Е.М. ЛАВРИЩЕВА

МЕТОДЫ ПРОГРАММИРОВАНИЯ

теория, инженерия, практика

Национальная академия наук Украины
Институт программных систем

Е.М. ЛАВРИЩЕВА

МЕТОДЫ ПРОГРАММИРОВАНИЯ

теория,
инженерия,
практика

КИЕВ НАУКОВА ДУМКА 2006

УДК 681.3.06

В монографии систематизированы и изложены теория, инженерия и практика современных методов программирования. Определены основы методов интеграции и преобразования разноязыковых программ и данных, рассмотрены методы спецификации и доказательства программ, верификации и тестирования в целях оценивания надежности и качества, а также представлены методы инженерии планирования и управления программными проектами. Показаны последние научные и практические достижения в области программирования, определен базис инженерии приложений и предметных областей, представлен комплексный подход к решению вопросов, возникающих при реализации программных проектов. Приведены перспективы развития программирования и инженерии предметных областей.

Для исследователей, разработчиков и менеджеров коллективов программистов, желающих ознакомиться с систематизированными знаниями по современной теории, инженерии и практике программирования и управления программными проектами, также для преподавателей информатики, которые готовят специалистов в области программной инженерии.

У монографії систематизовані і викладені теорія, інженерія та практика сучасних методів програмування. Визначені основи методів інтеграції і перетворення різномовних програм і даних, розглянуті методи специфікації і доведення програм, верифікації і тестування з метою оцінювання їх надійності і якості, а також представлені методи інженерії планування та керування програмними проектами. Показані останні наукові і практичні досягнення у галузі програмування, визначено базис інженерії застосувань і предметних областей, представлено комплексний підхід до вирішення питань, що виникають при реалізації програмних проектів. Наведені перспективи розвитку програмування і інженерії предметних областей.

Для дослідників, розробників і менеджерів колективів програмістів, що бажають ознайомитися з систематизованими знаннями із сучасної теорії, інженерії и практики програмування і керування програмними проектами, а також для викладачів інформатики, які готують спеціалістів у галузі програмної інженерії.

Утверждено к печати ученым советом
Института программных систем НАН Украины

Научно-издательский отдел физико-математической и технической литературы

Редактор М.К. Пунина

ISBN 978-966-00-0614-4

ОТ АВТОРА

В настоящее время во многих учебных заведениях в рамках компьютерных наук введена новая специальность – программная инженерия. Этим подтверждается, что в современных операционных условиях для разработки программного обеспечения требуются специалисты не только знающие конкретные языки программирования, средства и инструменты, но и владеющие основами инженерной деятельности – методами планирования, управления процессами разработки и оценивания качества, риска создания программного проекта и затраченных ресурсов (стоимостных, временных, технических и др.).

В данной книге представлена комплексная методология программирования, объединяющая теорию, инженерию и практику создания программного обеспечения с учетом современных сред и новых платформ компьютеров. В ней систематизированы базовые знания по двум основным направлениям программной инженерии:

– традиционное программирование – методы разработки требований, моделей предметных областей, проектирование и кодирование отдельных компонентов, их верификация или доказательство правильности и тестирование на наборах данных, а также методы интеграции и взаимодействия разноязыковых компонентов в новых средах, сетях и платформах компьютеров;

– инженерия программирования – методы повторного использования компонентов, оценки их применимости в новых проектах, планирование работ, распределение задач разработки по процессам жизненного цикла, управление коллективной разработкой, оценка качества, стоимости и сроков изготовления программного продукта.

Комплексная методология программирования – результат многолетних научных исследований и реализаций конкретных программных и фундаментальных проектов:

– системы программирования (АВТОКОД, АЛГАМС, язык типа SQL) для управляющей машины Днепр-2, которая применялась в 52 организациях бывшего СССР, в том числе и в Вычислительном центре двух металлургических комбинатов Германии для управления сложными технологическими процессами. Публикация – «АКД – автокод машины «Днепр-2» (Киев: ИК АН УССР, 1969);

– система автоматизации производства программ (АПРОП) для машин серии ЕС ЭВМ, в которой реализована связь разноязыковых модулей, записанных на языках программирования четвертого поколения (ФОРТРАН, АЛГОЛ, ПЛ/1, Ассемблер) путем генерации интерфейсных посредников, преобразующих нерелевантные типы данных для каждой

пары взаимодействующих модулей; созданы теоретические основы сборки программ, представленные в монографиях «Связь разноязыковых модулей в ОС ЕС» и «Сборочное программирование» (Москва: Финансы и статистика, 1985; 1991);

– технологическая подготовка разработки программных систем, прообраз современных производственных линий (Software Product Line), обеспечившая формализованное создание шести технологических линий для производства научно-аналитических задач и задач обработки данных в системе «ЮПИТЕР-470» (1987–1990);

– фундаментальные научные исследования в области программной инженерии в рамках десяти научных проектов Института программных систем НАН Украины, Госкомитета по науке и технике и Министерства науки и образования Украины. Опубликованы монография «Методы инженерии распределенных компьютерных приложений» (Киев, Наукова думка, 1997) и учебник «Основы программной инженерии» (Киев, Знання, 2001).

Автор в течение многих лет читает курс лекций «Технология программирования» студентам Киевского национального университета им. Тараса Шевченко (1978–1997), «Программная инженерия» – филиала Московского физико-технического института (2000–2006), Интернет-курсы Ukrsoftpro для повышения квалификации ИТ-специалистов (2003–2005), научный руководитель многих аспирантов и соискателей.

Выполненные проекты, проведенные многочисленные семинары, контакты с научными сотрудниками Института программных систем, других институтов, а также общения со студентами, аспирантами и выступления на международных конференциях УкрПрог (1998–2006) – источники формирования подхода автора к применению и развитию методов программирования, инженерных методов управления программными проектами и оценки их качества.

Автор благодарит коллег и сотрудников научного отдела Л.П. Бабенко, В.Н. Грищенко, Г.И. Коваль, Т.М. Коротун, Е.И. Моренцова и др., с которыми на протяжении многих лет совместной работы проводились исследования проблем программирования, разработки теоретического аппарата и программных средств для отдельных направлений программной инженерии, в том числе повторного использования, компонентного программирования, инженерии тестирования и качества, управления программными проектами, оценивания программных продуктов и процессов их изготовления.

СПИСОК УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ACM – Association for Computing Machinery
API – Application Program Interface
ER – Entity – Relationship
IEC – International Electro technical Commission
ISO – International System Organization
IDL – Interface Definition Language
ISO – International Standard Organization
ОМ – объектная модель
ООП – объектно-ориентированный подход
OMA – Object Management Architecture
ООМ – объектно-ориентированная методология
СОД – система обработки данных
ОС – операционная система
ORB – Object Request Broker
ПИК – повторного использования компонент
ПИ – программная инженерия
ПО – программное обеспечение
РП – распределенное приложение
ПС – программная система
COM – Component Object Model
CORBA – Common Object Request Broker Architecture
COP – Component-Oriented Programming
CBSE – Component-Based Software Engineering
CPM – Critical Path Method (метод критического пути)
PERT – Program Evaluation and Review Technique
PMBOK – Project Management Body of Knowledge
RMI – Remote Method Invocation
SQA – Software Quality Assurance (гарантия качества)
SWEBOK – Software engineering of body Knowledge
V&V – Verification and Validation (верификация и валидация)
ЖЦ – жизненный цикл

ПРЕДИСЛОВИЕ

Программирование возникло вместе с электронно-вычислительными машинами и является творческим процессом, который выполняет человек, обладающий математическими знаниями при осмыслении постановок задач, применении их к логике описания алгоритмов и составлении программ для получения решения на компьютерах.

Программа представляется в виде последовательности действий (команд, операторов в языке программирования, диаграмм, сценариев и др.), которая отображается транслятором или другим автоматизированным средством в последовательность инструкций, выполняемых компьютером.

Наиболее сложный и длительный путь, предшествующий достижению готовности программы решать конкретную задачу на компьютере – это проверка правильности ее составления. Для этого используются методы доказательства правильности программы с помощью набора аксиом и логических утверждений для вывода результата; методы поиска синтаксических и семантических ошибок системами программирования; верификация и тестирование программ на тестовых наборах и входных данных.

На протяжении полувековой истории возникали и исчезали те или иные направления в программировании, которые усовершенствовали процесс создания программ. В итоге программирование достигло такого уровня, что из творческого процесса превратилось в обыденную рутинную работу, которую могут выполнять все, кто поставил на стол компьютер и хочет его использовать для решения повседневных задач.

В результате, разработка и использование компьютерных программ стали массовой деятельностью. Около десятка миллионов человек занимаются программированием, а сотни миллионов – активно используют готовые программы в своей профессиональной деятельности. Сегодня практически нет ни одной сферы (экономика, медицина, бизнес, коммерция, промышленность и т.д.), где бы ни применялись компьютерные программы.

Основным источником компьютерной информации, используемой в программировании, является всемирная система Интернет,

содержащая множество готовых библиотек разного назначения, систем их поиска и инструментов для практического применения в создаваемых новых проектах. Несмотря на это богатство и достижения теории и практики программирования, многие новые проекты оказываются незавершенными из-за разного рода ошибок, несоблюдения требований, неслаженной работы коллектива разработчиков, недостаточного понимания применяемых методов, средств и инструментов.

Новые методы проектирования и спецификации программ (объектно-ориентированные, компонентные, сервисные, аспектные и т.д.) направлены на формализацию и систематизацию работ на проектах, наглядное представление их структуры (например, UML-метод) и снижение сложности за счет применения готовых программных продуктов. Современные методы верификации, доказательства правильности и тестирования уменьшают число ошибок и дефектов в программах.

Метод программирования – это конструктивные программные элементы и операции представления с их помощью алгоритмов программ в наглядном изображении (например, диаграммы UML, структурные блок-схемы и т.п.).

Разнообразие форм и языков описания структур программ, возможность рисования схем программ специалистами, не владеющими математическими знаниями и логикой описания алгоритмов, а также накопленные огромные интеллектуальные ресурсы в базах знаний, экспертных системах в 80-годы прошлого столетия привели к идее «программирования без программистов».

Эту идею предполагалось реализовать путем компьютеризации математических знаний (машины типа «Мир», японский проект «ЭВМ 5 поколения») для решения задач без программирования, а также фабрик программ по типу сборочного производства из готовых «деталей» (завод для сборки АСУ, система АПРОП и др.) и т.п. Однако уйти от программирования и освободить армию пользователей компьютеров от разработки программ не удалось.

Программы разрабатывались и будут, по-прежнему, разрабатываться, требуя все новых и совершенных методов и средств, а также методов эволюции и восстановления стареющих и уходящих в небытие программ в связи с появлением новых архитектур компьютеров и кластеров.

В настоящее время опять пришли к индустрии программных продуктов из повторно используемых компонентов (ПИК) в

рамках инженерии приложений (Engineering Application) и доменов (Domain Engineering), а также технологических линеек. Промышленный характер приобретает проблема преобразования и восстановления действующих программ для новой аппаратуры современных компьютеров. ПИК стал коммерческим продуктом, приносящим доход его разработчикам, дает значительную выгоду и при производстве новых продуктов, особенно при применении методов планирования и управления разработкой в целях балансирования сроков и стоимости. Этому в значительной степени способствует систематизация знаний в программной инженерии и стандартизация процессов разработки и управления проектами.

Систематизация знаний была проведена Международным комитетом при американском объединении компьютерных специалистов ACM (Association for Computing Machinery) и Институте инженеров по электронике и электротехнике IEEE Computer Society (1999–2004). В результате было создано ядро знаний SWEBOK (Software Engineering Body Knowledge) и определена дефиниция программной инженерии.

Программная инженерия – это система методов, способов и дисциплины планирования, разработки, эксплуатации и сопровождения программного обеспечения, способных к массовому воспроизводству.

Ядро SWEBOK стало справочным руководством по методам и средствам программной инженерии, которые могут применяться в процессах создания программного обеспечения:

– анализ требований к создаваемой системе;

– проектирование архитектуры системы;

– трансформация архитектуры в транслируемый код (кодирование);

– тестирование и верификация системы;

– обеспечение качества системы;

– проверка соответствия (верификация, валидация) реализованных спецификаций и требований заказчика;

– сопровождение и реинженерия программного продукта.

Стандарт ISO/IEC 12207:2002 регламентирует процессы жизненного цикла (ЖЦ) ПО (их более 40), устанавливает взаимосвязи между ними, определяет задачи и действия каждого процесса, а также рекомендует, что необходимо делать. Все процессы стандарта разделены на три категории:

– основные процессы (приобретение, поставка, разработка, эксплуатация и сопровождение);

– обеспечивающие процессы (документирование, управление версиями, верификация и валидация, просмотры, аудиты, оценивание и др.);

– организационные процессы (управление проектом, качеством, риском и др.), измерение продуктов, проверки качества и соблюдения стандартных положений.

Процессы, действия и задачи в стандарте приведены в наиболее общей естественной последовательности, из них можно создать свою модель ЖЦ, отображающую особенности реализации предметной области.

Модель жизненного цикла – это схема выполнения задач в рамках процессов, обеспечивающих разработку, эксплуатацию и сопровождение некоторого продукта, а также его развития, начиная с формулировки требований к нему и заканчивая прекращением применения.

Таким образом, стандарт определяет задачи процессов, а ядро знаний SWEBOK – методы и средства их выполнения.

Опыт разработки конкретных проектов, анализ направлений программирования и формирование новых теоретических основ определили содержание данной книги. В ней программирование рассматривается как процесс проектирования и разработки, а инженерия программирования как дисциплина планирования и управления разработкой, коллективом и риском, а также оценками продуктов и затрат на их изготовление.

В книге изложены современные методы программирования, их теоретические, инженерные и практические основы, в том числе методы разработки требований, моделей предметных областей, проектирование и кодирование отдельных компонентов, их верификация, доказательство правильности и тестирования, а также методы интеграции и взаимодействия разноязыковых компонентов в современных средах и платформах компьютеров. Рассмотрены инженерные методы планирования и управления программными проектами.

Книга состоит из четырех частей, заключения и двух приложений. Каждая часть состоит из глав, последовательно пронумерованных, и списка литературы для каждой главы. Рисунки, таблицы, теоремы и формулы пронумерованы по главам.

В первой части изложены современные, широко используемые прикладные (структурный, компонентный, аспектный и др.), теоретические (алгебраический, композиционный, алгебро-алгоримический, логико-алгебраический) и формальные (RAISE,

VDM) методы программирования. Определены их базовые понятия, свойства и процессы.

Во второй части проведен анализ и определены основы интеграции и преобразования разноязыковых программ и данных, передаваемых между взаимодействующими компонентами, в разных средах и платформах. Рассмотрены методы эволюции программ и систем (реинженерия, реверсная инженерия, рефакторинг), дана характеристика стандарта, определяющего независимость от языков типов и структур данных, а также рассмотрены принципы взаимодействия неоднородных компонентов в современных промежуточных средах.

Третья часть посвящена формальным спецификациям и методам доказательства правильности программ с помощью утверждений, пред- и постусловий; валидации требований и верификации спецификаций компонентов; методам тестирования и сбора данных о дефектах и отказах для использования их при оценке надежности программ.

В четвертой части определены инженерные основы программирования – ПИК и компоненты многоразового применения в семействах систем. Рассмотрены методы планирования и управления программными проектами, качеством и конфигурацией. Приведены модели инженерии качества и управления проектами.

В заключении описаны некоторые перспективные направления разработки и инженерии программирования.

Приложение 1. Приведено краткое описание областей знаний SWEBOK.

Приложение 2. Дана характеристика процессов ЖЦ стандарта ISO/IEC 12207.

ЧАСТЬ

1

ПРОГРАММИРОВАНИЕ: методы анализа и проектирования

ГЛАВА 1

МЕТОДЫ АНАЛИЗА ПРЕДМЕТНОЙ ОБЛАСТИ И ОПРЕДЕЛЕНИЯ ТРЕБОВАНИЙ

Разработка ПС начинается с анализа предметной области, которая автоматизируется, для выделения в ней объектов, отношений между ними и операций пополнения объектов модели ПрО новыми детализированными характеристиками. Наиболее разработанным методом программирования является объектно-ориентированный. Он положен в основу большого количества разных подходов к определению объектной модели и процесса проектирования по ней ПС. В общем случае построенная ОМ отображает концептуальную модель ПрО, способствует формированию требований к отдельным объектам системы или ко всей системе, позволяет спроектировать архитектуру системы и ее ПО.

При построении модели ОМ в других подходах главное внимание уделяется выявлению функциональных задач и требований к их реализации и выполнению.

Требования формируются разработчиком и заказчиком системы, они документируются и утверждаются, являясь при этом основой дальнейшей реализации проекта системы.

Далее рассматриваются особенности проведения анализа ПрО в целях построения ОМ и формирования требований к системе и ПО.

1.1. МЕТОДЫ АНАЛИЗА ПРЕДМЕТНОЙ ОБЛАСТИ

1.1.1. Краткий обзор методов анализа ПрО

В теории и практике программирования уже давно применяют методы анализа и проектирования моделей ПрО, в частности концептуальной модели. Среди них в практической деятельности наиболее используемым является классический объектно-ориентированный метод и его разновидности:

– метод объектно-ориентированного анализа систем (Object-Oriented system analysis by Shlaer and Mellor – OOAS-SM) [1] обеспечивает выделение сущностей, объектов ПрО, определение их свойств и отношений, а также создание информационной модели, модели состояний объектов и модели процессов прохождения потоков данных (dataflow) в системе;

– метод объектно-ориентированного анализа (Object-Oriented analysis by Coad and Yourdan – OOA) [2] обеспечивает моделирование ОМ, используя понятие «сущность–связь», спецификацию потоков данных и соответствующих им процессов с помощью диаграмм, а также формирование требований к ПрО;

– метод структурного проектирования (Structured Design by Yourdan and Constantine – SD) [3–5] позволяет представить спецификации требований, структуры данных и программ преобразования входных данных в выходные с помощью структурных карт Джексона;

– методология объектно-ориентированного анализа и проектирования (Object-oriented analysis and design by Martin and Odell – OOAD) [6] основывается на ER-моделировании [7] модели ПрО, определении функций системы и способов организации данных с использованием диаграмм «сущность-связь», структурных диаграмм и матрицы информационного управления;

– технология объектного моделирования (Object modeling Technique by Rumbaugh – OMT) [8, 9] включает в себя процессы анализа, проектирования и реализации, нотации для задания четырех моделей (объектной, динамической, функциональной и взаимодействия) на процессах ЖЦ;

– метод объектно-ориентированного программирования Буча [10] ориентирован на анализ и выделение объектов ПрО, объединение объектов в классы, суперклассы, а также представляет аппарат наследования, полиморфизма и упрятывания информации об объектах;

– метод вариантов использования Джекобсона [11] – это задание сценариев системы в виде диаграмм требований, целей и задач ПрО и набор правил их трансформации в систему взаимодействующих объектов системы;

– метод моделирования в системе CORBA [12, 13] базируется на эталонной объектной модели ПрО, строящейся из предметов реального мира со своими характеристиками, типами и операциями, группируемыми в классы или суперклассы. Для обеспечения функционирования объектов ОМ в распределенной среде система имеет набор сервисов (facilities) общего пользования;

– метод моделирования UML обеспечивает анализ ПрО, моделирование объектов ПрО, представление требований и архитектуры системы с помощью множества диаграмм в наглядной форме [14];

– метод генерации в порождающем (generative) программировании использует основные возможности ООП, компонентного и аспектного проектирования систем [15], а также компоненты многоразового использования, функциональные и нефункциональные требования к системе, отображенные в модели характеристик ПрО.

В о*бъектной модели* стандарта CORBA объект обозначает некоторую сущность ПрО, она именуется и определяет один ли более сервисов, ориентированных на пользователя. Объекты группируются в типы, которые определяют операции над экземплярами, объединенными в подтип/супертип, в случае их одинакового поведения. Объекты инкапсулируют методы их реализации, которые невидимы во внешнем интерфейсе.

К объектам применяются операции, вызывающие соответствующий метод обработки. Совокупность операций, которые связаны с объектом, определяют его поведение. Каждый объект имеет состояние, включающее в себя информацию, необходимую для выполнения операций. Объект состоит из типа, специализация которого определяется постепенно на этапах анализа, проектирования и реализации.

Исходя из краткого обзора приведенных основных подходов к проектированию ОМ ПрО, можно сделать вывод, что они имеют много общих черт (например, ER-моделирование) и свои специфические особенности, отличающие один метод от другого.

1.1.2. Метод анализа и построения объектной модели ПрО

Предлагается метод, основанный на объектно-ориентированном подходе, теории множеств для формализации выявления и представления объектов и их отношений в виде графовой структуры [16].

Объекты ПрО – это абстрактные образы ПрО со множеством свойств и характеристик, их определение зависит от уровня абстракции и совокупности полученных о них знаний. Спецификация объекта включает в себя:

<имя объекта > <концепт>,

где <имя объекта> – идентификатор, строка из литер и десятичных чисел; <концепт> – некоторый денотат, определяющий объект реального мира в соответствии с интерпретацией сущности моделируемой ПрО.

Под *предметной областью (доменом)* понимается совокупность объектов и связей между ними. Объекты снабжаются описанием свойств и характеристик, специфических для ПрО, и задач, которые выполняются в системе. Пространство ПрО делится на пространство задач и решений. Пространство задач – это сущности, концепты, понятия ПрО, а пространство решений – это множество функциональных компонентов, которым соответствуют задачи ПрО, заданные с помощью понятий, концептов этих задач.

Модель ПрО строится с использованием словаря терминов, точных определений терминов этого словаря, характеристик объектов и процессов, которые протекают в системе, а также множества синонимов и классифицированных логических взаимосвязей между этими терминами.

Концептуальная модель – это модель ПрО из сущностей и отношений, разработанная без ориентации на программные и технические средства выполнения задач ПрО, которые будут добавляться в дальнейшем в процессе проектирования системы.

Под концептом будем понимать абстрактное собрание сущностей ПрО, имеющих одни и те же характеристики. Все сущности ПрО, объединяющие концепты, согласуются с характеристиками, которые в системе ассоциируются с атрибутами. Каждый концепт в модели обозначается уникальным именем и идентификатором. Группа подобных концептов – это родительский концепт, который определяется заведомо известным набором общих атрибутов для данной группы концептов.

Атрибут – это абстракция, которой владеют все абстрагированные концепты сущности. Каждый атрибут обозначается именем, уникальным в границах описания концепта. Множество объединенных в группу атрибутов, имеет идентификатор группы атрибутов. Множество идентификаторов групп могут быть объединены в класс и иметь идентификатор класса.

Концепт вместе со своими атрибутами в концептуальной модели можно представить в графическом или текстовом виде.

Отношения – это абстракция набора связей, которые существуют или возникают между разными видами объектов ПрО, абстрагированные как концепты. Каждая связь имеет уникальный идентификатор. В информационной модели объекты и отношения представляются графически. Для формализации отношений между концептами к ним добавляются вспомогательные атрибуты, ссылки на идентификаторы отношений. Некоторые отношения образуются как следствие существования других отношений.

Выделение сущностей ПрО проводиться с учетом различий, определяемых соответствующими понятийными структурами. Объекты как абстракции реального мира обладают поведением, обусловленным свойствами и отношениями с другими объектами, они структурно упорядочиваются в графе посредством применения теоретико-множественных операций (принадлежности, объединения, пересечения, разности и др.) ко множеству (классу) объектов, позволяющих установить фактические отношения между объектами в виде

– элемент множества–элемент множества,
– элемент множества–множество,
– множество–элемент множества,
– множество–множество.

В результате проведения структурной упорядоченности объектов уточняется объектный граф $G = \{V, A\}$, определенный на множестве объектов V и связей A и удовлетворяющий следующим основным требованиям:

– множество вершин V представляет собою взаимнооднозначное отображение множества объектов ПрО;
– для каждой вершины должна существовать хотя бы одна связь, принадлежащая множеству отношений – связей A;
– существует хотя бы одна вершина, обладающая статусом множество–объект, отображающая ПрО в целом.

Под *связью* понимается абстракция отношения, систематически возникающего между различными абстрактными объектами.

Связь задается в графе дугой (линией) между двумя объектами и может быть следующих видов: один к одному, один ко многим, многие ко многим.

Связь один к одному соответствует отношению «элемент множества–элемент множества» и существует тогда, когда один экземпляр некоторого объекта множества связан с единственным экземпляром другого элемента множества (один интерфейс). Одиночная связь изображается на графе линией.

Связь один ко многим соответствует отношению «элемент множества–множество» и существует тогда, когда один экземпляр некоторого объекта связан с одним или более экземплярами другого объекта и каждый такой экземпляр связан только с одним экземпляром первого. Изображается на графе линией со стрелкой в сторону многих.

Связь многие ко многим соответствует отношению «множество–множество» и существует тогда, когда один экземпляр некоторого объекта связан с одним или бóльшим количеством экземпляров другого и каждый экземпляр второго объекта связан с одним или более экземпляром первого (множественный интерфейс). Множественная связь изображается на графе линией с двумя стрелками.

Пример построения графа $G = \{V, A\}$ (рис. 1.1) включает в себя множество элементов $V = \{O_1, O_2, ..., O_8\}$, расположенных в вершинах графа.

По мере детализации граф может дополняться новыми объектами, затем структурно упорядочиваться (снизу–вверх) для проведения контроля полноты, избыточности элементов графа и устранения дублирующих объектов-вершин в роли одного и того же элемента множества V или объектов, обнаруженных без связей.

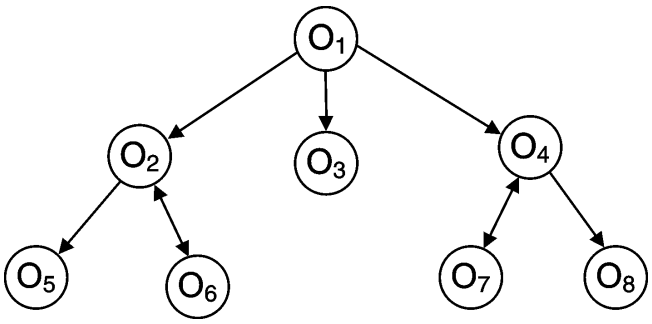

РИС. 1.1. Пример графа некоторой ПрО

Граф G и множество объектов ПрО, которые отличаются друг от друга статусом (элемент, множество или множество элементов) и взаимным порядком, образуют объектную модель. Для объектов ОМ определяются общие и индивидуальные свойства и характеристики, которые соответственно являются внешними и внутренними. Внешние характеристики отображают проблемную ориентацию объектов и их статус как элементов множества. Внутренние характеристики отображают внутренние свойства, входящие в состав отдельных внешних характеристик, которым соответствуют множества свойств объектов с различными типами.

Проверка свойств объектов в графе осуществляется с помощью операций экземпляризации, классификации, специализации, агрегации и др. путем попарного сравнения свойств внутренних характеристик объекта со множеством свойств внешних характеристик объекта. Свойства считаются проверенными, если выполняется условие соответствия внутреннего свойства объекта–множества эквивалентному ему внешнему свойству объекта–элемента. Если это условие не выполняется, то такой элемент удаляется соответственно из списка элементов множества и из графа.

1.1.3. Сценарный подход к определению модели ПрО

Рассматривается метод анализа ПрО, предложенный Джекобсоном [3]. Он обеспечивает последовательное выявление объектов, существенных для домена, и дает рекомендации относительно того, с чего начинать движение к пониманию проблемы и поиску существенных для ПрО объектов. Метод базируется на сценариях (use case) системы, которую требуется построить [17]. Разработка модели системы по данному методу начинается с осмысления того, для кого и для чего создается ПС.

Формируется общая цель системы, которая выражается через отдельные подцели. Цели могут отвечать функциональным или нефункциональным требованиям и проектным решениям. Цели являются источником требований к системе и способом их оценки, а также выявления противоречий между требованиями и реализуемыми функциями системы.

После определения целей определяются носители интересов, которым отвечает каждая цель, и возможные примеры удовлетворения целей в виде сценариев работы системы, помогающие пользователю получить представление о функциях системы и требованиях к ее разработке.

В результате последовательной декомпозиции проблемы и представления ее в виде совокупности целей, каждая из которых трансформируется в совокупность вариантов использования (сценариев), в процессе анализа трансформируются в совокупность взаимодействующих объектов.

Абстракция определенной роли пользователя, представленной сценарием, и обмена информацией с системой называется *актером*. Это абстрактное понятие обобщает понятие действующего лица системы. Роли являются носителями целей и постановщиками задач, для решения которых создается система.

Цепочка трансформации

проблема → цели → сценарии → объекты

отражает степень концептуализации понимания проблемы, последовательного снижения сложности и повышения уровня формализации. Трансформация выражается в терминах базовых понятий ПрО и отображает связь актеров и сценариев. Каждый сценарий инициируется определенным пользователем, являющимся носителем интересов системы.

Актер по отношению к системе является внешним фактором, его действия носят недетерминированный характер, чем подчеркивается разница между пользователем и ролью, которую может играть любое лицо в автоматизируемой системе.

В роли актера может выступать ПС, если она инициирует выполнение определенных работ данной системы. Соответственно, и актер, как абстракция внешнего объекта может быть человеком или внешней системой. Экземпляр актера запускает ряд операций в системе, которые задаются *сценарием*. Вариант сценария существует, пока он выполняется и его можно считать экземпляром класса, в роли которого выступает описание транзакции.

Сценарий обладает состоянием и поведением. Каждое взаимодействие между актером и системой это новый сценарий– объект. Если несколько сценариев системы имеют одинаковое поведение, то их можно рассматривать как класс сценариев. Модель системы основывается на сценариях и каждое внесение в нее изменений должно осуществляться повторным моделированием действий актеров и запускаемых ими сценариев.

Построенная с помощью сценариев модель ПрО отражает требования пользователей, которые могут изменяться по их предложениям, поскольку совокупность сценариев системы определяет все возможные пути использования системы, а актер обслуживает эту совокупность сценариев. Сценарий системы –

это полная цепочка событий, инициированных актерами, и специфицированными реакциями на них.

Между сценариями определены два типа отношений:

1. *Отношение "расширяет"* означает, что функции одного сценария является дополнением к функциям другого. Применяется это отношение в случае, когда имеется несколько вариантов одного и того же сценария. Инвариантная его часть изображается как основной сценарий, а отдельные варианты как расширения. При этом основной сценарий устойчив, не меняется при расширении вариантов функций и не зависит от них. При выполнении сценария расширение прерывает выполнение основного сценария, который будет продолжен после выполнения сценария расширения.

2. О*тношение "использует"* означает, что некоторый сценарий может быть применен как расширение для нескольких других сценариев (аналог процедуры в языках программирования).

Оба отношения – это средство определения наследования, если сценарии считать объектами. Различие между ними состоит в том, что при расширении функций системы, она рассматривается как дополнение к основной функции и может быть понятна только в паре с ней. Относительно понятия "использует" дополнительная функция имеет самостоятельное определение и может применяться во всех сценариях.

Таким образом, продукт анализа ПрО состоит из трех частей:

1) совокупность понятий (объектов) домена;

2) модель сценариев (диаграммы сценариев);

3) описание интерфейсов сценариев.

Модель сценариев сопровождается неформальным описанием каждого из сценариев. Нотация такого описания не регламентируется. Описание варианта сценария включает в себя:

– название, которое помечает сценарий на диаграммах и служит средством ссылки на сценарий;

– аннотацию (краткое содержание в неформальном представлении);

– актеров, которые могут запускать сценарий;

– определение всех аспектов взаимодействия системы с актерами (возможные действия актера и их последствий, запрещенные действия актера);

– предусловия, определяющие начальное состояние на момент запуска сценария, необходимое для успешного выполнения;

– функции, которые реализуются при выполнении сценария и определяют порядок, содержание и альтернативу действий, выполняемые в сценарии;

– нестандартные ситуации, которые могут появиться при выполнении сценария и требовать специальных действий для их устранения (например, ошибка в действиях актера, которую может распознать система);

– реакции на предвиденные нестандартные ситуации;

– постусловия, которые определяют конечное состояние сценария при его завершении.

На дальнейших стадиях сценарий трансформируется в сценарий поведения системы. К элементам модели добавляются элементы, связанные с проектированием проблемы и заданием нефункциональных требований, а именно:

– запуск сценария;

– ввод данных;

– отработка чрезвычайных ситуаций и др.

Следующим шагом работы со сценарием системы является тестирование продукта, полученного по заданному сценарию (см. главу 7).

1.1.4. Определение модели взаимодействия объектов для распределенной среды

Одним из условий построения модели взаимодействий объектов в распределенной среде является определение типов взаимодействий объектов ОМ между собой в зависимости от расположения объектов относительно друг друга (локально или удаленно) в распределенной среде. Аспектам взаимодействия объектов и агентов посвящен ряд работ [16–19], в том числе работа [20], в которой рассматриваются вопрос взаимосвязей объектов через интерфейс объектов и доказательство его корректности.

Преобразование ОМ к модели взаимодействия проводиться посредством добавления для каждой удаленной вершины графа новой дополнительной вершины, расположенной на следующем, нижнем уровне иерархии ветви дерева и выполняющей функции взаимосвязи объектов (stub-интерфейс [4, 9]). Это преобразование позволяет получить из объектного графа G модели ПрО расширенный граф, отображающий распределение объектов в среде,

$U = \{W, L\},$

где L – множество дуг – связей удаленных объектов; $W = \{V', I\}$ – множество вершин, включающее в себя подмножество $V' = \{v_1', v_2', ..., v_r'\}$, $r \in k$, подмножество удаленных объектов, распределенных в разных узлах сети, и $I = \{ i_1, i_2 ,..., i_r \}$ – множество интерфейсов для вершин множества V'.

Каждый элемент множества V' определяет интерфейс $\text{stub}_j = [v_j, b(v_j), sp(v_j)]$, который включает в себя: $v_j \in V'$ – объект, расположенный в некотором узле и содержащий описание внешних переменных, обеспечивающих взаимодействие с другим объектом множества V'; $b(v_j)$ – операторы передачи сообщений удаленной вершине, которые определены на множестве внешних переменных EX, распределенных переменных $R(V_j')$ или переменных интерфейса $b(v_j') = EX \cup R$; $sp(v_j)$ – спецификация интерфейса объекта – stub, определенная на множестве входных и выходных параметров из $EX\ X \cup R$ и задающая поведение вершины V_j' при отправке сообщения в сеть.

Stub-интерфейс j-объекта включает в себя:

– описание всех переменных, которыми обмениваются данный объект и теми объектами, с которыми он взаимодействует через протоколы сетевой среды;

– декларацию начальных значений распределенных переменных;

– тело интерфейса $b(v_j)$, состоящее из операторов вызовов удаленных объектов, эквивалентных операторам передачи сообщений по сети и описанных на множестве внешних, «private» и интерфейсных переменных;

– спецификацию объекта $sp(v_j)$ в виде определения распределенных интерфейсных переменных, задающих поведение объекта при передаче сообщений по сети.

Свойство удаленного распределения объектов, а также связанных с ними stub-интерфейсов отмечается в графе W, с затемненными вершинами (рис. 1.2). Граф с выделенными (затемненными вершинами) удаленными и интерфейсными вершинами $U = \{W,L\}$ будем называть *распределенным.*

Пути между взаимодействующими объектами в графе, в котором хотя бы один из объектов является, удаленным будем называть *распределенной схемой (m, n)* взаимодействия пары объектов (O_i, O_i') и (O_i', O_j).

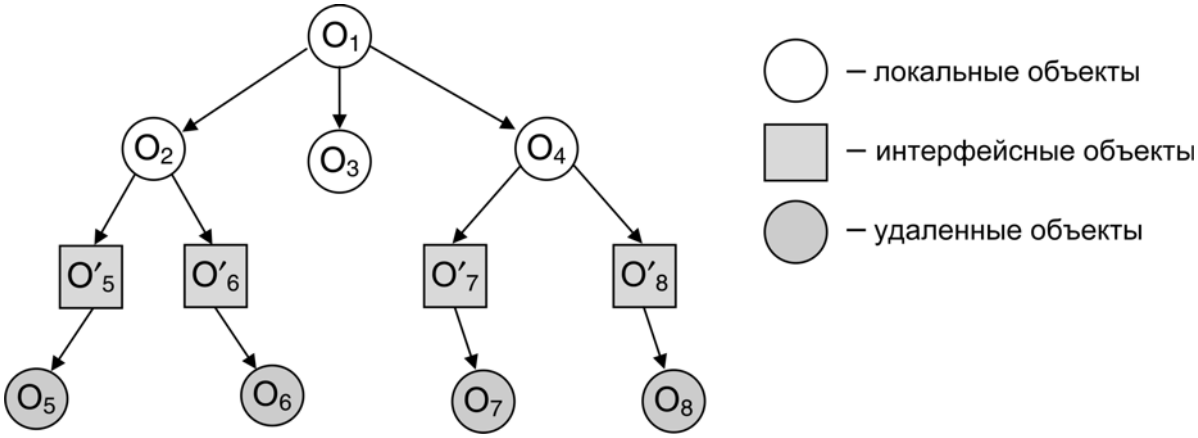

РИС.1.2. Пример модели распределенных объектов

В общем случае (x, y) − число входных и выходных параметров сообщений объектов множества O; $O = (O_1, O_2, ..., O_к)$ − исходные объекты схемы; $O' = (O'_1, ... O'_n)$ − E, интерфейсные объекты.

На содержательном уровне множество V' представляет собой набор методов реализации удаленных объектов, для каждого из которых существует элемент множества I , определяющий stub-интерфейс. Каждый метод реализации объекта задается графом управления, вершины которого распадаются на вершины ветвления и преобразования, которым сопоставляются операторы передачи сообщений, обеспечивающие связь между объектами графа распределения W.

Частным случаем распределенной схемы является последовательная (m, n)-схема, для которой во множестве W_i нет удаленных объектов. Такая схема соответствует локальной программе клиента и отображает традиционную структуру программы.

В распределенном графе установленные ранее связи объектов эквивалентны следующим типам взаимодействий:

Определение 1. Связь «локально-локальная» соответствует взаимодействию программа−программа, при котором программы обращаются друг к другу через механизм вызова процедур, расположенных на одном компьютере.

Определение 2. Связь «локально-удаленная» соответствует взаимодействию типа клиент−сервер, при котором локальная программа клиента посылает запрос-сообщение серверу, расположенному на другом компьютере.

Определение 3. Связь «удаленно-удаленная» соответствует взаимодействию типа сервер−сервер, при котором один удаленный объект обращается к другому для получения сервиса.

Определение 4. Связь «удаленно-локальная» соответствует взаимодействию типа сервер−клиент, при котором сервер посылает сообщение программе клиента с результатом выполнения сервиса.

Перечисленные типы взаимодействий, кроме первого, определяются на множестве внешних характеристик объектов. Их описание и задание удаленного вызова для каждого распределенного объекта представляется в виде дополнительного stub-объекта, выполняющего функцию интерфейса между двумя взаимодействующими объектами. Такой объект вносится в граф $U = \{W, R\}$ как промежуточная вершина между двумя взаимодействующими объектами.

Семантически граф распределения модели взаимодействия обеспечивает расположение методов (реализаций) удаленных объектов в разных узлах сетевой среды, т.е. на разных компьютерах, а интерфейсные объекты и неотмеченные вершины, принимаемые за локальные объекты, располагаются на одном компьютере, а именно, компьютере клиента. Для привязки графа распределения к конкретной распределенной среде, как правило, выполняются следующие действия:

− определение критериев и условий функционирования объектов в конкретной среде в целях эффективного их расположения в наиболее подходящих узлах сети;

− «привязка» объектов приложения к соответствующим узлам сети на основе связей объектов в графе распределения;

− определение средств задания интерфейсов объектов и соответственно типов протоколов передачи запросов от одного объекта к другому при их взаимодействии;

− интеграция реализаций объектов и их интерфейсов в систему связей и отношений для получения прототипа приложения.

Данные действия базируются на использовании стандартных сетевых средств обеспечения распределения данных, процессов и управления:

1. Распределение данных − совместное использование удаленно расположенных файлов или баз данных с помощью сервера, который для различных клиентов сети обеспечивает связь 3, 4 типов.

2. Распределение процессов соответствует модели распределения вычислительных ресурсов при выполнении обработки данных на синхронных и параллельных процессорах узлов сети модели клиент−сервер (для связи 2, 4 типов).

3. Распределение управления – это разделение функций распределенной системы на логически независимые программно-управляемые сервисные компоненты системы для обеспечения одновременного доступа к ним разных клиентов: управление данными, диалогом, транзакциями, безопасностью данных, вычислениями программных компонентов и др.

Фактически эти системные функции сети определяют сервис распределенной системы и на их основе проводиться реализация всех типов связей 1–4. Данные функции являются основополагающими при реализации задач удаленного вызова объектов приложения на основе графа распределения.

Характерной особенностью рассмотренных методов проектирования модели ПрО является ориентация на определение требований заказчика к ПрО, ПС, а также к функциям и задачам системы.

1.2. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К СИСТЕМЕ

Задачи формирования требований к ПС определены в стандарте ISO/IEC 12207: 2002 [19] (Приложение 2), в ядре знаний программной инженерии SWEBOK [20, 21] (Приложение 1), а также в ряде работ [22–25]. В них приведены основные понятия и принципы формирования требований. Главными процессами в стандарте являются: выявление требований и анализ требований к системе. В ядре SWEBOK приведены соответственно цели и задачи определения требований, а также инженерия требований.

1.2.1. Общие понятия проблематики определения требований

Процесс определения требований на разработку системы и ПО с применением инженерных технологий получил название инженерии требований.

Инженерия требований к ПО представляет собой дисциплину анализа ПрО для формирования и документирования требований, согласованных с заказчиком, а также преобразование заданных требований к системе в описание требований к ПО, их спецификация и верификация. Действующими лицами, обеспечивающими управление и формирование требований, являются представители организации разработки ПО и заказчика.

Требования – это свойства, которыми должно обладать ПО для адекватного задания функций, а также условия и ограниче-

ния на ПО, данные, среду выполнения и технологию обработки. Требования отражают потребности заказчиков и пользователей ПО, а также разработчиков, заинтересованных в создании ПО. Заказчик и разработчик совместно проводят сбор требований, их анализ, пересмотр, определение необходимых ограничений и документирование сформированных требований.

Различают требования к продукту и к процессу, а также функциональные и нефункциональные требования и требования к системе, которые определяют требования к процессу, ОС, режиму выполнения ПО, выбору платформы и т.п. Функциональные требования предъявляются к назначению и функциям системы, а нефункциональные – к условиям выполнения ПО. Требования к продукту – это требования к представлению структуры ПС, строящейся из взаимосвязанных программных и аппаратных подсистем и разных приложений. Требования к процессу предъявляются с учетом положений и рекомендаций стандартов.

Функция – это набор действий, выполнение которых возлагается на элемент системы при заданных требованиях, условиях и ограничениях, она отображает назначение, обязанность или роль элемента в системе.

Некоторые требования могут задаваться в количественном виде (например, количество запросов в секунду, средний показатель ошибок не должен превышать 1,5% объема вводимой информации и т.п.). Значительная часть требований относится к функциям и атрибутам качества: безотказность, надежность и др.

Управление требованиями к ПО заключается в планировании и в контроле формирования требований, а также в выполнении проектных решений и ресурсов в процессе разработки компонентов системы на этапах ЖЦ.

Процесс улучшения требований – это уточнение отдельных характеристик и атрибутов качества (надежность, реактивность и др.), которыми должна обладать система и ПО, методы их достижения на этапах ЖЦ и обеспечения адекватности процессов реализации требований.

1.2.2. Классификация требований

Формируемые требования к системе разделены на две категории:

– функциональные требования, которые отображают возможности проектируемой системы;

– нефункциональные требования, которые отображают ограничения, определяющие принципы функционирования системы и доступа к данным системы пользователей.

Первая из приведенных категорий дает ответ на вопрос "что делает система", а вторая определяет характеристики ее работы, например, вероятность сбоев системы на некотором промежутке времени, доступ до ресурсов системы разных категорий пользователей и др.

Функциональные требования характеризуют функции системы или ее ПО, а также способы поведения ПО в процессе выполнения функций и методы передачи и преобразования входных данных в результаты. *Нефункциональные требования* определяют условия и среду выполнения функций (например, защита и доступ к БД, секретность и др.). Разработка требований и их локализация завершаются на этапе проектирования архитектуры и отражаются в специальном документе, по которому проводится окончательное согласование требований для достижения взаимопонимания между заказчиком и разработчиком.

Функциональные требования связаны с семантическими особенностями ПрО, для которой планируется разработка ПС. Важным фактором является проблема использования соответствующей терминологии при описание моделей ПрО и требований к системе. Один из путей ее решения – стандартизация терминологии для нескольких ПрО (например, для информационных технологий, систем обеспечения качества и др.). Тенденция к созданию стандартизированного понятийного базиса для большинства ПрО отражает важность этой проблемы в плане обеспечения единого понимания терминов, используемых в документах, описывающих требования к системе и к ПО.

Нефункциональные требования могут иметь числовой вид (например, время ожидания ответа, количество обслуживаемых клиентов, БД и др.), а также содержать числовые значения показателей надежности и качества работы компонентов системы, период смены версий системы и др. Для большинства ПС, с которыми будут работать многие пользователи, требования должны выражать такие ограничения на работу системы:

– конфиденциальность;
– отказоустойчивость;
– одновременность доступа к системе пользователей;
– безопасность;
– время ожидания ответа на обращение к системе;

– ограничения на исполнительские функции системы (ресурсы памяти, скорость реакции на обращение к системе и т.п.);

– регламентации действующих стандартов, упрощающих процессы организации формирования требований и менеджмента.

Иными словами, для каждой системы формулируются нефункциональные требования, относящиеся к защите от несанкционированного доступа, регистрации событий системы, аудиту, резервному копированию и восстановлению информации. Эти требования на этапе анализа и проектирования структуры системы должны быть определены и формализованы аналитиками. Для обеспечения безопасности системы должны быть определены категории пользователей системы, которые имеют доступ к тем или иным данным и компонентам.

Определяются объекты и субъекты защиты. На этапе анализа системы решается вопрос: какой уровень защиты данных необходимо предоставить для каждого из компонентов системы, выделить критичные данные, доступ к которым строго ограничен. Для этого применяется система мер по регистрации пользователей, т.е. идентификации и аутентификации, а также реализуется защита данных, ориентированная на регламентированный доступ к объектам данных (например, к таблицам, представлениям). Если же требуется сделать ограничение доступа к данным (например, к отдельным записям, полям в таблице), то в системе должны предусматриваться определенные средства (например, мандатная защита).

Для обеспечения восстановления и хранения резервных копий БД, архивов баз данных изучаются возможности, предоставляемые СУБД, и рассматриваются способы обеспечения требуемого уровня бесперебойной работы системы, а также организационные мероприятия, отображаемые в соответствующих документах по описанию требований к проектируемой системе.

В целях описания нефункциональных требований к системе и фиксации сведений о способности компонента обеспечивать несанкционированный доступ к нему неавторизованных пользователей языки спецификаций компонентов дополняются средствами описания нефункциональных свойств. Эта группа свойств целиком связана с сервисом среды функционирования компонентов, ее способностью обеспечивать меры борьбы с разными угрозами, которые поступают извне от пользователей, не имеющих прав доступа к некоторым данным или ко всем данным системы.

Процесс формализованного описания функциональных и нефункциональных требований, а также требований к характеристикам качества с учетом стандарта качества ISO/IEC 9126–94 будет уточняться на этапах ЖЦ ПО и специфицироваться. В спецификации требований отражается структура ПО, требования к функциям, качеству и документации, а также задается архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные требования и требования к взаимодействию с другими компонентами и платформами (БД, СУБД, сеть и др.).

1.2.3. Модель требований

Модель требований дает обобщенное представление о будущих услугах системы для ее клиентов (актеров). Требования является предметом анализа ПрО в целях дальнейшего структурирования проблемы на отдельные функции и выработки правил их выполнения в заданной среде. Основу составляет объектная архитектура, результатом структурирования которой должна быть совокупность объектов, полученная путем последовательной декомпозиции каждого из сценариев на объекты с действиями сценария, а также их взаимодействие, определяемое функциональностью системы.

На стадии анализа требований определяются:

– функциональные и нефункциональные требования к системе;

– объекты системы и модель сценариев;

– описание интерфейсов сценариев и актеров;

– диаграммы взаимодействия объектов сценариев.

Основная стратегия формирования требований базируется на таких постулатах:

– каждое требование относится к условиям выполнения функций и данных системы (защита, конфиденциальность, тип доступа и др.);

– требования неизбежно изменяются;

– изменения соответствующих требований к системе приводит к модификации объекта;

– объект должен быть стойким к модификации системы, которая может изменить отдельные требования.

Объекты подразделяются на следующие типы:

– объекты-сущности;

– объекты интерфейса;

– управляющие объекты.

Объекты-сущности моделируют в системе информацию, хранящуюся после выполнения сценария. Обычно, им отвечают реальные сущности, которые находят свое отображение в базах данных. Большинство из них может быть выявлено из анализа проблемной области, на которую ссылаются в сценариях.

Объекты интерфейса включают в себя функциональности, зависимые непосредственно от окружения системы и определенные в сценарии. С их помощью актеры взаимодействуют со сценариями системы – их задачей является трансляция информации, которую вводит актер, в события, на которые реагирует система, и обратная трансляция событий системы в вывод для актера. Такие объекты определяются путем анализа описаний интерфейсов сценариев модели требований и анализа действий актеров по запуску каждого из соответствующих ему сценариев.

Управляющие объекты – это объекты, которые превращают информацию, введенную объектами интерфейса и представленную объектами-сущностями, в информацию, выводящуюся интерфейсными объектами. Они соединяют объекты в цепочку событий и взаимодействий объектов.

Такое разделение объектов служит целям локализации изменений в построенной системе. При преобразовании модели требований в процессе анализа каждый сценарий разбивается на эти три типа объектов. При этом один и тот же объект может присутствовать в нескольких сценариях, другие, чтобы унифицировать их функции и определять их как единый объект. Критерий распознавания такой: если различные сценарии ссылаются на один и тот же экземпляр объекта, то это один и тот же объект.

При выделении объектов формируется базис архитектуры системы как совокупности взаимодействующих объектов, для каждого из которых можно проследить связь с соответствующим сценарием модели требований и провести трассирование требований при переходе от модели требований к модели системы.

Дальнейшая детализация объектов реализуемой проблемы проводится на следующих этапах ЖЦ.

1.2.4. Базовые процессы инженерии требований

Основу инженерии требований составляет совокупность процессов, состав которых можно считать регламентированным в стандарте ISO/IEC 12207 и в ядре знаний SWEBOK. Основные из них следующие:

1. Обсуждение проекта системы. Начальные действия, которые позволяют сделать выводы относительно целесообразности и реальности выполнения проекта системы, которую планирует заказчик или уже заказал.

2. Выявление и сбор требований. Идентификация источников информации относительно системы и сбор информации относительно требований к проектируемой системе.

3. Анализ требований. Исследование собранной информации, определение разногласий в понимании отдельных целей или задач системы со стороны заказчика и разработчика, систематизация и структуризация собранной информации.

4. Управления требованиями и контроль за изменениями отдельных требований.

5. Проверка (инспекция) систематизированных требований заинтересованными сторонами в целях заверения, что требования отвечают целям системы. Согласование требований между их авторами и заказчиком завершается документированием в понятном для всех заинтересованных сторон виде.

Каждый из приведенных пяти процессов связан с выполнением определенной работы, завершение которой считается переходом к следующему процессу или возвращению к предыдущему для уточнения принятых требований. Результатом работы считается фиксация определенных соглашений и решений. Методы для выполнения отдельных работ являются взаимосогласованными, базируются на единой понятийной базе и в совокупности составляют методологию инженерии требований, результатом которой есть документы, фиксирующие требования.

Далее приводятся описание процессов формирования требований, основные задачи которых описаны в ядре SWEBOK (Приложение 1).

Обсуждение проекта системы. Этот вид работ ставит своей целью выработать первые выводы относительно целесообразности выполнения проекта и прогнозировать реальность его выполнения в заданные сроки и за стоимость, которые определяет заказчик.

ПС можно рассматривать как набор услуг для выполнения определенной профессиональной деятельности в заданной ПрО. Лицо, которое заказало проект системы, должно получить от системы определенные услуги, т.е. заказчик услуг является участником системы. К участникам относятся также операторы, менеджеры разных уровней, бухгалтеры и т.п. Именно они будут

обращаться к системе за услугами и получать от нее сообщения, реагировать на них в соответствии с профессиональными обязанностями.

Оценить возможность реализации услуг в проекте заказываемой системы могут только разработчики системы, роль которых заключается в ограничении желаемого и согласовании ограничений с другими участниками.

Среди участников системы назначается главный аналитик, ответственный за требования к системе, и главный программист, ответственный за их реализацию.

Все вместе проводят согласование требований на совместных переговорах об определении области действия проекта системы, что включают в себя:

– спектр проблем ПрО, при решении которых будут реализованы услуги системы;

– функциональное содержание указанных услуг;

– операционную среду работы системы.

Согласованная область действий позволяет составить оценки инвестиций в проект, заранее определить возможные риски и возможности разработчиков относительно выполнения проекта. Итогом обсуждения проекта может быть решение о развертывании реализационных работ по проекту или отказ от него.

Процесс выявления и сбора требований. Этот процесс предназначен для извлечения информации из разных источников заказчика (договоров, материалов аналитиков по задачам и функциям системы и др.), проведения технических мероприятий (собеседования, собраний и др.) для формирования отдельных требований для разработки.

Именно на этом процессе должны быть зафиксированы реальные потребности, касающиеся функциональных, операционных и сервисных возможностей ПО. Иными словами. осуществляется договоренность или составляется контракт между заказчиком и исполнителем ПС на языке, понятном обеим сторонам, и он служит заданием для исполнителей реализации поставленных задач ПрО.

Система, которая проектируется, должна реализовать некоторые функции ПрО, сбор требований к которым рекомендуется начинать с точного определения основного вида деятельности системы, предоставляющей услуги.

Целесообразно зафиксировать цели каждого из участников проекта для согласования сроков, целей и содержания работ, ко-

торые они укладывают в эти сроки. Для ПрО с высоким уровнем компьютеризации существуют терминологические стандарты и предметные тезаурусы, которыми необходимо пользоваться либо самостоятельно во время сбора требований составлять свой терминологический словарь. Кроме того, источниками информации при исследовании возможностей новой системы является:

– результаты опроса участников проекта;

– доступ до неформальных документов заказчика (отчеты, мониторинг рынка и т.п.);

– опыт использования прикладных систем, подобных создаваемой, и оценка возможностей технологии разработки данной системы.

Ключевой вопрос при сборе требований – выявление того, что может делать система. Фактически в процессе сбора требований накапливаются желаемые свойства и возможности будущей системы, которые отображаются в целях или результатах, достигаемых проектируемой системой. Поэтому функции системы наглядно и удобно представить в виде сценариев работы (функционирования или использования) системы, задающих последовательность действий или состояний, которые достигаются в определенном порядке при выполнении поставленных целей. Цель – это состояние системы, которою надо достичь.

Процесс анализа требований. Данный процесс ориентирован на изучение потребностей и целей пользователей, классификацию и их преобразование к требованиям системы, аппаратуре и ПО, установление и разрешение конфликтов между требованиями, определение приоритетов, границ системы и принципов взаимодействия со средой функционирования. Требования могут быть функциональные и нефункциональные, которые определяют соответственно внешние и внутренние характеристики системы.

На этапе анализа требований проводится структуризация собранных данных и представление их в более формализованном виде, например, UML, конструкции ***use case*** – вариантов использования или прецеденты. В отличие от диаграммы вариантов использования UML, далее основное внимание уделяется спецификации требований, а также неформальным объяснениям относительно функций и особенностей использования этих конструкций UML для задания сценариев.

Спецификации собранных требований представляются в виде декларативных знаний о функциях, рекомендациях и ограничениях на условия применения системы.

Требования, изложенные в спецификации на систему, проверяются (верифицируются) для того, чтобы убедиться, что они определяют данную систему и служат источником отслеживания требований. Заказчик и разработчик ПО проводят экспертизу сформированного варианта требований для того, чтобы разработчик мог далее проводить реализацию ПО. Под *верификацией требований* понимается процесс проверки правильности спецификации требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие действующим стандартам.

В результате проверки требований создается согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, а также возможность проведения проектирования ПО. Одним из методов валидации требований является прототипирование, т.е. быстрая отработка отдельных требований на конкретном инструменте и исследование масштабов изменения требований, измерение объема функциональности и стоимости.

Процесс управления требованиями. Данный процесс рассматривается как процесс систематического сбора, организации и документирования требований к системе, которая разрабатывается, и процесс установления и поддержки согласований между заказчиками и разработчиками при изменениях требований. Таким образом, управление требованиями является инженерией требований как на начальном этапе их формулирования, так и на протяжении всех изменений, а также включает в себя проведение мониторинга и восстановление источника требований.

Главная задача управления требованиями – это понимание проблем заказчика или будущего пользователя системы, ее высказывание на его языке и согласование того, как нужно построить систему. Для определения свойств будущей системы полученные требования надо трансформировать в требования к ее ПО. Таким образом, продукты отдельных процессов инженерии требований составляют цепочку последовательных преобразований типа: потребности–свойства–требования к ПО.

В зависимости от характера, степени сложности и структуры ПрО требования к ПО могут иметь множество разных понятий и элементов, а процессы инженерии требований должны быть приспособлены к особенностям конкретного приложения. При этом используется совокупность методов, которые адекватно отображают цели разработки, специфику проекта и зависимость от всех ресурсов (программных, системных, финансовых и др.).

К основным задачам инженерии требований к ПС относятся:

– определение базовых понятий ПрО;

– определение и фиксация проблем, которые должны быть отражены в требованиях;

– определение лиц, которые влияют на этот процесс, и видов работ, которые составляют процесс инженерии требований, и их алгоритмизация;

– определение языка спецификации требований, средств их верификации и трассирования на этапах разработки;

– управление требованиями в процессах ЖЦ и установление соответствия с требованиями, предъявляемыми к качеству программного продукта.

В связи с тем, что каждая система постоянно изменяется как при ее разработке, так и при ее использовании, возврат к этапу инженерии требований происходит неоднократно, итеративно для внесения изменений в требования после обсуждения их с заказчиком. Поэтому такой процесс часто называют процессом управления изменениями требований.

Сформулированные и задокументированные требования должны быть утверждены заказчиком и исполнителем.

Каждая ПрО имеет свою систему понятий, характерные свойства, отношения и правила поведения объектов. Они отображаются в документе (контракте), составленном между профессиональными специалистами ПрО, разработчиками ПС и заказчиком. Степень формализации этого документа должна быть достаточной, чтобы обеспечить точность и однозначность представления разных сторон деятельности ПрО.

Неотъемлемой составляющей процесса управления требованиями является трассирование требований для отслеживания правильности задания и реализации требований к системе и ПО на этапах ЖЦ, а также обратный процесс отслеживания от полученного продукта к требованиям.

1.2.5. Формулировка требований на основе прецедентов

Появление разнообразных инструментальных средств поддержки UML (Rational Rose, RUP [26, 27]) привело к созданию новых подходов к формированию и представлению требований к системе и ПО, основанных на использовании так называемых прецедентов (use case) [27].

Задача описания требований с помощью прецедентов сводится к анализу дерева целей системы и к описанию реакции системы

при недостижимости той или иной поставленной цели в системе. При этом необходимо, чтобы описание требований, выполненное с помощью прецедентов, было полным, отражало определенную часть системных требований, вокруг которой выстраиваются требования, касающиеся интерфейса пользователя, применяемых протоколов и форматов ввода-вывода. Содержательной стороной этой части требований является описание функций (бизнес правил), данных и принципов функционирования системы.

Методология формирования требований к ПО с помощью прецедентов реализована в рамках средств Rational Rose [28].

Экземпляр прецедента – это последовательность действий, выполняемых системой, она имеет наблюдаемый результат, который является ценным для конкретного субъекта. Прецедент – это набор экземпляров прецедента.

Функциональные возможности системы определяются набором прецедентов, каждый из которых представляет собой некоторый поток событий. Описание прецедента определяет то, что произойдет в системе, когда прецедент будет выполнен. Каждый прецедент имеет собственную задачу, которую он должен выполнить. Набор прецедентов устанавливает все возможные пути использования системы.

Прецеденты существуют в каждом из основных процессов проектирования: разработка требований, анализ и проектирование, выполнение и испытание системы.

Деловые прецеденты определяются в ходе делового моделирования.

В управляемом прецедентами проекте разрабатываются два представления системы – внешнее и внутреннее. Прецедент внешнего представления ПрО определяет, что нужно системе, чтобы обеспечить заказчику требуемые результаты. Это представление определяет принципы взаимодействия системы и субъектов при выполнении прецедента, оно разрабатывается, когда проводится обсуждение того, что должна делать система с каждым прецедентом.

Реализация прецедента является внутренним его представлением в системе и определяет принципы организации работ для достижения планируемых результатов. Реализация охватывает сущности, которые участвуют в выполнении прецедента, и связи между ними, необходимые для выполнения задания. Иными словами, необходимо разработать такое представление прецедентов, чтобы каждый из прецедентов выполнял определенное действие для достижения требуемых результатов.

Модель прецедентов создается при формировании требований, когда моделируется то, что должна делать система с точки зрения пользователей.

На этапе анализа и проектирования прецеденты реализуются в модели проекта. В этой модели определяются реализации прецедента, которые в терминах взаимодействующих объектов описывают, как прецедент выполняется. В терминах объектов проекта эта модель описывает различные части реализуемой системы и то, как должны взаимодействовать эти части, чтобы выполнить прецеденты.

Синтаксически модель проекта включает в себя следующие элементы:

<имя прецедента или действующего лица>, <имя роли или краткое описание роли действующего лица>, <описание границ системы>, <список всех заинтересованных лиц в анализе ключевых целей системы>, <исходные условия>, <результат успешного завершения определения целей системы>, <шаги сценария для формирования шаблона достижения целей проекта>, <описание дополнительной информации, которая может понадобиться разработчику в процессе реализации системы>.

Описание модели прецедента состоит из:

– целей и масштаба системы,
– используемых терминов (глоссария),
– основных действующих лиц и их целей,
– используемых технологий,
– принципов взаимодействия с другими системами,
– требований к форматам и протоколам взаимодействия,
– требований к тестированию и процедуре развертывания системы у заказчика,
– организационных вопросов управления процессом разработки системы.

Представление системы с помощью прецедентов задается в форме Vision-шаблонов в системе Rational Rose.

Таким образом, можно сделать вывод о том, что рассмотренные методы анализа ПрО позволяют построить модель ОМ, провести упорядочение объектов в этой модели и преобразование ее к модели взаимодействия удаленных объектов. В процессе обследования ПрО и проведения ее анализа определяются требования к системе и к ПО с использованием соответствующих методов ядра знаний SWEBOK и рекомендаций стандарта ISO/IEC 12207.

ГЛАВА 2

ТЕОРЕТИЧЕСКИЕ И ПРИКЛАДНЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ

Технология разработки программных систем находится на двух уровнях – практическом и теоретическом. Практики развивают методы проектирования и разработки для непосредственного применения, а теоретики ведут поиск таких принципов и теорий, которые могут быть положены в основу будущих технологий. Успехи теоретиков остаются весьма относительными, но вместе с практическими результатами они имеют огромное значение. Общее, что характерно для теоретиков – это использование математических знаний для формирования теории и объяснения математическими средствами некоторых важных аспектов программирования, формализации и доказательства правильности построения программ и др.

Наука развивается, продолжается поиск математически обоснованных методов программирования, которые позволяют более формально задавать структуры программ и данных. Вместе с тем профессиональные организации с помощью разных комитетов и институтов провели систематизацию накопленных знаний в области программирования и создали общее ядро знаний SWEBOK (Software Engineering Body Language, 1999 г.) (Приложение 1), а также создан кодекс этики программистов. Это является большим достижением в области программной инженерии, охватывающей все аспекты разработки и управления процессом создания качественного ПО.

Основу технологии программирования составляют методы проектирования и разработки программных систем. Многие методы являются формальными, они базируются на техниках спецификации и верификации программ. Некоторые формальные методы достаточно строги и позволяют обеспечить формальное описание, контроль корректности моделей концептуального проектирования, включая проверку на противоречивость и т.п. Современные формальные методы позволяют специфицировать требования, верифицировать их и по ним реализовывать автоматизированную генерацию программ. Некоторые из них используют неформальные процедуры и приемы без четкого математического обоснования операций проектирования ПС.

Математическая проблематика находит отражение не только в научных работах теоретиков в области программирования, но и в повседневной рутинной работе по составлению конкретных программ, их преобразованию и доказательству.

Среди практических методов наибольшее развитие и использование получил модульный и объектно-ориентированный подходы (ООП) [2–9] и их инструментальная поддержка (Rational Rose, Rational Software, RUP, Demral, OOram и др.). Они оказали влияние на появление и развитие компонентного, аспектно-ориентированного, генерирующего и другие методы [10–44]. Основное влияние на программирование оказывают понятия – модуль, объект, интерфейс и их свойства. В целом эти два подхода программирования создали базис инженерии приложений (Software Engineering), стали основой для дальнейшего развития готовых компонентов многоразового использования и создания соответствующих инструментов проектирования ПС.

В данной главе рассматриваются основные возможности и особенности широко распространенных методов практического (систематического) и теоретического программирования, включая формальные методы проектирования.

2.1. МЕТОДЫ СИСТЕМАТИЧЕСКОГО ПРОГРАММИРОВАНИЯ

К методам систематического программирования следует отнести такие:

- модульный, сборочный,
- структурный,
- объектно-ориентированный,
- UML-метод,
- компонентный,
- аспектно-ориентированный,
- генерирующий,
- агентный и др.

Остановимся на характеристике теоретических основ и возможностей этих методов, инструментальной поддержке и направлениях развития.

2.1.1. Модульное программирование

Одним из ключевых достижений раннего программирования (70–90 годы прошлого столетия) является формирование понятия модуль и использование модулей в качестве строительных

блоков новых ПС. Программирование с помощью модулей началось с библиотек стандартных подпрограмм вычислительной математики, банков модулей общего назначения, библиотек классов, библиотек методов, активных библиотек и т.п.

Вслед за стандартными подпрограммами в ЯП (Фортран, Алгол, ПЛ1 и др.) появился *аппарат функций и процедур*, включающих в себя средства их описания непосредственно в программе и механизм обращения к ним. Любые изменения в комплексной программе требовали повторной трансляции всех ее компонентов. Поэтому этот аппарат усовершенствовался и развивался по пути отделения процедур и функций от программы и стандартизации операторов их вызова (оператор CALL) в ЯП.

В 70–80 годы появилось модульное программирование, основу которого составляют методы декомпозиции и комплексирования модулей в модульные структуры. Модуль воплотил в себе аппарат процедур и функций и стал самостоятельным, отдельным артефактом деятельности программистов. Модули реализуют некоторые функции и сами используют функции других модулей.

Модуль – это логически законченная часть программы, выполняющая определенную функцию. Он обладает такими свойствами: завершенность, независимость, самостоятельность, раздельная трансляция, повторное использование и др. Модули накапливались в библиотеках или Банках программ и модулей, как готовые «детали», из которых собираются ПС и адаптируются к новым условиям среды обработки.

Теоретические и прикладные аспекты программирования с помощью готовых модулей и программ сформулированы академиком В.М. Глушковым в 1976 г. и реализованы в *сборочном программировании* (СП). В основе СП лежит метод проектирования ПС снизу–вверх, выбор готовых модулей из Банков модулей и их сборки в новые структуры ПС. С этого момента в Институте кибернетики АН УССР проводились разработки разных подходов к автоматизированной фабрике программ, в том числе система автоматизации производства программ – АПРОП (1976–1990гг.) [11, 12]. В этой системе главными составляющими были: паспорт программы или модуля, описание интерфейсного модуля с операторами сборки агрегатов из разноязыковых модулей в языке модульного конструирования сборки сложных программных агрегатов из готовых программ и Банка модулей (рис. 2.1).

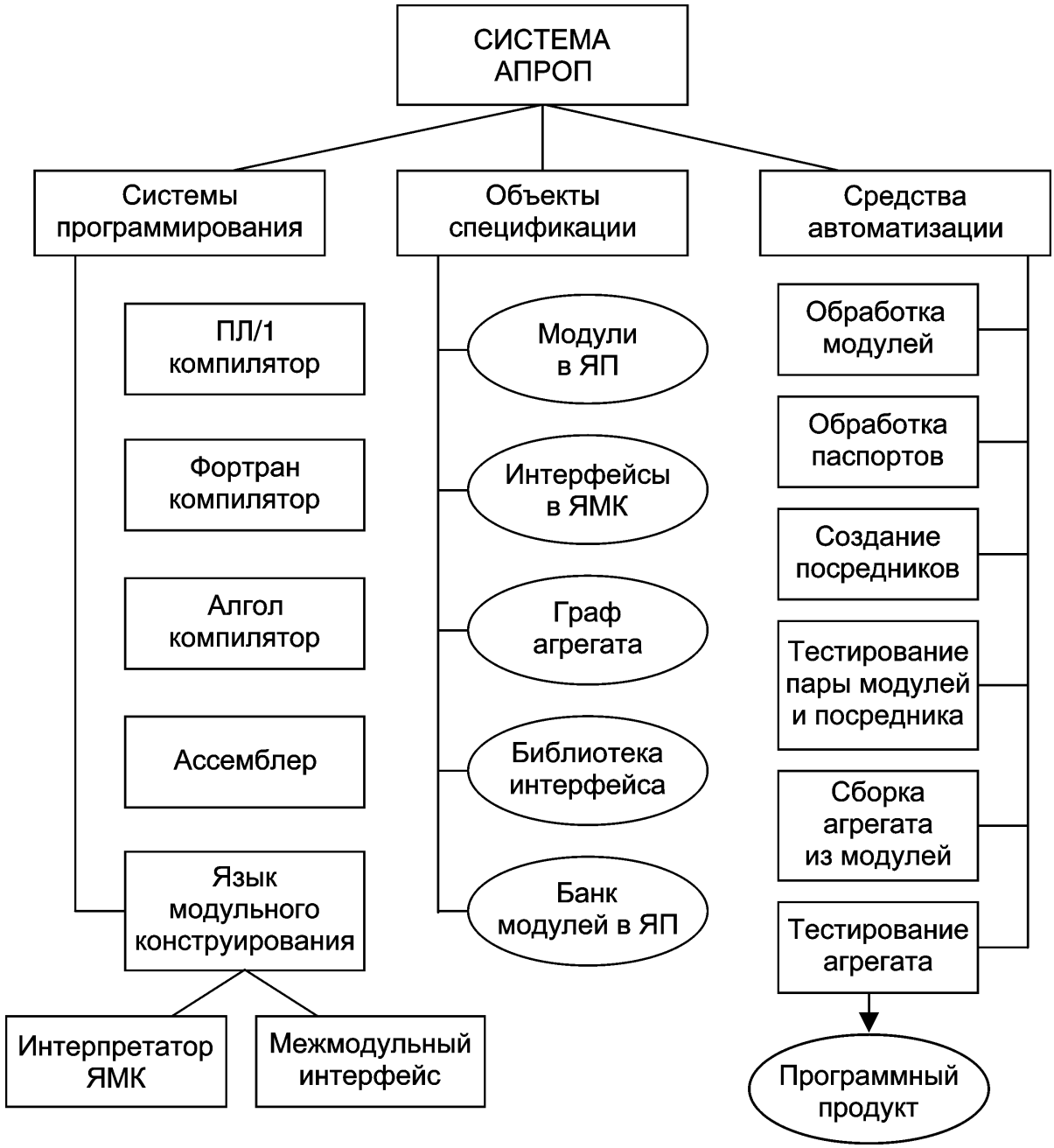

РИС. 2.1. Структура системы автоматизации производства программ

Базовой концепцией СП является интерфейс для связи модулей, которые записаны в ЯП (Фортран, ПЛ/1, Алгол, Ассемблер). Интерфейс был реализован на машинах ЕС ЭВМ и стал первой поисковой работой в этом направлении, прежде чем специалисты других стран пришли к необходимости объединения разных видов программных объектов на новых машинах и средах. Позже были созданы языки описания интерфейсов API (Application Program Interface), IDL (Interface Definition Language) и др.

Они и сейчас выступают в качестве главной доминанты взаимодействия компонентов, объектов, аспектов в современных распределенных средах.

Интерфейс двух модулей в сборочном программировании реализуется с помощью модуля-посредника, в котором задаются параметры, соответствующие операторам вызова, и их типы. Другой тип интерфейса – это интерфейс пары ЯП, сущность которого заключается в отсутствии некоторых типов данных или несоответствии структур и типов данных этих ЯП. Для обеспечения интерфейса ЯП разработана система преобразования, включающая в себя: алгоритм представления отсутствующих типов данных на одном из пары ЯП в существующие релевантные типы данных другого ЯП этой пары; функции и макросы релевантного преобразования.

В результате в модуле-посреднике для каждого вызова организуются обращения к указанным функциям библиотеки, которые выполняются в момент перехода одного модуля к другому через этого посредника. Такой механизм реализации связей модулей или компонентов имеется и в современных языках, с тем различием, что методы модулей и их интерфейсы хранятся в одной библиотеке.

Концепция интерфейса в классе ЯП положена в основу метода сборки (интеграции) модулей, который реализуется алгоритмом, включающим в себя следующие шаги:

– обработка паспортных данных модулей в ЯП;

– анализ описания параметров модулей, составление задания на обработку несоответствующих типов данных и проверка соответствия на правильность передачи параметров как по их количеству, так и по соответствию типов данных в классе ЯП;

– преобразование типов данных в ЯП (b – Boolean, c – character, i – integer, r – real, a – array, z – record и др.), размещаемых в модулях-посредниках сложного программного агрегата в виде обращений к функциям библиотеки интерфейса;

– реализация модуля-посредника в виде программы и составление таблицы связи пар компонентов;

– анализ оператора интеграции пар модулей и размещение их в составе программного агрегата;

– трансляция и компиляция элементов агрегата в виде готовой программной структуры;

– трассировка интерфейсов и отладка функциональности модулей в каждой паре агрегата;

– тестирование программного агрегата;

– форматирование программ запуска программного агрегата и документации.

Таким образом, основными объектами сборки ПС являются *модуль и интерфейс* (межъязыковый и межмодульный), позволяющие в процессе сборки построить промежуточные модули-связи для обеспечения взаимодействия пары модулей в ЯП между собой и с операционной средой.

Межъязыковый интерфейс – совокупность средств описания интерфейса модулей и методов релевантного преобразования структур и типов данных, представленного набором функций и макроопределений в классе типов данных рассматриваемого множества ЯП. Библиотека интерфейса – базовая компонента сборки, включала в себя более 60 функций преобразования одних типов данных в другие и практически использовалась многими программистами при работе с разными языками ОС ЕС [10, 11].

Межмодульный интерфейс – компонент системы сборки для генерации интерфейсных модулей-связи, взаимодействующих между собой модулей. Описание интерфейса на языке определения интерфейсов включает в себя операторы вызова модулей, описание типов параметров и проверку правильности обмена данными между объединяемыми модулями.

Данная концепция интерфейса модулей – одна из первых парадигм сборочного программирования сложных систем из модулей и их интерфейсов.

Аналогичные работы проводились и за рубежом (1980–1990 гг.). Например, MIL (Module Interface Language) – язык взаимосвязи модулей, SAA (System Application Architecture) фирмы IBM, система OBERON, система поддержки и связи компонент – COM [5] и др.

Метод сборки модулей в новые программные структуры включает в себя математические формализмы определения связей (по данным и по управлению) между объектами сборки и генерацию интерфейсных модулей для каждой пары объединяемых модулей. Основу задания связи модулей определяет оператор вызова CALL, в котором задается набор фактических параметров, передаваемых вызывающему модулю.

Сущность задачи сборки пары разноязыковых модулей состоит в определении взаимно-однозначного соответствия между задаваемым множеством фактических параметров $V = \{v_1, v_2, ..., v_к\}$ вызывающего модуля и множеством формальных параметров $F = \{f_1, f_2, ..., f_к\}$ вызываемого модуля, а также в отображении одних

параметров в другие. Если отображение построить не удается, то задача связи для данной пары модулей неразрешима и должна решаться другими методами.

Для формализованного описания типов данных использован алгебраический подход, в котором каждому типу данных ставится в соответствие алгебраическая система с множеством его значений и операций над объектами данного типа. Каждой операции преобразования типов данных соответствует изоморфное отображение одной алгебраической системы в другую.

В классе простых типов данных ЯП (Boolean, character, integer, real) и сложных типов данных (array, record) построены алгебраические системы и рассмотрены допустимые виды преобразований типов данных (см. более подробно главу 3).

Преобразования между массивами и записями сводятся к определению изоморфизма между основными множествами соответствующих алгебраических систем и включают в себя операции изменения уровня структурирования данных – операции селектора и конструирования.

Для массива операция селектора – это отображение множества индексов на множество значений элементов массива. Аналогично такая операция определяется для записи как отображение между селекторами компонентов и самими компонентами.

Для описания интегрированной среды функционирования совокупности разнородных ПИК предложены две модели – информационного объединения сопряжения и управления объектами в динамике их выполнения.

При описании модели информационного объединения используются построенные алгебраические системы, множество функций преобразования одних типов данных в другие и операции понижения уровня структурирования данных для сложных и структурных типов данных.

Модель управления включает в себя описание условий выполнения и выбора компонентов, входящих в интегрированный комплекс. С каждым компонентом комплекса связаны предусловия и постусловия, определенные на множестве управляющих переменных.

Метод сборки, или комплексирования, использует Банк готовых модулей и библиотеку интерфейса для установления взаимно-однозначного соответствия передаваемых типов данных и автоматической генерации интерфейсных модулей-связей. В результате создается программный комплекс жесткой структуры

для майнфреймов, обладающих большими объемами оперативной памяти.

В связи с появлением распределенных сред для функционирования создаваемых систем, жесткая структура заменена гибкой структурой, представляющей собой общую конфигурацию проектируемой системы, которая может адаптироваться к условиям функционирования компонентов, распределенных по разным узлам сети.

Инструменты. В системе АПРОП [11, 12] реализован метод сборки разноязыковых модулей, основанный на идеи интерфейса и отображения его в виде модуля-посредника. Позже в качестве элементов сборки стали применяться не только модули, а и объекты, компоненты, программы, каркасы и т.п., как результат реализации задач разных предметных областей. Возникли проблемы их использования как готовых элементов: создание языков вызова процедур − RPC (Remote Procedure Call), описание интерфейсов с помощью языков IDL и API; перенос готовых элементов на другую платформу или в другую среду функционирования; а также появление новых ЯП (C, C++, Java и др.).

Язык IDL обеспечивает связь готовых элементов с помощью описание входных и выходных данных объединяемых элементов. Появились конкретные системы обеспечения взаимосвязей указанных программных элементов. Одним из таких решений, является создание специального программного интерфейса − JNI (Java Native Interface). В нем допускается обращение Java-классов к функциям и библиотекам на других ЯП, которое включает в себя: анализ Java-классов для поиска прототипов обращений к функциям на C/C++; генерацию заглавных файлов при компиляции в C/C++; обращение из Java-классов к COM-компонентам [75].

Проблема связи реализована для платформы .Net с помощью языка CLR (Common Language Runtime), в который транслируются объекты в ЯП (C, Visual Basic, C++, Jscript). При этом для связи объектов независимо от ЯП используется библиотека стандартных классов и стандартные средства генерации в представление .Net-компонента [66, 67].

Среды CORBA и COM учитывают особенности ОС и архитектуру компьютеров, для которой разработаны компоненты, и используют их при интеграции [68, 69, 70, 75].

Разработан стандарт ГОСТ 30664 [34] для ISO/IEC 11404−1996, обеспечивающий независимое от ЯП описание типов дан-

ных и методику спецификации этих типов данных для разноязыковых компонентов. Каждый тип данных имеет набор свойств, достаточный для его выделения и сравнения с типами данных, содержащихся в другом ЯП. Стандарт представляет собой общую модель данных для обращения к нему других стандартов. Типы данных в стандарте описываются в независимом языке LI (Language Independent), который в отличие от описания типов данных в конкретном ЯП является более общим языком, содержащим все существующие типы в ЯП либо средства их конструирования. Средствами LI описываются параметры вызова, как элементы интерфейса, необходимые при обращении к стандартным сервисам и готовым программным компонентам. Цель этого стандарта состоит в том, чтобы обеспечить не только описание типов данных, но и их преобразование в типы данных конкретных ЯП с помощью специальных правил и операций агрегации стандарта в целях объединения типов данных в более сложные структуры ЯП. Данные средства описания типов данных могут использоваться при определении интерфейса компонентов, задаваемых такими языками, как IDL, RPC и API (более подробно см. главу 4).

2.1.2 Структурное программирование

Основу структурного программирования составляют строгие требования к организации проектирования с соответствующим использованием управляющих структур в ЯП, а именно, ветвления (if–then–else), цикла (while), case (выбор). Создаваемая системы декомпозируется на отдельные функции, которые, в свою очередь, декомпозируются на более мелкие. Для функций разрабатываются программа ее реализации с применением управляющих структур [14–18].

Метод структурного программирования базируются на двух общих принципах:

1) решение сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;

2) организация составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Приведенные принципы дополнены такими механизмами:

– абстракция – выделение существенных аспектов системы и отвлечение от несущественных;

– формализация, т.е. строгое формальное представление проблемы;

– непротиворечивость – обоснование и согласование элементов системы;

– структуризация данных согласно иерархической организации.

В рамках данного метода сформировался ряд моделей, главными из которых являются:

– SADT (Structured Analysis and Design Technique) – структурный анализ и техника проектирования [17];

– SSADM (Structured Systems Analysis and Design Method) – метод структурного анализа и проектирования [16];

– IDEF0 (Integrated Definition Functions) – метод создания функциональной модели, IDEF1 – информационной модели, IDEF2 – динамической модели и др.

На стадии проектирования эти модели дополняются структурными схемами и диаграммами. *Метод SADT* представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели ПрО, отображающей структуру функций объекта автоматизации, производимые действия и связи между ними. Проектируемую отдельную функцию графически можно представить в виде *блока* моделирования функции с интерфейсами и заданными входными и выходными дугами (рис. 2.2).

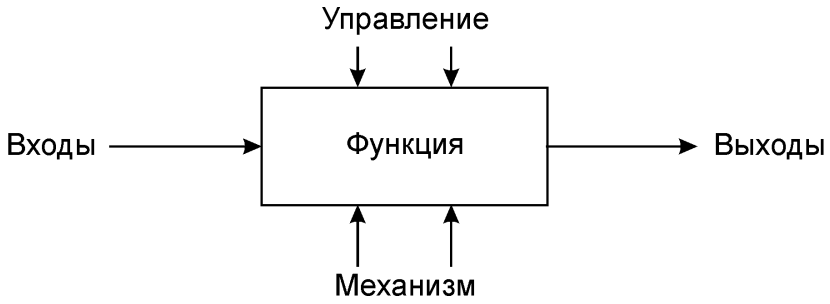

РИС.2.2. Описание функции с входами и выходами

При проектировании функциональной модели системы используются следующие правила:

– количество блоков на каждом уровне декомпозиции может быть от 3 до 6 блоков, связь диаграмм осуществляется через номера этих блоков;

– метки и наименования должны быть уникальными;

– входы и управления задаются раздельно;

– функциональная модель не связана с организационной структурой.

Между отдельно спроектированными функциями устанавливаются связи: логические, процедурные, коммуникационные, последовательные, функциональные и др.

Результатом применения метода SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария со ссылками на ее элементы, а также связями между функциями.

Метод SSADM базируется на таких структурных элементах: история существования сущности; распространения воздействий; структуры входных/выходных данных; пути доступа для выполнения запроса; структуры диалога; модели процесса обновления и выполнения запроса.

Структура компонентов системы представляется следующими базовыми диаграммами: последовательность, выбор и итерация.

Моделируемый объект задается сгруппированной последовательностью диаграмм, следующих друг за другом, операторами выбора элемента из группы и циклическим выполнением отдельных элементов. Каждый элемент изображается на диаграмме четырехугольником с прямыми или закругленными углами и дугами с входными и выходными данными.

Базовая диаграмма является иерархической и включает в себя следующее: список всех компонентов описываемого объекта; идентифицированные группы выбранных и повторяемых компонентов, а также последовательных компонентов. Модель процесса состоит из структурных диаграмм, которые в SSADM создаются в процессе:

– определения функций;

– моделирования взаимосвязей событий и сущностей;

– логического проектирования программ и данных;

– проектирования диалога;

– логического проектирования БД;

– физического проектирования.

В процессе проектирования строится ряд моделей.

Структурная модель состоит из диаграмм, представляющих собой модули, стадии и шаги проектирования, которые соответствуют ЖЦ (рис. 2.3).

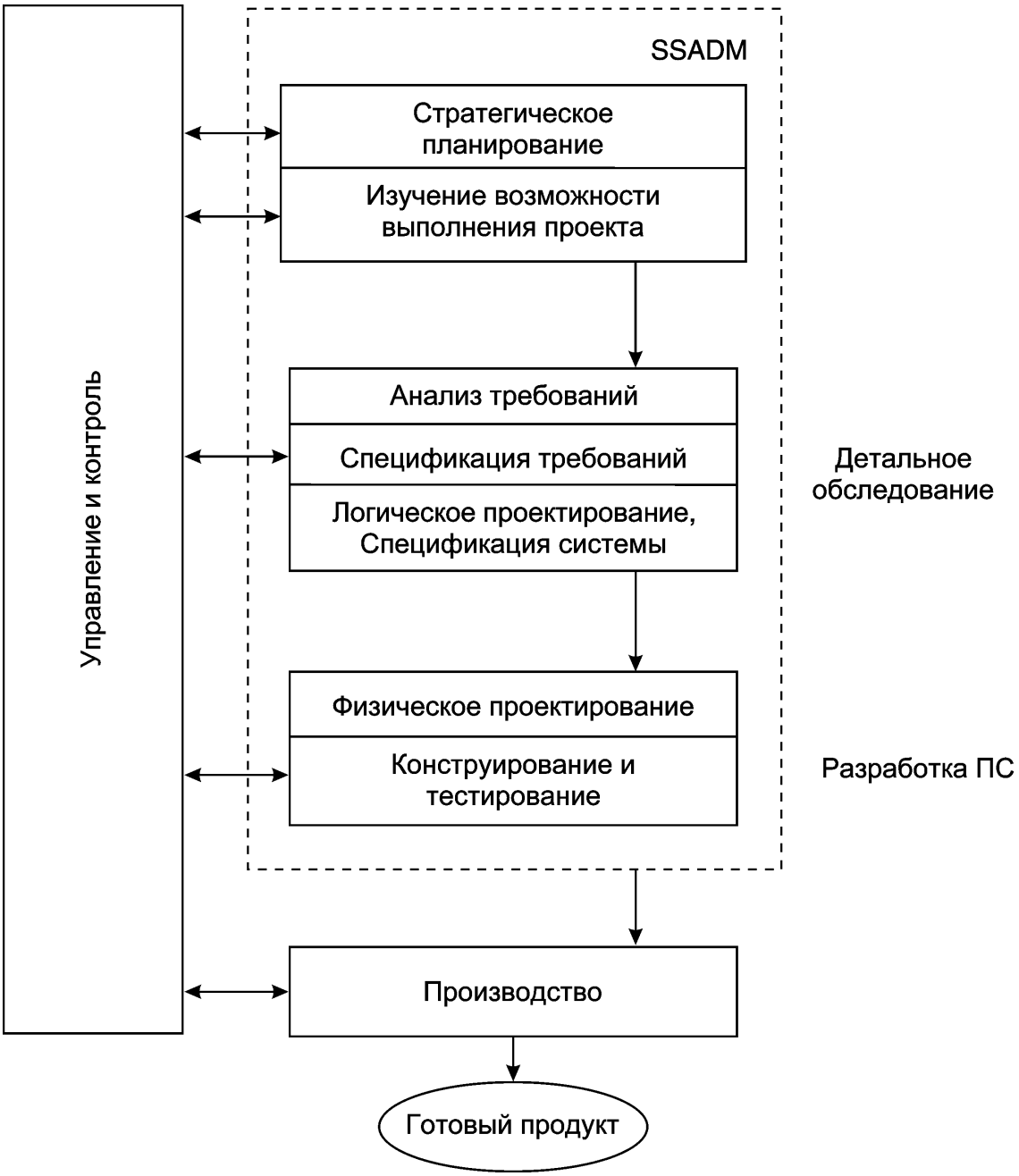

РИС. 2.3. Модель процесса ЖЦ разработки приложения в SSADM

В основе стратегического проектирования лежит анализ и определение требований и области действия приложения, существующие информационные потоки, формирование общего представления о затратах на разработку и подтверждение возможности дальнейшего использования приложения. Результатом является спецификация требований, которая применяется для логического проектирования системы.

Логическое проектирование включает в себя проектирование диалога и процесса обновления БД. Проектирование состоит в

создании логической модели и спецификации, в которой отображены входные и выходные данные, процессы выполнения запросов и процессов обновления на основе логической БД. Одной из целей логического проектирования является минимизация дублирования трудозатрат при физическом проектировании, обеспечение целостности на основе установления взаимосвязей между событиями и сущностями.

Физическое проектирование состоит в определении типа СУБД, необходимого для представления сущностей и связей между ними при соблюдении спецификации логической модели данных и учета ограничений на память и время обработки. Предусматривается реструктуризация проекта в целях изменения механизмов доступа, повышения производительности, объема логической структуры, добавления связей и документирования. Физическая спецификация включает в себя:

– спецификации функций и схемы реализации компонентов функций,

– описание процедурных и непроцедурных компонентов и интерфейсов,

– определение логических и физических групп данных с учетом ограничений оборудования и стандартов на разработку,

– определение групп событий, которые обрабатываются как единое целое с выдачей сообщений о завершении обработки и др.

Проект системы представляется в виде структурной модели, которая позволяет провести определение работ внутри системы SSADM и взаимосвязи между этими работами в виде потоков проектных документов, отображенных в сетевом графике работ.

Исполнительные процессы SSADM связаны с работами, управляющими потоками информации трех типов: поток работ; санкционированные потоки по контролю или управлению; отчеты о ходе разработки. В диаграммах структурной модели упорядочение процессов проведено слева направо и отражает развитие во времени, а не сами интервалы времени.

Модель потоков данных (Data Flow Model – DFM) используется для описания процессов обработки в системе и включает в себя:

– иерархический набор диаграмм потоков данных (Data Flow Diagram – DFD);

– описание элементарных процессов, потоков данных, хранилищ данных и внешних сущностей.

Каждая DFD отражает прохождение данных через систему в зависимости от уровня и назначения диаграммы. DFD преобразо-

вывает входные потоки данных в выходные потоки данных. Как правило, процессы, выполняющие такие преобразования, создают и используют данные (хранилища данных). Внешние сущности посылают и получают потоки данных в системе.

К объектам моделирования в SSADM относятся.

1. **Функции**, которые создаются на основе DFM и моделирования взаимосвязей событий и сущностей для исследования обработки данных в системе.

2. **События** – это некоторые прикладные действия, которые включают в себя (инициируют) процессы для занесения и обновления данных системы. Событие приводит к вызову процесса и исследуется посредством моделирования воздействий на сущности.

3. **Данные** представляются логической и физической моделями данных, реляционным анализом данных для выбора СУБД.

Наиболее распространенным средством моделирования данных являются диаграммы сущность–связь (ERD), с помощью которых определяются объекты (сущности) ПрО, их свойства (атрибуты) и отношения (связи) [18]. Сущность (Entity) – реальный либо воображаемый объект, имеющий существенное значение для ПрО. Каждая сущность и ее экземпляр имеют уникальные имена. Сущность обладает следующими свойствами:

– имеет один или несколько атрибутов, которые либо принадлежат сущности, либо наследуются через связь (Relationship);

– каждая сущность может обладать любым количеством связей с другими сущностями модели.

Связь – это ассоциация между двумя сущностями ПрО. Каждый экземпляр родительской сущности, ассоциирован с произвольным количеством экземпляров второй сущности (потомками), а каждый экземпляр сущности-потомка ассоциирован с одним экземпляром сущности родителя. Таким образом, экземпляр сущности-потомка может существовать только при наличии сущности родителя.

Метод IDEF1 основан на подходе ERD и позволяет построить информационную модель, эквивалентную реляционной модели. Данный метод постоянно развивается и совершенствуется (например, методология IDEF1X проектирования, ориентированная на автоматизацию – ERwin, Design/IDEF). Основная особенность состоит в том, что каждый экземпляр сущности может быть однозначно идентифицирован без определения его отношений с другими сущностями. Если идентификация экземпляра сущности зависит от его отношения к другой сущности, то сущность является

зависимой. Каждой сущности присваивается уникальное имя и номер, разделяемые косой чертой "/", и помещаемые над блоком, обозначающим сущность. Связь может определяться с помощью указания количества экземпляров сущности-потомка, т.е. для каждого экземпляра сущности-родителя можно иметь

– один или более связанных с ним экземпляров сущности-потомка;

– не менее одного связанного с ним экземпляра сущности-потомка;

– связь с некоторым фиксированным числом экземпляров сущности-потомка.

Если экземпляр сущности-потомка однозначно определяется своей связью с сущностью-родителем, то связь является идентифицирующей, иначе – не идентифицирующей. Связь между сущностью-родителем и сущностью-потомком изображается сплошной линией с точкой на конце линии у сущности-потомка. Идентифицирующая связь между сущностью-родителем и сущностью-потомком изображается сплошной линией без точки. Сущность-родитель в идентифицирующей связи может быть как независимой, так и зависимой от идентификатора сущностью, что определяется связями с другими сущностями. Пунктирная линия характеризует не идентифицирующую связь. Сущность-потомок в неидентифицирующей связи будет независимой от идентификатора, если она не является также сущностью-потомком в какой-либо идентифицирующей связи.

Атрибуты изображаются списком имен внутри блока сущности, первичный ключ размещается наверху списка и отделяется от других атрибутов горизонтальной чертой. Сущности могут иметь также внешние ключи, как часть или целое первичного ключа или не ключевого атрибута.

Средствами IDEF1 проводится сбор и изучение различных областей деятельности предприятия, определение потребностей в информационном менеджменте, а также:

– информации и структуры ее потоков, имеющих отношение к деятельности предприятия;

– правил и законов движения информационных потоков и принципов управления ими;

– взаимосвязей между существующими информационными потоками на предприятии;

– преодоления проблем, возникающих при некачественном информационном менеджменте.

Одной из особенностей данной методологии является обеспечение структурированного процесса анализа информационных потоков предприятия и возможности изменения неполной и неточной информации на этапе моделирования информационной структуры предприятия.

Инструменты. Основным инструментом, отображающим структурное проектирование в соответствии с ЖЦ, является комплекс инструментальных, методических и организационных средств системы SSADM. Эта система принята государственными органами Англии в качестве основного системного средства и используется многими организациями как внутри страны, так и за ее пределами. SSADM включает в себя пять главных модулей [16]:

1) изучения возможности выполнения проекта (Feasibility Study Module);

2) анализа требований (Requirements Analysis Module);

3) спецификации требований (Requirements Specification Module);

4) логической спецификации системы (Logical System Specification Module);

5) физического проектирования (Physical Design Module).

Проектирование с помощью системы SSADM представляется совокупностью мероприятий по разработке набора проектных документов в условиях использования соответствующих ресурсов при заданных ограничениях на стоимость разработки. Для управления ходом разработки проекта рассматриваются проектные работы и документы, организация разработки, планы разработки, мероприятия по руководству проектом и обеспечению качества. Различаются два типа проектных работ:

– обеспечение требований пользователя к качеству системы;

– управление разработкой проекта.

Структурная модель охватывает модули и стадии технологии SSADM, обеспечивает получение одних документов на основании других путем логических преобразований, каждое из преобразований выполняется над одной совокупностью документов для получения другой совокупности документов. Для установления последовательности работ и мероприятий по обеспечению качества разрабатывается сетевой график работ.

Мероприятия по обеспечению качества реализуется группой обеспечения качества, которая отвечает за поддержку целостности проекта. В нее входят специалисты, ответственные за функционирование организации (плановики, экономисты), пользова-

тели системы и разработчики, связанные с проектом от начала до конца. Плановики и экономисты отслеживают своевременность выполнения и финансирование работ, пользователи – свои интересы и требования, а разработчики выражают потребности в рамках своей компетенции.

Для управления проектом создается служба поддержки проекта, реализующая ряд административных функций либо специальных работ. Она осуществляет экспертизу при оценке, планирование и управление проектом, а также мероприятия по руководству конфигурацией, сущность которых состоит в отслеживании проектных документов и обеспечении информации об их состоянии в процессе разработки.

Проблема качества затрагивает два основных аспекта:

1. Совокупность функций, которая должна удовлетворять заданным требованиям в терминах реализуемых функций, надежности и производительности.

2. Способ реализации системы, т.е. то, как система сконструирована.

Обеспечение качества проводится проверками на этапах ЖЦ проекта указанных в требованиях одного или нескольких показателей качества: экономичность; документируемость; тестируемость; понимаемость; способность к повторному использованию; гибкость; способность к изменению; модульность; правильность; надежность; переносимость; эффективность.

Проверка качества конечного продукта состоит в проверке соответствия заданным стандартам и требованиям. Контроль качества включает в себя действия, которые позволяют проверить и измерить установленные показатели качества. Хорошее качество продукта означает, что система конструировалась в соответствии с установленными стандартами, облегчающими последующее сопровождение и модификацию, изменение требований или внесении исправлений в систему с минимумом затрат.

Идеология структурного программирования воплощена в ряд CASE-средств (Silver Run, Oracle Designers, Erwin для прямой и обратной связи с Rational Rose и др.), которые активно используются на практике.

2.1.3. Объектно-ориентированное проектирование (ООП)

ООП [2–9] представляет собой парадигму проектирования ПрО из объектов. Каждое понятие ПрО вместе с его свойствами и особенностями поведения в некоторой среде является отдельным

объектом, а вся ПрО – это совокупность объектов, связи между которыми определяются на основе отношений, установленных между ними. В роли объектов выступают как абстрактные образы, так и конкретные физические предметы или группы предметов с избранным подмножеством характеристик и функций.

Определение объекта как базисного понятия ПрО основывается на следующих положениях:

– каждый объект должен однозначно определять сущность ПрО, включая отображение необходимых связей и особенностей поведения;

– ПрО должна быть представлена объектной моделью с достаточной мерой полноты отображения всех ее сущностей и их поведения.

Полнота отображения сущностей объектами соответствует степени их абстракции. Объект определяется, как понятийная структура, которая характеризуется идентификатором и уровнем знаний о нем. *Объект* – это именованный предмет абстрактного компонента реального мира с поведением, которое целиком обусловлено его характеристиками и взаимоотношениями с другими объектами ПрО.

Объект интерпретируется, с одной стороны, как понятийная структура, которая состоит из идентификатора и уровня знаний о нем, и с другой – как образ предмета или абстрактного компонента ПрО, на который указывает идентификатор, и концепт, определяющий его назначение. Одному объекту могут отвечать несколько концептов, отображающих выбранный уровень абстракции.

Каждый объект, кроме собственного имени и соответствующего смысла, имеет внутренние и внешние особенности. К внутренним особенностям относятся структура и характеристики, которые скрыты от пользователя. Внешние особенности служат для отличия его от всех других компонентов ПрО и включают в себя сведения о принципах взаимодействия с другими объектами через интерфейс.

Объекты объединяются в классы – множество объектов, связанных общностью структуры и поведения. Каждый из них имеет описание всех атрибутов и операций. Класс обладает свойством наследования другого класса, если он полностью содержит все атрибуты и поведение наследованного класса или дополнительные атрибуты и/или поведение. Класс, который наследует, называют суперклассом, а класс, которого наследуют – подкласс.

Каждый реализованный класс имеет внутреннее представление – скрытую структуру и интерфейсную часть, в которой содержится описание внешнего представления класса, т.е. описание констант, переменных и др. особенностей, необходимых для установления отношений с этим классом: наследования, использования, наполнения.

Каждый объект является экземпляром класса.

Операции над объектами. Над объектами производятся следующие операции:

– ведение объектов (ввести, сохранить, удалить и др.);

– взаимодействие объектов посредством вызовов методов объектов, которые определяются на множестве входных и выходных интерфейсов.

Интерфейс называется входным, если объект с его помощью получает определенный сервис, и соответственно – выходным, если объект предоставляет этот сервис.

Каждая операция имеет имя, список входных параметров-интерфейсов и список выходных результатов, если они имеются.

Общая форма описания операции имеет вид

operation_name (param–1,..., param–n)

returns (res–1,..., res–m)

param–i ::= parameter _name : parameter_type

res–i ::= result_name : result_type

Иными словами, описание операции представляет собой структуру данных, в которой указывается набор входных и выходных параметров при вызове объекта.

w: (x_1:s_1, x_2:s_2, ..., x_n:s_n) —> (y_1:r_1, y_2:r_2, ..., y_m:r_m),

где w — имя операции;

x_1, x_2, ..., x_n — входные параметры и x_1 — управляющий оператор;

s_1, s_2, ..., s_n — типы входных параметров;

y_1, y_2, ..., y_m — выходные параметры;

r_1, r_2, ..., r_m — типы выходных параметров.

Основной операцией объекта является з*апрос,* в котором определены действие и список параметров, заданных клиентом для обращения к удаленному объекту для получения от него результата.

Запрос имеет силу, если типы параметров или результатов операции w принадлежат множеству типов входных и выходных интерфейсов.

Семантика исполнения операции состоит в связывании аргументов и переменных результата запроса с формальными параметрами операции.

На основе выполнения операций объект способен пребывать в различных состояниях. Каждое состояние определяется набором задаваемых атрибутов объекта и операций, особенностью которых является *полиморфизм*. Операции объекта позволяют получить сервис у объекта путем выполнения определенных вычислений, а затем полученный результат предоставить другим объектам.

Изменение реализации какого-нибудь объекта или добавление ему новых функций не влияет на другие объекты системы.

Процесс построения системы в среде ООП включает в себя в себя следующие этапы проектирования ПО (рис. 2.4) [1, 2]:

– *Анализ* ПрО. Создание объектной модели (ОМ) ПрО, в которой объекты отражают реальные ее сущности и операции над ними;

– *Проектирование* ПО. Уточнение ОМ с учётом описания требований для реализации конкретных задач системы;

– *Программирование* ПО. Реализация ОМ средствами C++, Java и др.;

– *Эволюция* ПО. Развитие системы путем внесения изменений как в состав объектов та и в методы их реализации;

– *Модификация* ПО. Изменение системы в процессе ее сопровождения осуществляется путем добавления новых функциональных возможностей, интерфейсов, новых операций и т.п.

Данные этапы могут выполняться итерационно друг за другом и с возвратом к предыдущему этапу. На каждом этапе может применяться одна и та же система нотаций.

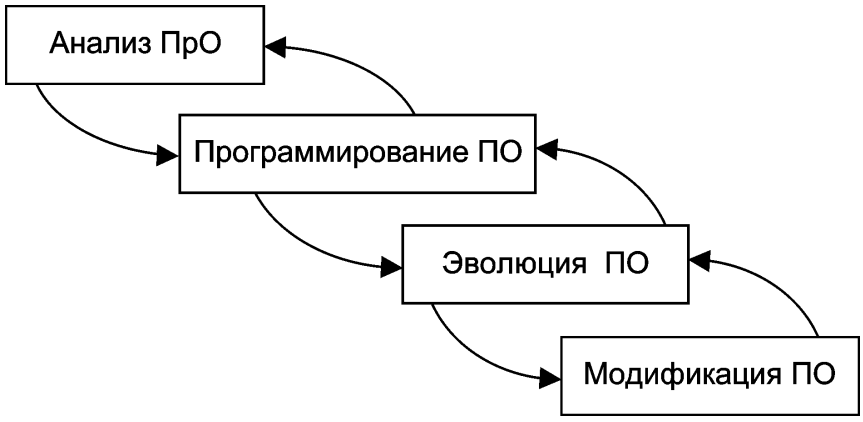

РИС. 2.4. ЖЦ разработки ПО с помощью ООП

Переход к следующему этапу приводит к усовершенствованию результатов предыдущего этапа путем проведения более детального описания определенных ранее классов объектов и новых классов.

На основе построенной модели ПрО строится программная система, которая отображает установленные связи между объектами, их состояния и набор операций для динамического изменения состояния других объектов. Объекты инкапсулируют информацию об их состоянии и ограничивают к ним доступ.

Результатом проектирования является исполняемая программа, в которой все необходимые объекты создаются динамически с помощью классов и выполняются соответствующими методами. Результат проектирования системы проверяется на качество после проведения тестирования.

Новый проект ПС можно разрабатывать на базе ранее созданных объектов, что снижает стоимость и уменьшает риск разработки новой системы.

ООП использует также возможности структурного подхода, например декомпозицию диаграмм использования и состояний, а также диаграмм потоков данных, широко используемых при логическом и физическом проектировании БД. На стадии проектирования требований в ООП прослеживается связь между диаграммами сущность–связь и диаграммами классов. Далее в процессе проектирования преимущественное значение имеет моделирование ПрО в терминах взаимодействующих объектов.

Инструменты. К инструментальным средствам поддержки ООП относятся: RUP [20, 21] – стандарт разработки одиночных ПС из элементов Use Case, отвечающих требованиям конечных пользователей; Rational Rose – инструмент для проектирования визуальной структуры системы; CORBA – архитектура брокера объектных запросов; COM – компонентная объектная модель; Java/Rmi – удаленный вызов объектных методов [68, 69].

RUP – методология создания ПО на основе регламентированных этапов разработки ПО, документов, сопровождающих каждый этап, и инструментальных средства поддержки каждого этапа ЖЦ (Rational Rose, Rational Software Documentation Automation, Requisite PRO, Clear Quest и др.). Компоненты процессов RUP описываются в категориях действий, рабочих потоков и действий исполнителей, которые измеряются и оцениваются. Главные элементы этого процесса включают в себя группы сценариев, описывающих реализацию ПС на этапах разработки, их интеграцию,

измерение времени выполнения этапов в контрольных точках ПС и компонентов на качество.

Rational Rose – средство проектировщиков, аналитиков и разработчиков объектно-ориентированных информационных систем с использованием UML [19]. Вначале проектируется общая модель процессов предприятия, а затем конкретная (физическая) модель. Результатом моделирования является файл с логической моделью, сформированной проектировщиком для кодировщика, который дополняет эту модель конкретными классами на ЯП (C++, Ada, Java, Basic, Xml, Oracle). Поддерживается обратное проектирование, состоящее в преобразовании готовой информационной системы (например, на C++) или база данных (на Oracle) к виду визуальной структурной модели, в которой определенны все виды диаграмм UML с заданием свойств, отношений и связей между ними.

Система ***CORBA*** базируется на объектной модели, которая включает в себя объекты со статическими типами и интерфейсами, определяемые в IDL, а также позволяет объявлять новые типы данных с помощью ключевых слов: sequence, struct, array. Объекты, обладающие общими свойствами, группируются в классы. Тип экземпляра класса рассматривается как отношение подтип/супертип. Каждому объекту отвечает одна или несколько операций, которые задают вызов методов. Операции определяют поведение объекта, и после их выполнения объект приобретает некоторое иное состояние, которое влияет на его поведение. Основу этой системы составляет эталонная архитектуры OMA [68, 69], которая включает в себя:

– объектную модель;

– язык IDL и транслятор интерфейсов компонентов приложений (Application Interface); общий объектный сервис (Common Object Services) для управления событиями, транзакциями, интерфейсами, запросами и др.;

– общие средства (Common facilities) используются группами компонентов и приложений (электронная почта, телекоммуникация, эмулятор программ и др.);

– брокер объектных запросов является главным связывающим звеном компонентов модели из разных приложений и функционирующих в разных средах.

В рамках системы CORBA создан трехзвенный объектный Web, который поддерживает связь с разными платформами (Unix, OS/2, OS/400, MacOS, MVS).Первое звено – это клиент Java-приложения (апплеты). Второе звено – это сервер приложе-

ний, на котором функционируют объекты и взаимодействуют с иным серверным приложением и компонентами клиента через ORB Интернета. Серверные приложения настроены на интерфейс и могут использовать в среде CORBA разные технологии (например, RAD − Rapid Application Development, Borland C++ Builder, S Builder, Delphi и др.), а также визуальные средства. Третье звено − это сама CORBA, имеющая доступ к первому и второму звену через монитор транзакций TP, MOM и др.

Система COM базируется на компонентной модели объектов (двоичные компоненты, документы и их композиция, программы на языке C и др.) и предоставляет разработчикам мощный набор средств для создания ПО. Эта система прошла путь от создания документов до распределенных объектов. Отсутствие формализованной архитектуры, общих решений и ориентации на отдельные языки является недостатком этой модели. Новый вариант COM+, средства интеграции MTS со службой асинхронного взаимодействия MSMQ, а также возможность наследования готовых приложений на мейнфреймах в среде Windows сделали DCOM базовой технологией [70].

JAVA−RMI реализует стандартный метод вызова RMI (Remote Method Invocation) удаленных компонентов среды и обеспечивает передачу сообщений, их синхронизацию, очистку памяти после выполнения компонентов и т.п.. Компоненты выполняются в виртуальной машине (virtual machine), которая работает с byte-кодами этих компонентов, после их интерпретации в код той машины, где компонент функционирует [74, 75].

Основной недостаток рассмотренных систем общего назначения − это отсутствие возможности синхронизации объектов, выполняемых в их средах.

2.1.4 Метод моделирования UML

Основу метода моделирования составляет язык UML (United Modeling Language) [18−21], предназначенный для визуализации, спецификации, конструирования и документирования артефактов ПС с помощью графической нотации, словаря и механизмов расширения.

Основная цель языка UML − предоставить разработчикам системы механизмы визуального (с помощью диаграмм) поуровневого моделирования моделей системы с заданием ее различных представлений в терминах классов, их свойств и методов, а также

взаимосвязей между ними. Процесс построения и дальнейшего применения моделей называется моделированием. Он не зависит от конкретного языка или платформы компьютера.

Визуальное моделирование в UML – это процесс поуровневого спуска от наиболее общей абстрактной модели системы к более наглядной логической, а затем и к физической модели системы. Основными понятиями процесса моделирования являются модели и виды деятельности, которые рассматриваются, как набор действий, приводящий к некоторому конечному результату процесса моделирования. Модели системы отображают различные виды деятельности, их взаимодействие с другими процессами, а также носителей интересов (актеров) через прецеденты.

Модели системы можно разбить на две группы.

В первую группу входят статическая и динамическая модели, которые отображают наиболее существенные закономерности системы и процессов ее функционирования.

Статическая (логическая) модель создается с помощью диаграмм классов, которые могут содержать интерфейсы, отношения, экземпляры объектов и связей. Основным элементом этой модели являются классификатор (классы, типы данных и др.) и отношения (ассоциация, обобщение, реализация и использование). Эта модель имеет логическое представление и представление реализации.

Динамическая модель описывает поведение объектов системы, задаваемых с помощью методов взаимодействия и изменения состояний. Такая модель определяет представление процесса функционирования и размещения (развертывания) компонентов системы.

Во вторую группу входит модель функционирования системы, задаваемая с помощью диаграмм последовательности классов, поведения и реализации.

Диаграммы вариантов использования и последовательности действий служат для описания функциональности или сервиса, который система предоставляет действующему лицу – актеру. Отдельные компоненты диаграмм изображаются прямоугольниками с типами отношений между актерами и другими вариантами использования. Актер является сущностью, которая взаимодействует с системой или ролью, которую выполняет пользователь в процессе взаимодействия с системой. Множество вариантов использования определяет все возможные стороны поведения системы. Между компонентами диаграмм могут быть установлены

различные отношения (ассоциация, расширение, обобщение и включение) по взаимодействию экземпляров актеров и вариантов использования с экземплярами других актеров и вариантов использования.

Диаграмма поведения отображает объект (класс, модуль, актер) как набор состояний и условий, по которым происходит переход от одного из них к другому. Состояние объекта может включать в себя подсостояния. Переходы происходят по событиям или прохождению определенного момента времени и др. Поведение системы – это отображение аспектов функционирования системы с помощью диаграмм состояний, деятельностей, последовательности и кооперации. Диаграмма состояний – это граф. Вершины – это состояния, а дуги – переходы. Этот граф задает динамическое поведение системы.

Диаграмма взаимодействия объектов модели может задаваться диаграммой последовательности, в которой отображается динамика взаимодействия объектов с помощью набора операций, и диаграммы сотрудничества, задающей виды взаимодействий между объектами. На горизонтальной линии диаграммы располагаются объекты, а вниз по вертикальной (пунктирной) линии для каждого объекта задается период жизни и уничтожения. Объекты связываются друг с другом ассоциациями и задаются сплошными и штриховыми линиями. Они показывают отношения между родственными элементами различного уровня (например, класс–объект).

Диаграмма кооперации позволяет графически представить не только последовательности взаимодействий, но и структурные отношения между объектами системы. Диаграмма реализации бывает двух видов: компонентной и развертывания. Компонентная диаграмма отражает физическую модель программного проекта в исходном коде с объектами и исполняемыми файлами. Диаграмма развертывания – это готовая скомпилированная программа с правилами ее расположения в среде функционирования и выполнения.

Таким образом, в состав диаграмм входят: class, use case, object, sequence, component, collaboration, statechart, activity, deployment, которые используются для графического изображения соответствующих элементов системы. Дадим краткую итоговую характеристику средств языка UML.

Диаграмма классов отображает онтологию домена, состав классов объектов, их взаимоотношения, список атрибутов класса и операций класса. Атрибутами могут быть:

– public (общий) – операция класса, вызванная из любой части программы, любым объектом ПС,

– protected (защищенный) – операция, которая вызывается объектом того класса, в котором она определена или наследована,

– private (частный) – операция, вызванную только объектом того класса, в котором она определена.

Под операцией понимается сервис, который экземпляр класса может выполнять, если к сервису будет произведено соответствующее обращение.

Классы могут находиться в следующих отношениях.

Ассоциация – зависимость между объектами разных классов, каждый из которых является равноправным ее членом и обозначает количество экземпляров объектов каждого класса, которые принимают участие в связи (0 – если ни одного, 1 – если один, *n* – если много).

Зависимость между классами, при которой класс использует определенную операцию другого класса.

Экземпляризация – зависимость между параметризированным абстрактным классом–шаблоном (template) и реальным классом, который инициирует параметры шаблона (например, контейнерные классы языка C++).

Пакет – это совокупность элементов (объектов, классов, подсистем и т.п.), представленная подсистемами разного уровня детализации. В пакете задается модель прецедентов, описывающих важные функциональные возможности и требования. Пакет определяет пространство, занимаемое элементами, т.е. его составляющими и ссылками на него. Объединение элементов в пакеты происходит тогда, когда они используются совместно или касаются такого аспекта, как интерфейс с пользователем, ввод/вывод и т.п. Пакет может быть элементом конфигурации, структуры ПС из отдельных модулей или из заведомо определенного состава их вариантов.

Подход к проектированию ПС. Базовыми понятиями UML–метода являются:

– онтологии домена для определения состава классов объектов домена, их атрибутов и взаимоотношений, а также услуг, которые могут выполнять объекты классов;

– модели поведения, предназначенной для определения возможных состояний объектов, инцидентов, инициирующих переходы из одного состояния к другому, а также сообщений, которыми обмениваются объекты;

– модели процессов, которая определяет действия, выполняемые объектами;

– технологии проектирования с помощью диаграмм UML для получения прототипа создаваемой системы, его генерации в выходной код и тестирование.

По мере осмысления поставленных перед разрабатываемой системой задач, рассматриваются и подбираются диаграммы или уточняются ранее созданные. Осмысление проблемы происходит путем ее декомпозиции на более простые составляющие. Следуя объектно-ориентированной (ОО) парадигме, можно сказать, что декомпозиция проблемы состоит в определении состава объектов и особенностей их взаимодействия.

Наличие спектра областей интересов или мнений при рассмотрении каждой системы обусловили структуру средств моделирования UML как собрания нотаций, каждая из которых отвечает определенной структуре, аспекту или интересу рассматриваемой системы.

Таким образом, применение UML приводит к декомпозиции проблемы, решаемой целевой системой, при этом каждая декомпозиция задается своим набором диаграмм и отвечает одному из приведенных выше аспектов системы.

Средства моделирования UML разрешают автономно рассматривать и последовательно детализировать отдельно выделенные цели проблемы (которые может решать программная система, и которые формулируются в требованиях к ней).

Далее определяются носители интересов (актеров), которым соответствует любая из целей, и возможные варианты удовлетворения этих целей с помощью вариантов использования отражающих представление пользователей о назначении и функциях системы и считающихся первой итерацией требований к разработке ПС.

Для вариантов использования определяется состав объектов, взаимодействие которых обеспечивает решение задачи, детализуются свойства объектов, их ассоциации и характер взаимодействия.

В терминах элементов моделирования UML последовательные шаги по декомпозиции сложной системы представляются последовательностью диаграмм, а именно:

1) сложная проблема трансформируется в совокупность целей;

2) любая из целей трансформируется в совокупность диаграмм вариантов использования;

3) варианты использования трансформируются в совокупность взаимодействующих объектов, для каждого из которых оп-

ределяются его особенности – атрибуты и ассоциации, представленные диаграммами классов;

4) для объекта определяется поведение, а именно, операции, которые он может выполнять (методы классов), состояния, в которых он может находиться, правила переходов от состояния к состоянию (диаграммы состояний) и действия, которые сопровождают переходы (диаграммы действий);

5) для каждого из вариантов использования определяются его поведенческие особенности: последовательность взаимодействий объектов во времени (диаграммы последовательности), особенности кооперативного поведения объектов, задействованных в варианте использования (диаграммы кооперации).

Определенная таким образом цепочка трансформаций *проблемы–цели–варианты использования–объекты* отображает ступени концептуализации, т.е. достижения последовательного снижения сложности ее частей.

UML – стандартный метод, которому соответствуют этапы ЖЦ – выработка требований, проектирование архитектуры системы с помощью диаграмм, обеспечение перехода от диаграмм к автоматизированному программированию системы и получения ПС в виде пакета (диаграмм компонентов и размещения). Для полученного варианта системы проводится тестирование компонентов системы. Общая схема трансформации проблемы в программу ее решения средствами UML приведена на рис. 2.5.

Основной особенностью этого метода является спецификация ОО приложений за счет:

– формальной метамодели для описания синтаксиса и семантики элементов UML, а также для унификации многих аспектов проектирования на объединенном (архитектурном, инскрементальном и итерационном) процессе проектирования объектных приложений;

– совокупности диаграмм и объектов с параллельными и распределенными функциями для графического представления ОО приложений;

– множества разнообразных идиом для отображения конкретной модели ПрО средствами языка UML.

Унифицированный процесс – UP (Unified Process) – это процесс, который задается упорядоченным набором этапов ЖЦ разработки ПС и выполняется итеративно.

Формализация моделей процессов в RUP обеспечивается средствами UML и дает возможность описывать требования и преобразовывать их постепенно к готовому продукту.

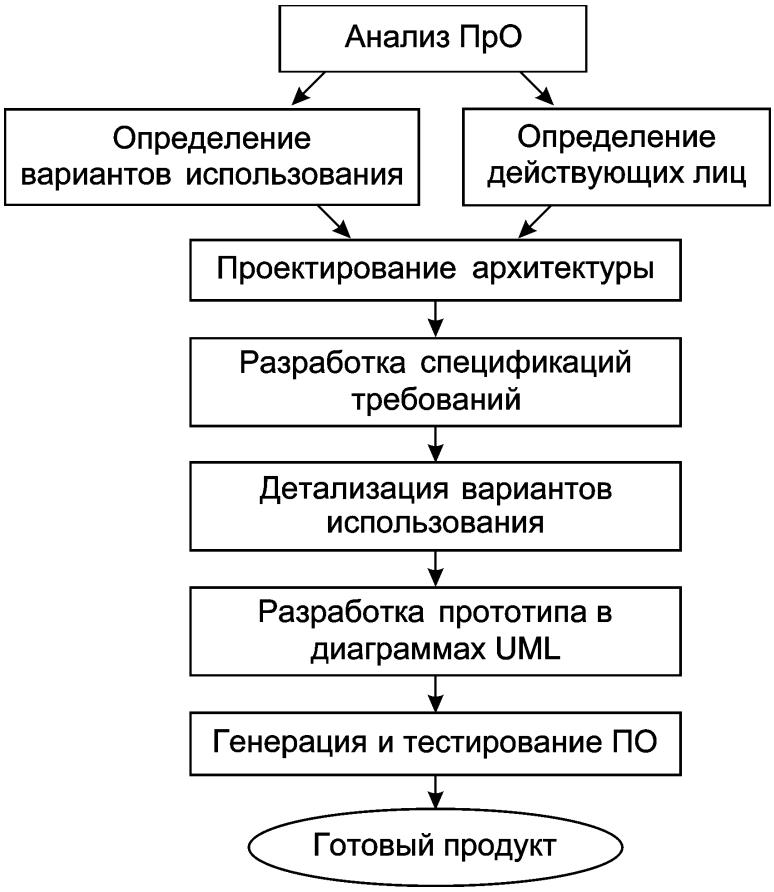

РИС. 2.5. Схема визуального моделирования и проектирования в UML

UP базируется на вариантах использования для описания требований [20, 21] к системе. Главный элемент проектирования – модель вариантов использования системы, на основе которой разрабатываются модели анализа, проектирования и реализации. Каждая модель последовательно анализируется на соответствие модели вариантов использования, в которую входят входные данные для спецификации классов и подсистем, для подбора и спецификации тестов, а также при планировании итераций разработки и интеграции ПС. Каждая итерация состоит из потока работ по разработке требований, проектированию и реализации ПС и включает в себя создание следующих моделей (рис. 2.6):

– вариантов использования, отражающих взаимодействие между пользователями и ПС;

– анализа, обеспечивающего спецификации требований к системе и описание вариантов использования как кооперации между концептуальными классификаторами;

– проектирования, ориентированного на создание статической структуры и интерфейсов системы, реализацию вариантов

использования в виде набора кооперaций между подсистемами, классами и интерфейсами;

– реализации, включающей в себя компоненты системы в исходном виде на ЯП;

– размещения компонентов в операционной среде компьютеров;

– тестирования.

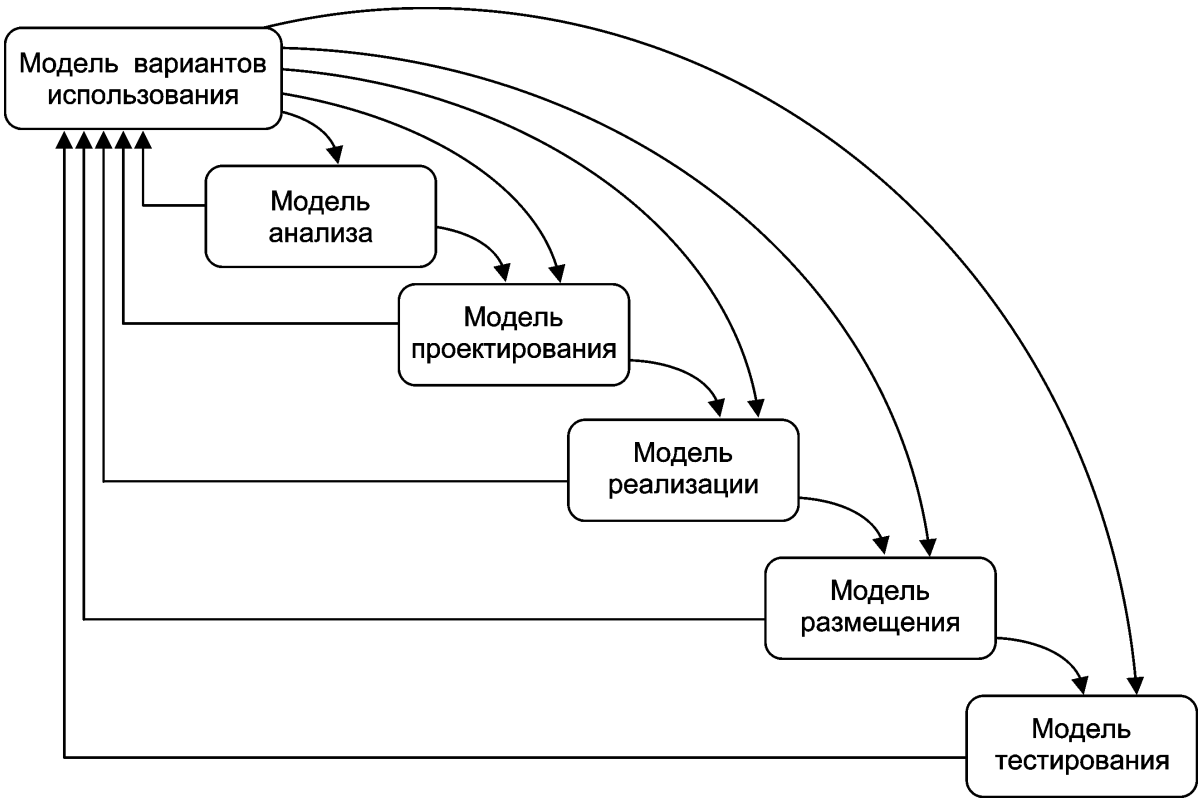

РИС. 2.6. Связь моделей в UP

В этих моделях применяются разные виды диаграмм, например, в модели вариантов использования системы применяются все диаграммы use case, в модели анализа – диаграммы классов, в модели реализации – диаграммы кооперaций и состояний. Зависимости между моделями служат уточнением понятия “диаграмма состояний”. Вместе с тем, все перечисленные модели являются взаимосвязанными, семантически пересекаются и определяют систему как единое целое. Элементы одной модели имеют отношение к элементам других моделей. Например, вариант использования может иметь отношение зависимости к кооперации в модели проектирования, задающей его реализацию.

Отношение между моделями задаются артефактами – диаграммами. Каждая модель создается из нескольких артефактов.

Например, модель анализа состоит из диаграмм классов, диаграмм состояний и диаграмм кооперации.

Артефакты одной модели связаны между собой и должны быть совместимы друг с другом. Отношения между моделями являются не полностью формальными, поскольку части моделей специфицированы на языке метамодели, а другие описаны на естественном языке. Спецификации диаграмм UML является также полуформальными.

Варианты использования могут быть специфицированы различными способами. В общем случае применяется язык, структурированный по типу диалога между действующим лицом (актером), пользователем и системой.

Диалог определяет запрос пользователя и соответствующие ответы системы на высоком уровне абстракции и включает в себя упорядоченную последовательность действий или альтернатив.

В языке UML вариант использования является разновидностью классификатора, имеющего набор операций с соответствующими методами. Операции задают сообщения, которые могут получать экземпляры конкретного варианта использования. Методы описывают реализацию операций в терминах последовательностей действий, которые выполняются экземплярами варианта использования. В общем случае вариант использования имеет операцию, а метод, реализующий эту операцию, содержит последовательность действий, выполняемых последовательно или альтернативно [20].

Пусть *uc* – переменная для обозначения варианта использования (use case), операция которого выполняется над учетной записью. Дадим определение *uc*, например, для этой записи, применяя стандартные обозначения и метамодели UML в следующем виде:

uc.operations = <*op1*>
op1.name = запрос и обновление учетной записи
op1.method.body = {<проверка идентификации пользователя и наличия запроса о долгах, обновление учетной записи>,
<проверка идентификации пользователя, отклонение учетной записи>,
<проверка идентификации пользователя и наличия сервиса, отклонение учетной записи>,
<проверка идентификации пользователя и наличия сервиса, запроса о долгах и запроса на оплату, обновление учетной записи>}.

Тело метода – процедура, специфицирующая реализацию операций в виде последовательности действий, которая обозначается *op.method.body*.

Между именами действий варианта использования и именами действий в кооперации устанавливается соответствие. Каждое имя в варианте использования отображается на имя действия в кооперации, что обеспечивает гибкость в процессе разработки и модификации имен действий.

Вариант использования реализуется кооперацией, если роли классификаторов в данной кооперации взаимодействуют для того, чтобы обеспечить поведение, заданное данным вариантом использования. Если кооперация включает в себя более сложное поведение, то этот вариант использования является частичной спецификацией поведения кооперации.

Кроме того, варианты использования специфицируют действия, видимые за пределами системы, но не специфицируют внутренних действий: создание и удаление экземпляров классификаторов, взаимодействие между экземплярами классификаторов и т.д.

Определение расширения включает в себя как условие расширения, так и ссылку на точку расширения в целевом варианте использования, которая является точкой внутри этого варианта использования. Как только экземпляр целевого варианта использования достигает точки расширения, на которую ссылается отношение расширения, проверяется условие данного отношения. Если условие выполняется, последовательность, удовлетворяющая условию в экземпляре варианта использования, расширяется таким образом, чтобы включать в себя последовательность расширяемого варианта использования.

С практической точки зрения UP представляется упорядоченным набором шагов и этапов ЖЦ, которые выполняются итеративно. Этот процесс является управляемым как при задании требований, так и реализации функциональных возможностей ПрО с заданным уровнем качества и затратами согласно графику работ.

Шаги при выполнении UP управляются прецедентами, технологическим маршрутом делового моделирования, начиная с требований и заканчивая испытаниями ПС. Экземпляр прецедента – это последовательность действий, выполняемых системой с наблюдаемым результатом для конкретного субъекта. Функциональные возможности системы определяются набором

прецедентов, каждый из которых представляет собой некоторый поток событий. Описание прецедента определяет то, что произойдет в системе, когда прецедент будет выполнен. Каждый прецедент ориентирован на задачу, которую он должен выполнить. Набор прецедентов устанавливает всевозможные пути (маршруты) выполнения системы.

UP содержит пять основных этапов – процессов разработки ПС. Завершение этих этапов называется итерацией, которая заканчивается созданием промежуточного продукта. На каждой итерации цикл повторяется, начиная со сбора и уточнения требований.

Этап формирования требования. На этом этапе проводится сбор функциональных, технических и прикладных требований к проекту. На основе требований заказчика и пользователей система описывается так, чтобы достичь понимания между заказчиком и проектной группой. Информация собирается с учетом особенностей существующих систем и документов, подготовленных заказчиком, и включает в себя

– модель ПрО;

– модель схем использования с описанием функциональных и общих требований в форме наборов диаграмм и детального описания каждой схемы;

– дизайн и прототип интерфейса пользователя для каждого актера;

– список требований, которые не относятся к конкретным схемам использования.

Этап анализа. Сформулированные требования уточняются и отображаются в модели сценариев использования. Кроме того, создается модель системы, которая включает в себя формализмы для анализа внутренней структуры системы, определения классов и превращения этой модели в проектные концепции и схемы их реализация.

Этап проектирования служит для уточнения классов и описания их относительно четырех уровней: пользовательского интерфейса, бизнес-решений, уровня доступа к методам и уровня данных. Создаваемая проектная модель системы состоит из подсистем, их распределения между уровнями, интерфейсов классов и объектов, связей классов с узлами в модели развертывания.

На этапе реализации выполняется построение прототипа из компонентов; создание тестов по схемам использования; интеграция компонентов; проверка итерации.

На этапе тестирования тестовая модель уточняется путем исключения неактуальных тестов, создания схемы регрессионного тестирования и добавления тестов для собираемых компонентов. Каждый тест создается на основе вариантов использования и реализует конкретный метод проверки функций системы соответственно условиям тестирования на наборах входных данных.

Инструменты. Основным CASE-инструментом проектирования ОО приложений является Rational Rose.

CASE – это средство, графические возможности которого основаны на использовании языка UML и позволяют проектировать разные виды диаграмм с заданием всех свойств, отношений и взаимодействий. Средства системы обеспечивают проектирование и моделирование общей модели процессов предприятия до конкретной физической модели классов создаваемой ПС. Допускается прямое и обратное проектирование, т.е. доработка старой системы. Результатом моделирования является визуальная логическая модель системы, которая дополняется моделями конкретных классов на ЯП.

В рамках Rational Rose применяются Rose Data Modeler для проектирования системы и баз данных, Rational Rose Professional – для прямого и обратного проектирования шаблонов системы на ЯП, Rose Enterprise для проектирования предприятия и др.

2.1.5. Компонентное программирование

По оценкам экспертов 75 % работ по программированию в мире дублируются. Поэтому переход к повторному использованию компонентов (ПИК) является наиболее производительным. Этот процесс происходил эволюционно: от подпрограмм, модулей и объектов до компонентов путем совершенствования этих элементов, методов их спецификации и композиции. Программирование с помощью компонентов обобщает модульное и объектно-ориентированное программирование. Объекты рассматриваются на логическом уровне проектирования ПС, а компоненты – как непосредственная физическая их реализация. Один компонент может быть реализацией нескольких объектов или даже некоторой части объектной системы, полученной на уровне проектирования [26–33].

Применение ПИК в разработке ПС получило статус типового элемента в программировании. Компонентная программная инженерия (Component–based software engineering – CBSE) [26] является обобщением идеи объектно-ориентированной парадигмы,

повторного использования, абстрактных архитектур и формальных спецификаций. Развитию CBSE способствовала промышленность и рынок программных компонентов. Согласно CBSE компонент является некоторой функцией, которая обеспечивает взаимодействие со средой, а атрибуты, т.е. “не функциональные характеристики” компонента определяют поведение компонента (надежность, защита, безопасность и т.п.). Хотя объектный и компонентный подходы очень близки, они имеют различия. Компонентный подход ориентирован на интерфейс компонентов, повторное его использование и на поведение компонента во время выполнения. Объектно-ориентированный анализ и проектирование используется как метод разработки компонентов и в этом их сходство.

Компонент *– это* самостоятельный продукт, который поддерживает объектную парадигму, реализует часть или отдельную ПрО и взаимодействует с другими компонентами через интерфейсы. Следующее определение.

Программный компонент – это независимый от ЯП, самостоятельно реализованный программный объект, который обеспечивает выполнение определенного множества сервисов и представлен как контейнер с доступом к нему через интерфейс.

Компонент, как элемент композиции ПС, специфицируется с помощью интерфейсов и инкапсуляции его реализации, чем отличается от объектно-ориентированного представления, в котором реализация класса отделяется в зависимости от определения класса. Важным свойством компонента является доступ до него через его интерфейс, полная спецификация которого включает в себя функциональные и нефункциональные атрибуты, использующие разные типы интерфейсов. Спецификация функциональных свойств состоит из описания синтаксиса операций и атрибутов, а их семантика и нефункциональные характеристики задаются с помощью специальных средств – компонентной модели и правил.

Современная компонентная среда является расширением классической модели “клиент–сервер” с учетом специфики построения и функционирования программных компонентов, а также результатов практических реализаций и их апробирования. По отдельным вопросам и концепциям построения архитектуры и функционирования компонентной среды имеется довольно большая литература, например, [15–18]. Рассматривается представление обобщенной архитектуры как некоторой абстракции существующих моделей, сред, распределенных систем и их реализаций.

Основой компонентной среды служит множество серверов компонентов (часто их называют серверы приложений – application servers). Внутри сервера разворачиваются компоненты, представленные как контейнеры. Для каждого сервера может существовать произвольное количество контейнеров.

Контейнер – это оболочка, внутри которой реализуется функциональность компонента. Взаимосвязь и взаимодействие контейнера с сервером строго регламентированы и осуществляется через стандартизированные интерфейсы. Контейнер руководит порождаемыми компонентами или экземплярами компонента с соответствующей функциональностью. В общем случае внутри него может существовать произвольное количество экземпляров-реализаций, каждая из которых имеет уникальный идентификатор.

С каждым контейнером связаны два типа интерфейсов: для взаимодействия с другими компонентами и интерфейс системных сервисов, необходимых для функционирования самого контейнера и реализации специальных функций, например, поддержка распределенных транзакций, в которых принимают участие несколько компонентов.

Первый тип интерфейса (Home интерфейс) обеспечивает управление экземплярами компонента с обязательными реализациями методов поиска, создания и удаления отдельных экземпляров.

Ко второму типу относятся интерфейсы, которые обеспечивают доступ к реализации функциональности компонента. Фактически с каждым экземпляром связан свой функциональный интерфейс.

Экземпляры внутри контейнера могут взаимодействовать друг с другом с помощью системных сервисов к экземплярам, расположенным в других компонентах. Сами компоненты могут размещаться как внутри одного сервера, так и в разных серверах для разных платформ. Такое взаимодействие обеспечивает уникальная идентификация компонентов и экземпляров, а также регламентированные методы взаимодействия с помощью интерфейсов и системных функций.

Архитектура компонентной среды состоит из следующих типов программных объектов:

– серверы компонентов;

– контейнеры компонентов;

– реализаций функциональности, представленных как экземпляры контейнеров-компонентов;

– среды, обусловленные компьютерными платформами, реализациями компонентных моделей, которые обеспечивают установку и конфигурирование компонентов;

– клиентские компоненты интерфейсы, которые обеспечивают конечного пользователя, реализованы в виде разных типов клиентов (веб–клиенты, реализации графического интерфейса и т.д.);

– компонентное приложение.

Каждый из типов объектов может реализовываться отдельно, так как для каждого из них существуют свои спецификации и требования, а также правила взаимодействия с другими объектами компонентного программирования. Все типы объектов образуют цепочку, которая определяет порядок реализации компонентного приложения. Каждый тип объектов может реализовываться отдельным разработчиком и соответственно этому определяется его роль в процессе создания компонентной программы.

В соответствии с делением объектов компонентного программирования на типы и учитывая определенное место каждого из них в процессе создания компонентного приложения, следует сделать вывод, что ЖЦ компонентной программы значительно сложнее, чем ЖЦ в других подходах к программированию. Фактически речь идет о нескольких отдельных ЖЦ для каждого типа объектов. Обобщения этих данных ЖЦ сведены в единую таблицу (табл. 2.1.) с определением их характеристик.

ТАБЛИЦА 2.1. Распределение ролей, объектов и процессов ЖЦ

Роль	**Объект компонентного программирования**	**Базовая концепция и спецификация**	**Модель ЖЦ объекта**
Разработчик сервера	Сервер компонентов	Спецификация сервера. Спецификация компонентной модели. Спецификация системных сервисов.	Модель ЖЦ сервера
Разработчик контейнера	Контейнер	Спецификация контейнера. Спецификация компонентной модели. Спецификация системных сервисов.	Модель ЖЦ контейнера
Разработчик функциональности компонентов	Реализация компонента	Спецификация прикладных интерфейсов Спецификация компонентной модели.	Модель ЖЦ реализации компонента

ПРОДОЛЖЕНИЕ ТАБЛИЦЫ 2.1.

Роль	Объект компонентного программирования	Базовая концепция и спецификация	Модель ЖЦ объекта
Разработчик компонентной среды, специалист по инсталляции компонентов	Инсталляция и параметризация компонента в компонентной среде	Спецификация размещения и настройки компонентов. Спецификация компонентной модели. Спецификация системных сервисов.	Модель развертывания компонента. Модель построения каркаса.
Разработчик клиентских объектов	Клиентские компоненты, GUI-объекты, Веб-интерфейсы	Спецификации клиентских интерфейсов Спецификация компонентной модели.	Модель ЖЦ клиента.
Разработчик приложения	Компонентное приложение	Спецификация приложения. Спецификация размещения и настройки компонентов.	Модель ЖЦ компонентного приложения.

Интеграция компонентов и развертывание не зависят от ЖЦ разработки компонентов, замена любого компонента новым компонентом не должна приводить к перекомпиляции или перенастройке связей в ПС.

Интерфейс компонента может быть определен в виде спецификации точек доступа к компоненту, используя которые, клиент получает сервис в клиент-серверной среде. Так как интерфейс не предоставляет реализацию операций, то можно изменять реализацию без изменения интерфейса и таким образом улучшать исполнительные или функциональные свойства компонента без перестройки ПС в целом, а также прибавлять новые интерфейсы (и реализацию) без изменения существующей реализации ПС.

Семантика интерфейса может быть представлена с помощью контрактов, которые определяют внешние ограничения и поддерживают компонент-инвариант. Кроме того, для каждой операции компонента контракт может определять ограничения, которые должны быть выполнены клиентом перед вызовом операции (предусловие), и условия, которое компонент выполнит после завершения операции (постусловие). Предусловие и постусловие определяют спецификацию поведения компонента и зависят от состояния, которое поддерживает компонент, а также интерфейсов и связанных с набором инвариантов.

Контракты и интерфейс связаны между собою, но содержат разные сущности. Интерфейс представляет собою коллекцию операций или функциональных свойств спецификации сервисов, которые поддерживает компонент. Контракт задает описание поведения компонента, нацеленное на взаимодействие с другими компонентами, и отражает семантику функциональных свойств компонента.

Таким образом, спецификации семантики компонента определяет его интерфейс и ограничения. Каждый интерфейс состоит из набора операций (сервисов, которые он предлагает или требует). С каждой операцией связан набор предусловий и постусловий.

Нефункциональные свойства определяют характер поведения компонента, который не может выражаться через стандартные интерфейсы, а являются специальными расширениями интерфейса, которые обеспечивают принципы взаимодействия с другими компонентами или средой.

Компонент может использоваться многократно в качестве ПИК, к которым относятся формализованные артефакты деятельности разработчиков ПС при реализации некоторых функций и которые могут применяться в новых разработках. А*ртефактом* может быть реальная порция информации, создаваемая при выполнении деятельности, связанной с разработкой ПС систем различного назначения, в частности промежуточные продукты процесса разработки ПС: требования, постановки задач, заготовки программ, программы, комплексы, системы и т.п. Ими также могут быть: спецификации, модели, архитектура, каркас (framework) и т.п., а также готовые ПС.

Для объединения разных видов компонентов применяются шаблоны развертывания, которые сохраняются в скрытой части абстракции компонента. К спецификации компонента могут прибавляться новые шаблоны интеграции или изменяться старые. В некоторых классах ПИК параметры интеграции в новую среду включаются в интерфейс компонента, что сужает круг задач компонента для его повторного использования. Интеграция компонентов в архитектуру целевой ПС в CBSE состоит из нескольких типичных методов объединения, среди которых наибольшее распространение получили паттерны, каркасы и контейнеры [8, 26].

Паттерн – абстракция, которая содержит описание взаимодействий совокупности объектов, ролей участников и их ответственности. Он практически определяет повторяемое решение в

проблеме объединения ПИК в программную структуру. Для каждого объединения определяется взаимодействие объектов при совместной кооперативной деятельности с заданием абстрактных участников, их ролей, распределения полномочий (или обязанностей). Паттерны классифицируются по трем уровням абстракции:

– высокий уровень связан с глобальными свойствами и архитектурой системы, скомпонованной из компонентов в виде архитектурного паттерна, который охватывает общую структуру и организацию ПС в виде набора подсистем с определением их ролей и отношений между ними;

– средний уровень абстракции уточняет структуру, поведение отдельных подсистем и компонентов ПС, а также взаимодействие между ними;

– нижний уровень представляет собой абстракцию определенного вида (например, объединение компонентов по аналогии), которая зависит от выбранной парадигмы и ЯП.

Кроме паттернов, в сборочной стратегии широко используется *каркас* – типовая повторно возникающая ситуация на уровне модели ПС, которая определяет структуру проекта и имеет недоопределенные элементы, обозначенные пустыми слотами для расположения в них новых определенных компонентов. В этом смысле каркас становится ПИК со свойством экземпляризации и представленный высокоуровневой абстракциею проекта ПС, в которой отделены функции компонентов от задач управления ими. В бизнес-функциях компонентов каркас задает надежное управление ими. Каркас объединяет множество взаимодействующих между собою объектов в некоторую интегрированную среду, предназначенную для решения заданной конечной цели типа “белый или черный ящик”.

Каркас типа “белый ящик” включает в себя абстрактные классы для представления цели объекта и его интерфейсов. При реализации эти классы наследуются в конкретные классы с указанием соответствующих методов реализации, как это принято в ООП. Каркас типа «черный ящик» характеризуется тем, что в видимой, информационной, части имеется описание точек входа и выхода. Через эти точки можно входить и выходить из компонента не обязательно через конечный оператор. Таким образом, каркас определяет контекст интеграции отдельно построенных программных частей. Физически он реализуется с помощью одного или больше паттернов, которые в свою очередь выступают в роли инструкций по реализации проектных решений.

Заполненный компонентами каркас становится *контейнером* и сам может быть компонентом ПС, обеспечивает инкапсуляцию компонентов и доступ к компонентам через каркас, который определяет возможные варианты наполнения контейнера. В нем заданы функции, порядок их выполнения, вызываемые события, сервис, интерфейс, необходимый для обращения к провайдеру сервиса. При этом контракты выражают общие принципы и определяют спецификацию отношений между конкретными компонентами, которые могут отличаться от спецификации компонентов как частей композиции.

Создание компонентной системы начинается с построения к*омпонентной модели* (рис. 2.7), включающей в себя проектные решения по композиции компонентов, разные типы паттернов, связи между ними, способы взаимодействия и операции развертки ПС в среде функционирования.

Композиция компонентов из каркасов включает в себя следующие типы:

– компонент–компонент обеспечивает непосредственное взаимодействие компонентов через интерфейс на уровне приложения;

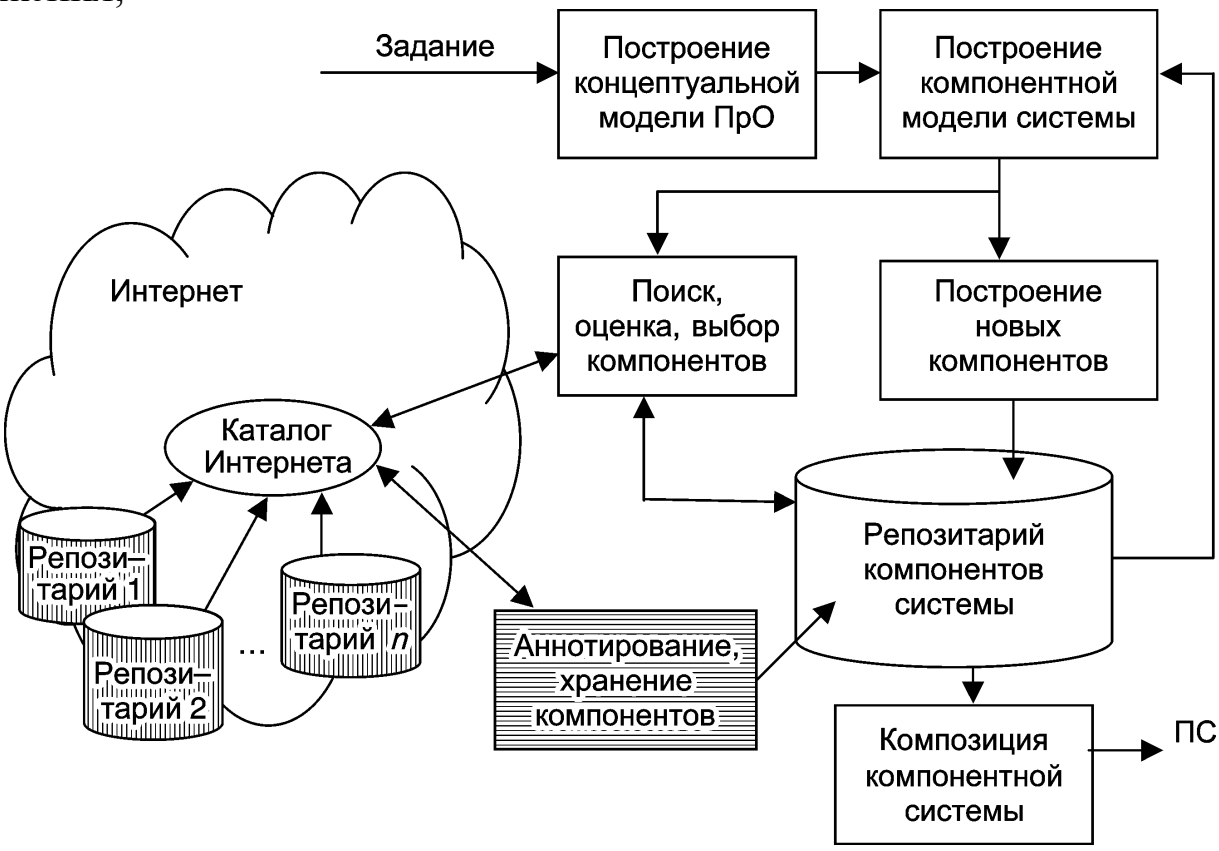

РИС. 2.7. Концептуальна схема построения компонентных ПС в среде Интернета

– каркас–компонент обеспечивает взаимодействие каркаса с компонентами, управление ресурсами компонентов и их интерфейсами на системном уровне;

– композиция компонент–каркас обеспечивает взаимодействие компонента с каркасом по типу «черный ящик», в видимой части которого находится спецификация для развертывания и выполнения функции на сервисном уровне;

– каркас–каркас обеспечивает взаимодействие каркасов, каждый из которых может загружаться в гетерогенной среде и взаимодействовать через интерфейсы на сетевом уровне.

Готовые компоненты, как и ПИК, накапливаются в хранилищах (библиотеках, репозитариях) конкретных ПС, либо в репозитариях Интернета и могут многократно использоваться при построении ПС [30–31]. Каждый репозитарий Интернета ориентирован на конкретную предметную область.

Инструментальными системами разработки компонентных ПС являются RSEB, OOram, CORBA и др. [42–45].

Система RSEB реализует метод разработки ПС, основанный на нотации UML, объектно-ориентированном методе и использовании ПИК многократного применения. Построение разных ПС проводится с помощью Use Case. Компонент в RSEB — это тип, класс или любое другое программное средство (например, Use Case анализа, проектирования или реализации), разработка которого производится с учетом его многократного применения.

Компонентная система в RSEB рассматривается как ПС, содержащая ряд ПИК, и повторно используемые нефункциональные характеристики (надежность, безопасность и др.). В качестве примера компонентной системы можно привести повторно используемые каркасы графических пользовательских интерфейсов и математические библиотеки [43–44]. Практически компонентная система, изготовленная из этих заготовок, позволяет производить композицию или генерацию ПС, других прикладных систем, если в них явно выделены родовые подсистемы с многократно используемыми решениями. Инженерия предметной области в системе RSEB состоит из двух процессов:

1) конструирование семейства ПС путем разработки и сопровождения общей многоуровневой архитектуры систем;

2) конструирование компонентных систем из элементов семейства. Процесс разработки различных отдельных частей ПС осуществляется путем их конструирования и реализации из надежных, расширяемых и гибких ПИК.

Система OOram обеспечивает поддержку инженерии разработки ПС с помощью ПИК и следующих процессов [44]:

– создания компонентной модели, разработки спецификации компонентов модели, ПИК и их размещение в репозитарии системы для последующего использования;

– разработки системы, начиная с формулирования требований и заканчивая введением в действие ПС;

– производства ПС из набора компонентов многократного применения с учетом анализа рынка, маркетинга, продажи продукта, его упаковки и др.

CORBA технология обеспечивает получение сервиса от компонента, который представлен в ЯП (например, Lava) и имеет спецификацию интерфейса в языке IDL. Система CORBA генерирует Java – компонент в виде файла для сервера или клиента, а также интерфейсный IDL – файл для их взаимодействия. Для стороны клиента создается форма IOR (Interoperable Object Reference), которая поддерживает наименование компонентов в сервере. Браузер CORBA при запросе компонентом сервиса проверяет содержащиеся имена, генерирует выходной код и вставляет его в файл клиента. Сервер дополняется кодом, содержащим связи экземпляра сервента с именами сервисов [28–30, 68–70, 74].

В среде CORBA формируются шаблоны для интегрирования компонентов:

– клиент–класс (Client class) вызывает метод, который будет выполнен сервером;

– стаб–класс (Stub class) конвертирует метод инициации работы клиента и может быть использован при связывании компонентов на стороне клиента;

– брокер–класс (ORB class) управляет передачей данных и методов;

– исполнитель-класс (Implementation class) содержит сервент, который регистрируется в ORB и может быть использован клиентом;

– сервер–класс (Server class) создает сервент, выполняет ссылку к IOR, делает доступной и записывает сервент в стандартный файл;

– скелетон–класс (Skeleton class) конвертирует инициирующий метод в формат ORB, который может быть прочитан экземпляром сервента.

Для реализации ПИК типа CORBA используется также шаблон, который обеспечивает реинженерию компонента. Есть так-

же шаблоны, поддерживающие работу с адаптером POA (Portable Object Adapter), создающим экземпляр объекта, доступный разным ORB. Для обеспечения доступа к ресурсам этого сервера на веб-сервере выполняется небольшая программа – сервлет на языке Java, которая не зависит от платформы, может размещаться в разных средах (без ретрансляции), используя все возможности библиотеки классов Java. Создание сервлетов выполняет инструментарий JSDK (Java Servlet Development Kit).

По запросу клиента к веб-серверу взаимодействие с ним выполняет CGI программа (Common Gateway Interface), которая разрабатывается на любых ЯП и создает процесс обработки этого запроса.

Для реализации ПИК типа сервлет используются следующие типы шаблонов:

– веб-модуль (WebModule) поддержки функционирования сервлету на основе файлов с Java-классами для beans-компонентов, статических HTML документов и аплетов;

– веб-группы модулей (WebModuleGroup), предназначенные для создания групп взаимодействующих веб-модулей на сервере с помощью сервлетов;

– сервлеты и HTML файлы.

JAVA-технология базируется на стандартной модели EJB (Enterprise Java Beans), предназначенной для обеспечения взаимодействия разных компонентов с помощью вызова удаленного метода RMI (Remote Method Invocation) языка Java. Согласно этой модели программные компоненты группируются в прикладную программу для работы в любой среде на виртуальной машине JVM (Java Virtual Machine). Beans-компонент разрабатывается, как ПИК для разных сред. Механизм развертывания beans-компонентов на сервере базируется на программах, записанных в языке Java. Есть специальная утилита построения приложения для конфигурирования, объединения его компонентов и подключения к ним новых. Компонентное приложение включает в себя следующее [72]:

– сеансы, которые ориентированы на состояние конкретного доступа клиента или общего бизнес-ресурса независимого от клиента;

– сущности для связи с БД и представления данных в объектной форме;

– управление событиями с помощью сообщений;

– загрузки (deployment) со сведениями об отладке, администрировании и управлении компонентов;

– отображения возможностей одних компонентов в другие компоненты, способные к анализу и динамическому выполнению;

– интроспекции анализа beans-компонентов для определения возможностей и разной информации о компоненте.

При создании beans-компонентов используется интерфейс RMI для определения бизнесов-методов и их реализации. Каждый такой компонент имеет свой контейнер, который вызывает и регулирует все аспекты ЖЦ.

Для определения функциональных свойств компонентов, ориентированных на ПИК, используются проектные шаблоны свойств и событий. К свойствам beans-компонентов относится подмножество состояний, значения которых определяют поведение и внешний вид компонента. Beans-компоненты генерируют события или посылают их другим объектам, а проектные шаблоны обеспечивают идентификацию этих событий.

2.1.6. Аспектно-ориентированное программирование

Аспектно-ориентированное программирование (АОП) [35–41] – это парадигма построения гибких к изменению ПС за счет добавления новых функций, средств безопасности и взаимодействия компонентов с другой средой, синхронизации одновременного доступа частей ПС к данным, вызова новых общесистемных средств, управления событиями и др.

Аспект – это некоторая заготовка, к которой проявляется интерес одного или нескольких заинтересованных лиц в программировании [40, 41]. Аспектом может быть некоторая функция, ПИК, элемент готовой программы, отдельный компонент, концепция взаимодействия, защиты и др. Созданная средствами АОП ПС из отдельных программ семейства может включать в себя набор ПИК, объекты, методы и аспекты, используемые как средство дополнения ПС необходимыми концепциями взаимодействия или защиты программ и данных в новой среде и которые могут пересекать (переплетать) компоненты, усложняя процесс вычислений.

Для преодоления сложности, аспект представляется модулем (или функцией), обеспечивающим связь с другими объектами ПС и снижение количества переплетений программных компонентов за счет кодов аспектов, встроенных в некоторые компоненты системы. Реализация аспектов в различных частях программного кода ПС решается также путем установления перекрестных ссылок

и точек соединения, посредством которых осуществляется связь аспекта с транзакциями, распределенным доступом, защитой данных и т.п.

Основой АОП является метод разбиения задач ПрО на ряд функциональных компонентов, использование разных видов аспектов (синхронизация, взаимодействие, защита и др.) для встраивания их в выделенные отдельные компоненты, как некоторые реализации, оказывающие влияние на выполнение объединенных программных компонентов ПС.

В качестве аспекта может использоваться также некоторая практическая задача (идея), которая интересует нескольких заинтересованных членов проекта и представлена в виде варианта использования, функции, как обязательного элемента компонента или программы. Некоторые аспекты могут выполняться на этапах ЖЦ процесса разработки, они кодируются, тестируются и улучшают результат разработки ПС.

Создание конечного продукта ПС выполняется по технологии, соответствующей разработке компонентных систем, с тем различием, что здесь используются аспекты, которые задают условия выполнения компонентов (безопасность, защиту, взаимодействие и др.) в среде функционирования. В процессе разработки взаимодействующие лица выполняют разные роли, которые, в частности, могут выполнять современные программные агенты, такие как определение архитектуры, управление проектом, повышение качества и продуктивности разработки ПС.

Так как современные ЯП не содержат средств инкапсуляции аспектов в проектных решениях системы, то в некоторые инструментальные системы компонентного типа введен дополнительный механизм фильтрации входных сообщений, с помощью которых выполняется изменение параметров и имен текстов аспектов в конкретно заданном компоненте системы. В результате появляется «нечистый» код компонента (т.е. код с пересекаемыми его аспектами), который требует разработки новых подходов к композиции компонентов, ориентированных на ПрО и выполнение ее функций.

Общие средства композиции объектов ООП или компонентов (вызов процедур, RPC, RMI, IDL и др.) в АОП являются недостаточными, так как аспекты требуют декларативного сцепления между частичными описаниями, а также связывания отдельных обрывков описаний из различных объектов. Один из механизмов композиции – фильтр композиции, суть которого состоит в об-

новлении заданных аспектов синхронизации или взаимодействия без изменения функциональных возможностей компонента с помощью входных и выходных параметров сообщений, которые проходят фильтрацию и изменения, связанные с переопределением имен или функций самих объектов. Фильтры делегируют внутренним компонентам параметры, переадресовывая ранее установленные ссылки, проверяют и размещают в буфере сообщения, локализуют ограничения на синхронизацию, и готовят компонент для выполнения.

В связи с тем, что в ОО-программах может содержаться много мелких методов, которые самостоятельно не выполняют расчеты и обращаются к другим методам, расположенным в областях внешнего уровня, Деметер [41–44] сформулировал закон, согласно которому не разрешаются длинные последовательности методов, связанные с передачами параметров с помощью внутренних объектов. В результате создается код алгоритма, который содержит имена классов, не задействованных в выполнении расчетных операций. При необходимости внесения изменений в структуру классов, создается новый дополнительный класс, который расширяет ранее созданный код и не вносит качественных изменений в расчетные программы.

С точки зрения моделирования аспекты можно рассматривать как каркасы декомпозиции системы, в которых отдельные аспекты синхронизации и взаимодействия пересекают ряд многократно используемых ПИК (рис. 2.8).

Разным аспектам проектируемой системы могут отвечать и разные парадигмы программирования: объектно-ориентированные, структурные и др.

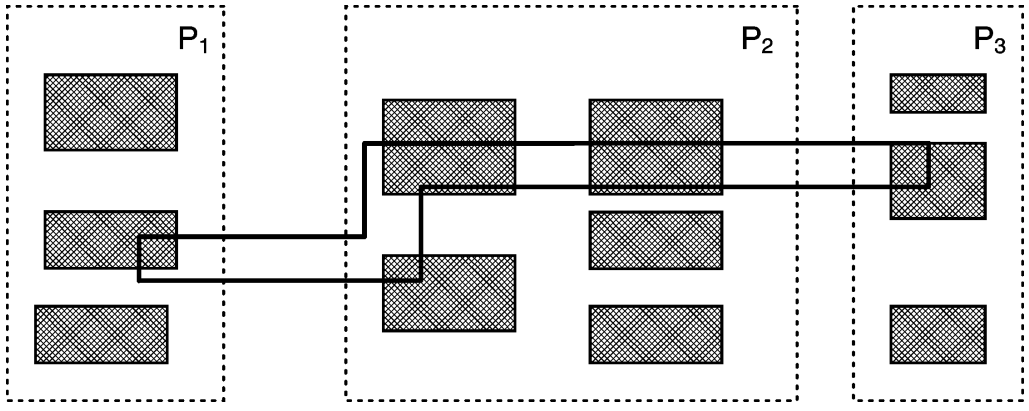

РИС. 2.8. Пересечение функциональных модулей программ P_1, P_2 и P_3 аспектами защиты

Они относительно проектируемой ПрО образуют мультипарадигмную концепцию аспектов, такую как синхронизация, взаимодействие, обработка ошибок и др., и требуют значительных доработок процессов их реализации. Кроме того, можно устанавливать связи с другими предметными областями для описания аспектов приложения в терминах родственных областей. Появились языки АОП, которые позволяют описывать пересекающиеся аспекты в разных ПрО. В процессе компиляции переплетения объединяются, оптимизируются, генерируются [41] и выполняются в динамике.

Существенной чертой любых аспектов является модель, которая пересекает структуру другой модели, для которой первая модель является аспектом. Так как аспект связан с моделью, то ее можно перестроить так, чтобы аспект стал, например, модулем и выполнял функцию посредника, беря на себя все варианты взаимодействия. Однако решение таким образом проблемы пересечения может привести к усложнению и понижению эффективности выполнения созданного модуля или компонента. Переплетение может проявиться на последующих этапах процесса разработки, когда реализуются аспекты, и они делают запутанным выходной код. Один из путей оптимальной реализации аспектов – это минимизация сцепления между аспектами и компонентами с помощью ссылок на языковые конструкции, варианты использования и образцы. Реализация аспектов в различных блоках кода позволяет устанавливать перекрестные ссылки между ними, тем самым осуществляется связь с соответствующей моделью. В этих блоках появляются точки соединения сообщений, которые обеспечивают связь между транзакциями, обработкой ошибок и т.п.

Связь между характеристиками и аспектами может быть выявлена в ходе анализа ПрО. Создается динамическое связывание с помощью косвенного кода или виртуальных таблиц для повторного связывания или статическое или «жесткое» связывание в период компиляции.

АОП стимулирует разработку новых механизмов композиции, ориентированных на ПрО, и выполнение ее задач. Аспекты с точки зрения моделирования можно рассматривать как каркасы декомпозиции системы с многократным использованием. АОП соответствует мультипарадигмовой концепции, сущность которой состоит в том, что разным аспектам проектируемой ПС, должны отвечать разные парадигмы программирования: объектно-ориентированные или структурные. Каждая из парадигм относительно реализации разных аспектов ПС (синхронизации, внедрения, обработки оши-

бок и др.) требует усовершенствования и обобщения применительно к каждой ПрО.

В АОП используется модель модульных расширений, создаваемая в рамках метамодельного программирования. Эта модель предлагает оперативное использование новых механизмов композиции в отдельные части ПС или их семейств с учетом предметно-ориентированных возможностей языков (например, SQL) и каркасов, которые поддерживают разные аспекты [41].

Технология разработки прикладной системы с использованием АОП базируется на технологии ООП и включает в себя такие этапы (рис. 2.9):

1. Декомпозиция функциональных задач с предположением многоразового применения соответствующих модулей и выделение аспектов, т.е. свойств их выполнения (параллельно, синхронно и т.д.).

2. Анализ способов и языков спецификации аспектов.

3. Определение в модулях точек соединения аспектов или ссылок на них.

4. Определение механизмов композиции (вызовов процедур, методов, сцеплений) функциональных модулей многоразового применения и аспектов в точках их соединения, как фрагментов модулей с обеспечением свойств управления выполнением этих модулей, или ссылок из этих точек на другие модули.

РИС. 2.9. Технологическая схема проектирования ПС в АОП

5. Создание объектной или компонентной модели и дополнение ее входными и выходными фильтрами сообщений, посылающих другим объектам, на которые есть ссылки, заданий на выполнение их методов или управления синхронизацией, защитой и т.д. Система фильтров по своей семантике отображает модель EJB, работающую на стороне сервера и управляющую их безопасностью, транзакциями и защитой доступа и др.

6. Анализ библиотеки расширений для выбора некоторых функциональных модулей, необходимых для реализации задач домена.

7. Компиляция модулей и аспектов в прикладную программу и ее отладка.

Технология разработки программного продукта является итеративной и последовательной. За время разработки происходит многократное возвращение к каждому предыдущему этапу для улучшения промежуточных продуктов и в конечном итоге конечного результата. В процессе разработки всегда решаются также задачи профилирования, трассировки, соблюдение проектных соглашений, слежения за корректностью входных и выходных данных на разных уровнях абстракции, отслеживается поведение объектов в многопоточной среде. Особенностью технологии является анализ и использование ПИК, а также методов сквозной функциональности этих компонентов.

Процесс разработки ПС может проводиться с оценкой характеристик системы системным тестировщиком, который собирает ряд количественных данных о процессе выполнения программы, например, сколько раз и какими модулями используется аспект в программе, к каким файлам обращается программа, и сколько времени она на это тратит и др.

В рамках АОП разработана технология использования шаблонов, как способа реализации протокола взаимодействия объектов ПС. Под *шаблоном* понимается описание взаимодействия компонентов, адаптированных для решения общей задачи ПС. Кроме того, он именует и идентифицирует ключевые аспекты структуры ПС, а также вычленяет функции и участвующие компоненты с их ролями и отношениями.

Абстрактная модель шаблона может быть неоднократно использована. Компоненты шаблона не всегда удается применять в другом шаблоне, так как компоненты сильно связаны в контексте поведения реализуемого шаблона. Поэтому добавление и уда-

ление реализации шаблона из кода системы является достаточно сложной задачей, требующей проведения рефакторинга.

В работе [41] проведен сравнительный анализ аспектной реализации шаблонов проектирования и сравнение с "традиционной" объектно-ориентированной реализацией. Отмечается, что модульность кода предполагает локализацию сквозной функциональности абстрактной модели шаблона в коде аспекта. Для повторного использования шаблона разработан общий протокол взаимодействия компонентов. Определяется метрика способности к встраиванию, которая характеризует возможность удалить или добавить в систему аспектный код, реализующий данный шаблон, без модификации компонентов шаблона.

Инструменты. Для эффективной реализации аспектов разработаны AspectJ, İP – библиотека расширений, активные библиотеки, а также проведено расширение ЯП Smalltalk и др. средствами описания аспектов (**AspectC++, AspectC, AspectC#, JAC).**

Система AspectJ. Исследовательский центр Xerox PARC разработал данную систему поддержки АОП, базисом которой является язык Java, она может встраивается в другие системы: Eclipse, Sun ONE Studio, Forte 4J и Borland JBuilder [40].

Язык системы AspectJ является расширением языка Java средствами АОП, любая программа, написанная на Java, будет выполняться в системе AspectJ. Она состоит из компилятора (ajc), отладчика (ajdb), и генератора документации (ajdoc).

Компилятор выдает байт-код, совместимый с виртуальной машиной Java. Расширение языка Java касается способов описания правил интеграции аспектов и Java-объектов, к которым относятся следующие ключевые понятия языка AspectJ:

– точка выполнения JoinPoint в программе, ассоциированная с контекстом выполнения (вызов метода, конструктора, доступ к полю класса и др.);

– набор Pointcut (срез) для точек JoinPoint и удовлетворяющих заданному условию;

– набор инструкций Advice в языке Java, выполняемых до, после или вместо каждой из точек выполнения JoinPoint, входящих в заданный срез;

– единица модульности Aspect и задание срезов точек выполнения и инструкций, которые выполняются в Advice;

– способность аспекта Introduction изменять структуру Java-класса путем добавления новых полей и методов, а также иерархию класса.

Точки среза и набор инструкций определяют правила интеграции и вместе формируют модуль на срезе системы.

При разработке ПС с использованием средств языка AspectJ выделены три принципа АОП:

– выделение в отдельные модули сквозную функциональность путем аспектной декомпозиции;

– реализация каждого требования отдельно;

– интеграция аспектов в программный код.

Интеграция аспектов (weaving) происходит в момент компиляции. Модель построения готовой ПС при использовании компилятора ajc приведена на рис. 2.10 [40].

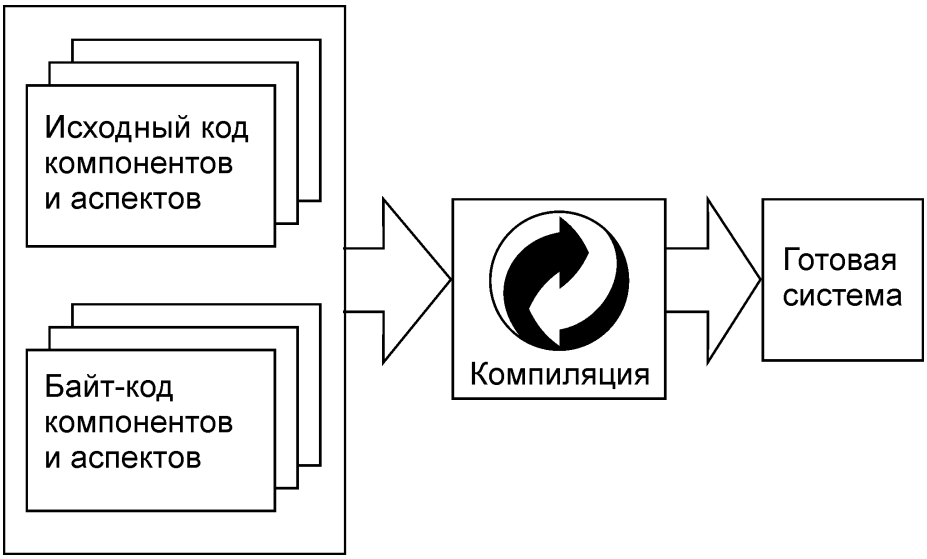

РИС. 2.10. Интеграция аспектов и компонентов

После компиляции компонентов и аспектов получается готовая система с интегрированной сквозной функциональностью по правилам, описанным в аспектных модулях.

Имеются и другие реализации АОП: **AspectC++, AspectC, AspectC#,** такие как расширение языков C++, C, C# аспектами;. **JAC** – основа, написанная на языке Java, для создания распределенных АОП ПС; **Weave.NET** – проект реализации механизма поддержки АОП без привязки к конкретному ЯП внутри компонентной модели .NET Framework и др.

В IP-библиотеке размещены некоторые функции компиляторов, методов, средства оптимизации, редактирования, отображения. и др. Например, библиотека матриц, с помощью которой вычисляются выражения с массивами, обеспечивает скорость выполнения, предоставления памяти и т.п. Использование таких

библиотек в расширенных средах программирования называют родовым программированием, а решение проблем экономии, перестройки компиляторов под каждое новое языковое расширение, использование шаблонов и результатов предыдущей обработкой относят к области ментального программирования [39, 41, 43]. Данная библиотека включают в себя отдельные функции компиляторов, средств оптимизации, редактирования, отображения понятий, перестройки отдельных компонентов компиляторов под новое языковое расширения, а также средства программирования на основе шаблонов и т.п. Библиотеки с такими возможностями получили название библиотек генерирующего типа.

Другой вид библиотек АОП – *активные библиотеки*, которые содержат не только базовый код реализации понятий ПрО, а и целевой код, обеспечивающий компиляцию, оптимизацию, адаптацию, отладку, визуализацию и редактирование. Библиотека предоставляет разным инструментальным средствам (компиляторам, анализаторам кода, отладчикам и т.п.) описание самих элементов.

Активные библиотеки пополняются средствами и инструментами интеллектуализации агентов, с помощью которых создаются специализированные агенты, предназначенные для решения конкретных задач реализуемой ПрО.

2.1.7. Генерирующее (порождающее) программирование

Порождающее программирование (generate programming) – это парадигма разработки ПС, основанная на генерации и моделировании групп или отдельных элементов ПС из разных продуктов программирования: объектов, компонентов, аспектов, сервисов, ПИК, систем, характеристик, каркасов и т.п. Базисом этого программирования является ООП с механизмами использования готовых компонентов, отсутствующих в ООП, а также свойств изменчивости, взаимодействия, синхронизации и др. [42]. В нем могут применяться другие методы программирования для поддержки инженерии ПрО, как дисциплины проектирования семейств ПС из разных, ранее указанных, продуктов программирования. В рамках данного программирования построена объединенная и стройная технология генерации как отдельных ПС, так и их семейств. В результате сформирован базис будущего программирования, включающий в себя современные методы программирования, новые формализмы и объединяющие модели, посредством которых можно будет создавать более долговременные качественные программные изделия семейств ПС по принципу конвейера.

Главным элементом программирования является не уникальный программный продукт, созданный из ПИК для конкретных применений, а семейство ПС или конкретные его экземпляры. Элементы семейства не создаются с нуля, а генерируются на основе общей генерирующей модели домена (generative domain model), т.е. модели семейства, включающей в себя средства определения членов семейства, компоненты реализации и ПИК, из которых собирается любой представитель этого семейства, и базы конфигурации, отображающей спецификации членов семейства.

В созданном программном члене семейства отражается максимум знаний о его производстве, а именно, конфигурация, инструментарий измерения и оценки, методы тестирования и планирования, отладка, визуальное представление и пр. Эти аспекты отображают специфику ПрО многократно используемых ПИК, представленных в активных библиотеках [43, 44].

Активные библиотеки содержат не только базовый код реализации понятий ПрО, но и целевой код по обеспечению компиляции, оптимизации, отладки, визуализации и др. Этот код представляется разными инструментальными системам в виде описания функций. Фактически компоненты активных библиотек выполняют роль интеллектуальных агентов, в процессе взаимодействия которых создаются новые агенты, ориентированные на предоставление пользователю возможности решать конкретные задачи ПрО.

Для связи агентов при выполнении задач генерации, преобразования и взаимодействия разных объектов создается инфраструктура, т.е. расширяемая среда программирования. Используя эту среду, можно конструировать ПС из компонентов библиотек, а также из специальных метапрограмм среды, которые осуществляют редактирование, визуализацию, взаимодействие компонентов в расширяемой среде. Вместе с тем имеется возможность пополнять эту среду новыми сгенерированными компонентами в рамках отдельных ПС семейства, которые относятся к числу компонентов многоразового применения. Кроме того, ЯП компонентов расширяется новыми аспектами, которые расширяют одновременно и ПрО новыми возможностями.

Главной целью порождающего программирования является разработка готовых компонентов для целого семейства и автоматическое их предоставление другим членам семейства. Реализации этой цели соответствует два сформировавшихся направления использования ПИК:

1) *прикладная инженерия* (application engineering) – процесс производства конкретных ПС из ПИК, созданных ранее в среде самостоятельных ПС, или как отдельных элементов процесса инженерии некоторой ПрО;

2) *инженерия ПрО* (domain engineering), включающая в себя методы разработки, поиска, классификации, адаптации, сбора и т.п. ПИК из созданных ПС или из частей систем семейства многоразового применения ПрО путем сбора, систематизации и сохранения прежде наработанного и заимствованного опыта у соответствующей области. Используются и создаются инструментальные системы поддержки этих методов и внедрения их в новые члены семейства ПС. Технологическая схема инженерии ПрО приведена на рис. 2.11.

РИС. 2.11. Технологическая схема инженерии ПрО

Основным требованием к *инженерии ПрО* является обеспечение многоразового применения используемых решений для семейства ПС, а в *инженерии приложений* – производство (линейка) одиночной системы из ПИК по требованиям к ней.

Архитектура ПрО представляет собой высокоуровневую структуру проекта, в которой интерфейсы компонентов формально определены, является каркасом (фреймворком) для конструирования программных продуктов из ПИК.

Основными этапами инженерии ПрО являются:

– анализ ПрО и выявление объектов и отношений между ними;

– определение области действий объектов ПрО;

– определение общих функциональных и изменяемых характеристик, построение модели характеристик, устанавливающей зависимость между различными членами семейства, а также в пределах членов семейства системы;

– создание базиса для производства конкретных программных членов семейства с механизмами изменчивости независимо от средств их реализации;

– подбор и подготовка компонентов многократного применения, описание аспектов выполнения задач ПрО;

– генерация отдельного домена, члена семейства и ПС в целом.

В основе генерации доменной модели для семейства ПС лежит модель характеристик и набор компонентов реализации задач ПрО. Используя данную модель, знания о конфигурации и спецификации компонентов автоматически генерируется отдельный член семейства.

Инженерия ПрО включает в себя вспомогательные процессы:

– *корректировка процессов* для разработки решений на основе ПИК;

– *моделирование изменчивости и зависимостей,* которое начинается с разработки словаря описания различных понятий, фиксации их в модели характеристик и в справочной информации сведений об изменчивости моделей (объектных, Use Case, взаимодействия и др.). Фиксация зависимостей между характеристиками модели избавляет пользователей от некоторых конфигурационных манипуляций, которые выполняются, как правило, вручную;

– *разработка инфраструктуры ПИК* – описание, хранение, поиск, оценивание и объединение готовых ПИК.

При определении членов семейства ПрО используются такие понятия, как пространство проблемы, а в технологии реализации

компонентов на основе каркаса конфигураций – пространство решений.

Пространство проблемы (space problem). Областью разработки является семейство систем, в которых используются компоненты многократного применения, объекты, аспекты и др. Процесс разработки с повторным использованием организуется таким образом, чтобы в нем использовались не только ПИК, но и инструменты, созданные в ходе разработки ПрО. В рамках инженерии ПрО разрабатывается модель характеристик, которая объединяет функциональные характеристики системы, характеристики определения свойств выполнения компонентов и изменяемые параметры разных частей (компонентов) семейства, а также решения, связанные с особенностями выполнения групп ПС.

В рамках инженерии ПрО используются горизонтальные и вертикальные типы компонентов, предложенные OMG-комитетом в системе объектного проектирования CORBA [68, 69]. К горизонтальным относятся общие системные средства: графические пользовательские интерфейсы, СУБД, системные программы, библиотеки расчета матриц, контейнеры, каркасы и т. п. К вертикальным типам относятся прикладные системы (медицинские, биологические, научные и т.д.), методы инженерии ПрО и некоторые методы горизонтального типа по обслуживанию архитектуры многократного применения, интерфейсов, библиотек и др.

Пространство решений (space solution) состоит из компонентов, каркасов, образцов проектирования, а также средств их соединения и оценки избыточности. При этом каркас оснащается изменяемыми параметрами модели, которые требуют лишнюю фрагментацию из «множества мелких методов и классов». С помощью каркаса обеспечивается динамическое связывание аспектов и компонентов, а также реализация изменчивости компонентов в разных приложениях. *Образцы* проектирования обеспечивают создание многократно используемых решений в различных типах ПС. Для задания и реализации таких аспектов, как синхронизация, удаленное взаимодействие, защита данных и т.д. применяются компонентные технологии ActiveX и JavaBeans, а также новые механизмы композиции, метапрограммирования и др.

Инструменты. Примером систем поддержки инженерии ПрО и реализации горизонтальных методов является система DEMRAL [41, 71, 75], предназначенная для разработки библиотек: численного анализа, контейнеров, распознавания речи, графовых вычислений и т.д. Основными видами абстракций этих библиотек ПрО являются абстрактные типы данных (abstract data types–

ADT) и алгоритмы. DEMRAL позволяет моделировать характеристики ПрО в виде высокоуровневой характеристической модели и предметно-ориентированных языков конфигурации.

Система конструирования RSEB [42] базируется на вертикальных методах, ПИК и ориентирована на использование Use Case элементов при проектировании крупных ПС. Эффект достигается, когда вертикальные методы инженерии ПрО «вызывают» различные горизонтальные методы, относящиеся к разным прикладным подсистемам. При работе над отдельной частью семейства системы могут быть задействованы такие основные аспекты — взаимодействие, структуры, потоки данных и др. Главную роль, как правило, выполняет один из методов, например, графический пользовательский интерфейс в бизнес-приложениях и метод взаимодействия компонентов в распределенной, открытой среде (например, в CORBA).

2.1.8. Агентное программирование

Понятие интеллектуального агента появилось более 20 лет назад, его роль в программной инженерии все время возрастает. Так, Джекобсон считает [76], что перспективными ролями агентов будут разработчики архитектуры системы, вариантов использования, тестирования ПС, менеджера проекта и др. Использование агентов в этих ролях в ближайшем будущем будет способствовать повышению производительности, качества и ускорению разработки ПС.

Основным теоретическим базисом данного программирования являются темпоральная, модальная и мультимодельная логики, дедуктивные методы доказательства правильности свойств агентов и др. Рассмотрим сущность основных прикладных аспектов агентной тематики [45–52].

С точки зрения программной инженерии, агент – это самодостаточная программа, способная управлять своими действиями в информационной среде функционирования для получения результатов выполнения поставленной задачи и изменения текущего состояния среды, т.е. агент выполняет задачи, достигая заданных целей путем выполнения программы и изменения состояния среды. Он обладает такими свойствами:

– автономность – способность действовать без внешнего управляющего воздействия;

– реактивность – способность реагировать на изменения данных и среды, и воспринимать их;

– активность – способность ставить цели и выполнять заданные действия для достижения этой цели;

– социальность – способность к взаимодействию с другими агентами (или людьми).

В задачи программного агента входят:

– самостоятельная работа и контроль своих действий;

– взаимодействие с другими агентами;

– изменение поведения в зависимости от состояния внешней среды;

– выдача достоверной информации о выполнении заданной функции и т.п.

С интеллектуальным агентом связываются знания типа: убеждение, намерение, обязательства и т.п. [45, 48, 52]. Эти понятия входят в концептуальную модель и связываются между собой операционными планами реализации целей каждого агента. Для достижения целей интеллектуальные агенты взаимодействуют друг с другом, устанавливают связь между собой с помощью сообщений или запросов и выполняют заданные действия или операции в соответствии с имеющимися знаниями.

Агенты могут быть локальными и распределенными. Процессы локальных агентов протекают в клиентских серверах сети, выполняют заданные функции и не влияют на общее состояние среды функционирования. Распределенные агенты, расположенные в разных узлах сети, выполняют автономно (параллельно, синхронно, асинхронно) предназначенные для них функции и могут влиять на общее состояние среды (рис. 2.12).

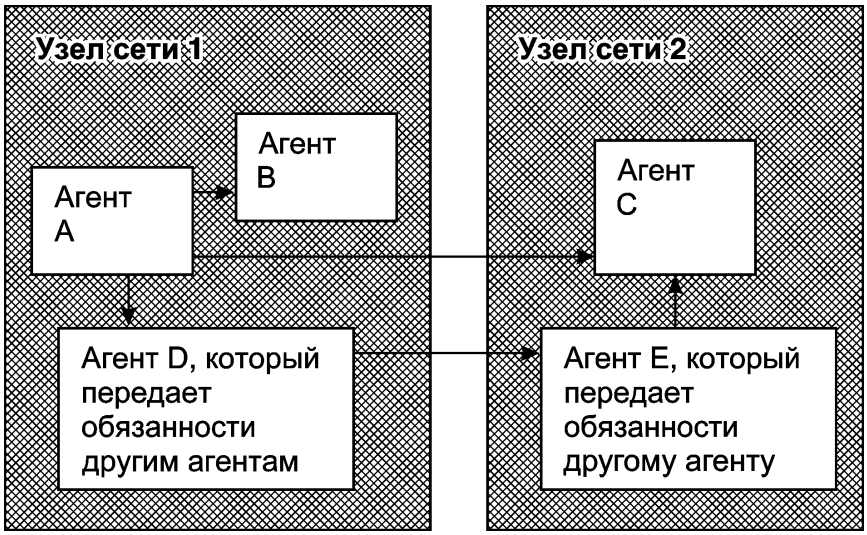

РИС. 2.12. Пример взаимодействия агентов в распределенной среде

В обоих случаях характер взаимодействия между агентами зависит от таких факторов: совместимость целей, компетентность, реакция на нестандартные ситуации т.п.

Основу агентно-ориентированного программирования составляют:

– формальный язык описания ментального состояния агентов;

– язык спецификации информационных, временных, мотивационных и функциональных действий агента в среде функционирования;

– язык интерпретации спецификаций агента;

– инструменты конвертирования любых программ в соответствующие агентные программы.

Спецификация агента уточняется, интерпретируется и компилируется в вычислительное представление агентной программы, пригодное для выполнения в среде функционирования. Агенты взаимодействуют между собой через координацию, коммуникацию, кооперацию или коалицию.

Координация агентов – это процесс, с помощью которого агенты обеспечивают последовательное функционирование при согласованности их поведения и без взаимных конфликтов. Координация агентов определяется:

– взаимозависимостью целей других агентов-членов коалиции, а также возможного влияния агентов друг на друга;

– ограничениями, которые принимаются для группы агентов коалиции в рамках общего их функционирования;

– компетенцией – знаниями условий среды функционирования и степени их использования.

Главным средством коммуникации агентов является транспортный протокол TCP/IP или протокол агентов ACL (Agent Communication Languages). Управления агентами (Agent Management) выполняется с помощью таких сервисов: передача сообщений между агентами, доступ агента к серверу и т.п. Координационные механизмы могут стать обязательством и для управления ими создаются соглашения, которые определяются стоимостью и полезностью, выражаемой прибылью от полученного соглашения между агентами [51–54].

Коммуникация агентов базируется на общем протоколе, языке HTML и декларативном или процедурном (Java, Telescript, ACL и т.п.) описании этого протокола.

Начиная с 1990 г. начало быстро развиваться новое направление – использование агентов [13–16, 30, 32–35, 43, 49], как

средств поиска и обработки информации в Интернете. На международных конференциях неоднократно (1992–1999 гг.) рассматривалась эта проблематика. Ею начали заниматься ряд компаний и академических центров за границей. Их усилиями был создан международный проект FIRA (Foundation for Intelligent Physical) для выполнения исследований, определения базовых понятий и унификации разных подходов в агентной проблематике. Более 10 лет исследованиями этой проблемы занимаются в ИПС НАНУ [47, 48].

Агенты в среде Интернета обеспечивают доступ к информации через серверы информационных ресурсов Интернета. Выбранную информацию агенты в зависимости от своего статуса (агент архива, агент пользователя, агент-диспетчер, информационный агент и др.) обрабатывают, анализируют, фильтруют и передают результат клиенту. Каждый агент выполняет те действия, которые ему предназначены. Модель этой среды состоит из базы знаний и базы данных, модели информационных ресурсов, их свойств, правил работы с ними и типов сообщений. В результате выполнения функций агенты создают поведение среды, которое в любой момент времени находится в некотором состоянии, а агент, выполняя заданные действия, изменяет его в целевое состояние с учетом возникновения нерегулярных состояний (тупиков, нехватки ресурса и др.).

В общем случае среда, в которой действует агент, имеет определенное поведение, которое может быть известно полностью или частично. Состояние среды зависит от информации, имеющейся у агента, а также от таких ее свойств: дискретность состояния, детерминированность (или нет) действий, динамичность или статичность среды, синхронное или асинхронное изменение состояния и т.п.

Для управления работой агентов разработаны мультиагентные поисковые система (МАПС), которые выполняет многоразовые запросы, связанные с профессиональной деятельностью пользователей по поиску в Интернете разного рода семантической информации. В этом поиске агенты обеспечивают более быстрое и точное представление релевантной информации на запрос пользователя сети. МАПС реализует эффективный обмен информацией между ресурсом и пользователями (рис. 2.13).

Главная особенность этой модели – промежуточная среда, которая выполняет дополнительные функции поиска информации с помощью информационного агента, агента-пользователя, агента-поставщика и др.

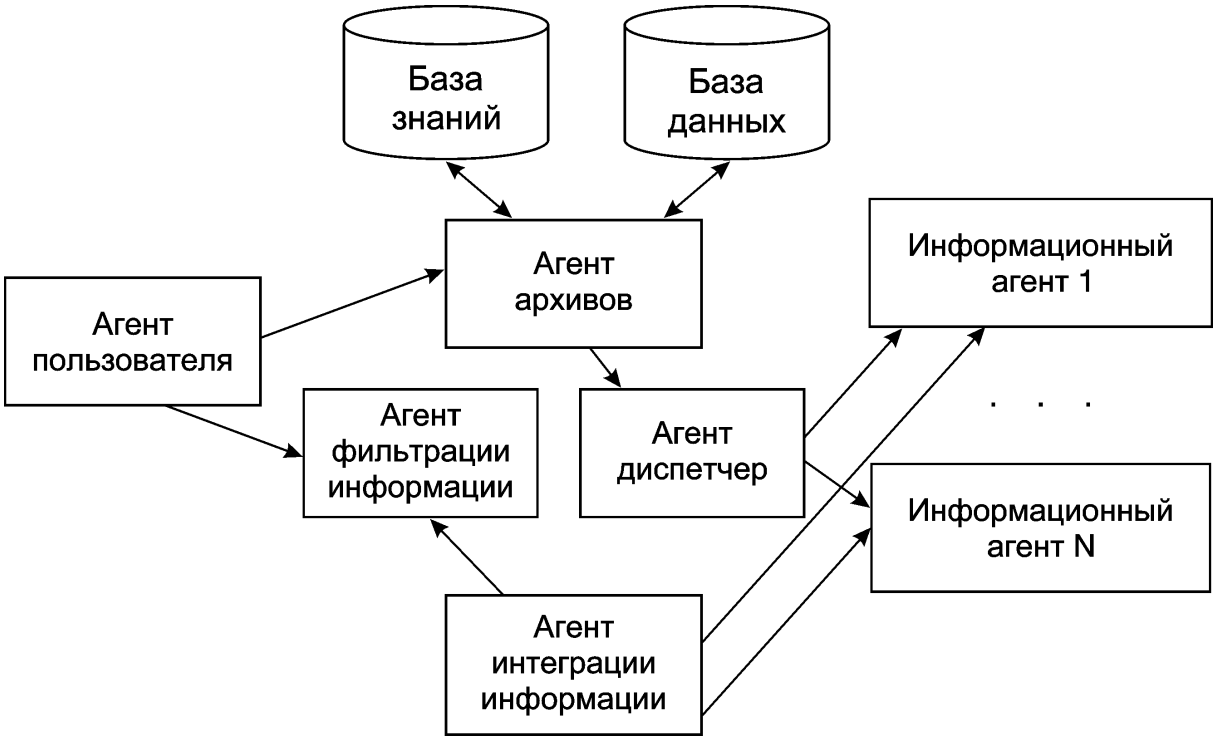

РИС. 2.13. Общая структура МАПС для поиска ресурсов Интернета

В приведенной структуре МАПС агент архива обеспечивает сопоставление добытой в Интернете информации, заданной в запросе, и при ее актуальности система выдает агенту результат на запрос пользователя.

Если информация не найдена, агент архива передает запрос агенту-диспетчеру для продолжения поиска. Этот агент анализирует параметры и данные, запрашивает у агента сервера информационных ресурсов потребную информацию. Полученная информация оценивается на релевантность, и результат передается другим агентам.

В функции агента-интегратора входит объединение ответов на запросы разных агентов информационных ресурсов в единый список для передачи его агенту, фильтрующему этот список, который передает его агенту-пользователя.

Инструменты. Одной из систем построения агентов, основанной на обмене сообщениями в ACL, является JATLite. Она включает в себя Java-классы для создания новых агентов, ориентированных на вычисление функций в распределенной среде. Система Agent Builder предназначена для конструирования программных агентов, которые описываются в языке Java и могут взаимодействовать на основе языка KQML (Knowledge Guery and Manipulation Language). Построенные агенты выполняют сле-

дующие функции: поиск ключевых слов документов по онтологиям, визуализации, отладки и др. На реализацию механизмов взаимодействий агентов ориентирована и система JAFMAS. Ряд мультиагентных систем описано в [48].

2.2. ТЕОРЕТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Теоретическое программирование основывается на функциональных математических дисциплинах (логика, алгебра, комбинаторика) и отражает математический метод анализа ПрО, осмысление постановок задач и разработку программ для получения на компьютере математических результатов. Специалисты с математическим образованием развивают отдельные направления в программировании, объясняя некоторые закономерности в структуре программ и их определении с различных точек зрения: аппарата функций (функциональное программирование, композиционное программирование и др.).

Алгебраисты использовали алгебраический математический аппарат для объяснения действий над объектами программ, выполнения математических операций над их элементами и принципов обработки, исходя из базовых основ алгебры – алгебраическое программирование, алгоритмика и др.

Совместные знания в области математики и алгоритмики послужили основой для решения таких проблем, как постановки некоторых математических задач (например, задача Гильберта, формализация бесконечных рядов, и др.), которые считаются алгоритмически неразрешимыми задачами. В результате был создан общематематический процедурный язык, предназначенный для формальной постановки задач, поиска их решения и алгоритмизации. Такой язык назвали концепторным языком, по своим вожможностям он выходит за рамки алгоритмических языков и в некоторой степени приближается к языку современной математики [79].

Таким образом, разработаны теоретические методы программирования (алгебраическое, композиционное, экспликативное, алгеброалгоритмическое и др.), которые основываются на математических, алгебраических и логико-алгоритмических подходах и методах формального построения и доказательства программ. Авторами украинской теоретической школы программирования были созданы новые формальные методы для решения ключевых проблем программирования [54–63, 78–80, 86–90].

Теория алгебраического программирования (А.А. Летичевского и др.) обеспечивает описание математических конструкций, вычислений, алгебраических преобразований, доказательство математических теорем, а также содержит новые механизмы создания интеллектуальных агентов [54–55].

Экспликативное программирование (В.Н. Редько) определяет теорию дескриптивных и декларативных программных формализмов для адекватного задания моделей структур данных, программ и средств их конструирования. Создана программология – наука о программах, которая объединяет идеи логики, конструктивной математики и информатики и на единой концептуальной основе предоставляет общий формальный аппарат конструирования программ [60–63].

Теоретико-множинный концепторный язык (В.Н. Коваля.) базируется на математической теории решения сложных задач, работающих с неконструктивными элементами, и ориентирован на описание дискретных аппаратно-программных систем [78–83].

Алгебра алгоритмики (Г.Е. Цейтлин) представляет математические механизмы построения алгоритмов в виде схем, задаваемых графами, элементами которых являются конструкции, производные от структурных конструкций. К операциям алгебры относятся суперпозиция, свертка, развертка, а также операции над множествами [86–90].

Далее рассмотрим перечисленные формальные методы программирования с математическим базисом представления объектов и операций над ними. Дадим краткую характеристику этим методам и основным направлениям их развития.

2.2.1. Алгебраическое программирование (АП)

Парадигмой АП является определение интеллектуальных агентов и сред их функционирования с применением математического аппарата, в качестве базиса которого используется понятие транзитивной системы [54–58]. Данный аппарат позволяет определить поведение систем и их эквивалентность. В качестве транзитивных систем в общем случае могут быть компоненты, программы и их спецификации, объекты, взаимодействующие друг с другом и со средой их существования. Эволюция такой системы описывается с помощью истории функционирования систем, которая может быть конечной или бесконечной, и включать в себя обзорную часть в виде последовательности действий и скрытую часть в виде последовательности состояний.

История функционирования включает в себя успешное завершение вычислений в среде транзитивной системы, тупиковое состояние, когда каждая из параллельно выполняющихся частей системы находится в состоянии ожидания событий и, наконец, неопределенное состояние, возникающее при выполнении алгоритма, например, с бесконечными циклами. Расширением понятия транзитивных систем является множество заключительных состояний с успешным завершением функционирования системы и без неопределенных состояний.

Главное инвариантное состояние транзитивной системы – поведение системы, которое можно задать выражениями алгебры поведения *F(A)* на множестве операций алгебры *A*, а именно две операции префиксинга $a \cdot u$, задающие поведение *u* на операции *a*, недетерминированный выбор $u+v$ одного из двух поведений *u* и *v*, который является ассоциативным и коммутативным. Конечное поведение задается константами: Δ, $\perp$, 0, обозначающими соответственно состояния успешного завершения, неопределенного и тупикового. Алгебра поведения частично задается отношением $\leq$, для которого элемент $\perp$ – наименьший, а операции алгебры поведения – монотонные. Теоретически алгебра поведения *F(A)* проверена путем доказательства теоремы про наименьшую неподвижную точку.

Транзитивные системы называют бисимуляцийно эквивалентными, если каждое состояние любой из них эквивалентно состоянию другой. На множестве поведений определяются операции, которые используются для построения программ агентов, а именно, такие операции: последовательная композиция *(u; v)* и параллельная *u//v*.

Среда E, где находится объект, определяется как агент в алгебре действий *A* и функции погружения от двух аргументов *Ins (e, u) = e[u].* Первый аргумент – это поведение среды, второй – поведение агента, который погружается в эту среду в заданном состоянии. Алгебра агента – это параметр среды. Значение функций погружения – это новое состояние одной и той же среды.

Разработанная общая теория выходит за рамки определения вычислительных и распределенных систем, а также механизмов взаимодействия со средой. Базовым понятием является “действие”, трансформирующее состояние агентов, поведение которых, в конце концов, изменяется.

Поведение агентов характеризуется состоянием с точностью до бисимиляции и, возможно, слабой эквивалентности. Каждый

агент рассматривается как транзитивная система с действиями, определяющими недетерминированный выбор и последовательную композицию (т.е. примитивные и сложные действия).

Взаимодействие агентов может быть двух типов. Первый тип выражается через параллельную композицию агентов над той же самой областью действий и соответствующей комбинацией действий. Второй тип выражается через функцию погружения агента в некоторую среду, и результатом трансформации является новая среда.

Язык действий А имеет синтаксис и семантику. Семантика – это функция, определяемая выражениями языка и ставящая в соответствие программным выражениям языка значения в некоторой семантической области. Разные семантические функции дают равные абстракции и свойства программ. Семантика может быть вычислительной и интерактивной. Доказано, что каждая алгебра действий является гомоморфным образом алгебры примитивных действий, когда все слагаемые разные, а представление однозначно с точностью до ассоциативности и коммутативности в детерминированном выборе. Установлено, что последовательная композиция – ассоциативная, а параллельная композиция – ассоциативная и коммутативная. Параллельная композиция раскладывается на комбинацию действий компонентов.

Агенты рассматриваются как значения транзитивных систем с точностью до бисимиляционной эквивалентности. Эквивалентность характеризуется в алгебре поведения непрерывной алгеброй с аппроксимацией и двумя операциями: не детерминированным выбором и префиксингом. Среда вводится как агент, куда погружена функция, имеет поведение типа агент и среда. Произвольные непрерывные функции могут быть использованы как функции погружения, а эти функции определены значениями логики переписывания. Трансформации поведения среды, которые определяются функциями погружения, составляют новый тип эквивалентности – эквивалентность погружения.

Создание новых методов программирования с введением агентов и сред позволяет интерпретировать элементы сложных программ как самостоятельно взаимодействующие объекты.

Теоретические аспекты АП. В АП интегрируется процедурное, функциональное и логическое программирование, используются специальные структуры данных – граф термов, который разрешает применять разные средства представления данных и знаний о ПрО в виде выражений многоосновной алгебры данных. Наибольшую

актуальность имеют системы символьных вычислений, которые дают возможность работать с математическими объектами сложной иерархической структуры. Многие алгебраические структуры (группы, кольца, поля) являются иерархически-модулярными. Теория АП обеспечивает создание математической информационной среды с универсальными математическими конструкциями, вычислительными механизмами, учитывающими особенности разработки ПС и функционирования. АП является основой формирования нового вида программирования – инсерционного, обеспечивающего программирование систем на основе моделей поведения агентов, транзитивных систем и бисимуляционной эквивалентности [55].

Инструменты. Система алгебраического программирования (АПС) основывается на типах системных объектов, базовых вычислительных механизмах и языковых конструкциях AL [56, 58]. Она объединяет процедурный и алгебраический методы программирования, разрешает использовать не только канонические, но и другие системы уравнений. ПрО реализуется: алгебраическими программами (АП-модуль), алгебраическими модулями (А-модуль), интерпретаторами алгебраических модулей и т.п. А-модуль – это представление структуры данных, которые определяются в АП-модулях. Он наследует тип, начальное именование и имеет динамический характер, т.е. состояние, которое может изменяться во времени. Техника программирования в АПС базируется на технике переписывания терминов и используется при автоматизации доказательства теорем, символьных вычислениях, обработке алгебраических спецификаций. В АПС допускается модельно-имитационный класс, т.е. мониторинг данных, моделирование ситуации, условное прерывание, управление экспериментами и др. К дедуктивно-трансформационному классу АПС относятся методы наблюдений, доказательства и повторного использования программ.

2.2.2. Экспликативное программирование (ЭП)

Данное программирование [59–63] ориентировано на разработку теории дескриптивных и декларативных программных формализмов, адекватных моделям структур данных, программ и средств конструирования из них программ. Для этих структур решены проблемы существования, единства и эффективности. Теоретическую основу ЭП составляет логика, конструктивная

математика, информатика, композиционное программирование и классическая теории алгоритмов. Для изображения алгоритмов программ используются разные алгоритмические языки и методы программирования: функциональное, логическое, структурное, денотационное и др.

Принципами ЭП последовательных и многоуровневых программ и их уточнений являются следующие [59].

Принцип развития определения понятия программы в абстрактном представлении и постепенной ее конкретизации с помощью экспликаций.

Принцип прагматичности, или полезности, определения понятия программы выполняется с точки зрения понятия "проблема" и ориентирован на решение задач пользователя.

Принцип адекватности ориентирован на абстрактное построение программ и решение проблемы с учетом *информационности данных* и *апликативности,* т.е. рассмотрение программы, как функции, вырабатывающей выходные данные на основе входных данных. Функция является объектом, которому сопоставляется *денотат* имени функции с помощью отношения *именования* (*номинации).*

Развитие понятия функции осуществляется с помощью *принципа композиционности,* сущность которого состоит в построении программы (функции) с помощью композиций из более простых программ. При этом создаются новые объекты с более сложными именами, описывающими функции и включающими номинативные (именные) выражения, языковые выражения, термы и формулы. Согласно теории Фреге, сложные имена являются дескрипциями.

Принцип дескриптивности позволяет трактовать программу как сложные дескрипции, построенные из более простых программ и композиций отображения входных данных в результаты на основе п*ринципа вычислимости.*

Таким образом, процесс развития программы осуществляется с помощью цепочки понятий: данные–функция–имя функции–композиция–дескрипция. Понятия данные–функция–композиция задают *семантический аспект* программы, а данные–имя функции – дескрипция – *синтаксический аспект.* Главным в ЭП являются семантический аспект, система композиций и номинативности (КНС), ориентированные на систематическое изучение номинативных отношений при построении данных, функций, компози-

ций и дескрипций. КНС задают специальные языковые системы для описания разнообразных классов функций и называются композиционно-номинативными языками функций.

Такие системы тесно связаны с алгебрами функций и данных, они построены в семантико-синтаксическом стиле. КНС отличается от традиционных систем (моделей программ) теоретико-функциональным подходом, классами однозначных *n*-арных функций, номинативными отображениями и структурами данных. Они используются для построения математически простых и адекватных моделей программ параметрического типа с применением методов универсальной алгебры, математической логики и теория алгоритмов. Данные в КНС рассматриваются на трех уровнях: абстрактном, булевом и номинативном. Класс номинативных данных обеспечивает построение именных данных, многозначных номинативных данных или мультиименных данных, задаваемых рекурсивно.

Разработаны новые средства [60] для определения систем данных, функций и композиций номинативного типа, имена аргументов которых принадлежат некоторому множеству имен *Z,* т.е. композиция определяется на *Z-номинативных* наборах именных функций.

Номинативные данные позволяют задавать структуры данных, которым присущи неоднозначность именования компонентов типа множества, мультимножества, реляции и т.п.

Функции обладают свойством апликативности, их абстракции задают соответственно классы слабых и сильных аппликативных функций. Слабые функции позволяют задавать вычисление значений на множестве входных данных, а сильные – обеспечивают вычисление функций на заданных данных.

Композиции классифицируются уровнями данных и функций, а также типами аргументов. Экспликация композиций соответствует абстрактному рассмотрению функций как слабо аппликативных, и их уточнение строится на основе понятия детерминанта композиции, как отображения специального типа. Класс аппликативных композиций предназначен для конструирования широкого класса программ. Доказана теорема о сходимости класса таких композиций в классе монотонных композиций.

Инструменты. Введенные системы данных, функций и композиций реализованы в *классе манипуляционных* данных в БД,

информационных системах и позволяют значительно ускорить процесс обработки запросов, заданных в SQL-подобных языках [61–63]. Практическая проверка теоретического аппарата формализации дедуктивных и ОО БД прошла в ряде экспериментальных проектов.

2.2.3. Доказательное проектирование на основе концепторных языков

Область искусственного интеллекта (игры, распознавание, обучение, индуктивные и дедуктивные построения) самая богатая, но не единственная как источник решения сложных задач, имеющих математическую постановку [78–80]. К другим источникам можно отнести области управления процессами, оптимизации систем, планирования экспериментов и др. Эти задачи, как правило, решаются неочевидным способом, который объясняется их не тривиальностью. Существуют и *языковые* трудности, так как практически отсутствует язык спецификации постановок сложных математических задач для обеспечения их решений.

Между первоначальным описанием сложной задачи и ее решением существует большое расстояние, для преодоления которого выделены три этапа: 1) переход от первоначального описания задачи к формальной постановке; 2) принципиальное обоснование решения задачи; 3) переход от принципиального обоснования к алгоритмическому описанию.

Понятие "принципиальное обоснование решения задачи" обозначает формулировку решения задачи на более высоком уровне абстракции, приближенном к условиям задачи, чем конкретное решение, которое требуется с позиций современной математики, когда формальная постановка задачи рассматривается как принципиальное решение.

Для решения сложных математических задач предложен *общематематический процедурный язык*, так называемый концепторный язык – КЯ, обеспечивающий формальное описание суммирования бесконечных рядов, теоретико-множественных операций с бесконечными множествами, гильбертов оператор и др. [78–85]. Эти особенности выводят КЯ за рамки алгоритмических языков и приближают их к языку современной математики без потери императивных возможностей. Тексты в КЯ являются *концепторами* и в общем случае не алгоритмами.

Основу КЯ составляет теоретико-множественный язык, который содержит декларативные и императивные средства теории

множеств Цермело–Френкеля. Этот язык имеет *ядро* с набором элементов (типы, выражения, операторы) и средствами определения новых типов, выражений и операторов.

Декларативную часть КЯ образует типизированный, многосортный логико-математический язык *выражений X,* объектами которого являются типы и выражения. *Тип*, как и в ЯП, – это средство построения выражений и структуризации множества значений (денотаты). *Выражения* состоят из термов и формул. *Термы* обозначают объекты ПрО, а *формулы* – утверждения об объектах и отношениях между ними. Конструкторы составных типов и формул разделены на четыре категории: функторы и предикаты, конекторы и субнекторы.

Функтор – это конструктор, преобразующий термы в термы. *Предикаты* превращают термы у формулы, *конекторы* включают в себя логические связи и кванторы и служат для преобразования одной формулы в другую. Субнектор (дескриптор) – это конструктор построения термов из выражений, которые содержат формулы. Конструкторами термов являются традиционные арифметические и алгебраические операции над числовыми множествами и вещественными функциями. Приведены таблицы конструкторов множеств, кортежей, отношений, семейств, произведений множеств и др. Конструкторы формул включают в себя предикаты, состоящие из предикатных и числовых символов, а также конекторы, состоящие из логических связок, кванторов и конструкторов теории множеств.

Императивные средства КЯ – это операторы и процедуры, предназначенные для описания объектов ПрО с помощью концепторов, которые содержат разделы для определения объектов решаемой задачи и действий над ними. Описание принципиального решения задачи представляется в виде одного или нескольких иерархически связанных между собою концепторов, каждый из которых представляет собой именованный набор определений и действий следующей структуры:

концептор K (< список параметров >)
  < список импортных параметров>
  < определение констант, типов, предикатов>
  < описание глобальных переменных>
  < определение процедур>
начало K
  < тело концептора>
конец K.

Таким образом, *концептор K* объединяет декларативное описание объектов и императивное тело. Рассматривается два случая: 1) декларативный концептор состоит из определений; 2) императивный концептор − из операторов тела. Декларативные концепторы накапливают определение объектов и понятий, связанных с задачей и ее подзадачами.

После уточнения постановки задачи или разработки метода ее решения проводится построение тела с помощью императивных концепторов. В результате получается принципиальное решение задачи в структурированном виде с формальной спецификацией, необходимой для разработки и верификации алгоритмически сформулированного решения.

О постановке и решении сложных задач. После формулировки задачи на естественном языке осуществляется переход к формальной постановке решения задачи средствами КЯ без заботы о процедурной эффективности. Полученный концептор может использоваться для исследования свойств решаемой задачи и спецификации для разработки алгоритмического решения, приближенного к концепторному решению.

Если полученный концептор описывает неэффективную процедуру, то необходимо найти более эффективную, алгоритм которой эквивалентен данному концептору. Для получения алгоритмического решения задачи используются допустимые аппроксимации концепторного решения в целях замены неконструктивных объектов и неэффективных операций конструктивными и более эффективными аналогами.

Для решения многих проблем КЯ имеет дополнительные возможности. Например, при существовании общего концепторного решения для алгоритмически неразрешимой проблемы эта проблема становится основой для поиска решения классов таких проблем, для которых концептор может быть превращен в эквивалентный алгоритм.

Методы формализации КЯ. Общая схема формализации декларативной и императивной частей КЯ строится аналогично многосортным логико-математическим языкам, а императивный подъязык включает в себя традиционные операторы структурного программирования (присваивание, последовательность, цикл и т.п.). Для формализации неконструктивной семантики КЯ используется теоретико-модельный (денотационный) и аксиоматический подходы.

Денотационный подход состоит в определении *семантики* (*интерпретации*) языка путем подстановки каждому выражению

соответствующего элемента из множества денотатов с помощью функции φ интерпретации символов сигнатуры языка. Каждой константе $c \in C$, каждому функциональному символу $f \in F$ и каждому предикатному символу $p \in P$ сопоставляется объект из множества денотат. Этот способ интерпретации семантики выражений и операторов языка аналогичен *денотационной семантики* ЯП. Главное отличие семантики КЯ от семантики программ – это ее не конструктивность. Действия, описываемые концепторными операторами, не обязательно являются эффективными.

Возможной альтернативой формализации языка КЯ является не использование понятия интерпретации. С каждым КЯ можно связать некоторую дедуктивную теорию, которая отражает свойства концепторов этого КЯ.

На основе данного языка строится формальная дедуктивная теория путем выделения из множества всех формул подмножества аксиом и правил вывода. Для каждой пары R_1, R_2 формул дедуктивной теории и каждого оператора I создается *операторная* формула $\{R_1\}\, I\, \{R_2\}$ с таким утверждением: если R_1 истинно именно перед выполнением оператора I, то завершение оператора I обеспечивает истинность R_2. Иными словами, формула R_1 есть предусловие, а R_2 — постусловие оператора I.

В работе Хоара [85] показано, что аксиомы и правила вывода для операторов ЯП, можно непосредственно использовать в соответствующих концепторных операторах. Они не требуют эффективной вычислимости выражений и операторов. Поэтому в дедуктивной теории, которая содержит эти аксиомы и правила вывода, можно адекватно описывать свойства неэффективных процедур с помощью неконструктивных объектов и неразрешимых формул.

Аксиоматическое описание КЯ – это возможность использования аппарата теории доказательств при исследовании КЯ и верификации концепторов с помощью дедуктивной теории. Благодаря наличию в концепторных языках декларативных средств во многих случаях удается в компактной форме описывать разные ПрО, а наличие императивных средств КЯ разрешает представить решение задач в процедурном (концепторном) виде. На протяжении последнего десятилетия КЯ активно применяются для постановок и принципиального решения задач распознавания динамических обстановок в гидроакустике, радиолокации и т.п. в целях создания прикладного программного обеспечения в составе соответствующих объектов новой техники, которая выпускается предприятиями Украины и России [78–80].

Технология доказательного проектирования. В настоящее время во многих областях науки и техники возникает потребность в ПС, исходные тексты которых содержат много строк, а в аппаратных схемах миллионы логических элементов. Но современные ЯП и технологии проектирования ПС не обеспечивают быструю и надежную их разработку. Поэтому в последнее время исследованы принципиально новые технологии проектирования, которые применяют язык спецификации и методы автоматического доказательства теорем как средства более точного абстрактного описания ПС, тестирования и использования средств автоматического доказательства. Реализация этих принципов основана на формализованном описании (концепторному) поведения дискретной системы, т.е. на логико-алгебраической спецификации, которая базируется на операционной и логической семантиках для доказательства утверждений о свойствах аппаратно-программных систем.

На первом этапе проектированная дискретная система *S* рассматривается как черный ящик с конечным набором входов, выходов и состояний. Области значений входов и выходов являются произвольными, а функционирование системы *S* определяется набором частичных отображений как носителей алгебраической системы с набором операций, которые входят в состав сигнатуры и образовывают *частичную алгебру,* формальное описание которой выполняется с помощью алгебраических спецификаций [82]. Полученная спецификация является программой моделирования состояний дискретной системы. Иными словами, первый этап завершается разработкой алгебраической спецификации для частичной алгебры, которая отвечает системе *S.*

На втором этапе система *S* детализируется в виде совокупности взаимозависимых подсистем S_1 ,..., S_n, каждая из которой описывается алгебраической спецификацией. Получается новая спецификация системы *S* на основе определенных функций переходов и выходов, для которых необходимо доказывать корректность и тот факт, что новые функции выходов совпадают с начальными.

Процесс детализации выполняется на уровне элементной базы или элементарных программ и сопровождается доказательством их корректности. В конечном итоге создается система *S,* которая с функциональной точки зрения эквивалентна исходной абстрактной спецификации, что соответствует доказательному проектированию [78-80].

Логико-алгебраические спецификации системы. В основе использования данных спецификаций лежит представление ПрО в

виде алгебраической системы, которая задается с помощью соответствующих носителей, сигнатуры и следующих принципов. *Первый принцип* логико-алгебраического подхода к спецификациям ПрО состоит в уточнении понятий ПрО. *Второй принцип* состоит в описании свойств ПрО в виде аксиом, которые формулируются языком предикатов первого порядка и хорновскими атомарными формулами. *Третий принцип* состоит в формулировке так называемых термальных моделей, строящихся из основных термов спецификации и выбранной минимальной модели.

Для любого непротиворечивого множества хорновских аксиом наименьшая термальная модель всегда существует и вместе со всеми изоморфными ей моделями образует абстрактный класс инициальных моделей. Поэтому в логико-алгебраических спецификациях обычно ограничиваются хорновскими формулами из-за простоты аксиом, которые упрощают процесс автоматического доказательства теорем. Можно избавиться от отношений в сигнатурах спецификаций, заменяя их булевыми функциями, которые отвечают характеристическим функциям этих отношений.

При этом алгебраические системы становятся многоосновными алгебрами, а аксиомы – спецификациями: тождественными и квазитождественными. Именно *алгебраические* спецификации являются наиболее используемыми, поскольку для них существуют относительно эффективные алгоритмы выполнения, которые преобразуют спецификации ЯП высокого уровня. Поэтому такие языки называют языками выполненных логико-алгебраических спецификаций. Их операционная семантика основана на *переписывании термов,* а создаваемая алгебраическая спецификация получает логическую семантику, используемую при доказательстве теорем.

2.2.4. Алгоритмика программ

На протяжении многих лет Г.Е.Цейтлин занимается разработкой теоретических аспектов алгоритмов, представляемых блок-схемами как способа детализации и реализации алгоритмов. Аппарат блок-схем пополнен математическими формулами, свойственными математике, которые используются при аналитических преобразованиях и обеспечивают улучшение качества алгоритмов. Построение и исследование алгебры алгоритмов впервые осуществил В.М. Глушков в рамках проектирования логических структур ЭВМ. В результате была построена теория систем алгоритмических алгебр (САА), которая затем была положена

в основу обобщенной теории структурированных схем алгоритмов и программ, называемой сейчас алгоритмикой [86–89].

Объектами алгоритмики являются модели алгоритмов и программ, представляемые в виде схем. Метод алгоритмики базируется на компьютерной алгебре и логике и используется для проектирования алгоритмов прикладных задач. Построенные алгоритмы описываются с помощью ЯП и реализуются соответствующими системами программирования в машинное представление. В рамках алгоритмики разработаны специальные инструментальные средства реализации алгоритмов, которые базируются на современных объектно-ориентированных средствах и методе моделирования UML.

Таким образом, обеспечивается полный цикл работ по практическому применению разработанной теории алгоритмики при реализации прикладных задач, начиная с постановки задачи, формирования требований и разработки алгоритмов и кончая получением программ решения этих задач.

Алгебра алгоритмов. Под алгеброй алгоритмов $AA = \{A, \Omega\}$ понимается основа A и сигнатура Ω операций над элементами основы алгебры. С помощью операций сигнатуры может быть получен произвольный элемент $q \in AA$, который называется системой образующих алгебры. Если из этой системы не может быть исключен ни один элемент без нарушения ее свойств, то такая система образующих называется базисом алгебры.

Операции алгебры удовлетворяют следующим аксиоматическим законам: ассоциативности, коммутативности, идемпотентности, закону исключения третьего, противоречия и др. Алгебра, которой удовлетворяют перечисленные операции, называется булевой.

В алгебре алгоритмов используется алгебра множеств, элементами которой являются множества и операции над множествами (объединение, пересечение, дополнение, универсум и др.).

Основным объектами алгебры алгоритмики являются схемы алгоритмов и их суперпозиции, т.е. подстановки одних схем в другие. С подстановкой связана развертка, которая соответствует нисходящему процессу проектирования алгоритмов, и свертка, т.е. переход к более высокому уровню спецификации алгоритма. Схемы алгоритмов соответствуют конструкциям структурного программирования:

Последовательное выполнение операторов A и B записывается в виде композиции $A*B$; альтернативное выполнение операторов A и B (f (A, B)) означает, если u истинно, то выполняется

A иначе B; цикл (u (A, B)) выполняется, пока не станет истинным условие u (u − логическая переменная).

С помощью этих элементарных конструкций строиться более сложная схема $П$ алгоритма:

$П ::= \{(u_1)\ A_1\}$,

$A_1 ::= \{(u_2)\ A_2 * D\}$

$A_2 ::= A_3 * C$,

$A_3 ::= \{(u)\ A, B\}$, $u ::= u_2 \wedge u_1$.

Проведя суперпозицию путем свертки приведенной схемы алгоритма $П$, получаем формулу:

$П ::= \{(u_1)\ (u_2)\ ((u_2 \wedge u_1)\ A, B)\ * C\ * D)\}$.

Важным результатом работы [86] является эквивалентное представление и толкование известных алгебр с помощью алгебры алгоритмики.

Алгебра Дейкстры АД = {CAA, $L(2)$, СИГН} − двухосновная алгебра, основными элементами которой являются множество CAA операторов, представленные структурными блок-схемами, множество $L(2)$ булевых функций в сигнатуре СИГН, в которую входят операции дизъюнкции, конъюнкции и отрицания, принимающие значения из $L(2)$. С помощью специально разработанных механизмов преобразования АД в алгебре алгоритмики установлена связь между альтернативой и циклом, т.е. $\{(u)\ A\} = \{(u)\ E, A * \{(u)\ A\}\}$, произвольные операторы представлены суперпозицией основных операций и констант. Операция фильтрации $Ф\ (u) = \{(u)\ E, N\}$ в АД представлена суперпозицией тождественного E, неопределенного N оператора и альтернативы алгебры алгоритмики, где N − фильтр разрешения выполнения операций вычислений. Оператор цикла **while do** также представлен суперпозицией операций композиции и цикла в алгебре алгоритмики.

Алгебра схем Янова (АЯ) включает в себя операции построения неструктурных логических схем программ. Схема Янова состоит из предикатных символов множества $P\{p_1, p_2,...\}$, операторных символов множества $A\{a_1, a_2,...\}$ и графа переходов. Оператором в данной алгебре является пара $A\{p\}$, состоящая из символов множества A и множества предикатных символов. Граф перехода представляет собой ориентированный граф, в вершинах которого располагаются преобразователи, распознаватели и один оператор останова. Дуги графа задаются стрелками и помечаются знаками + и −. Преобразователь имеет один преемник, а распознаватель − два. Каждый распознаватель включает в себя условие выполнения схемы, а преобразователь представляет собой операторы, состоящие из логических переменных, принадлежащих множеству $\{p_1, p_2,...\}$.

Каждая созданная схема АЯ отличается большой сложностью, требует серьезного преобразования при переходе к представлению программы в виде соответствующей последовательности действий, условий перехода и безусловного перехода. В работе [86] разработана теория интерпретации схем Янова и доказательство эквивалентности двух операторных схем, исходя из особенностей алгебры алгоритмики.

Для представления схемы Янова аппаратом алгебры алгоритмики в сигнатуру операций АЯ вводятся композиции A^* В и операции условного перехода, которая в зависимости от условия u выполняет переход к следующим операторам или к оператору, помеченному меткой (типа goto). Условный переход трактуется как бинарная операция $П(u, F)$, которая зависит от условия u и разметок схемы F. Кроме того, производится замена альтернативы и цикла типа while do. В результате выполнения бинарных операций получается новая схема F', в которой установлена $П(u)$ вместо метки, и булевы операции конъюнкции и отрицания. Эквивалентность выполненных операций преобразования обеспечивает правильность неструктурного представления.

Система алгебр Глушкова $АГ$ = {ОП, УС, СИГН }, где ОП и УС − множество операторов суперпозиции, входящих в сигнатуру СИГН, и логических условий, определенных на информационном множестве ИМ, СИГН = {СИГНад ∪ Прогн.}, где СИГНад − сигнатура операций Дейкстры, а Прогн. − операция прогнозирования. Сигнатура САА включает в себя операции алгебры АД, операции обобщенной трехзначной булевой операции и операцию прогнозирования (левое умножение условия на оператор $u = (A^* u')$, порождающую предикат u = УС, такой, что $u(m) = u'(m')$, $m' = A(m)$, $A \in$ ОП. ИМ − множество обрабатываемых данных и определения операций из множеств ОП и УС. Сущность операция прогнозирования состоит в проверке условия u в состоянии m оператора A и определения условия u', вычисленного в состоянии m' после выполнения оператора A.

Данная алгебра ориентирована на аналитическую форму представления алгоритмов и оптимизацию алгоритмов по выбранным критериям.

Алгебра булевых функций и связанные с ней теоремы о функциональной полноте и проблемы минимизации булевых функций также сведены к алгебре алгоритмики. Этот специальный процесс отличается громоздкостью и рассматриваться не будет [86].

Алгебра алгоритмики и прикладные подалгебры. Алгебра алгоритимки пополнена двухуровневой алгебраической системой и механизмами абстрактного описания данных (классами алгоритмов).

Под многоосновной алгоритмической системой (МАС) понимается система $S = \{\{D_i / i \in I\}, СИГН_0, СИГН_n\}$, где D_i – основы или сорта, $СИГН_0$, $СИГН_n$ – совокупности операций и предикатов, определенных на D_i. Если они пусты, то определяются многоосновные модели – алгебры. Если сорта интерпретируются как множество обрабатываемых данных, то МАС представляет собой концепцию АТД в виде подалгебры, широко используемую в объектно-ориентированном программировании. Таким образом, устанавливается связь с современными тенденциями развития современного программирования.

Практическим результатом исследований и развития алгебры алгоритмики является построение оригинальных инструментальных систем проектирования алгоритмов и программ на основе современных средств поддержки ООП.

Читателям предоставляется возможность познакомиться более подробно с приведенными источниками.

2.3. ФОРМАЛЬНЫЕ МЕТОДЫ

Формальные методы более всего связаны с математическими техниками спецификаций, верификации и доказательства правильности создаваемых программ. Эти методы содержат математическую символику, формальную нотацию, аппарат вывода и поэтому они трудно используются на практике рядовыми программистами. Кроме того, эти постоянно развиваемые средства не вкладываются в стандартизованные процессы ЖЦ, в которых регламентированы все основные и дополнительные процессы (управление качеством, проектом, ресурсами и др.). За многие годы своего существования (более 20лет) такие известные формальные методы, как VDM, Z, RAISE [82–84] используются редко, эпизодически в реальных проектах, чаще всего в университетских и академических организациях, а до промышленного производства фактически не дошли.

Бытует мнение, что формальные техники трудно использовать практически особенно в таких системах, как системы реального времени, где особенно важно применение формальных методов для создания более надежных программ, стоимость разработки которых возрастает, так как затрачивается много времени на их формальную спецификацию и верификацию.

Отметим также, что специалисты международного класса в области ПО, которые провели систематизацию накопившихся знаний, т.е. практических методов, подходов и средств технологии программирования, не отвели в своем документе SWEBOK [90] место для рассмотрения формальных методов.

Существует разрыв между бурно развивающимися инженерными методами (например, оценка качества, стоимости ПС, измерение процесса, продукта и др.) проектирования и формальными методами спецификации и верификации программ.

О стандартизации одного из языков спецификации говорить преждевременно, поскольку требуется длительная практическая работа и апробация их в конкретных условиях создания ПС в соответствии с этапами ЖЦ.

Можно привести отдельные примеры использования формальных методов: в Минобороне Великобритании для спецификации задач безопасности военных систем и проверки их на полноту и непротиворечивость, а также тщательной верификации программных компонентов; в атомной энергетике Канады при создании безопасных атомных электростанций и др.

В технике формального проектирования ПС отмечен значительный прогресс в создании обобщенного UML языка визуальной спецификации требований системы и модели ПрО. Этот язык прошел массовую апробацию в практически создаваемых системах и в конкретных инструментальных поддержках. В результате UML доведен до стандарта разработки ПС и отодвинул на второй план такие средства спецификации, как RAISE, несмотря на то, что для него существует программная поддержка Raise Tools, которая обеспечивает спецификацию системы, ее верификацию и доказательство корректности отдельных свойств системы.

Далее рассматриваются некоторые техники спецификации моделей программ и методы формального доказательства.

2.3.1. Формальное описание моделей системы

Приведем наиболее известные из этих моделей:

– обобщенная модель, компоненты которой задаются с помощью языка спецификации Z-схем [82];

– модель системы в виде ациклического графа, формально создаваемая с помощью композиционной теории;

– формальная модель модульной структуры, которая задается в виде графа, имеет язык его спецификации.

Обобщенная модель является формальной, создается с помощью языка спецификации Z-схем и включает в себя описание состояний модели следующими операциями:

– создание узла управления моделью;

– назначение метки узла графа;

– задание внешних характеристик и параметров модулей.

Созданная модель включает в себя:

– совокупность Z-схем с набором деклараций и ограничений;

– инварианты схем для верификации модели.

На определение обобщенной модели повлияли следующие операции:

– выбор модуля из библиотеки шаблонов, переименование шаблонов или их конкретизация соответственно определения узла графа, где зафиксирован шаблон;

– связь модулей в модульную структуру системы, модули которой имеют интерфейс *порта* после конкретизации *слот* и отмечаются они меткой соответствующей Z-схемы;

– верификация связей модулей, которая включает в себя проверку атрибутов и типов данных;

– создание нового шаблона, инкапсуляция его интерфейса и соответствующей схемы.

Графова модель создается с помощью композиционной теории в виде ациклического графа, узлы которого – модули, а дуги – интерфейс между модулями.

Каждому модулю модели соответствует сервис как провайдера (provider), так и потребителя (consumer) услуг. Дуга графа от модуля *M* к *N* определяет интерфейс, который может предоставлять модуль *N* для модуля *M*.

Концепция сервиса поддерживается протоколами сети при взаимодействии компонентов, которые транспортируются через сеть в целях посылки множеств значений параметров серверу сети.

Базис формального метода представления модели – спецификация интерфейсов модулей для двух выше указанных пользователей сервисов. Интерфейс удовлетворяет следующим свойствам:

– разделение (separable) модулей с точки зрения проектирования и спецификации, интерфейс которых не требует описания среды модуля;

– композиция (composition), когда модули соединяются с помощью механизмов композиционной теории в виде системы с гарантированными связями между ними.

Данная теория базируется на модели интерфейса для взаимодействия модулей системы между собою через сетевые протоколы (TCP, UDP и др.). Каждый протокол передает разным модулям этой модели параметры для нахождения требуемого сервиса.

Модель интерфейса задает связь модуля со средой в виде дискретного события (event), которое возникает только тогда, когда модуль и среда действуют вместе, и создают событие, рассматриваемое на стороне интерфейса.

Таким образом, интерфейс специфицирован как множество последовательностей событий для наблюдения за поведением модулей модели.

Предположим, что *S* определяет спецификацию модуля *M,* которая включает в себя все возможные случаи его поведения и задает разные ситуации, которыми могут быть:

– событие интерфейса или состояние наблюдателя системы;

– анализ является конечным или представлен неопределенной последовательностью;

– формализм определения этой последовательности;

– условия выполнения события.

Эти предположения разрешают обеспечить разноуровневую систему взаимодействия модулей модели через механизм протоколов и систему наблюдения за событиями.

Проект модели VM системы состоит из двух модулей: модуля управления CONT и модуля распределения памяти STOR.

Модуль CONT имеет следующую спецификацию действий:

CONT = (coin —–> reguest —> response —> choc —> cont)

Общее назначение этой спецификации состоит в том, что потребитель может вставлять данные в coin для отправки запроса (reguest) модулю памяти. Этот модуль дает ответ (response) на запросы и имеет следующую спецификацию :

STOR = (reguest —> response —> stor)

Модель VM определяет параллельную обработку модулей CONT, STOR, а также связь со средой. При этом спецификация VM имеет вид

VM = (CONT ‖ STOR) \ {reguest, response} = (coin —> choc —> VM).

2.3.2. Формальный метод и спецификация RAISE

RAISE-метод и RSL-спецификация (RAISE Specification Language) [84, 85] были разработаны в 1985–1998 гг. как результат предварительного исследования формальных методов. Метод со-

держит нотации, техники и инструменты для использования формальных методов при конструировании ПС и доказательства их правильности. Метод имеет программную поддержку в виде набора инструментов и методик, которые постоянно развиваются и используются для конструирования и доказательства правильности программ, описанных в RSL и ЯП (C++ и Паскаль).

Язык RSL содержит абстрактные параметрические типы данных (алгебраические спецификации) и конкретные типы данных (модельно-ориентированные), подтипы, операции для задания последовательных и параллельных программ, предоставляет аппликативный и императивный стиль спецификации абстрактных программ, а также формальное конструирование программ в других ЯП и доказательства их правильности. Синтаксис этого языка близок к синтаксису языков C++ и Паскаль.

В RSL-языке имеются предопределенные абстрактные типы данных и конструкторы сложных типов данных, такие как произведение (product), множества (sets), списки (list), отображения (map), записи (record) и т.п. Далее рассмотрим как примеры некоторые конструкторы сложных типов данных.

1. Произведение типов – это упорядоченная конечная последовательность типов $T_1, T_2, ..., T_n$ произведения (product) $T_1 \times T_2 \times, ..., \times T_n$. Представитель типа имеет вид $(v_1, v_2, ..., v_n)$, где каждое v_i является значением типа T_i. Компонент произведения можно получить с помощью операции get и переслать с помощью set, т.е.

get component (i, d) = getvalue (i, d),

set component $(d, i,$ val$) = d \Rightarrow \nabla\ (I \rightarrow$ val).

Количество компонентов произведения d находится таким образом:

size $(d) = id\ \nabla$ (null (couter inc(counter))).

Конструктор произведения d_1 и d_2 строит произведение $d_1 \times \times d_2$ вида

product $(d_1, d_2) = id\ \nabla$ (size $(d_1) \Rightarrow$ couter 1) ∇ (null (couter 2) inc couter 2))).

Для каждого конкретного типа product $(T_1 \times T_2 \times, ..., \times T_n)$ можно построить конструктор значения этого типа из отдельных компонентов произведения таким образом:

make product (value $_1$, ..., value $_n$) = (value $_i \Rightarrow$ 1) $\nabla ... \nabla$ (value $_n \Rightarrow n$), где каждое значение value $_i$ имеет тип T_i, а результирующее значение – тип произведения $T_1 \times T_2 \times, ..., \times T_n$.

2. Списки типов – это последовательность значений одного типа списка list T, могут быть конечным списком типов T^k и не-

конечными списком типов T^n. В качестве структур данных типа списка может быть бинарное дерево, в котором есть голова (head) и сын (tail), следует за ним в списке и хвост. Операциями для списка является операция hd взятия первого элемента списка, т.е. головы, и операция *tl* − хвоста остальных элементов.

Функция

Caddr (I) = L ⇒ tail ⇒tail ⇒ Head, которая выбирает из списка I −элемент.

Индекс элемента помогает выбрать нужный элемент списка

Index(I, idx) = L (idx) =

while (¬ is null(idx))do((L ⇒ tail ⇒L) ∇dec(idx)) L ⇒ Head

Для определения количества элементов в списке выполняется функция

len (L) = (ld ∇ null (result))

while (L ⇒) do ((L ⇒ tail ⇒ tail ⇒L) ∇ inc (result))

result ⇒

Элемент списка находится так:

elem (L) = (ld ∇ empty (result))

while (L ⇒) do ((L ⇒ tail ⇒L) ∇

(result ↑ (L⇒ head ⇒) ⇒ elem) ⇒ result)

result ⇒.

Аналогично можно представить функции конкатенации, преобразование типов данных, добавления элемента в голову и хвост списка и др.

3. Отображение − это структура (map), которая ставит в соответствие значениям одного типа значение другого типа. Вместе с тем отображение является бинарным отношением декартова произведения двух множеств, как совокупности двухкомпонентных пар, в которой первый компонент arg содержит элементы аргументов отображения, а второй компонент res − соответствующие элементы значений этого отображения. В языке представлены разные допустимые операции над отображениями: наложение, объединение, композиция, срез и др. Среди этих видов отношений рассмотрим только композицию отображений (m_1, m_2).

(ld ∇ (compose (m_1, m_2) ⇒ m)) apply (m, elem)

apply to composition (m_1, m_2, elem) =

(ld ∇ (image (elem, m_1) ⇒ s) restrict (m_2, s) ⇒ map

(ld ∇ (map getname elem ⇒ name)) getvalue (name, map).

При этом используются функции:

Apply (m, elem) = image (elem, m) elem ⇒,

Apply (m, elem) = getvalue (elem, m) elem ⇒.

4. Запись − это совокупность именованных полей. Этот тип соответствует типу **record** в языке Паскаль и struct в языке C++. В языке RAISE определено два конструктора типа, record, shurt record, которые описываются в виде − type record id =

type mk_id (short _record id) ::=

$destr\_id_1$: $type\_expr_1 \leftrightarrow recon\_id$

. . .

$destr\_id_n$: $type\_expr_n \leftrightarrow recon\_id$.

Идентификатор mk_id является конструктором типа record, для которого даны деструкторы $destr\_id_n$, как функции получения значения компонентов записи.

5. Объединение − это конструктор Union**,** обеспечивающий объединение типов

type id = id_1 , id_2 ,..., id_n

при котором тип id получает одно из значений в списке элементов.

Конструктор типа имеет вид:

type id = id_from_ id_1 (id_to_ $id_{1:}$ id_1) |... | id_from_ id_n (id_to_ $idn_{:}$ id_n).

Операции над самим типом не определены в языке RAISE.

Рассмотренные формальные структуры данных языка RAISE, позволяет математически описывать и конструировать новые структуры данных в проектируемых программах. Их проще проверять на правильность методами верификации.

2.3.3. Описание базовых конструкций языка VDM

Язык VDM появился в венской лаборатории как метод для описания языка ПЛ/1, разработки трансляторов и систем со сложными структурами данных [44, 45]. Главная цель этого языка − спецификация программ и данных. С небольшой скоростью разработки и не высокой эффективностью, не всегда удобный в использовании, язык предоставляет математическую символику, которая легко воспринимается студентами последних курсов за 5−6 лекций.

В языке содержатся следующие элементы данных:

X − натуральные числа с нулем,

N − натуральные числа без нуля,

Int − целые числа,

Bool − булевы,

Qout − строки символов,

Token − знаки и специальные обозначения операций.

Функция в языке – это определение свойств структуры данных и операций над ними, аппликативно или императивно. В первом случае функция специфицируется через комбинацию других функций и базовых операций (через выражения), что соответствует синониму функциональный. Во втором случае значение определяется описанием алгоритма, что соответствует синониму алгоритмический. Например, спецификация функции вычисления минимального значения из двух значений имеет вид

min N_1 N_2 $\rightarrow$ N_3.

Пример алгоритмического описания значения этой функции

min (x, y) = if $x < y$ then x else y.

Объекты языка VDM. Все объекты строятся иерархически. Элементы данных, которыми оперируют функции, могут образовывать множества, деревья, последовательности, отображения, а также формировать новые более крупные объекты.

Множество может быть конечное и обозначаться X–set. При работе с множеством используются операции $\in$, $\subseteq$, $\cup$, $\cap$ и др. Язык включает в себя правила проверки правильности задания этих операций. Пример: $x \in A$ будет корректным только тогда, когда A является подмножеством из множества, которому принадлежит x.

Дистрибутивное объединение подмножеств покажем на примере:

union {(1, 2), (0, 2), (3, 1) } = (0, 1, 2, 3).

Списки (последовательности) – это цепочки элементов одинакового типа из множества X. Операция **len** задает длину списка, а **inds** – номер элемента списка.

Например, **inds** lst =($i \in X$ [$f \leq i \leq$ len]).

К операциям относятся: взятие первого (головы) элемента списка – **hd** и остатка (хвоста) после удаления первого элемента из списка – **tl**.

Например, **hd** (a, b, c, d) = (a), **tl** (a, b, c, d) = (b, c, d).

Могут использоваться также операция конкатенация (соединение двух списков) и операция дистрибутивной конкатенации.

Дерево – это конструкция **mk**, позволяющая объединять в комплекс объекты разной природы (последовательности, множества и отображения). В деревьях могут использоваться также конструктор для обозначения составных объектов и деструктор для именования констант, вносимых в ранее определенный составной объект.

Например, пусть t – переменная типа Время, значением которой является 10.30, тогда конструкция let **mk** – Время (h, m) = t **tin** определяет значение $h = 10$, а $m = 30$.

Отображение – это конструкция **map**, позволяющая создавать абстрактную таблицу из двух столбцов: ключей и значений. Все объекты таблицы принадлежат одному типу данных – множеству. Операция dom строит множество ключей, а rng – множество всех его значений. Кроме того, есть такие операции: исключить строку, слить две таблицы и др.

Приведенные конструкции используются для спецификации программы и, в частности, начального состояния с инвариантными свойствами этого состояния. Инвариантным свойством (inv) является функция, содержащая описание типов аргументов, результата и самой функции. При спецификации программы средствами VDM задаются пред- и постусловия, аксиомы и утверждения, необходимые для проведения доказательства правильности специфицированной программы. *Предусловие* – это предикат с операцией, к которой будет обращаться программа после получения ее начального состояния или в случае нерегулярной ситуации.

Утверждение задает описание операций проверки правильности программы в разных ее точках. Операторы программы изменяют ее состояние, а операции утверждений извлекают информацию из нее после изменении состояния (например, после операции работы с БД). На основе этой информации устанавливается правильность или неправильность выполнения операции. При возникновении непредвиденной ситуации, которая не предусматривались при описании программы в аксиомах, к программе должны предусматриваться соответствующие действия.

Постусловие – это предикат, который является истинным после выполнения предусловия, завершения текущей операции и выполнения инвариантных свойств программы.

Метод VDM ориентирован на пошаговую детализацию спецификации программ. На первом уровне строится грубая спецификация – модель программы в языке VDM, которая постепенно уточняется, пока не получится окончательный текст в ЯП. Разработка спецификации проводится по следующей схеме.

1. Определение терминов, которыми будет специфицироваться программа.

2. Описание понятий и объектов, для обозначения которых используется денотат, идентифицируемый с помощью некоторого имени (или фразы).

3. Описание инвариантных свойств программы.

4. Определение операций структуры программы (например, ввести объект, удалить и др.), которые изменяют ее состояние и сохраняют инвариантные свойства.

При переходе от одного шага детализации к другому модель программы детализируется и постепенно становится ближе к конечному описанию. Функции – это операции на некотором шаге, которые уточняются при детализации структуры программы и задания поведения модели.

Реальная разработка спецификации производится итерационно, на первом уровне она служит для проверки только свойств модели программы при заданных ограничениях и независимо от среды. Далее уточняется и расширяется спецификация и набор формальных утверждений. И так до тех пор, пока окончательно не будет завершен процесс спецификации программы и проведено доказательство.

Для демонстрации возможностей VDM языка рассмотрим задачу поиска («Поиск») имени компонента C в каталоге (catalR) репозитария компонентов и сравнения его с заданным именем в запросе пользователя. В случае совпадения имен проверяются параметры, из каталога извлекается код компонента для передачи пользователю.

Спецификация переменных программы «Поиск»

1. repoz :: = developers : dl → **Init –set**
2. catalR : cat → **Init –set**
3. role : **inst** → facet
4. facet :: = autors: Milk
5. title : N
6. user :: = developer : **Itn**
7. free : **Bool**,

где developers – сведения о разработчике компонента C;

facet – переменная, в которую посылается код компонента, выбранного из каталога catalR репозитария repoz при совпадении имен в каталоге и запросе;

role – переменная, в которой хранится текущий элемент из репозитария, найденный по фасете компонента с номером N для user;

autors: Milk – имя разработчика компонента;

free : Bool – переменная, которая используется для задания признака : компонент не найден или к нему никто не обращался.

Описание инвариантных свойств программы

8. **type inv** − repoz: repoz → Bool

9. **inv** − repoz (dev) =

10. **let** mk repoz (*cd*, *c*, role) = dev **in**

11. (∀ i ∈ dom *cd*)

12. (∀ *i* ∈ *cd* (*i*)

13. ((∃ *j* ∈ dom cd & *i* ∈ *c*) & ∃ a ∈ elems role (*i*), autors (*a* ∈ dom *cd*) &

free = false ⇔ developer = catalR & facet (N) = role

Проверяется список имен в каталоге, который содержит *N* элементов, представляющих собой элементы типа **set**, совпадающие с именем в запросе, после чего результат сохраняется в переменной role.

Доказательство инвариантных свойств программ должно проводиться автоматизированным способом с помощью специально созданных инструментальных средств поддержки VDM−языка.

Таким образом, можно сделать вывод, что системный анализ рассмотренных методов систематического и теоретического программирования показывает постоянное их развитие, совершенствование, пополнение новыми возможностям, а также появление новых ЯП и средств их поддержки. Классическим примером объединения является язык UML, благодаря чему ООП обогатился новыми возможностями, которые удовлетворили многих пользователей визуальным и наглядным моделированием ПС на основе моделей и разнообразных диаграмм. Алгебраическое и экспликативное программирование вобрало в себя не только возможности логического, функционального и процедурного программирования, но и специфические особенности абстрактного представления функций, программ и данных.

Большой интерес представляет объединенная концепция порождающего программирования, ориентированная на инженерию ПрО, и позволяющая генерировать компоненты и члены семейства ПС. Каждый вид программирования, используемый в среде этого программирования, может развиваться как самостоятельно, так и вместе с другими.

ЧАСТЬ

2

ПРОГРАММИРОВАНИЕ: методы интеграции, преобразования и изменения программ и данных

ГЛАВА 3

ОСНОВЫ ИНТЕГРАЦИИ ОБЪЕКТОВ, МЕТОДОВ И ТЕХНОЛОГИЙ

Под термином интеграция понимается метод объединения отдельных программных компонентов в конфигурацию, необходимую для обеспечения их взаимодействия в разных средах, а также пополнения методов программирования дополнительными свойствами и технологии проектирования ПС необходимыми процессами (например, управления конфигурацией, качеством и т.п.).

Средством интеграции являются: сборка, комплексирование, генерация, композиция и др. Такие средства, как сборка и комплексирование означали объединение компонентов в единое целое (в комплекс, систему, агрегат и т.п.) в объеме 100–200 тыс. команд на больших машинах (mainframes). Одновременно с этим интеграция включала в себя и данные (файлы, БД, интегрированные БД и др.).

Методы интеграции существенно зависят от возможностей новых компьютеров, локальных и глобальные сетей, а также от общесистемных программных средств. Интеграция приобрела новый смысл и заменила понятие единого комплекса новым понятием – интегрированная, распределенная система или среда, включающие в себя набор компонентов и системные средства обеспечения их взаимодействия (например, стандарт взаимодействия открытых систем OSI – Open Systems Interconnection, системы поддержки взаимодействия – CORCA, COM и др.).

Главным элементом интеграции программных объектов является интерфейс – связник объектов. В общем случае им является оператор вызова компонента, в котором задается список параметров передаваемых другому компоненту и получаемых от него результатов. Такими механизмами связи взаимодействующих компонентов и объектов являются:

– оператор обращения к процедуре в ЯП;

– RPC – язык (Remote Procedure Call) вызова удаленных процедур [1, 2];

– язык описания интерфейсов IDL [3, 4];

– вызов удаленных методов – RMI в языке JAVA [6, 7].

RPC-механизм – это оператор вызова удаленной процедуры и передачи параметров на языке RPC, связник между вызывающей программой клиента (stub-клиента) и вызываемой процедурой сервера (stub-сервера). Роль посредника выполняет специальный диспетчер.

Язык IDL в системе CORBA предназначен для описания интерфейса stub-клиент, как запроса к серверу на выполнение метода/функции, и интерфейса skeleton-сервера для удовлетворения запроса клиента. Роль посредника между клиентом и сервером выполняет брокер ORB в среде клиент–сервер.

Вызов удаленного метода RMI описывается в языке JAVA, аналогичен запросу и обеспечивает интерпретацию на виртуальной машине (virtual machine), вызываемой программы в ЯП (JAVA, C++ и др.), представленной скомпилированным byte-кодом.

Кроме механизмов интеграции компонентов и объектов, получили практическое развитие интеграционные аспекты в виде расширения методов и технологий программирования новыми дополнительными возможностями, отображающими потребности пользователей.

В данной главе рассмотрены теоретические и прикладные аспекты интеграции компонентов, современных методов и технологий программирования.

3.1. ТЕОРЕТИЧЕСКИЕ И ПРАКТИЧЕСКИЕ ОСНОВЫ МЕТОДА ИНТЕГРАЦИИ ОБЪЕКТОВ

Излагается систематизированный подход к проблеме интеграции элементарных компонентов, который определен в ряде работ с участием автора [8–11], он включает в себя базовые модели и методы интеграции модулей, записанных в разных ЯП.

Процесс интеграции компонентов в интегрированную структуру выполняется на основе базовых моделей взаимосвязи компонентов, модели управления и алгебраических систем преобразования типов данных взаимодействующих разноязыковых компонентов.

Модель интеграции компонентов определена на заданном множестве компонентов $K = \{K_i\}$, множестве интерфейсов I компонентов и данных, на которых определены фактические и формальные параметры объектов и система преобразования несовпадающих в них типов данных.

3.1.1. Модель интеграции объектов

Определим формально эту модель.

Пусть задано множество программных объектов $K = \{K_i\}_{i=1,N}$ в ЯП и множество данных $D = \{D\}_{ii=1,M}$, на котором определяются элементы множества K.

Каждый элемент $D_i = \{d_i^{\,j}\}_{j=1,t}$ определяется тройкой: именем $N_i^{\,j}$, типом данных $T_i^{\,j}$ и значением этого типа $V_i^{\,j}$.

Типы данных, которыми обмениваются пара компонентов K_i и K_j, могут быть эквивалентными, если они имеют одинаковую семантическую структуру и обработку, или обмениваться неэквивалентными типами $D_i^{\,j}$ и $D_i^{\,k}$ для K_i и K_j, которые требуют их преобразования с помощью функций, представленных такими отображениями:

$$FN_{i\,k} : N_i \rightarrow N_k ,$$
$$FT_{i\,k} : T_i \rightarrow T_k ,$$
$$FV_{i\,k} : V_i \rightarrow V_k .$$

Между множествами $D_i^{\,j}$ и D_i^k может не существовать взаимно однозначного соответствия, например, при условии, когда нескольким элементам множества D_i^j соответствует один элемент из множества D_i^k, и наоборот. Тогда строится отображение нескольких типов до одного общего типа с помощью функции конструирования: $C(d_i^{\,j1}, \ldots, d_i^{\,j\,k}) = d_i^{\,j}$, в которой $di^{\,j}$ является элементом из множества $D_i^{\,j}$.

Отображения FN_{ik}, FT_{ik}, FV_{ik} содержат одинаковое количество элементов. В задачу преобразования FN_{ik} входит упорядочение имен переменных в описании интегрированных программных компонентов. Отображение типов данных FT_{ik} базируется на множестве типов данных

$$T = (X, \Omega),$$

где X – множество значений, которые могут принимать переменные типа и элементы множества V, Ω – множество операций для преобразования типов.

Иными словами множество T будем рассматривать как алгебраическую систему преобразования типов данных из множества V.

Под преобразованием типа $T_i^j = (X_i^j, \Omega_i^j)$, содержащиеся во множестве T подмножества $T_i^k = (X_i^k, \Omega_i^k)$, понимается приведение значения переменной X_{ij} к значению X_{ik} множества X, при котором операции семантического преобразования Ω_i^j эквивалентны операциям из Ω_i^k. В общем случае приведение типа T_i^j к типу T_i^k может быть односторонним.

В случаях вызовов отдельных компонентов, которые обрабатывают одни и те же структуры данных, выполняется прямое и обратное их преобразования. Для этого необходимо, чтобы отображение между типами T_i^j и T_i^k было изоморфизмом, т.е. построенное преобразование между двумя типами данных должно отвечать изоморфному отображению соответствующих им двух алгебраических систем.

Таким образом, модель интегрированного комплекса из объектов для множества компонентов P_i с учетом типов данных и функций отображения имеет вид

$$MI = \{\{N_i, T_i, V_i\}, \{N_k, T_k, V_k\}, \{FN_{ik}, FT_{ik}, FV_{ik}\}\}.$$

Случай, когда нескольким типов из множества D_i соответствует несколько типов из D_k, рассматривается как удаленный обмен данными между компонентами и объектами распределенной среды, которые выполняются на разных компьютерах или процессорах этой среды.

Обмен данными между удаленно расположенными объектами осуществляется с помощью операторов вызова типа *CALL* – оператора обращения к процедуре или функции (F_1, F_2) в ЯП, расположенного в вызывающей программе (рис. 3.1).

Модель интеграции имеет такой общий вид: $M_1 = \{P, F, D\}$, где P – программный объект (на рисунке – P_1, P_2, P_3), который содержит обращение к функциям F и определен на множестве данных D.

В операторах вызова функций задаются фактические параметры, которые должны соответствовать порядку и типам соответствующих формальных параметров.

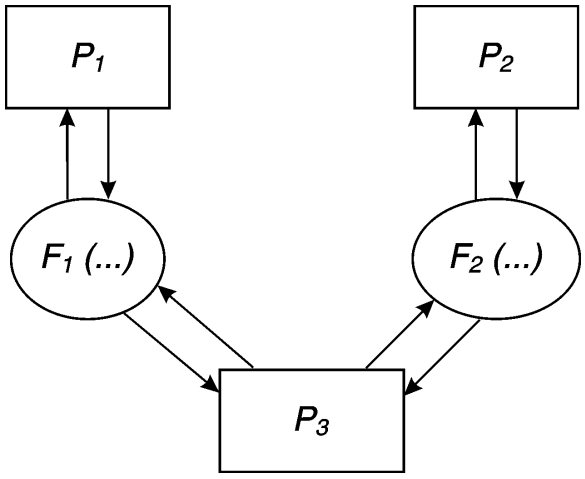

РИС. 3.1. Пример модели с вызовом функций F_1 и F_2

Если не соблюдался порядок расположения параметров или их количество, а также имело место несоответствие некоторых типов передаваемых параметров, традиционно перед оператором вызова непосредственно проводится необходимое преобразование типов передаваемых данных и после выполнения вызванной функции – обратное преобразование соответствующего параметра в теле программы P сразу после оператора вызова.

Общая задача обеспечения взаимосвязи пары компонентов, описание которых сделано на разных ЯП высокого уровня, состоит в построении взаимно однозначного соответствия между множеством фактических параметров $V = \{v_1, v_2, ..., v_{1k}\}$ – вызывающего компонента и множеством формальных параметров $F = \{f_1, f_2, ..., f_{kl}\}$ – вызываемого компонента.

Для множеств фактических и формальных параметров V и F взаимодействующих компонентов необходимые отображения параметров выполняются с помощью операций преобразования простых типов данных из множества Ω, а также специальных операций преобразования сложных типов данных (массивов, записей и др.) селектора S и конструктора C, обеспечивающих выбор необходимых простых операций из Ω, и конструирования из них передаваемых сложных типов данных с соответствующим преобразованием.

Множество операций конструирования имеет такой общий вид: $C = \cup C\alpha$, $C\alpha = \{Cm, Cz, Cf, ...\}$, где m – массив, z – запись, f – файл и т.п.

Для каждой пары взаимодействующих компонентов, описанных в разных ЯП, выполняется:

– построение отображения между множествами параметров фактических и формальных параметров;

– выбор необходимых операций селектора S и конструирования C сложных типов данных из простых;

– изоморфное преобразование типов данных для множества K.

Если отображение построить не удастся, то задача для таких типов данных пары объектов решается методами эвристики.

Теоретические аспекты преобразования простых и сложных типов данных на основе аппарата алгебраических систем описаны в следующей главе данной работы и в [9–11].

3.1.2. Модели взаимосвязи объектов в интегрированной среде

Для описания взаимосвязи объектов в интегрированной среде используются две модели – информационного объединения и управления объектами в динамике их выполнения.

Модели информационного объединения включает в себя средства преобразования типов данных компонентов в одном ЯП к типам данных в другом ЯП, а также операции снижения уровня структурирования данных для сложных и структурных типов данных.

Модель управления объектами в интегрированной среде включает в себя описание условий выполнения и выбора объектов. С каждым объектом среды связаны пред- и постусловия, определенные на множестве управляющих переменных. Дадим формальное представление этой модели.

Вначале во множестве данных D выделим два подмножества

$$\underset{i=1}{\overset{s}{D}} = (\cup D_i) \cup D_c ,$$

где D_c – обозначает множество управляющих данных, в котором содержатся переменные, определяющие условия выполнения компонентов, имена компонентов интегрированного комплекса и условия среды их выполнения. В состав этого множества могут также входить специальные функции управления данными и операциями вызова компонентов.

Для каждого объекта K_i из множества K определим предусловие $R_i \in R$ и постусловие $Q_i \in Q$, которые задаются на множестве управляющих переменных D_c. Предусловие R_i проверяется перед вызовом компонента K_i и определяет условие его вызова.

После этого привлекаются механизмы анализа и преобразования типов данных и их подготовка для запуска вызванного компонента. Постусловие Q_i проверяется после выполнения K_i как условие завершения основной программы и преобразования результата к требуемому виду вызвавшей компоненты, а также проверяется готовность к выбору нового компонента.

Таким образом, множества $R = \{R_i\}$ и $Q = \{Q_i\}_{i=1,s}$ определяют *модель управления* для выбора одного из объектов множества P и анализа множества R в целях нахождения первого правильного условия выполнения элемента из множества K. Если соответствующее условие из R истинно, то происходит вызов этого компонента, иначе, ищется новое условие из R и процесс повторяется, пока не произойдет выбор необходимого условия преобразования передаваемых параметров, выполнение выбранного из K компонента и отправка результата после анализа соответствующего условия Q_i.

3.1.3. Средства взаимосвязей объектов в сетевой среде

В интегрированной сетевой среде клиент–сервер обмен данными между объектами/компонентами проводится с помощью операторов удаленного вызова (RPC и RMI), в которых пересылаются параметры друг другу. При этом в рамках распределенных систем сформировалось несколько видов интерфейсных посредников (сервер файлов, портмейкер RPC, брокер ORB, VM–машина), поддерживающих соответствующие модели интеграции программных объектов в интегрированную программную структуру P [3–7, 29–34]. Рассмотрим их.

Посредник – сервер файлов. Интерфейс между взаимодействующими программными объектами в среде клиент–сервер выполняет *сервер файлов* (рис. 3.2).

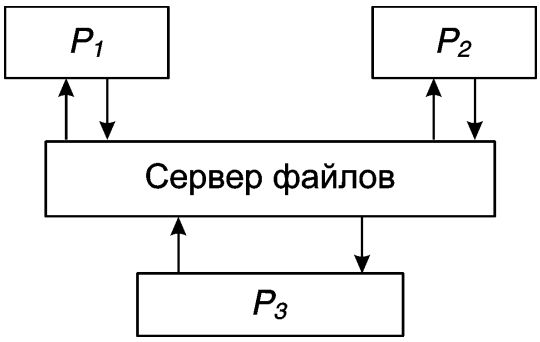

РИС. 3.2. Модель связи объектов через сервер файлов

Модель взаимосвязи объектов $M_2 = \{P \{ P_1, P_2, P_3\}, D\}$ через сервер файлов определяется на множестве объектов P и данных *D,* на котором определяются передаваемые серверу данные.

Связь объектов через stub. Средой поддержки взаимодействия компонентов через stub-клиента и stub-сервера (рис. 3.3) является ONC SUN, OSF DSE [1, 2] и CORBA.

В stub содержится описание в языке высокого и низкого уровней интерфейса взаимодействующих удаленных компонентов. В нем приводятся RPC-операторы, описанные на языке низкого уровня для прохождения оператора по сети (тип протокола, размер буфера данных и др.).

Модель связи $M_3 = \{P, I, RPC, D\}$ задает гибкую структуру, компоненты которой заданы на множестве языков I и соответствующих интерфейсах в языке RPC.

Описание интерфейса обеспечивает вызов удаленных процедур из библиотеки системы и передачу параметров в запросе транспортным протоколом (UDP, TCP/IP) сетевой среды ONC SUN [1, 2].

Связывающим звеном между вызываемым и вызывающим объектами является stub-клиента, который и посылает сообщение к stub-сервера для выполнения заданной удаленной процедуры. Соответственно stub-сервера инициирует выполнение этой процедуры и возвращает результат обратно через сеть клиенту.

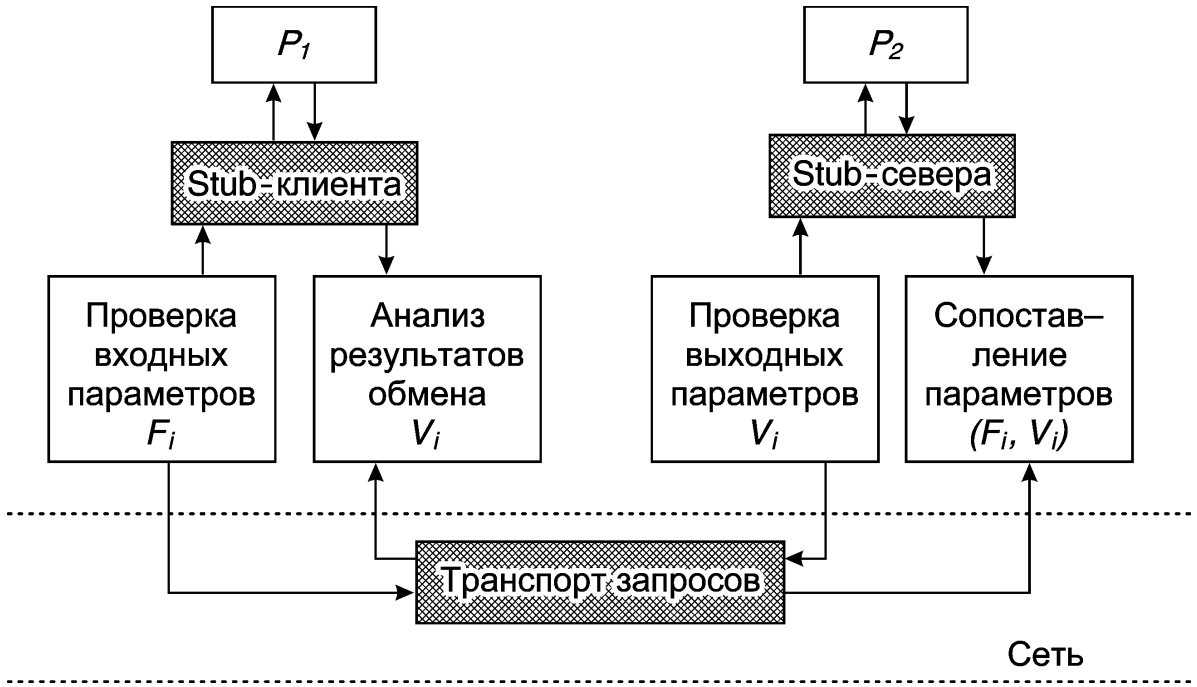

РИС. 3.3. Модель взаимосвязи объектов в сетевой среде

Входные F_i и выходные V_i параметры проверяются на типы данных, которые могли быть представлены на разных машинах и потребовать декодирования их в виду платформы сервера и обратного перекодирования к виду платформы клиента при отправке результата. Кроме того, могут возникнуть и другие проблемы, связанные с необходимостью преобразования типов данных параметров, например, целых к действительному типу и т.п.

RPC – механизм связи объектов в среде DCOM. Модель объектных компонентов DCOM задает связь распределенных объектов и документов в версии IDL, а система CORBA обеспечивает взаимодействие объектов через брокера ORB.

Система DCOM включает формализмы:

– механизм RPC – вызова удаленной процедуры и передачи ей данных;

– сетевой обмен данными;

– системы передачи данных, преобразования данных, кодирования и декодирования данных для разных архитектур машин.

Брокер ORB – интегратор объектов. Модель интеграции компонентов создаваемой системы в распределенной среде имеет следующий общий вид: $M_4 = \{ P, I, ORB, D, W \}$, где ORB – интегратор объектов в распределенной среде W.

Средой поддержки взаимодействия компонентов является система CORBA [4–7], в которой главную связующую роль между программными объектами выполняет брокер ORB и соответствующие stub/skeleton, описываемые в языке IDL (рис. 3.4).

Формализмами интеграции в системе CORBA являются:

– механизмы передачи запросов удаленным объектам через интерфейсные объекты stub и skeleton;

– сетевой обмен данными и их преобразование (кодирования и декодирования при расхождении архитектуры программных компонентов);

– системы преобразования типов данных объектов, которыми обмениваются между собой компоненты каждой пары в ЯП: С ⇔ Смолток, Смолток ⇔ Ада, Ада ⇔ Кобол, Кобол ⇔ Java и др.

Для задания взаимодействия объектов в системе CORBA используется язык IDL, который независим от языков описания объектов (С, С++, Паскаль и др.). Интерфейсы объектов в этом языке запоминаются в репозитории интерфейсов (Interface Repository), а реализации объектов – в репозитарии реализаций (Implementation Repository). Независимость интерфейсов от реализаций объектов позволяет их использовать в статике и в динамике.

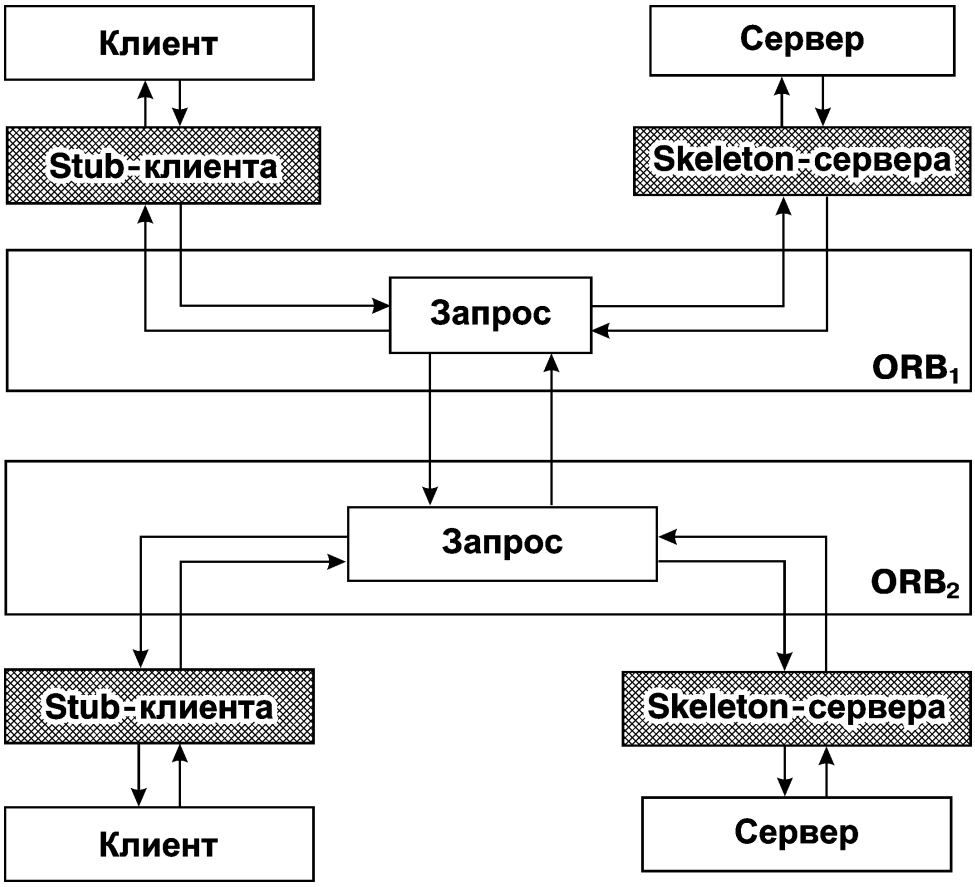

РИС. 3.4. Схема интеграции программ клиента и сервера двумя брокерами

Интерфейс клиента *(Client Interface)* состоит из трех интерфейсов:

– stub-интерфейса, содержащего описание внешне видимых параметров и операций объекта в языке IDL, генерируется в статическую часть программы клиента и хранится в репозитарии интерфейсов;

– интерфейса динамического вызова DII (Dynamic Invocation Interface) объекта, определяемого во время выполнения программы клиента посредством поиска описания интерфейса в репозитарии интерфейсов;

– интерфейса сервисов ORB (ORB Services Interface), содержащего набор сервисных функций, которые клиент запрашивает у сервера.

Сервис предоставляется следующими видами адаптеров:

– объектный адаптер (*Object Adapter*), который обеспечивает экземплярам объектов обращаться к сервисным функциям;

– базовый адаптер (*Basic Object Adapter*), который может обеспечивать выполнение объектов независимо от брокера;

– библиотечный адаптер (*Library Adapter*), обеспечивающий выполнение объектов, хранящихся в библиотеке объектов и вызываемых из прикладной программы клиента;

– адаптер БД (*Database Adapter*), обеспечивающий доступ к объектно-ориентированным БД.

Запросы реализуются с помощью шаблонов интеграции компонентов:

– Client class для вызова метода, который будет выполнен сервером.

– Stub class для конвертации метода на стороне клиента;

– ORB class управляет методами передачи данных и вызовами методов между процессами;

– Implementation class содержит деловую логику сервера, экземпляр этого класса – сервент – регистрируется в ORB и может использоваться клиентом для запуска другого процесса;

– Server class создает сервлет и ссылку, которую он записывает в стандартный выходной файл;

– Skeleton class конвертирует инициирующий метод с помощью ORB брокера в формат, который может прочитать экземпляр сервента.

– адаптер POA (Portable Object Adapter) обеспечивает доступ к разным ORB.

При выполнении сервисных функций брокер ORB использует следующие общесистемные средства CORBA:

– объектные сервисы, которые предназначены для логического моделирования и физического хранения объектов (например, их имен, событий, взаимодействий и т.п.), поддержки работы с объектами, обеспечения их существования и независимости от приложений, которые к ним обращаются;

– средства обслуживания облегчают построение приложений, которые будут функционировать в среде ORB, предоставляют унифицированную семантику общих компонентов и взаимодействие с другими объектами через брокер ORB и объектный интерфейс;

– объектные приложения – это приложения, разрабатываемые независимыми разработчиками на основе объектного подхода с использованием стандартного интерфейса и брокера ORB.

Формализация взаимосвязей компонентов в среде JAVA. Язык JAVA ориентирован на проектирование распределенных приложений, которые взаимодействуют между собою с помощью метода вызова RMI [6, 7]. Инструментом интеграции является virtual machine, которая работает с byte–кодами компонентов в ЯП.

Взаимодействие компонентов на языках JAVA и C++ осуществляется с помощью брокера ORB, рассмотренного выше. При этом преобразование типов данных этих двух языков относится к bite-кодам оттранслированных программ, которые интерпретируются на виртуальной машине.

Формализмы в системе JAVA включают в себя:

– механизм вызова удаленных методов RMI;

– сетевой обмен данными между удаленными компонентами;

– виртуальную машину для интерпретации кодов компонентов компилятора C++ в среде JAVA.

Альтернативой интерфейса компонентов является CGI (Common Gateway Interface), через который взаимодействует клиент с WEB-сервером. Для доступа к информации и формирования ответа клиенту используется сервлет, который описывается на языке JAVA, не зависит от платформы и использует библиотеку классов JAVA. Реализация сервлетов в JAVA осуществляется с помощью инструмента Servlet Development Kit.

3.1.4. Формальные основы языка описания интерфейсов объектов

Язык IDL позволяет описывать типы данных, интерфейсы объектов, которые вызываются для выполнения, а также предоставляет средства для описания параметров объектов, передаваемых в сообщении другим объектам. В нем описываются интерфейсные программы клиента и сервера (stub-клиент и skeleton-сервер), а сами программы описываются в ЯП (C++, JAVA, PASCAL и др.) [7, 13, 34].

Описание интерфейсов начинается с ключевого слова **interface**, за которым следует идентификатор (имя интерфейсной программы), образующие вместе заголовок, и тело, содержащее описание типов параметров для обращения к объекту. Пример описания заголовка описания интерфейса:

interface A { ... }
interface B { ... }
interface C: B,A { ... }.

Тело интерфейса содержит описание: типов данных (type_dcl), констант (const_dcl), исключительных ситуаций (except_dcl), атрибутов параметров (attr_dcl), операций (op_dcl).

Описание типов данных начинается ключевым словом typedef, за которым следует базовый или конструируемый тип и его иден-

тификатор. В качестве константы может быть некоторое значение типа данного или выражение, составленное из констант. Типы констант могут быть: integer, boolean, string, float, char и др.

Описание операций op_dcl включает в себя: атрибуты операции, тип результата, наименование операции интерфейса, список параметров (от нуля и более) и др.

Атрибуты параметров могут начинаться следующими служебными словами:

in – при отсылке параметра от клиента к серверу;

out – при отправке результатов от сервера к клиенту;

inout – при передаче параметров в оба направления (от клиента к серверу и от сервера к клиенту).

Описание интерфейса может наследоваться другим объектом, тогда такое описание интерфейса становится базовым, пример приведен ниже:

```
const long l=2
interface A {
void f (in float s [l]);
   }
interface B {
const long l=3
   }
interface C: B, A { }.
```

В нем интерфейс С использует интерфейс В и А и их типов данных, которые по отношению к С – глобальные. Имена операций могут использоваться во время выполнения интерфейсного посредника (Skeleton) для динамического вызова интерфейса, пример приведен ниже:

```
interface Vlist {
status add_item (
  in Identifier item_name,
  in type Code item type,
  in void * value,
  in long value Len,
  in Flags item flags
     );
status free ( );
status free memory( );
status get count (
  out long count);
     };
```

Описание модуля в языке IDL начинается с ключевого слова module, за которым следует имя модуля и описание его тела.

Типы данных. Язык IDL позволяет описывать типы данных, которые задают параметры, передаваемые от объекту к объекту, и подразделяются на базовые, сложные и конструируемые.

К базовым типам относятся фундаментальные типы данных:

16- и 32-битовые (короткие и длинные) со знаком и без знака двухкомпонентные целые;

32- и 64-битовые числа с плавающей запятой, что соответствует стандарту IEEE;

символьные (symbol);

8-битовый непрозрачный тип данных, обеспечивающий преобразование данных в момент пересылки между объектами;

булевы (true, false);

строка, которая состоит из массива одинаковых длин символов, допустимых во время выполнения;

перечисляемый тип, включающий в себя упорядоченную последовательность идентификаторов;

произвольный тип any, который представляет собой любой базовый или конструируемый тип данных;

логический тип boolean;

octet – специальный 8-разрядный базовый тип данных, который не требует перекодировки при переносе с одной платформы на другую.

Конструируемые сложные типы создаются из базовых типов и включают в себя:

– структуру (struct), состоящую из совокупности разнородных базовых элементов;

– объединение (union), содержащее дискриминатор, за которым располагается подходящий тип и значение;

– последовательность (sequence), представляющая собой массив, компоненты которого имеют переменную длину и одинаковый тип;

– массив (array), состоящий из компонентов фиксированной длины одинакового типа;

– интерфейс (interface), специфицирующий операции, которые клиент может послать в запросе.

Тип struct аналогичный языку C++, типы sequence и array – массивы содержат элементы одинакового типа переменной и фиксированной длины соответственно. Тип union семантически соответствует union в языке C++ и имеет дополнительно дескриптор вариантов. Каждому типу данных соответствует значение, которое

задается в запросе клиента или объекта, отправляющего ответ на запрос.

3.1.5. Подход к формальному описанию взаимодействия объектов

При анализе ПрО и проектировании распределенных систем (РС) с применением метода ООП и концепции распределения объектов по разным узлам сети создаются интерфейсы в виде stub/skeleton для обеспечения их взаимосвязи. В [8, 9] предложен формальный аппарат описания взаимодействия объектов, которые разделены на объекты двух различающихся семантикой родов.

Объекты 1-го рода – это объекты или компоненты, которые соответствуют прикладным функциям ПрО и обеспечивают решение задач конечного пользователя. К *объектам 2-го рода* относятся объекты-интерфейсы, включающие в себя операции вызова методов объектов 1-го рода и играющие роль посредников при взаимосвязи с объектами 1-го рода, удаленными друг от друга.

К операциям взаимодействия относятся интерфейсные операции (запросы), операция проекции (взаимодействие 1-го рода – рис. 3.5) и операция наследования (взаимодействие 2-го рода – рис. 3.6).

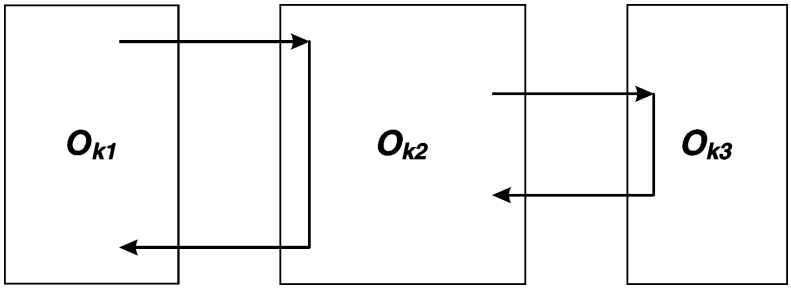

РИС. 3.5. Взаимодействие объектов 1-го рода

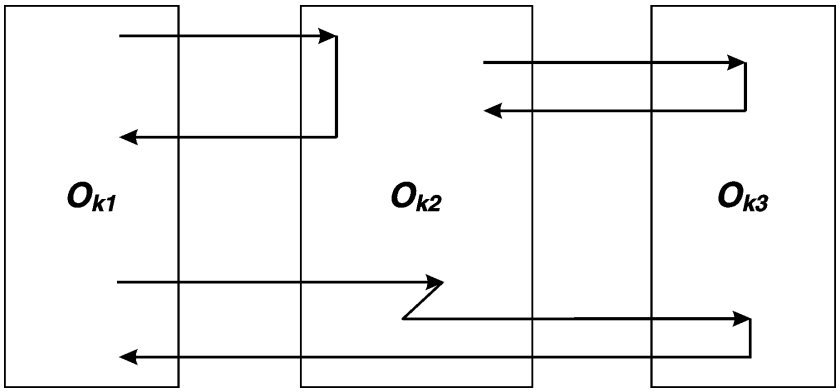

РИС. 3.6. Взаимодействие объектов 2-го рода

Операции и специфические для них интерфейсы определяются в соответствующем объекте 1-го рода и описываются в ЯП (например, C++, Паскаль).

Проблема взаимодействия объектов изучалась многими авторами, которые давали ей свои толкования. Так, Питер Вагнер выдвинул парадигму перехода от алгоритмов вычислений к взаимодействию [33]. По его мнению вычисление и взаимодействие – это две ортогональные концепции. Взаимодействие рассматривается как некоторое действие, а не вычисление. В качестве примера можно привести сообщение (это не алгоритм, а действие), ответ на которое определяется при выполнении последовательности операций (Op) некоторой локальной программы на состоянии разделенной (shared state, ss) памяти (рис. 3.7.). Операции Op1 и Op2 порождены интерфейсом взаимодействия и относятся к классу неалгоритмических.

Модель взаимодействия можно рассматривать как обобщение машины Тьюринга. В общем случае машина Тьюринга реализует модель вычислений алгоритмов. Обобщенную модель этой машины будем называть распределенной интерактивной машиной с моделью взаимодействия объектов и компонентов. Эта машина расширяет машину Тьюринга входными и выходными (input, output) действиями (actions), производящими динамическую генерацию и продвижение потенциально бесконечного потока.

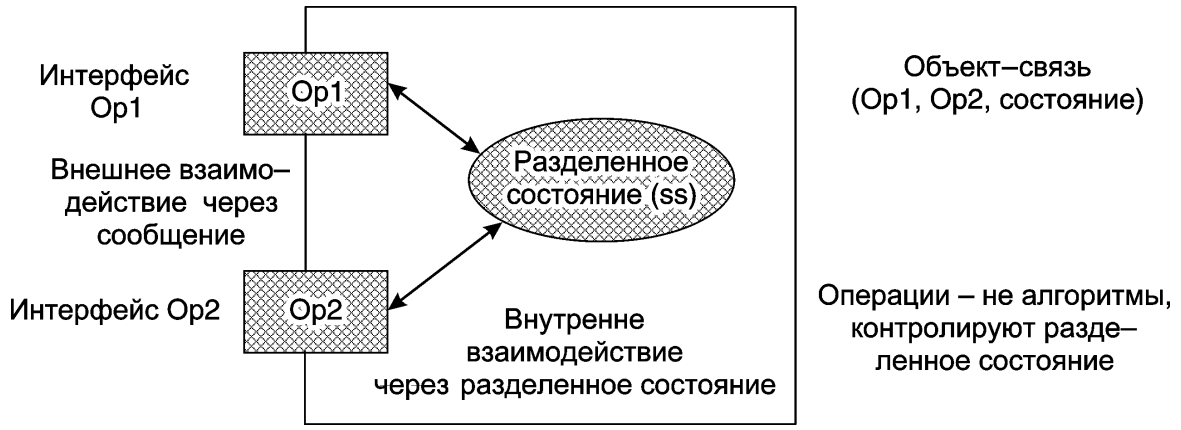

Рис. 3.7. Интерфейс взаимодействия операций

Распределенная интерактивная машина включает машину Тьюринга для проведения вычислений, т.е. ее модель допускает вычисления и обмен сообщениями в заданный интервал времени, в то время как в машине Тьюринга на вычисление алгоритмов фактор времени не влияет. Иное важное различие между

этими машинами состоит в том, что машине Тьюринга соответствует входная лента с конечным числом символов, а распределенной интерактивной машине – неограниченный входной поток (запросов, пакетов).

Таким образом, интерпретация модели взаимодействия с помощью распределенной интерактивной машины основывается на действиях и состояниях общей (разделенной) памяти взаимодействующих объектов.

Формальное описание объектов распределенной среды. Введем формальное определение объектов РС с использованием языка спецификаций интерфейсов IDL, а именно:

F – множество функций;

I – множество интерфейсов;

O – *м*ножество объектов-процессов;

$O_k \in O$, $in(O_k)$ – множество входных интерфейсов, которые объект использует, как клиент;

$O_k \in O, Out(O_k)$ – множество выходных интерфейсов, которые объект предоставляет, как сервер.

Взаимодействие объектов 1-го рода. Результатом взаимодействия двух объектов является объект, у которого множество входных интерфейсов совпадает со множеством выходных интерфейсов объекта-сервера, а множество выходных интерфейсов – с множеством выходных интерфейсов объекта-клиента:

$$O_k = \left(Out(O_k), In(O_k)\right),$$
$$O_k \cdot O_l = \left(Out(O_k), In(O_l)\right).$$

Не все объекты могут взаимодействовать друг с другом, поэтому наложим определенные условия.

Взаимодействие объектов $O_k \cdot O_l$ является корректным, если объект-сервер полностью обеспечивает сервис, необходимый объекту-клиенту $\forall I_m \in In(O_k) \Rightarrow \exists I_n \in Out(O_l) \wedge I_m = I_n$.

Интерфейс является входным для объекта, если объект получает сервис с помощью этого интерфейса, выходным, если с помощью этого интерфейса объект предоставляет сервис объектам.

Проекция объекта. Результатом проекции объекта на интерфейс будет объект, у которого множество входных интерфейсов содержит один элемент, а множество выходных интерфейсов содержит только те интерфейсы, которые необходимые для предоставления сервиса этому интерфейсу:

$$O_k = \left(Out(O_k), In(O_k)\right),$$

$$O_k[I_m] = if\left(I_m \in Out(O_k)\right) then\left(\{I_m\}, \{I_n : I_n \in In(O_k) \wedge exec(I_m, I_n)\}\right),$$

$$else(\{\ \}, \{\ \})$$

где *exec(I_m, I_n)* − предикат, который указывает на необходимость интерфейса I_n для выполнения I_m интерфейса.

Результатом проекции объекта на объект является проекция объекта на множестве входных интерфейсов объекта:

$$O_k[O_l] = O_k[In(O_l)].$$

Из определения операций проекции объекта и взаимодействия между объектами вытекает равенство:

$$O_k \cdot O_l = O_k \cdot O_l[O_k].$$

Операция проекции объекта имеет высший приоритет, чем операция взаимодействия (что задается конкатенацией), т. е.

$$(O_k \cdot O_l)[I_m] = O_k[I_m] \cdot O_l.$$

Операция наследования (взаимодействие 2-го рода). Наследование среди объектов РС происходит на уровне интерфейсов. Если объект наследует интерфейсы другого объекта ($O_k \leftarrow O_l$), то он предоставляет сервис всего множества исходных интерфейсов этого объекта: $O_k \leftarrow O_l \Rightarrow Out(O_k) \subseteq Out(O_l)$.

Таким образом, наследование объектом другого объекта — это объект, у которого множество выходных интерфейсов содержит все интерфейсы как 1-го, так и 2-го рода объекта, а множество входных интерфейсов содержит только те интерфейсы, которые необходимые для предоставления сервиса этих интерфейсов:

$$O_k \leftarrow O_l = \left(\begin{array}{l} Out(O_k) \cup Out(O_l), \\ \left\{ \begin{array}{l} I_m : \left(I_m \in In(O_k) \cup In(O_l)\right) \\ \wedge \exists I_n \in Out(O_k \leftarrow O_l) : exec(I_n, I_m) \end{array} \right\} \end{array} \right).$$

Объект, который наследуется, делегирует другому объекту свои интерфейсы.

Из определения операции наследования интерфейсов объектов вытекают свойства:

транзитивности − $\forall O_{1,2,3} \in O : O_1 \leftarrow O_2, O_2 \leftarrow O_3 \Rightarrow O_1 \leftarrow O_3$.

симметричности − $\forall O_k \in O \Rightarrow O_k \leftarrow O_k$.

Проекция над объектами, между которыми существует взаимодействие 2-го рода имеет вид $(O_1 \leftarrow O_2)[\ ] = O_1[\ ] \leftarrow O_2[\ ]$.

Описание распределенной среды. Язык спецификации среды функционирования РС, близкий к языку описания этого РС.

Последовательную среду можно описать с помощью описания программы, которая выполняется в этой среде. Поскольку основным различием между последовательной и распределенной средой является возможность параллельного выполнения объектов, то вводится операция параллельного выполнения РС: $P_{\text{о}} \,||\, P_{\text{г}}$.

Если в распределенной среде выполняется несколько РС, то распределенная среда описывается так: $P_{\text{о}} \,||\, ... \,||\, P_{\text{г}}$.

Операцию параллельного выполнения можно расширить и перенести на множество объектов, поскольку РС рассматривается, как объект. Но результат параллельного выполнения объектов – это не объект, а среда, в которой между объектами не возникает взаимодействия ни 1-го, ни 2-го рода.

Взаимодействие объекта и среды (взаимодействие 1-го рода*)*. Результатом взаимодействия объекта и среды будет объект, у которого множество входных интерфейсов совпадает с множеством входных интерфейсов объекта-сервера, а множество выходных интерфейсов является объединением множеств выходных интерфейсов объектов среды:

$$O_k \cdot \left(O_{l_1} \,||\, \ldots \,||\, O_{l_n}\right) = \left(Out\left(O_k\right), \bigcup_{j=1}^{n} In\left(O_{l_j}\right) \right).$$

В дальнейшем будем исходить из того, что множество входных интерфейсов при таком взаимодействии – пустое (это ограничение). Поскольку при взаимодействии объекта $O_k \cdot \left(O_{l_1} \,||\, \ldots \,||\, O_{l_n}\right)$ возникает недетерминированное взаимодействие (когда объекты среды взаимодействует с объектом-сервером), то

$$O_k \cdot \left(O_{l_1} \,||\, \ldots \,||\, O_{l_n}\right) = \left(Out\left(O_k\right), \{\ \}\right).$$

Взаимодействие объекта и среды $O_k \cdot \left(O_{l_1} \,||\, \ldots \,||\, O_{l_n}\right)$ является корректным, если выполняется условие: среда полностью обеспечивает сервис, необходимый объекту-клиенту

$$\forall I_m \in In\left(O_k\right) \Rightarrow \exists I_n \in \bigcup_{j=1}^{n} Out\left(O_{l_j}\right) \wedge I_m = I_n .$$

Проекция среды. Результатом проекции среды на интерфейс является среда, у которой все объекты являются проекциями объектов среды:

$$(O_k \| \ldots \| O_l)[I_m] = O_k[I_m] \| \ldots \| O_l[I_m]$$

Аналогично определяем проекцию среды на множество интерфейсов и объектов. Из определения операций проекции объекта и взаимодействия между объектами вытекает равенство

$$O_k(O_{l_1} \| \ldots \| O_{l_n}) = O_k(O_{l_1} \| \ldots \| O_{l_n})[O_k].$$

Операция наследования (взаимодействие 2-го рода). Наследование объектом интерфейсов РС − объект, который наследует интерфейсы всех объектов среды. Данная операция называется множественным наследованием. В дальнейшем будем использовать этот термин как синоним.

При наследовании интерфейсов среды возникает неопределенность, когда существуют два (или несколько) объекта, которые предоставляют сервис через один интерфейс. Для исключения неопределенности будем считать, что операция параллельного выполнения не симметричная, т.е.:

$$O_k \leftarrow (O_{l_1} \| O_{l_2}) \neq O_k \leftarrow (O_{l_2} \| O_{l_1}).$$

Описание классов РП. Расширим множество объектов, которые будут описываться языком IDL. Объект может получать сервис не только от одного объекта, но и от многих объектов. Все запросы относительно интерфейса и сервиса объект O_{k_1} получает только от объекта O_{k_2}, который в свою очередь получает сервисы от $O_{l_1} \ldots O_{l_n}$.

Иными словами, между объектами O_{k_1} и O_{k_2} существует взаимодействие 1-го рода, а объекты $O_{k_1} \ldots O_{k_n}$ выполняются параллельно, и определяют РС в виде $O_{k_1} \cdot O_{k_2} \cdot (O_{l_1} \| \ldots \| O_{l_n})$.

Запрос на интерфейс и сервис объект O_{k_1} получает от объекта O_{k_2}, если этот объект предоставляет сервис, по этому интерфейсу, в противном случае, от первого из объектов $O_{l_1} \ldots O_{l_n}$, который предоставляет этот сервис.

Описание объектов 2-го рода в язык IDL включает в себя описание типов данных параметров интерфейсов объектов 1-го рода и операции вызова метода для посылки запроса по сети другому объекту, удаленному от него. Эти объекты являются по-

средниками между клиентом и сервером (*stub* для клиента и *skeleton* для сервера). Их описания отображаются в те ЯП, в которых описаны соответствующие им объекты 1-го рода.

3.2. ИНТЕГРАЦИЯ МЕТОДОВ ПРОЕКТИРОВАНИЯ МОДЕЛЕЙ ПРЕДМЕТНЫХ ОБЛАСТЕЙ

К настоящему времени зафиксировано более пятидесяти методов проектирования ПО и более десяти из них ориентированы на детальное определение структур моделей ПрО и поддержку процесса проектирования ПС. Методы проектирования модели ПрО базируются на структурном и объектно-ориентированном подходах, что дает возможность с разных точек зрения определить основные сущности ПрО и их отношения, начиная с этапа анализа ПрО на ЖЦ.

Основным базисным понятием рассматриваемых ниже методов является объект. Это понятие и его свойства используются разными авторами не всегда одинаково, что создает некоторые проблемы при взаимном применении отдельных артефактов (например, нотаций, процессов, метрик качества и др.) из других методов.

3.2.1. Краткая характеристика методов объектно-ориентированного анализа

Среди многочисленных методов рассмотрим основные, широко используемые:

– метод объектно-ориентированного анализа систем (Object–Oriented system analysis by Shlaer and Mellor – OOAS–SM) [13] базируется на объектах реального мира, выделяет сущности, свойства и отношения объектов и на их основе определяет информационную модель, модель состояний объектов и процессов представления потоков данных (dataflow) между объектами;

– метод объектно-ориентированного анализа (Object–Oriented analysis by Coad and Yourdan – OOA–CY) [14] основан на анализе требований к ПрО, моделировании объектов с помощью понятия «сущность–связь», спецификации потоков данных и соответствующих им процессов в виде диаграмм и каталога (справочника) всех элементов модели и их атрибутов;

– метод структурного проектирования (Structured Design by Yourdan and Constantine – SD–YC) [15–17] позволяет представить с помощью структурных карт Джексона спецификации тре-

бований, структуры данных и программы преобразования входных данных в выходные;

– методология объектно-ориентированного анализа и проектирования (Object–Oriented analysis and design by Martin and Odell – OOAD–MO) [18] ориентирована на построение модели ПрО с помощью ER-моделирования [19] понятий и их отношений, а также на представление функций системы и способов организации данных с использованием структурных диаграмм, диаграмм «сущность–связь» и матрицы информационного управления;

– технология объектного моделирования (Object Modeling Technique by Rumbaugh – OMT–RU) [20, 21] включает в себя процессы (анализа, проектирования и реализации), методы их поддержки и нотации для представления моделей (объектной, динамической, функциональной и взаимодействия) на процессах ЖЦ;

– метод объектно-ориентированного программирования Буча [22] состоит из классов, суперклассов и операций наследования, полиморфизма и упрятывания информации;

– сценарии вариантов использования Джекобсона [23] задаются диаграммами, ориентированными на описание целей и задач ПрО и их трансформацию в систему взаимодействующих объектов;

– метод моделирования объектной модели CORBA [4, 5] основан на представлении объекта как некоторого предмета реального мира, имеющего характеристики, типы и операции обработки экземпляров класса или суперкласса, и включает в себя эталонную модель с набором сервисных системных компонентов (facilities) общего пользования для обеспечения функционирования объектных компонентов распределенных приложений;

– метод генерации в порождающем (generative) программировании использует базовые возможности ООП, компонентного и аспектного проектирования систем [24], базируется на компонентах многоразового использования, функциональных и нефункциональных требованиях, отображенных в модели характеристик;

– другие методы.

Приведенные основные методы проектирования имеют общие артефакты: объект (абстракция – abstaction, наследование – inheritance, инкапсуляция – encapsulation, агрегация – aggregation и др.), ER-модель (entity – relationship), модель потоков данных (dataflow model), интерфейс (interface) и другие, различают метод друг от друга. Существуют специфические особенности, отли-

чающие один метод от другого: терминологический аппарат, представление процессов и результатов разными диаграммами при моделировании и др.

Каждый отдельный метод имеет до десяти ключевых концепций, сотни проектных нотаций, руководящих принципов, критериев, процессов и еще большее число взаимосвязей между этими артефактами. И несмотря на это, часто такого набора средств у одного метода недостаточно для отображения некоторых особенностей процесса проектирования специфических ПрО, возникает необходимость обращаться к другим методам за недостающими артефактами.

Возникает задача объединения разных свойств методов для получения нового метода. Для этого необходимо определить вид интеграции, используя критерии и руководящие принципы, а затем провести анализ и провести дополнение отдельных черт из одного метода проектирования в другой.

Более подробно будут рассматриваться принципы интеграции методов Буча, Румбауха − OMT− и Джекобсона для создания метода моделирования UML (Unified Modeling Language) [25, 26], артефакты которого получены объединением указанных трех методов.

3.2.2 Принципы интеграции метода UML

Метод UML получен путем интеграции ранее разработанных базисных положений методов Буча, Румбауха и Джекобсона и предназначен для моделирования ПрО в терминах взаимодействия объектов [22, 23. 25, 26]. В него вошли:

1) парадигма объектного подхода Буча (классы объектов, экземпляры классов, операции наследования, полиморфизма и др.);

2) диаграммный метод use case Джекобсона, который обеспечивает представление вариантов или сценариев использования объектов ПрО и целей проектируемой системы;

3) метод Румбауха как средство проектирования основных элементов проектируемой системы на этапах ЖЦ (анализа требований, проектирования, реализации и др.).

Из метода Буча в UML входят классы и соответствующие им диаграммы. Классы обеспечивают отделение функций от описания данных с применением принципов инкапсуляции и наследования данных.

Диаграмма класса может отображать имена, соответствующие атрибуты классов и операции (методы) классов.

Атрибуты могут иметь такие значения:

– общий (public) означает, что операция класса может быть вызвана из любой части программы любым объектом;

– защищенный (protected) означает, что операция может быть вызвана лишь объектом того класса, в котором она определена;

– частный (private) означает, что операция может быть вызвана только объектом того класса, в котором она определена.

Операция определяет экземпляр класса, который может выполняться, если будет вызов со списком аргументов.

Между объектами задаются отношения:

– ассоциация означает зависимость между объектами разных классов, каждый из которых является равноправным членом, и принимают значение 0 – если ни одной связи, 1 – если одна связь и * – если много связей;

– агрегация означает время существования объекта-части или объекта-целого;

– наследование – это отношение обобщения и специализации;

– экземпляризация – зависимость между параметризированным абстрактным классом-шаблоном (template) и реальным классом, который инициирован путем определения параметров шаблона.

Из метода Джекобсона в UML входят диаграммы вариантов или сценарий представления целей проектируемой системы.

Сформированная цель системы трансформируется в совокупность вариантов использования системы для достижения этих целей в которые трансформируются в совокупность взаимодействующих объектов. Цепочка трансформаций <проблема → цели → сценарии → объекты> отражает степень концептуализации, достижения понимания проблемы путем последовательного снижения сложности ее частей.

Каждый сценарий инициируется с определенным пользователем, как носителем интересов, соответствующих определенной цели проекта системы. Абстракция роли личности пользователя представляется сценарием *актера,* обобщающим понятие действующего лица системы. В роли актера может выступать программная система, если она инициирует выполнение определенных работ в данном проекте.

Актер в модели системы представлен классом, а пользователь – экземпляром класса. Одно и тоже лицо может быть экземпляром нескольких актеров. Экземпляр актера запускает ряд операций в системе, соотнесенных со *сценарием*.

Сценарий определяет протекание событий в системе и имеет состояние и поведение. Если несколько запусков сценария системы имеют одинаковое поведение, то такой сценарий можно рассматривать как класс сценариев.

Моделирование системы по сценариям осуществляется с помощью актеров, запускающих эти сценарии. Совокупность сценариев определяет все возможные пути использования системы. Каждого актера обслуживает своя совокупность сценариев.

Модель сценариев системы сопровождается неформальным описанием каждого из сценариев, нотация такого описания включает в себя:

– название сценария на диаграммах;

– аннотации – краткое содержание в неформальном представлении;

– задание актера, запускающего сценарий;

– предусловия, которые определяют начальное состояние на момент запуска сценария;

– функции, которые реализуются при выполнении сценария;

– исключительные или нестандартные ситуации;

– постусловия, которые определяют конечное состояние сценария при его завершении.

Сценарии трансформируются в сценарий поведения системы, ввода данных и обработки возникающих чрезвычайных ситуаций.

Из метода Румубаха в UML входят диаграммы моделирования системы на этапах ЖЦ (анализа требований, проектирования, реализации и др.) и поведение, которое определяется обменом сообщений между объектами системы и обеспечением взаимодействия с помощью диаграмм:

– последовательности для упорядочивания взаимодействия объектов;

– сотрудничества для задания ролей во время взаимодействия объектов для достижения определенных целей;

– активности, задаваемой потоками управления при взаимодействии объектов;

– состояний, показывающих динамику изменения объектов под влиянием возникающих событий.

На заключительных этапах ЖЦ создаются диаграммы реализации, а именно:

– диаграмма компонентов, которая отображает структуру системы и связей между отдельными ее элементами;

– диаграмма размещения, которая определяет состав физических ресурсов системы, отношений между ними и операций запуска системы;

– диаграмма пакета, которая отображает функции подсистемы или системы в виде модели *конфигурации* системы, включающей в себя отдельные экземпляры модулей из состава вариантов использования, формируемых на разных этапах ЖЦ.

Характеристика языка UML. Данный язык предназначен для визуализации, спецификации, конструирования и документирования артефактов ПС с помощью рассмотренных стандартных диаграмм. Язык включает в себя общий словарь, механизмы расширения и графическую нотацию [25].

Словарь UML содержит три категории элементов: предметы (things), связи (relationships) и диаграммы (diagrams).

Предметы имеют следующие разновидности:

– структурные (use case, class, active class, interface, component, collaborations, node);

– поведенческие (interaction, state machine);

– групповые (package, model, subsystem, framework);

– аннотационные (note).

В состав диаграмм входят: class, use case, object, sequence, component, collaboration, state chart, activity, deployment. Они используются для графического изображения соответствующих им элементов системы.

Механизмы расширения предназначены для уточнения синтаксиса и семантики языка, используемых при проектировании программных проектов и систем или процессов разработки, и включают в себя:

– стереотипы расширения и уточнения семантики уже существующих элементов (things, relations);

– средние значения для определения новых свойств существующих элементов;

– констрейны для задания среднего значения общезначимых правил элементов и их свойств.

Графические нотации – это структуры, задающие наглядное представление ПрО с помощью объектов и их связей языка UML.

В результате интеграции нотаций независимо существующих методов проектирования получен объединенный метод моделирования UML для спецификации представления дизайна объектно-ориентированных приложений разнообразными нотациями, входящими в состав язык UML:

– метамодель для единообразного описания синтаксиса и семантики элементов UML, а также для унификации многих аспектов проектирования на объединенном (архитектурном, итерационном и др.) процессе проектирования объектных приложений;

– набор нотаций для структурного представления объектов проектирования с помощью диаграмм, объектов с параллельными и распределенными функциями;

– множество различных идиом для отображения конкретной модели ПрО средствами UML.

Методология проектирования на основе UML широко используется в практике программирования вместе с многочисленными его программными поддержками (Rational Rose, RUP и др.). Теоретические и практические результаты применения и развития UML отображены в многочисленных публикациях и монографиях.

Задача объединения методов проектирования не исчерпала себя, она продолжает развиваться и инициировать исследование и разработку новых способов их интеграции.

3.3. МОДЕЛЬ И ЯЗЫК ИНТЕГРАЦИИ МЕТОДОВ ПРОЕКТИРОВАНИЯ

Наряду с подходом, использованным при интеграции языка моделирования UML, в [27] рассматривается более общий подход к интеграции существующих методов проектирования, основанных на едином формализованном базисе – *каркасе* (framework), включающие в себя две модели: компонентную модель методов проектирования и модель артефактов.

Методы проектирования имеют общие артефакты: объект (абстракция, наследования, инкапсуляция и др.), ER-модель, модель потоков данных, интерфейс и др.

Артефакты – это способ представления метода, отличающегося от другого метода терминологией, формой представления процессов, видами диаграмм, моделями, нотациями и др.

Каждый метод характеризуется своими ключевыми концепциями, нотациями, принципами, процессами и способами взаимосвязи между ними. При формировании метода для конкретных целей, поучается, что набора имеющихся средств в нем недостаточно для полного отображения процесса проектирования некоторой ПрО, поэтому возникает необходимость обращения к другим методам за отсутствующими артефактами для пополнения этого метода.

3.3.1. Компонентная модель интеграции методов

Ниже рассматривается общий подход к интеграции методов проектирования, которые базируется на модели *каркаса* [27]. В состав каркаса входят методы проектирования из компонентов верхнего и нижнего уровней (рис. 3.8).

В состав *компонентов верхнего уровня* входят:

– модель артефактов со структурами и паттернами для спецификации и организации артефактов проектирования (например, объектная модель в методе OMT);

– свойства характеристик артефактов проекта, представленных в модели артефактов (например, артефакт – понимаемость);

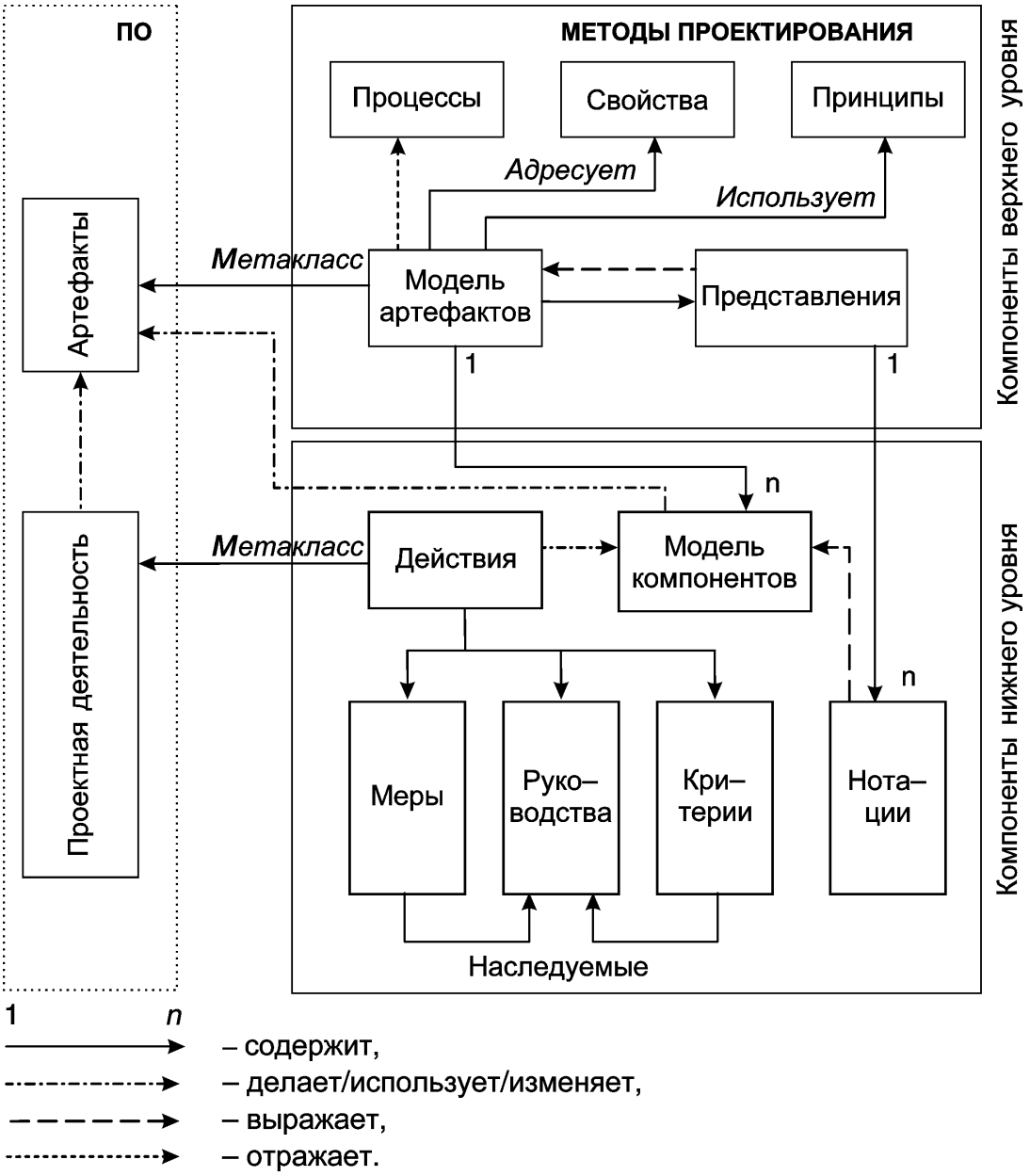

РИС. 3.8. Структура каркаса общего метода проектирования

– принципы, содержащие концепты для производства артефактов с учетом требований заказчика (например, принцип упрятывания информации), а также представлений артефактов, таких как диаграммы потока данных;

– процессы, задающие множество шагов по разработке ПО с использованием модели артефактов.

Фактически интеграция методов включает в себя виды интеграции, выполняющиеся на различных уровнях.

Основу видов интеграции методов составляет базовая модель потоков данных (dataflow). Каждый вид интеграции аналогично включает в себя:

модели для определения того, как сущности в модели потока данных должны относиться к методу;

принципы для гарантии того, что модели потоков данных были совместимы с принципами существующего метода;

процессы интеграции диаграмм потоков данных в существующем методе;

представления интеграции моделируемого представления потока данных с существующим методом.

Каркас использует композиционную модель метода и множество процедур анализа структур методов и их описаний в модели. Она в свою очередь использует структурную модель, модель потоков данных, иерархическую модель и модель «сущность–связь» ER.

Компоненты нижнего уровня определяют типовые проектные действия (activities), производимые операциями метода на основе артефактов компонентной модели – метакласса этих действий. К ним относятся:

– модель компонентов, являющаяся составной частью компонентой модели артефактов, служит для определения и спецификации сущностей (например, класс в OMT);

– критерии и руководства, являющиеся правилами для принятия решений по управлению разработкой ПО;

– меры, определяющие аспект измерения согласно стандарту оценки качества артефактов модели;

– нотации, являющиеся частью используемых представлений, которые выражаются через артефакты модели компонентов (например, диаграмма в UML);

– действия, определяющие шаги разработки новых артефактов для модели артефактов, и которые сами могут образовывать, использовать и изменять артефакты.

Компоненты нижнего уровня метода проектирования могут наследоваться методами верхнего уровня.

При проведении интеграции методов на основе каркаса применяют два подхода, основанные на функции и качеством управляемой интеграции (рис. 3.9).

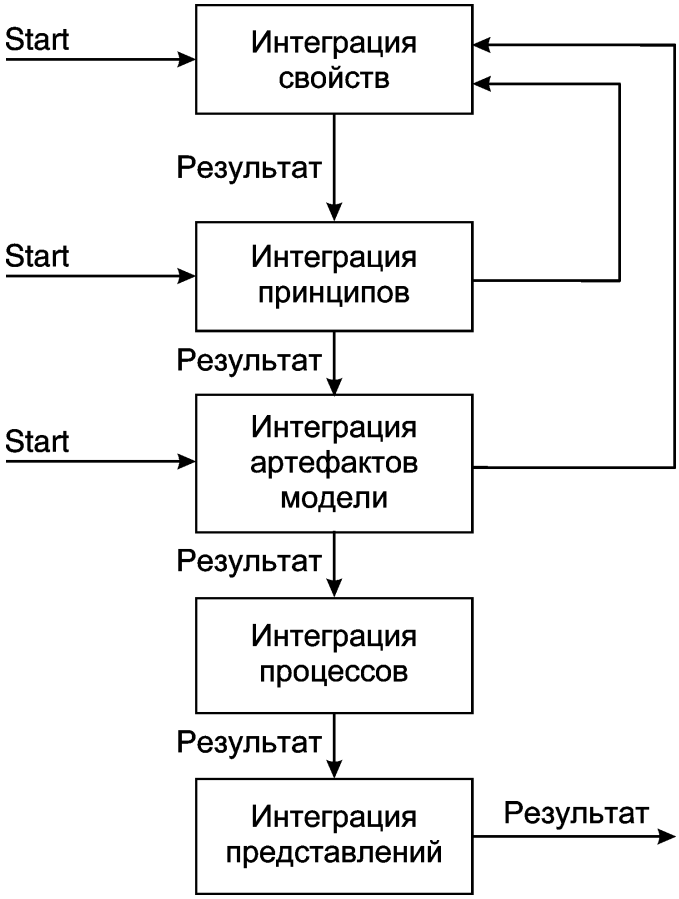

РИС. 3.9. Структура функцией управления интеграции для компонент верхнего уровня

Добавление новой функции для моделирования задач ПрО и поддержки разработки ПО на этапах ЖЦ конкретным методом (например, объединение структурного и объектно-ориентированного методов) образует цель первого подхода.

Модель артефактов пополняется новым артефактом из компонентной модели вместе с ассоциированными множествами элементов управления качеством. Как видно из рис. 3.9, интеграция, в зависимости от задаваемой функции, может начинаться (start) со свойства, принципа или модели артефактов. Результат каждого вида интеграции может интегрироваться и на верхнем уровне путем добавления функция, которая не меняет компонентов модели.

Качеством управляемая интеграция добавляет к объединяемому методу проектирования не новые качественные функции, а компоненты улучшения качества (например, меру повышения тестируемости). Эта интеграция может сделать существующий метод более понимаемым, эффективным и качественным. Артефакты повышения качества берутся из компонентной модели и добавляются в модель артефактов метода. В результате повышается качество применения конкретного метода.

Добавление нового функционального компонента или компонента управления качеством может привести к повторной интеграции принципов, проектных представлений и процессов рассматриваемого конкретного метода.

Таким образом, обе вида интеграции расширяют возможности метода проектирования, а не ПО.

Формальный аспект интеграции. Опыт работы по обеспечению интерфейса ЯП [8–11] показывает, что системная работа по интеграции языковых средств предполагает формальные подходы для отображения структур данных одного языка метода проектирования в структуры данных языка другого метода.

При интеграции языковых средств методов проектирования возникают аналогичные проблемы отображения одних нотаций в другие, как и в случае стандартного описания интерфейса [12].

В виду отличий в представлениях синтаксиса и семантики нотаций методов проектирования требуется провести системную работу по сопоставительному анализу языковых средств каждого из рассматриваемых методов проектирования для построения алгебраических систем отображения релевантных структур нотаций каждой пары интегрируемых методов, а также разработать необходимые функции преобразования для нерелевантных семантически неодинаковых элементов нотаций.

Каждому множеству нотаций определенного метода можно поставить в соответствие алгебраическую систему, элементами которой являются типы компонентов модели и множество операций над ними. Для интеграции нотаций разных методов в объединенное множество требуется обеспечить изоморфное отображение одной алгебраической системы в другую путем преобразования релевантных структур нотаций.

3.3.2. Концепция языка описания методов проектирования

Современные проблемно-ориентированные методы разработки ПС накапливаются в библиотеках для того, чтобы ими можно было воспользоваться при разработке новых ПС или ис-

пользовать возможности этих методов для создания других методов. Для этого предлагается подход, который связывает задачи методов, базу знаний и библиотеку. Такое связывание обеспечивает не только поиск метода, но и присоединение к нему программных модулей отображения. Способом связывания методов является *язык описания методов* (ЯОМ) или MDL − (Method Definition Language) [20, 27].

При разработке конкретного метода необходимо обратиться к библиотеке методов и найти пригодный метод, это экономит усилия на его разработку, за счет его формального описания и тестирования.

Для занесения метода в библиотеку выполняются требования:

1) методы в библиотеке должны быть индексированы, что помогает осуществить поиск и получение аннотированной информации;

2) методы должны быть специфицированы в ЯОМ, т.е. задается аннотация и *tt* отображение в базе знаний в неформальном или формальном виде.

Средства описания методов. Существующие на данный момент средства описания методов не удовлетворяют приведенному требованию. Например, каждый метод в системе CORBA [4−7] включает в себя спецификацию языка IDL, в частности описание характеристик внешнего интерфейса метода: типы и структура входов и выходов. Однако в ЯОМ отсутствует информация о семантике метода.

Средства языка ориентированы на отображение в базе знаний, а также на задачи поиска и получения методов из библиотеки методов.

Аспекты языка описания метода**:**

1) формализация языка ограничивается средствами формальной проверки (верификации) корректности отображения метода в базу знаний;

2) управление логикой метода рассматривается как «черный ящик»;

3) ориентация ЯОМ на отображение метода в базу знаний.

Требования к ЯОМ. Конструирование отображений методов в базе знаний еще недостаточно исследовано. В то же время поддержку процесса конструирования отображений можно возложить на саму библиотеку методов, используя для этой цели ЯОМ. При конструировании таких отображений предусматривается автомати-

ческая проверка того, что отображение сохраняет семантику знаний о переводе требований проблемно-ориентированного метода к виду формальных спецификаций входов и выходов в ЯОМ.

Выбор методов из библиотеки может быть выполнен путем описания знаний, а решение задачи отображения осуществляется на уровне описания входов и выходов в символьном виде.

3.4. ОСНОВНЫЕ ПОЛОЖЕНИЯ СБОРКИ ТЕХНОЛОГИЙ ПРОГРАММИРОВАНИЯ

Дальнейшим развитием идеи объединения методов является метатехнология, как более высокий уровень проектирования технологий программирования путем интеграции методов, средств и нотаций, способных обеспечить реализацию конкретной ПрО. Данный подход соответствует современному процессному подходу, зафиксированному в новой серии стандартов ISO/IEC в области программной инженерии.

Предшественником процессного подхода к созданию ПС была Комплексная программа научно-технического прогресса стран-членов СЭВ до 2000 года, в рамках которой программная продукция впервые была отнесена к продукции производственно-технического назначения и сформулирована проблема «Развитие технологии разработка и промышленного производства программных средств вычислительной техники».

Участвуя непосредственно в реализации этой программы, нами разработаны основы *технологической подготовки разработки* (ТПР) программных изделий, как этапа, который предшествует непосредственному созданию ПС, и предназначен для построения технологического процесса разработки программных продуктов в организации-разработчике ПС и адаптации его для каждого конкретного программного проекта [28].

ТПР включал в себя набор документов (методик и стандартов предприятия), в которых регламентировались технологические процессы разработки разных видов ПС, в частности СОД, устанавливались требования к структуре и содержанию программно-технологических документов – карт технологических линий (ТЛ) и процессов (ТП), технологических маршрутов, форм и т.п. В задачи ТПР входило системное обследование ПрО и формализованное определение специализированных технологий программирования для реализации задач в АСУ, СОД, АСНИ и др.

После распада СССР эти работы были прекращены, а «процессный подход» представлен в международном стандарте

ISO/IEC 12207−2002 [29], соответствующем гармонизованном национальном стандарте ДСТУ 3918−99 и фактически получен новый виток разработки базового технологического процесса, ориентированного на управление качеством ПС [30, 31].

3.4.1. Основы технологической подготовки разработки ПС

ТПР включает в себя следующие основные элементы:

− объект разработки (его начальное, промежуточные и конечное состояния);

− методы программирования, средства и инструменты, обеспечивающие изменение состояний объекта;

− модели ТП и ТЛ;

− инженерные методы управления процессом разработки качественных программ по ТЛ [28].

Для построения новых ТП и ТЛ в рамках ТПР определены классы объектов и язык спецификации ТЛ. Классы объектов ПрО разделяются на понятийные, технологические и инструментальные.

В класс понятийных объектов ПрО входят модели данных, задач и программ. Все задачи делятся по функциональному признаку и определяются на подмножестве данных и функций. Для типовых задач определены модели типовых и специализированных ТЛ реализации функций ПрО (например, ввода и вывода отчетных документов СОД и др.).

К классу технологических объектов отнесены модели состояний объектов и процессов их разработки. В их состав входят модели:

− для фиксации проектных решений и промежуточных результатов проектирования ПС на каждом ТП линии;

− ТП и ТЛ для формализации деятельности исполнителей при исполнении их операций, связанных с преобразованием результатов предшествующего процесса к последующему, и оценивания этих результатов;

− качества программных объектов для управления качеством ПС на всех операциях ТП и ТЛ;

− эксплуатационных документов для формирования документации в процессе разработки ПС на промежуточных операциях ЖЦ;

− производственной системы, представляющая собой логически связанную совокупность ТП и ТЛ, набор методик и рекомендаций по управлению разработкой, изготовлением и сопровождением ПС.

Определены процессы ТЛ и основные этапы жизни отдельных объектов ПС, каждый из которых отображается в одном или нескольких ТП линии. Каждый ТП – это вероятностный автомат, переводящий объект разработки из одного состояния в другое, которому соответствует промежуточный результат проектирования ПС. Заключительным состоянием является готовый программный продукт.

К классу инструментальных объектов относятся методы, средства и инструменты, из которых подбираются более подходящие для создаваемой ТЛ и осуществляющие преобразование состояний объекта разработки на каждом ТП. Разработан язык спецификации ТЛ, являющийся первой попыткой формализованного описания ТЛ. Элементами этого языка являются ТП и технологические операции разработки объекта, документирования и контроля качества.

Каждая операция разработки включает в себя: состояние объекта, входные и выходные данные, метод и средство разработки.

Описание операции контроля качества состоит из названия показателя качества, контролируемого оценочного элемента, метрики и метода оценки. Операция документирования включает в себя указание вида модели документа и набор точек для внесения соответствующих текстов в формируемую документацию. Формой спецификации ТЛ являются карты в табличнографическом виде.

Данный метод апробирован при создании конкретных ТЛ программирования прикладных программ, работающих с базами данных в классе СОД.

3.4.2. Инфраструктура линейки программных продуктов

Наряду с ТПР идея технологичности (управление, оценка) процесса разработки продуктов ПС постоянно обсуждалась и развивалась. Так, на базе использования готовых к употреблению компонентов (ПИК) возникло новое направление – инженерия приложений и инженерия ПрО [24, 30, 32]. Инженерия приложений ориентирована на создание готового продукта для конкретного приложения. Инженерия ПрО ориентирована на создание семейства программных продуктов, отдельные члены этого семейства могут быть компонентами многоразового применения, в том числе они включают в себя и продукты инженерии приложения.

В рамках инженерии ПрО формулируются требования к семейству систем, причем инвариантные свойства, присущие каждому из членов семьи, отделены от свойств, специфических для отдельных представителей семейства.

Свойства классифицируются как обязательные, необязательные и альтернативные. К обязательным относятся такие свойства, которые присутствуют обязательно в каждом из членов семейства систем, а реализация каждого отдельного представителя системы может иметь некоторые отличия. К альтернативным свойствам относятся свойства, которые отображают особенности выбора представителей семейства, как многократно используемых. Необязательными являются свойства, которые у некоторых представителей семейства могут отсутствовать.

Создание семейства программных продуктов в инженерии ПрО требует наличия определенной компонентной платформы, включающей в себя совокупность готовых компонентов и инструментов генерации отдельных членов семейства ПС. Такими готовыми для применения могут быть не только компоненты, ПИК и программы семейства, но и спецификации требований и архитектуры системы.

Американский институт программной инженерии SEI [1] предложил ***линию продуктов*** или семейства продуктов, как множество программ, ПИК и ПС, которых объединяет общий управляемый набор показателей качества, удовлетворяющих потребностям некоторого рынка программной продукции. Множество компонентов и систем образуют семейство продуктов, если они имеют общие свойства, а каждый член семейства имеет свои индивидуальные особенности.

Понятие линейки программных продуктов (Framework for Product Line Practice) сформировалось, как поддержка инженерии ПрО, в задачу которой входит построение разных видов программных продуктов с применением соответствующих подходов и методов.

При этом исследуются рынок и потребности покупателей, строится производственный план, процессы и определяется организационная структура. На основе анализа потребностей рынка строится технология линии продукта, в которую включаются методы разработки, тестирования и оценки процессов и продуктов.

Инфраструктура разработки линейки продуктов, в которую входят различные методы и средства, необходимые для построения и эксплуатации продуктов приведена на рис. 3.10 [24, 36].

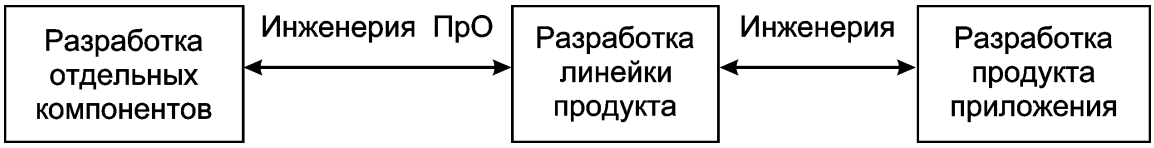

Рис. 3.10. Инфраструктура построения линеек продуктов

При построении конкретной линейки разработки программного продукта для некоторого представителя (члена) семейства домена определяются:

– ограничения, свойственные продуктам линейки;

– образцы и каркасы, которые могут использоваться на линии;

– производственные стратегии и методы;

– набор средств и инструментов для разработки продукта на линии.

На основе этих данных определяются область действия линейки и набор базовых средств, строится план создания продукта на линейке, который учитывает сроки, стоимость и требования к управлению производством продукта путем:

– контроля плана работ и отслеживания хода построения продукта;

– выявления рисков и управления ими в процессе деятельности на процессе проектирования члена семейства;

– прогнозирования стоимостных и других ресурсов;

– применения методов управления конфигурацией продукта;

– измерения и оценки качества продукта.

Данное направление развивается за счет использования готовых компонентов, накапливаемых в репозитариях, что улучшает процесс выбора готовых ПИК на производственной линейке.

Таким образом, рассмотрены модели, методы и подходы к интеграции, используемые в практике программирования. Проведена их систематизация и определены три направления интеграции: интеграция компонентов для обеспечения их взаимодействия в современных средах, интеграция методов Буча, Раумбаха и Джекобсона в общий метод моделирования UML, интеграция методов программирования в новые технологии для реализации задач разных предметных областей. Дан анализ двух технологических подходов – технологическая подготовка разработки ПС и инфраструктура построения линеек продуктов.

ГЛАВА 4

МЕТОДЫ ВЗАИМОДЕЙСТВИЯ И УСТРАНЕНИЯ НЕОДНОРОДНОСТИ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И БАЗ ДАННЫХ

При работе клиента с разными программными компонентами в распределенной среде клиент–сервер возникают проблемы взаимодействия компонентов при передаче и обработке данных. Клиент обращается к серверу с запросом, в котором передаются фактические параметры для выполнения программы, расположенной на компьютере этой среды с различающейся архитектурой ОС, ЯП. Сервер принимает запрос и решает вопрос о передаче параметров для обеспечения выполнения программы.

Взаимодействие между клиентом и сервером осуществляют специальные промежуточные среды: ONC RPC [1], DCE RPC [2], OMG CORBA [3–5], COM [6, 7] и др. Каждая из этих сред решает вопросы неоднородности данных и ЯП с помощью стандартных правил кодирования (декодирования) языков форматирования XDR, NDR, CRD и др., а также межсетевых средств обеспечения взаимодействия ЯП.

Если программы расположены на разных платформах и взаимодействуют друг с другом, то передача данных между ними в общем случае предполагает стандартное преобразование форматов данных одной платформы в представление другой платформы, а также устранение неоднородности представления типов и структур данных взаимодействующих компонентов, записанных в разных ЯП и функционирующих не только на разных платформах компьютеров, а и в разных промежуточных средах.

Выделим три основных направления в решения проблемы неоднородности данных:

1) разные форматы представления данных и их передача программам, которые расположены в разных средах или на разных платформах;

2) различие в представлении типов данных современных ЯП, неоднородность их реализации соответствующими системами программирования, разные способы преобразования типов данных в одном ЯП к типам данных другого языка;

3) форма и структура представления данных в разных моделях БД СУБД и их неоднородность при замене одной БД другой [5, 9, 10].

Первое направление – следствие различных видов кодировки данных и их отображения на платформы компьютеров с разной структурой памяти. Это потребовало преобразования форматов данных передающего и принимающего их компьютеров. Процедура преобразования данных из одного формата в другой получила название маршаллинг (marshalling) данных, она включает в себя линеризацию сложных структур данных с учетом порядка расположения байтов и стратегию выравнивания данных до установленных границ на каждой платформе. Например, в системе CORBA в этих целях используется стандарт формата представления данных – CDR.

При втором направлении проблема неоднородности связана с разнообразными средствами описания и реализации типов данных в системах программирования ЯП, с обменом данных между разноязыковыми программами [21], а также через механизмы удаленного вызова RPC [1,2], IDL запросов и межсетевых протоколов передачи данных между общесистемными средами CORBA, DCOM, JAVA [11–13], а также с рекомендациями стандарта (ISO/IEC 11404–96) [8] по обеспечению независимости типов данных от средств ЯП.

В случае **третьего направления** проблема преобразования данных возникает в связи с заменой старой БД новой БД, которые основываются на различающихся моделях данных (иерархические, сетевые, реляционные), функционирует в разных СУБД и обмениваются данными, например, через DBF-файлы, транзитные файлы и др.

Предметом рассмотрения указанных трех направлений являются теоретические и прикладные методы представления данных, устранения их неоднородности путем преобразования типов данных ЯП, а также преобразования данных, изменяемых БД, приведенных в работах [1–14, 21].

4.1. МЕТОДЫ ПРЕДСТАВЛЕНИЯ И ПРЕОБРАЗОВАНИЯ ДАННЫХ

Под *парадигмой преобразования* данных будем понимать формализмы описания базовых типов и структур данных в современных ЯП, методы преобразования форматов данных (кодирование и декодирование) на одном компьютере к соответствующему пред-

ставлению на другом компьютере, механизмы передачи данных по сети [1– 5], методы устранения различий в описании типов данных в разных ЯП, а также методы представления и преобразования данных для СУБД, связанные с заменой БД.

К *средствам представления* данных и их форматов относятся:

– стандарты кодировки данных XDR, CDR, NDR и трансформация их к этим представлениям;

– ЯП программных компонентов и средства обращения компонентов друг к другу посредством механизма вызова;

– языки описания интерфейсов компонентов – RPC, IDL и RMI для передачи данных между разными компонентами.

К *методам преобразования* форматов данных относится набор правил кодирования и декодирования (маршаллинг) данных, линеризация сложных структур и порядка расположения данных в передающей и соответственно в принимающей платформах компьютеров.

Механизмами передачи данных являются:

– протоколы передачи данных (TCP/IP, UDP, GIOP и др.) [16];

– классы функций преобразования различающихся типов и структур данных ЯП и генерации соответствующих новых типов данных [14, 16–21];

– системные процедуры по обеспечению маршаллинга данных между разными объектами распределенной среды неоднородных компьютеров [5, 14, 21].

При передаче данных от компонента в одном ЯП компоненту на другом языке устраняются различия в представлении типов данных в этих ЯП путем систематического и эквивалентного преобразований типов и структур данных одного ЯП к соответствующим типам и структурам данных другого ЯП. Соответствие преобразуемых типов данных устанавливается с помощью специальных функций, общесистемных средств, либо стандарта, регламентирующего независимые от языков типы данных (ISO/IEC 11404–96).

4.1.1. Методы преобразования форматов данных

На каждой действующей платформе компьютера используются соглашения о кодировке символов (например, ASCII, EBCDIC), о форматах целых чисел и чисел с плавающей точкой (например, IEEE, VAX, IBM и др.). Для представления целых типов (short, long) используется дополнительный код, для типов float и double – стандарт ANSI / IEEE, а для char – множество значений ISO Latin /1 .

Порядок расположения байтов зависит от структуры процессора платформы (Big Endian или Little Endian) – от старшего к младшему байту и от младшего к старшему (например, в сетях Ethernet байты кодируются, начиная с младшего по значению бита). При этом существуют процессоры, которые поддерживают обе возможности (UltraSPARC, PowerPC) учитывают возможное несовпадение порядка байтов [1].

Имеется несколько видов стандартов, которые обеспечивают маршаллинг данных внутри соответствующей промежуточной среды и между разными. Рассмотрим их.

XDR-стандарт содержит язык описания структур данных произвольной сложности и средства преобразования данных, передаваемых на другие платформы (Sun, VAX, IBM, Cray). Программы, написанные на разных ЯП, могут использовать одни и те же данные в XDR-формате, несмотря на то, что компиляторы с ЯП выравнивают их в памяти машины по-разному [1].

Если один из компьютеров с адресацией «от младшего» обменивается целыми числами с компьютером «от старшего», то отсылающий компьютер преобразует их в соответствии с XDR-стандартом в целые с порядком байтов «от младшего» к порядку байтов «от старшего». Принимающий компьютер делает обратные преобразования.

При преобразовании форматов данных большая часть времени, требуемого для подготовки форматов данных к передаче, тратится не на преобразование, а на анализ элементов структуры данных. Принятый 4-байтный размер XDR-блока передачи данных максимизирует производительность передачи для всех архитектур без большого расхода памяти.

Для осуществления кодирования (code) или декодирования (decode) данных используется библиотека XDR-процедур, представленная в табл.4.1. В ней определены основные xdr-процедуры преобразования форматов данных, заданных в XDR-языке к виду представления в языке C++.

Кодирование – это преобразование из локального представления в XDR-представление и запись полученного результата в XDR-блок. Декодирование – это чтение объекта из XDR-блока и преобразование его в локальное представление принимающей платформы. Библиотека содержит процедуры форматирования для простых и сложных типов данных XDR-стандарта.

ТАБЛИЦА 4.1. Основные элементарные процедуры преобразования

Тип данных в языке C++	XDR-процедура	Тип данных в XDR-стандарте
Char	xdr_char( )	Int
short int	xdr_short ( )	Int
Unsigned short int	xdr_u_short ( )	unsigned int
Int	xdr_int ( )	Int
Unsigned int	xdr_u_int ( )	unsigned int
Long	xdr_long ( )	Int
unsigned long	xdr_long ( )	unsigned int
Float	xdr_float ( )	float
Double	xdr_double ( )	double
Void	xdr_void ( )	void
Enum	xdr_enum ( )	int

Для выравнивания данных в памяти используется стратегия размещения значений базовых типов с адреса, кратного их действительному размеру в байтах (2, 4, 8, 16) и выравнивания границ данных по наибольшей длине. В памяти могут быть «дырки» от предшествующей информации, которые учитываются при выравнивании последующих данных. Некоторые компиляторы оптимизируют расположение полей памяти под сложные структуры данных с помощью специальных процедур и функций, которые проводят анализ форматов представления данных и их преобразование к формату принимающей платформы. Компилятор с XDR-языка генерирует к ним обращения и размещает данные для платформ клиента и сервера. Принимающая платформа обрабатывает полученные данные, а затем декодирует их обратно к виду формата платформы, отправившей эти данные.

Таким образом, для передачи по сети XDR-стандарт преобразования форматов данных включает в себя две взаимно обратные процедуры: кодирование и декодирование. Алгоритмы этих процедур достаточно симметричны и соответствуют базовым операциям (put и get).

CDR-стандарт реализован в среде CORBA, обеспечивает преобразование данных в формат с учетом различных платформ передающей и принимающей сторон.

Процедура маршаллинга обеспечивается средствами кодогенерации и специальным буфером для размещения данных, размер которого определяется при компиляции с выравниванием промежуточных полей некоторых типов данных. Маршаллинг выполняет брокер ORB путем интерпретации или компиляции

данных и применения универсальной структуры интерпретатора TypeCode для формирования необходимых процедур преобразования форматов данных и значений базовых типов. Процедуры преобразования сложных типов используют базовые примитивы и следующие элементы:

– дополнительный код для представления целых чисел;

– числа с плавающей точкой (стандарт ANSI / IEEE);

– символы ISO Latin /1;

– схема выравнивания значений базовых типов, независимо от составных типов или конкретных данных, реализуемая всеми современными компиляторами;

– базовые типы языка IDL (64-разрядный целый тип – signed и unsigned), плавающий тип двойной точности и др.

Процедуры преобразования типов данных интерпретируются TypeCode для каждого типа языка IDL (базового и составного) как интерфейса, содержащего поле со значением типа TCKind, дополнительные параметры, соответствующие этому значению, множество базовых типов (tk_long, tk_float) и вид конструктора составных типов (tk_struct, tk_sequence). Константы TypeCode порождаются компилятором на основе спецификации интерфейса в языке IDL.

Преобразование данных осуществляют процедуры encoder ( ) и decoder ( ) интерпретатора TypeCode. Параметрами encoder( ) являются данные для преобразования и указатель на буфер. Интерпретатор определяет тип исходных данных и вызывает соответствующий базовый примитив для записи информации в буфер. Для сложного типа интерпретатор вычисляет размер и границы выравнивания данных с помощью значений типа TCKind, зафиксированного в стандарте CORBA. Размер и граница выравнивания базовых типов зависит от конкретной платформы.

Процедура decoder ( ) выполняет функцию, обратную encoder ( ), имеет аналогичную структуру и выполняет дополнительные действия по выделению памяти, необходимой для расположения данных переменной длины (sequence или string).

Таким образом, интерпретатор TypeCode обеспечивает единообразное преобразование типов произвольной сложности, а также типов, которые задаются динамически.

4.1.2. XML-стандарт для устранения неоднородности взаимосвязей компонентов

Язык XML (Extensible Markup Language) [20] – базовое средство обеспечения взаимосвязей и преобразования передаваемых данных компонентам, записанных в разных ЯП, а также предос-

тавления единого формата данных. XML-ориентированные программные средства и функции могут работать на разных платформах и средах. Современные промежуточные среды (CORBA, DCOM, JAVA и др.) имеют в своем составе в той или иной форме реализованные специальные функции, аналогичные разным аспектам XML как альтернативы сервисам CORBA при обеспечении взаимосвязей компонентов, записанных на разных ЯП.

XML имеет множество различных системных поддержек, например, браузер – Internet Explorer для визуализации XML-документов, объектная модель DOM (Document Object Model) консорциума W3 для отображения XML-документов в объектно-ориентированной среде. При этом объекты этой модели, имеющие интерфейс, описанный в языке IDL, могут распределяться и обрабатываться системой CORBA.

Тексты в XML описываются с помощью стандартного формата данных ASCII. XML является удобным средством кодирования собственных типов данных с помощью общепринятых файловых форматов. Если прикладная или информационная система не работала ранее с XML, то необходимо провести переформатирование данных в формат XML, и наоборот.

Для компонентных моделей использование средств XML означает, что их отдельные (или все) составные элементы подаются в виде XML-документов, обработка которых зависит от назначения модели. Как правило, применяются два подхода.

1. На основе описания XML-документа могут быть сгенерированы тексты на ЯП, которые после компиляции включаются в состав среды, реализующей представление соответствующей объектной модели. Все особенности взаимодействия отдельных компонентов (поиск и установления связей, обращение к функциям, обмен данными) отображаются в сгенерированном тексте.

2. Исходный вид XML-документ обрабатывается в режиме интерпретации, при этом применяются средства синтаксического разбора XML-конструкций, т.е. парсеры и прикладные интерфейсы.

Таким образом, XML-язык позволяет представлять объекты для разных объектных моделей на единой концептуальной, синтаксической и семантической основе. Он не зависит от платформы и среды задания моделей взаимодействия компонентов прикладного уровня, упрощает обработку документов, а также работу с БД стандартными методами и средствами (XML-парсеры, DOM-интерфейсы, XSL-отображение и др.).

4.2. ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ ПРЕОБРАЗОВАНИЯ ТИПОВ ДАННЫХ В ЯП

Основными ЯП описания отдельных компонентов для работы в распределенных средах, являются C++, Паскаль, JAVA, Ада, Smalltalk и др. Характерной особенность этих ЯП является неоднородность как в смысле представления типов данных ЯП, так и платформ компьютеров, где реализованные системами программирования ЯП выполнятся.

Причиной неоднородности также являются различные способы передачи параметров между объектами в разных промежуточных средах, поддерживающие разные объектные модели и форматы данных для хранения параметров и результатов выполнения запросов объектов.

В общем, ЯП имеют следующие виды различий:

– разное представление типов данных;

– разные средства конструирования типов данных;

– разные способы программирования интерфейсов (в некоторых ЯП интерфейсы отделены от реализации);

– разная трактовка понятия наследования;

– разные способы задания операторов вызова (статически или динамически) для обеспечения взаимосвязи.

Реализация компонентов в ЯП в системах программирования имеет следующие общие особенности и различия:

– компиляторы одного и того же ЯП могут иметь разные двоичные представления, если они реализованы на разных компьютерах;

– представление машинного кода компиляторов ЯП зависит от аппаратной платформы;

– разные виды связей между объектами предоставляются разными средами;

– размещение параметров и объектов может различаться.

К общим принципам взаимосвязи ЯП, характерных для современных распределенных сред, отнесем следующие:

– связь двух ЯП – двунаправленная;

– атрибуты вызовов ЯП отображаются в операции, а операции в методы;

– связь с ЯП реализуется ссылками в компиляторе и осуществляется через отдельные интерфейсные посредники для каждой пары ЯП или через общий корневой объект для всех ЯП промежуточного слоя распределенной среды.

В общем виде связь устанавливает соответствие между интерфейсами ЯП и промежуточной средой, отображает конструкции ЯП в операции интерфейса, и наоборот. В промежуточный слой среды входят интерфейсы L_i , L_n по количеству пар ЯП, взаимодействующих между собой (рис. 4.1).

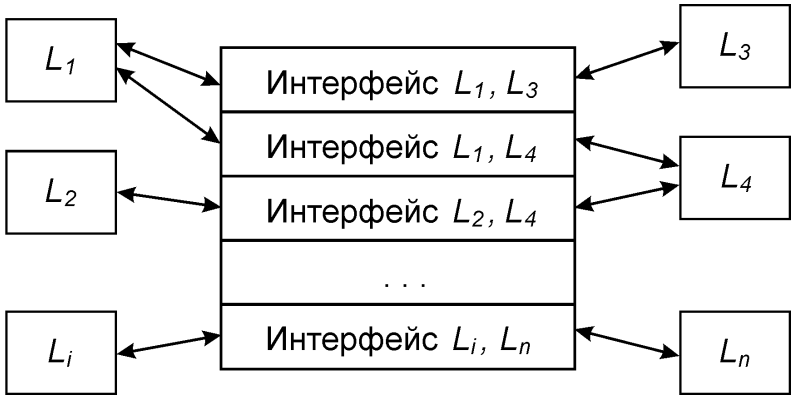

РИС. 4.1. Связь между языками L_1 , L_2 , ..., L_n через интерфейсы для каждой пары ЯП

Далее описано решение проблемы неоднородности ЯП с помощью метода доказательного преобразования типов данных ЯП и системного преобразования типов данных в промежуточном слое распределенной среды CORBA.

4.2.1. Метод формального преобразования типов данных ЯП

Данный метод преобразования типов данных компонентов в одном ЯП к представлению типов данных компонентов в другом ЯП основан на формализованном определении их интерфейсов и разработан с участием автора [17–19]. При вызове разноязыкового компонента в одном из ЯП высокого уровня проводится проверка соответствия типов данных формальных параметров $F = \{ f_1, f_2 , ..., f_{к1} \}$ этого компонента и фактических параметров $V = \{ v_1, v_2 ,... v_к \}$, которые передаются вызывающему компоненту и проверяется на неоднородность типов данных для их отображения (mapping).

В общем случае задача отображения $A: П \rightarrow Ф$,
где $П = \{ V^1, V^2 ,..., V^m \}$, $Ф = \{ F^1, F^2 ,..., F^m \}$, состоит в построении отображений вида:

$$\bigcup_{t=1}^{m} V^t = V\ ,\ V^t \cap V^{t'} = \varnothing \ при\ t \le t'$$

$$\bigcup_{t=1}^{m} F^t = F\ ,\ F^t \cap F^{t'} = \varnothing \ при\ t = t'$$

Отображение выполняется за два этапа.

1). Построение операций преобразования типов данных $T_\alpha = \{T_\alpha^t\}$ для множества языков $L = \{l_\alpha\}_{\alpha=1,\,n}$.

2). Преобразование типов данных для каждой пары взаимодействующих компонентов в ЯП $l_{\alpha 1}$ и $l_{\alpha 2}$ с применением операций селектора S и конструктора C.

При преобразовании типов данных используется алгебраический подход, при котором каждому типу данных T_α^t ставиться в соответствие алгебраическая система:

$$G_\alpha^t = <X_\alpha^t\ ,\ \Omega_\alpha^t\ >,$$

где t – тип данных, X_α^t – множество значений, которые могут принимать переменные этого типа данных, Ω_α^t – множество операций над типами данных.

Для простых типов данных ЯП ($t = b$ (bool), c (char), i (int), r (real)) и сложных типов данных ($t = a$ (array), z (record), u (union), e (enum)), как комбинации простых типов данных, построены соответственно две алгебраические системы:

$$G_1 = \{\ G_\alpha^b\ ,\ G_\alpha^c\ ,\ G_\alpha^i\ ,\ G_\alpha^r\ \},$$
$$G_2 = \{\ G_\alpha^a\ ,\ G_\alpha^z\ G_\alpha^u\ ,\ G_\alpha^e\ ,...\}. \qquad (4.1)$$

Каждая из систем определяется на множестве значений типов данных и операций над ними:

$G_\alpha^t = <X_\alpha^t\ ,\ \Omega_\alpha^t\ >$, где $t = b, c, i, r, a, z, u, e$.

Операциям преобразования каждого t типа данных соответствует изоморфное отображение двух алгебраических систем.

В классе систем (4.1) преобразование типов данных $t \to q$ для пары языков l_α и l_b обладает такими свойствами:

1. Системы G_α^t и G_β^q являются изоморфными (типы q, t определены на том же множестве).

2. Между G_1 и G_2, а также X_α^t и X_β^q существует изоморфизм. Множества операций Ω_α^t и Ω_β^q различны.

Если множество операций $\Omega = \Omega_\alpha^t \cap \Omega_\beta^q$ не пусто, то имеем изоморфизм двух систем:

$G_{\alpha}^{t'} = < X_{\alpha}^{t}\ ,\ \Omega_{\alpha}^{t} >,\ G_{\beta}^{q'} = < X_{\beta}^{q}\ ,\ \Omega_{\beta}^{q} >.$

Если тип t – строка, а тип q – вещественный, то между множествами X_{α}^{t} и X_{β}^{q} не существует изоморфного соответствия.

3. Мощности алгебраических систем должны быть равны $|\ G_{\alpha}^{t}\ | = |\ G_{\beta}^{q}\ |$.

Алгебраические системы линейно упорядочены и поэтому любое отображение 1, 2 сохраняет линейный порядок.

Докажем изоморфизм двух алгебраических систем G_{α}^{t} и G_{β}^{q} с помощью функций $\varphi_1\ (X_{a\,\max}^{t})$ и $\varphi_2\ (X_{a\,\max}^{t})$, лемма 4.1 и теоремы 4.1–4.5.

Лемма 4.1. *Для любого изоморфного отображения φ между алгебраическими системами G_{a}^{t} и G_{β}^{a} выполняются равенства* $\varphi_1\ (X_{a\,\min}^{t}) = X_{\beta\,\min}^{a},\ \varphi_2\ (X_{a\,\max}^{t}) = X_{\beta\,\max.}^{a}$.

Здесь $X_{\min}$ и $X_{\max}$ означают соответственно минимальное и максимальное значение элементов множества X с числовыми типами и входящими в состав простых типов данных. Отсюда следует, что эти элементы принадлежат и алгебраическим системам (4.1). Так как структурные типы строятся из простых типов с помощью конечного множества операций, то их множества значений тоже конечны и сохраняют линейную упорядоченность.

Теорема 4.1. *Пусть φ - отображение алгебраической системы G_{α}^{c} в алгебраическую систему G_{β}^{c} (c-char). Для того чтобы отображение φ было изоморфизмом, необходимо и достаточно, чтобы φ изоморфно отображало X_{α}^{c} на X_{β}^{c} с сохранением линейного порядка.*

Необходимость. Пусть φ – изоморфизм. Тогда при отображении сохраняются все операции множества $\Omega = \Omega_{a}^{c} = \Omega_{\beta}^{c}$, в том числе и операция отношения, которая определяет линейный порядок на множествах значений X_{a}^{c} и X_{β}.

Достаточность. Пусть φ изоморфно отображает X_{a}^{c} в X_{β}^{c} с сохранением линейного порядка. Необходимо проверить сохранность операций pred и succ. Операция отношения pred выполняется согласно упорядоченности. Операция succ доказывается с помощью леммы 4.1 и множества значений $\varphi\ (X_{a\,\min}^{c.}) = X_{\beta\,\min}^{c}$.

Последовательно применяя операцию succ к данному равенству и учитывая линейную упорядоченность множества X_{a}^{c} и X^{c}_{β} (x < succ(x)), получаем, что для любого $x^{c}_{a} \in X^{c}_{a}$ *и* $x^{c}_{a} \neq X_{a\,\max}^{c}$ из равенства $\varphi\ (x_{a}^{c}) = x^{\,c}_{\beta}$, где $x^{c}_{\beta} \in X^{\,c}_{\beta}$ следует равенство

$$\varphi\ (succ(x^{c}_{a})) = succ(x^{c}_{\beta}). \qquad (4.2)$$

Аналогично доказывается операция pred с помощью $\varphi(X_{a\,\max}^{c}) = X_{\beta\,\max}^{c}$.

Теорема 4. 2. *Любой изоморфизм φ между алгебраическими системами G_{α}^{b} и G_{β}^{b} (b-bool) является тождественным изоморфизмом*:

$$\varphi\,(\,X_{\alpha.false}{}^{b}\,) = X_{\beta.false}{}^{b}\,,$$
$$\varphi\,(\,X_{\alpha.true}{}^{b}\,) = X_{\beta.true}{}^{b}\,. \qquad (4.3)$$

Д о к а з а т е л ь с т в о. При отображении G_{α}^{b} и G_{β}^{b} всегда справедливо $X_{\alpha.false}{}^{b} < X_{\beta.true}{}^{b}$. Поэтому, учитывая, что φ сохраняет линейный порядок, единственным изоморфизмом является (4.3).

Теорема 4.3. *Любой изоморфизм между алгебраическими системами с соответствующими числовыми типами является тождественным автоморфизмом.*

Доказательство аналогично доказательству теоремы 4.2.

Теорема 4.4. *Пусть G_{α}^{a} и G_{β}^{a} − алгебраические системы, соответствующие типам данных массивов, и φ_i и φ_v - изоморфные отображения множеств индексов и значений элементов массивов, сохраняющие линейный порядок. Тогда изоморфизм определяется соответственно*:

$\varphi_i : I_{\alpha}^{a} \to I_{\beta}^{a}$,
$\varphi_v : Y\,(\,X_{\alpha}^{a}\,) \to Y\,(\,X_{\beta}^{a}\,)$.

Изоморфизм между алгебраическими системами G_{α}^{a} и G_{β}^{a} полностью определяется отображениями φ_i и φ_v, которые сохраняют линейный порядок, а значит и упорядоченность элементов массива.

Теорема 4.5. *Пусть G_{α}^{z} и G_{β}^{z} − две алгебраические системы, соответствующие типам данных: запись и $x_{\alpha}^{z} \in X_{\alpha}^{z}$, $x_{\beta}^{z} \in X_{\beta}^{z}$* . Тогда, если между последовательностями компонентов записей x_{α}^{z} и x_{β}^{z} существует взаимно однозначное соответствие, то изоморфизм φ между алгебраическими системами G_{α}^{z} и G_{β}^{z} определяется изоморфными отображениями, соответствующих компонентам записи.

Преобразования для смешанных простых типов вида $c \to i$, $c \to b$, $i \to c$, $i \to b$, $b \to i$, $b \to c$ теоретически рассматривается как преобразование для перечисленных типов. Преобразование вещественных типов предполагает использование частных случаев, так как отсутствует изоморфизм основных множеств алгебраических систем. Преобразования между массивами и записями сводятся к преобразованию простых типов данных их элементов.

При преобразовании простых и структурных типов используются операции селектора S и конструирования C для изменения уровня структурирования данных Операция селектора S для массива определяется как ограничение отображения:

$$M : I \to Y \text{ на } I',\ E \# M : I \to Y ,$$

где E – вложение $I' \in I$. Тогда $M|\{ k \}$ соответствует k – элементу массива при $I' = \{ k \}$. Аналогично эта операция определяется и для записи: $M|\{ S_{vm} \}$, где M – отображение между селекторами компонентов и самими компонентами S_{vm} определяет соответствующий компонент записи.

Операция конструирования C массива заключается в формальном упорядочивании компонентов и определении соответствия между множеством индексов и множеством элементов массива. Аналогично эта операция определяется для записи, union и др.

Для каждой пары взаимодействующих ЯП соответствующие операции прямого и обратного преобразования простых и сложных типов данных выполняются статически с помощью специальных функций, которые при компиляции включаются в интерфейс, являющийся посредником пары ЯП (см. рис. 4.1).

Множество операций селектора для массива и записи в l_α языке имеет вид:

$$S = \bigcup_{\alpha=1}^{n} S_\alpha,\ S_\alpha = \{ S_\alpha^a ,\ S_\alpha^z \} .$$

Операция конструирования массива построена путем формального упорядочивания его компонентов и определения соответствия между множеством элементов массива и их индексов. Аналогично построена операция конструирования для записи.

Окончательно множество операций конструирования можно представить в виде:

$$C = \bigcup_{\alpha=1}^{n} C_\alpha ,\ C_\alpha = \{ C_\alpha^a ,\ C_\alpha^z \} ,$$

где C_α^a , C_α^z – операции конструирования массива и записи соответственно в языке l_α. Полученные множества операций S и C выполняют роль элементарных правил сборки объектов, представленных разными ЯП.

Для описания интегрированной среды функционирования совокупности разноязыковых компонентов ПС используют мо-

дель информационного объединения и модель управления объектами в динамике их выполнения.

При описании модели информационного объединения используются построенные алгебраические системы $(G_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r, G_\alpha^a, G_\alpha^z)$, множество функций преобразования одних типов данных в другие и операции понижения уровня структурирования данных для сложных и структурных типов данных.

Таким образом, множество операций S и C определяют элементарные правила выбора и конструирования сложных типов данных взаимодействующих компонентов, которые описаны в разных ЯП.

4.2.2. Общий подход к обеспечению неоднородности ЯП в системе CORBA

На эффективное использование ЯП в качестве средства разработки ПС оказывает влияние наличие разных видов компьютеров и систем программирования ЯП, способов их размещения на разных компьютерах и установления связей между объектами ПС. Связывание объектов определяется не только возможностями ЯП, но и способами передачи параметров между ними в разных средах, поддерживающих различающиеся объектные модели и форматы данных для хранения параметров и результатов выполнения запросов объектов [5].

В промежуточный слой системы CORBA входят интерфейсы по количеству пар ЯП и общий корневой объект, который обеспечивает трансляцию и компиляцию интерфейсных описаний stub и skeleton, как интерфейсов прикладных объектов клиента и сервера, для каждой пары взаимодействующих ЯП (рис. 4.2).

Среда CORBA реализует общее связывание ЯП через корневой объект связи, которое для некоторой сгенерированной среды системы CORBA можно подсчитать для конкретных ЯП (два, три) по формуле $n \times (n \times (n\text{-}1) / 2$ и столько же связей будет между каждым ЯП и IDL.

Общий принцип решения проблемы неоднородности в промежуточном слое системы CORBA заключается в единой схеме обработки запроса, в котором задаются ссылки на описание объектов-клиентов и объектов-серверов в любых ЯП, семантика которой состоит в следующем:

РИС. 4.2. Связывание ЯП L_1, L_2, ..., L_6 через общий корневой объект

– описание в ЯП, указанное в запросе клиента, отображается в операции IDL;

– операции IDL преобразуются в конструкции ЯП;

– результат преобразования передается серверу через брокера ORB для исполнения запроса.

Сервер реализует полученный запрос путем связывания ЯП и типов данных, представленных в stub, в типы данных клиента, являющихся классами.

В распределенную среду CORBA включаются объектные модели для всех объектно-ориентированных ЯП (CORBA, COM, JAVA), в которой общая объектная модель системы осуществляет взаимодействие программ с помощью брокера ORB.

Объектно-ориентированная парадигма определяет общий принцип взаимодействия объектов, который состоит в том, что любой объект выполняет метод (функцию, сервис, операцию) при условии, если другой объект, который выступает в роли клиента для него, посылает ему запрос для выполнения этого метода. Объект может выполнять совокупность методов, которые определяют интерфейсы, и отвечать за выполнение действий сам, инициировать его выполнение другим или координировать действия других. Обязанности между объектами распределяются произвольным образом, например, обязанность выполнения одной функция или сервиса может быть выполнена одним методом или несколькими из разных классов.

Специфика выполнения методов и их индивидуальные особенности определи типичные правила организации взаимодействия объектов в распределенной среде. Они отображены в ряде объектных моделей промежуточных слоев в виде механизмов взаимодействия типа клиент–сервер.

Связывание с ЯП в промежуточном слое состоит в отображении объектных типов в типы клиентских и серверных стабов, которые выступают в роли классов и определяют интерфейсы, используемые реализациями клиента и сервера.

ЯП, используемые в системе CORBA, могут реализовываться на разных платформах и в разных средах и иметь двоичное представление для конкретной аппаратной платформы. Для всех ЯП (C++, JAVA, COBOL и др.) в системе CORBA предусмотрена общая объектная модель в среде брокера и расположение параметров методов объектов в промежуточном слое. Связь устанавливает соответствие между объектными моделями каждого ЯП систем COM и JAVA распределенной среды CORBA и дополняют ее новыми возможностями (рис. 4.3). Рассмотрим их.

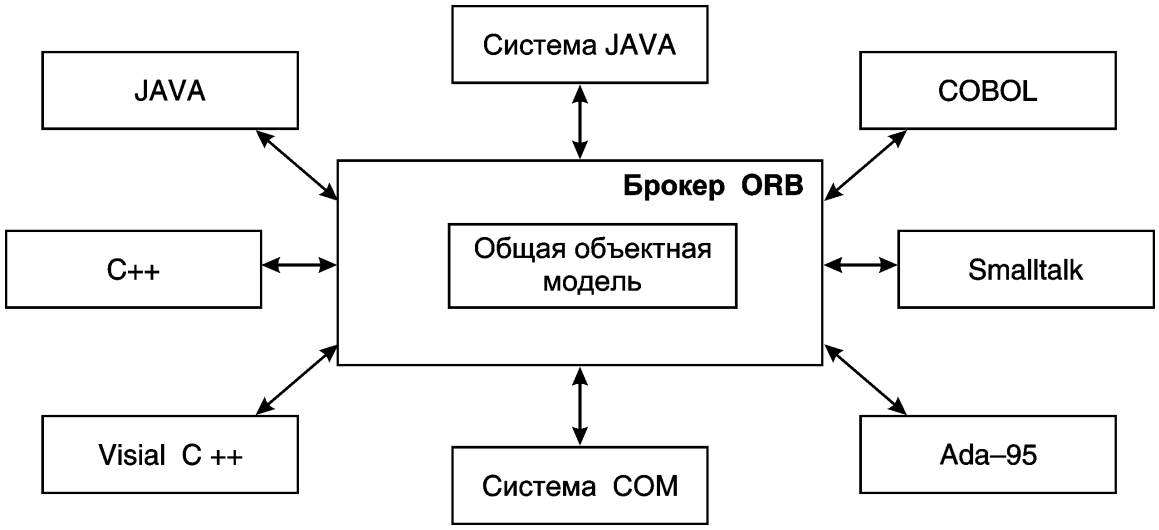

РИС. 4.3. Интегрированная среда системы CORBA

Объектная модель CORBA является статически типизированной. В ней простые типы данных ЯП определяются статически. Сложные типы данных, экземпляры которых являются значениями, делаются ссылочными, а их параметры и атрибуты получают статические типы в IDL и в представлениях объектов-клиентов и объектов-серверов в ЯП. Для всей распределенной среды имена объектов должны быть уникальными. Механизм языка IDL позволяет объявлять интерфейс, типы данных, константы и исключения. Модель допускает динамический или статический удаленный вызов типа RPC.

В *объектной модели COM*, входящей в общую объектную модель системы, типы данных также определяются статически, а

конструирование сложных типов данных допускается только для массивов и записей. Операции вызова включают в себя типы параметров – in, out, inout.

Методы объектов модели используются в двоичном виде, основанные на совместимости использования машинного кода объекта, созданного в одной среде разработки, для работы в другой среде. Совместимость разных ЯП достигается также двоичным представлением и отделением интерфейсов объектов от реализаций.

JAVA/RMI объектная модель обеспечивает вызов удаленного метода объекта посредством ссылок на объекты, задаваемых указателями на адреса памяти. Типы данных – атомарные, классы и интерфейсы – являются расширением Remote интерфейса и определяются статически. Интерфейс – это объектный тип, реализуемый классами, он обеспечивает удаленный доступ к нему сервера. Интерфейсы, расширяющие принцип удаленности, называют удаленными интерфейсами. Компилятор JAVA создает байт-код, который интерпретируется виртуальной машиной, обеспечивающей переносимость байт-кода на разные платформы и однородность представления данных на всех платформах, в том числе в среде CORBA.

4.3. ПРИКЛАДНЫЕ СРЕДСТВА ВЗАИМОДЕЙСТВИЯ И ПРЕОБРАЗОВАНИЯ ДАННЫХ

Вопросы реализации процедур преобразования данных рассмотрим на примере языка RPC, широко используемого в средах ONC и DCE, а также языка IDL среды CORBA [3–5] для программ клиента и сервера, реализованных на языке C++.

Данные среды имеют общий механизм посылки запроса от одного объекта к другому, который основан на интерфейсном посреднике и обеспечивающим взаимодействие между клиентом и сервером.

Для сред ONC и DCE взаимодействие выполняет монитор на стороне клиента и портмеппер (portmapper) на стороне сервера, а для среды CORBA – брокер объектных запросов.

Портмеппер пересылает запросы в XDR-стандарте от клиента к серверу и обратно. Параметры запросов преобразуются к виду платформы сервера. При этом программа клиента состоит из заголовочного файла, stub-клиента, прикладных функций клиента и монитора для связи с сервером. Программа сервера включает в себя stub-сервер, выходной файл сервер–процедур и порт-

меппер. При поступлении на сервер запроса от клиента, портмеппер регистрирует обращение клиента к серверу, определяет из RPC-библиотеки заданную удаленную функцию, преобразует для нее данные из XDR-стандарта к внутреннему представлению и выполняет ее. После того, как эта функция будет выполнена, она передает stub-серверу полученные результаты для обратного преобразования к XDR-стандарту и передачи результата клиенту.

К функциям взаимодействия брокера ORB относятся:

– нахождение серверного объекта;

– упаковка запроса и его параметров, передача запроса по сети;

– ожидание результатов, их распаковка, обработка и передача результата клиенту.

Дополнительно, кроме этих функций, брокер ORB обеспечивает хранение данных, синхронизацию взаимодействия и др.

Как правило, при взаимодействии клиента и сервера принимает участие два брокера ORB: один на стороне клиента, другой на стороне сервера. Клиентский брокер передает запрос, полученный от объекта-клиента, на сторону сервера, а затем принимает от него результаты. Брокер на стороне сервера, в свою очередь, принимает запрос от клиента, передает его серверному объекту и возвращает результаты клиенту. Если объект является и клиентом, и сервером, то брокер ORB передает и принимает сообщения.

4.3.1. Взаимодействие объектов в системе CORBA

Стандарт консорциума OMG определяет распределенные объектные средства, позволяющие приложениям взаимодействовать прозрачным образом (независимо от платформы) с помощью брокеров ORB, языков IDL, CDR и протокола передачи данных GIOP.

CORBA базируется на объектной модели, являющейся классической моделью взаимодействия клиент–сервер на основе механизма передачи сообщений. В этой модели интерфейсы отделены от реализаций, и сама модель определяет интерфейсы в терминах языка IDL, независимо от конкретных ЯП, включает в себя полную сигнатуру операций обращения (имя объекта, имя метода, типы передаваемых параметров и тип возвращаемого результата). Спецификация объекта в языке IDL не включает в себя операторы и другие конструкции, она близка средствам описания классов в C++ и отображается в ЯП клиентских и серверных объектов.

Все типы данных IDL подразделяются на две группы: базовые и составные.

К базовым типам относятся: целочисленный тип (signed и unsigned integer), 32 и 64-разрядные числа с плавающей точкой IEEE, символы ISO Latin /1, логический тип boolean и некоторые другие. Octet − специальный 8-разрядный базовый тип данных, не требующий преобразования при переносе с одной платформы на другую.

Составные типы − это типы более высокого уровня, к которым относятся: тип interface − тип IDL посредника между клиентом и сервером, struct, union, sequence и array. Тип struct аналогичен языку C++; sequence и array − последовательность и массив, содержащие элементы одинакового типа и фиксированной длины. Тип union семантически соответствует типу union в языке C++ с дополнением дескриптора вариантов.

На более низком уровне стандарт взаимодействия и представления данных задается с помощью протоколов TCP/IP и GIOP, которые передают сообщения между брокерами ORB и включают в себя CDR формат представления данных и протокол сообщений GIOP.

Дополнительный протокол IIOP (Internet Inter − ORB Protocol) является частным случаем отображения GIOP на конкретный транспортный протокол.

CDR задает набор правил представления данных, включающих в себя использование специального знака в сообщениях GIOP для указания используемого порядка байтов и соглашений о выравнивании базовых типов. Этот язык охватывает все множество типов IDL и понятие потока октетов как абстракцию буфера памяти, которая реализуется в виде класса в C++.

4.3.2. Средства обеспечения интероперабельности

Под *интероперабельностью* понимается способность совместного взаимодействия разнородных компонентов системы для решения определенной задачи. Основу такого взаимодействия составляет концепция промежуточного слоя (middleware). Интероперабельность достигается общим механизмом поддержки взаимодействия компонентов (например, ORB), введением базовой модели компонентов, унифицированного языка описания спецификаций интерфейсов, отделением реализации компонентов от их интерфейсов. Этим достигается однородность представления компонентов и их взаимодействий.

Для создания сложных систем с множеством межкомпонентных связей используются технологии (объектные, компонентные и др.), поддерживающие принцип декомпозиции ПрО на отдельные объекты, определение интерфейсов между компонентами, повторное использование в смысле как функциональности, так и архитектуры системы.

Международный комитет по стандартизации ISO, консорциум OMG и компания Microsoft приняли в качестве технологии распределенного взаимодействия объектно-ориентированную парадигму. Кроме того, были созданы стандарты открытых систем и промышленные продукты поддержки промежуточного слоя – CORBA, DCOM, LAVA и др., а также всевозможные сервисы разработки и выполнения приложений в распределенной среде, предоставляющие процедуры маршаллинга данных взаимодействующих компонентов. Системные сервисы систем промежуточного слоя подключаются к приложению с помощью брокера и тем самым расширяют его прикладную функциональность.

К средствам обеспечения интероперабельности и передачи данных между разными средами и платформами относится, например, стандартный механизм связи между JAVA и C/C++ компонентами, основанный на применении концепции Java Native Interface (JNI), реализованной как средство обращения к функциям из JAVA-классов и библиотек, разработанным на других языках.

Эти средства включает в себя анализ JAVA-классов в целях поиска прототипов обращений к функциям, реализованным на языках C/C++, и генерацию заголовочных файлов для использования их при компиляции C/C++ программ. В JAVA классе известно, что в нем содержится обращение не к JAVA-методу (он называется native и для загрузки необходимых C/C++ библиотек добавляется вызов функции), а ориентируется именно на такую связь. Данная схема действует в одном направлении – от JAVA к C/C++ и только для такой комбинации ЯП.

Аналогичную задачу реализует технология Bridge2Java, которая обеспечивает возможность обращения из JAVA-классов к COM-компонентам. В этой схеме представление COM-компонента расширяется для “понимания” его JAVA-классом. В этих целях генерируется оболочка для COM-компонента, которая включает в себя прокси-класс, который обеспечивает необходимую трансформацию данных на основе стандартной библиотеки преобразований типов и вызов COM-функций с соблюдением всех правил работы

с COM-компонентами, в частности используется стандартная иерархия интерфейсов. Данная схема является более гибкой, она не требует изменений в исходном Java–классе и, кроме того, COM-компоненты могут быть написаны в разных языках.

Механизм интероперабельности реализован также на платформе .Net, основу которого составляет промежуточный язык CLR (Common Language Runtime). В этот язык транслируются коды, написанные в разных ЯП (C#, Visual Basic, C++, Jscript). CLR разрешает не только интегрировать компоненты, разработанные в разных ЯП, но и использовать библиотеку стандартных классов независимо от языка реализации. Такой подход разрешает реализовать доступ к компонентам, которые были разработаны раньше без ориентации на платформу .Net, например к COM–компонентам. Для этого используются стандартные средства генерации оболочки для COM–компонента, с помощью которого он представляется как .Net–компонента. При такой схеме реализуются все виды связей и для всех ЯП данной среды [21].

Реализация механизмов передачи данных в распределенной среде включает в себя дополнительные возможности, так как учитывает не только имеющиеся различия в ЯП, но и в операционных средах и в архитектурах используемых компьютеров. Такие механизмы реализованы в среде CORBA, в частности иерархия средств передачи данных от самого верхнего уровня ЯП до самого нижнего и преобразование форматов данных. Иными словами, система CORBA включает в себя оба типа механизмов преобразования: унифицированное представление типов и структур данных, объектных ссылок, системных данных, которые не зависят от платформ и ЯП; правила и средства, обеспечивающие реализацию совместимости среды конкретного ЯП со средой CORBA., Такой подход создается для C++, JAVA.

4.3.3. Преобразование IDL в C++

При отображении некоторого интерфейса I, записанного в языке IDL, в язык C++ создается два типа ссылок, различающихся схемой работы с памятью:

– объектная ссылка I_ptr является указателем на реализацию объекта, связанного с этим интерфейсом I;

– специальный объект I_var, имеющий конструктор и деструктор, владеющий памятью, на которую он указывает [14].

Уничтожение объекта I_var обязательно влечет за собой и уничтожение памяти, на которую он ссылается. При этом значе-

ние переменной любого из этих типов может быть присвоено переменной другого типа и определено преобразование типа I_var в I_ptr. Причем дублирование памяти и контроля операций с данными этих типов не производится.

При вызове деструкторов для var1 и var2 будет дважды уничтожаться одна и та же память, на которую они ссылаются, т.е. возникает ошибки при работе с динамической памятью. Данная схема требует определения в процессе программирования взаимосвязей между всеми объектными ссылками.

При работе с объектными ссылками в системе CORBA преобразование проводится с помощью операций duplicate и release. Первая из них копирует ссылку, вторая – ее уничтожает.

Использование таких операций предполагает прослеживание за взаимосвязями всех объектных ссылок. Любой лишний вызов duplicate или не поставленный в нужном месте вызов release может привести к тому, что память данных объектной ссылки не будет уничтожена. Это приведет к увеличению размера используемой динамической памяти.

Запоминаемая часть преобразования представляется набором правил передачи параметров и результатов. Для каждого типа данных языка IDL параметры специфицируются так: in (входные), out (выходные) и inout (результат).

Схема передачи параметров выделена для простых типов данных (short, long, unsigned short, unsigned long, float, double, boolean, char, octet, enum).

Эти типы данных разделены на данные с фиксированной длиной (fixed-length) и переменной длиной (variable-length). Типами данных переменной длины являются Any, строки (bounded и unbounded string), последовательности (bounded и unbounded sequence), объектные ссылки, структуры и объединения, а также элементы массивов, имеющие переменную длину. Такое разделение типов данных влияет на отображение параметров структур, объединений и массивов.

Эта схема преобразования служит для учета различий типов при написании сигнатур методов серверных объектов, при организации возврата данных из этих методов и вызовов этих объектов.

Если при изменении типа поля некоторой структуры изменяют типы «fixed или variable», то это означает, что требуется переписать текст программ во многих местах как для клиента, так и для сервера. В связи с этим в клиенте и в сервере, где используются параметры out и result, они переписываются в тексте программы.

Для любого составного типа данных T в отображение вводится специальный тип указателей – T_var. Схема работы с параметрами со стороны клиента в посреднике одинакова для всех таких типов. Все параметры для серверного объекта также передаются посредством типа T_var. Параметры типа inout удаляются из памяти при возврате других данных.

Конструктор вида T_var (T) используется объектом T для динамического выделения памяти, которая будет удалена либо присвоена другому объекту. Данная схема требует явного копирования данных перед заполнением их в T_var. Например, заполнение типа String_var var вида: CORBA::String_var var = «some string».

Копирование строк проводится с помощью функции string_dup:

CORBA::String_var var = CORBA::string_dup («some string»).

Для остальных типов данных функция string_dup не предусматривает отображения, поэтому вопрос о выделении динамической памяти и ее заполнении решается непосредственно пользователем.

При заполнении T_var с помощью указателя T_ptr на данные типа T, в нем запоминается значение указателя без копирования данных. Тип T_var, заполненный по умолчанию, не может использоваться для доступа к данным типа T, поскольку в нем хранится нулевая ссылка, поэтому он не может использоваться для возврата out результата метода.

Для занесения массива типа A в объект типа Any введен деструктор. Тип данных Any отличается от A_var, тем, что он не уничтожает массив, на который этот объект ссылается. Компилятор не отличает массив от указателя и при реализации операции заполнения Any из A может возникать конфликтная ситуация.

Преобразование сложных типов данных осуществляется с помощью функций отображения типов, описанных в спецификации IDL.

Тип данных struct преобразуется путем трансформации всех полей в порядке, указанном в спецификации функции в языке IDL, которая отображается в язык C++.

Компилятор IDL отображает конструкции IDL в соответствующие конструкции C++ и в набор вспомогательных процедур и функций, необходимых для общения с брокером ORB.

Функции преобразования базовых типов, содержащие информацию о границе выравнивания и размере соответствующих данных, совпадают с методами класса CDR. Преобразование со-

ставных типов, имеющих вложенную структуру, проводится с помощью базовых процедур и функций. В случае типа данных array обращение к функциям преобразования может осуществляться из других процедур.

Типизированные функции и процедуры encoder ( ) и decoder ( ) имеют симметричные структуры с точностью до базовых процедур, используемых для отображения типов данных в IDL в типы данных C++.

4.4. СТАНДАРТ ISO/IEC 11404–1996 ДЛЯ ПРЕОБРАЗОВАНИЯ ТИПОВ ДАННЫХ

Стандарт ISO/IEC 11404–1996 [8] обеспечивает независимое от ЯП описание типов данных, объявление типов данных и генерацию типов данных в язык LI (Language Independent) этого стандарта. Цель этого стандарта состоит также в преобразовании типов данных конкретных ЯП в LI-язык, и наоборот. Определены специальные правила и операции генерации примитивных типов данных, объединенных и агрегатных типов данных LI-языка в более простые структуры данных ЯП.

Средства описания независимых от ЯП типов данных могут использоваться при определении параметров интерфейса компонентов, задаваемых в IDL, RPC и API.

Независимое от ЯП типы данных в LI-языке подразделяются на примитивные, агрегатные, сгенерированные типы данных (рис.4.4.), семейство типов данных и генератор типов данных.

Каждый тип данных имеет набор свойств, достаточный для его выделения и сравнения с типами данных, содержащихся в ЯП. Стандарт представляет общую модель данных для обращения к нему других стандартов.

Типы данных должны описываться в независимом языке LI, который в отличие от средств описания типов данных в конкретных ЯП является более общим языком, содержащим все существующие типы в современных ЯП, а также обобщенные типы данных и средства, ориентированные на генерацию типов данных. Средствами LI описываются параметры вызова, например, элементы интерфейса, необходимые при обращении к стандартным сервисам и готовым компонентам.

В стандарте представлен раздел объявления типов данных (рис. 4.5), который включает в себя объявление новых типов и переименование существующих, объявление новых генераторов,

значений и результатов. Каждый объявляемый тип данных имеет шаблон, состоящий из словесного описания типа данных, спецификатора типа данных, значения в пространстве значений, синтаксического описания операции.

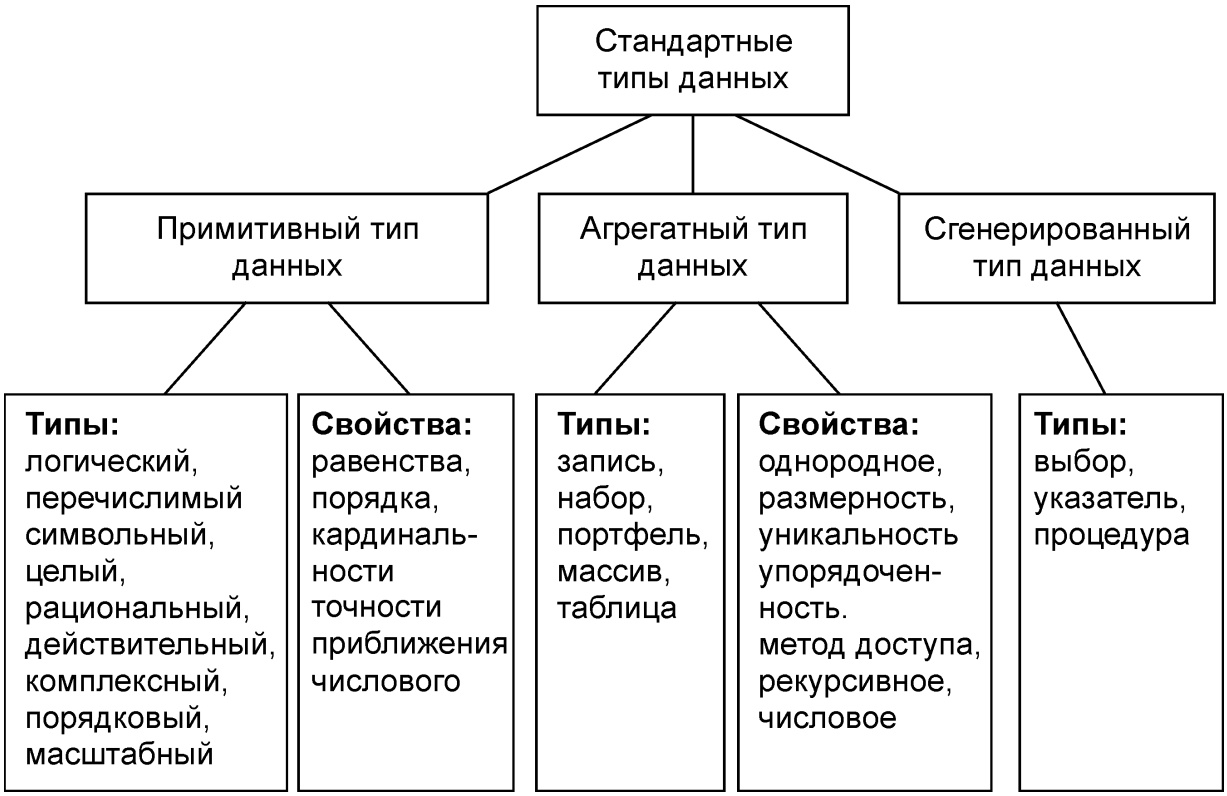

РИС. 4.4. Независимые от ЯП типы данных стандарта ISO/IEC 14044

РИС. 4.5. Объявление типов данных в стандарте ISO/IEC 14044

В стандарте предусматривается преобразование типов данных ЯП и спецификаций в LI-типы данных. Имеются примитивные типы данных, которые не генерируются из внутренних типов данных ЯП, и генераторы типов данных, используемые при создании новых типов и объектов с новыми типами данных.

4.4.1. Виды преобразований типов данных

Каждая система программирования имеет свои типы данных и методы их реализации. Программы в ЯП реализуются этими системами и не требуют дополнительного преобразования в рамках одной среды. Как только программа, написанная в одном языке, обращается к программе на другом ЯП, могут возникнуть коллизии при различающихся типах данных. LI-язык – посредник преобразования данных между ЯП и этим языком, в нем осуществляются следующие виды преобразования:

– внешнее преобразование из внутренних типов данных ЯП в LI-типы данных;

– внутреннее преобразование из LI-типа данных в тип данных ЯП;

– обратное внутреннее преобразование.

Внешнее преобразование типов данных и генераторов типов данных обладает следующими свойствами:

а) преобразование связывается с одним соответствующим LI-типом данных для каждого примитивного типа сгенерированного внешнего типа данных;

в) преобразование определяет связь между каждым допустимым значением внутреннего типа данных и эквивалентным значением соответствующего LI-типа данных для каждого внутреннего типа данных;

с) для каждого внутреннего значения типа данных преобразование определяет является ли это значение образом какого-либо значения любого LI-типа данных и как он преобразуется.

Данное преобразование должно документировать возникающие аномалии в идентификации внутренних типов данных с LI-типами данных, которые имеются в ЯП и не отличаются от LI-типа данных и значений, недопустимых для ЯП. Внешнее преобразование гарантирует интерфейс между программными компонентами и задается сервисным средством, которое игнорирует окружение ЯП.

Внутреннее преобразование связывает примитивный тип данных или сгенерированный в LI-тип данных с конкретным внут-

ренним типом данных ЯП, имеющим синтаксис и семантику этого языка. Представители LI-типа данных могут преобразовываться в различные внутренние типы данных ЯП. Аналогично, внутреннее преобразование генератора LI-типа данных связывает генератор внутреннего типа данных, синтаксис и семантика которого заданы в ЯП.

Данное преобразование обладает следующими свойствами:

а) для каждого LI-типа данных (примитивного или сгенерированного) преобразование определяет, поддерживается ли этот тип данных ЯП или нет;

в) для каждого поддерживаемого LI-типа данных преобразование определяет отношение между каждым допустимым значением этого типа и эквивалентным значением соответствующего внутреннего типа ЯП;

с) для каждого значения внутреннего типа данных преобразование определяет является ли это значение образом (после преобразования) какого-то значения LI-типа данных и как он преобразуется.

Обратное внутреннее преобразование преобразует значение внутреннего типа данных LI-типа в соответствующее значение типа ЯП путем установления соответствия. Это преобразование для ЯП является коллекцией обратных внутренних преобразований LI-типа данных.

При обоих объявленных типах преобразование предусматривает поддержку свойств: равенства, порядка, ограниченности, кардинальности, свойств точного и приближенного значений LI-типа данных.

4.4.2. Примеры преобразований

Стандарт содержит набор приложений, в которых приведены отдельные общие вопросы проблем преобразования типов данных. Дадим краткую характеристику содержания некоторых его приложений.

В приложении А приведен перечень действующих стандартов (около 40), определяющий наборы символов, которые надо поддерживать ЯП.

Для обеспечения совместимости используемых и реализуемых компонентов ***приложение В*** содержит рекомендации по расположению аннотаций (атрибутов данных, параметров, процедур) при идентификации их типов данных.

Приложении C рекомендует иметь в каждой компьютерной системе свои внутренние типы данных, которые должны преобразовываться в LI-типы данных с соблюдением размера, режима, выравнивания и т.п. для совместимости на разных платформах.

В приложении D отмечается, что синтаксис LI-языка является подмножеством стандарта (ISO/IEC) IDM−95 нотации интерфейса (Interface Definition Notation), которое предназначено для описания интерфейса в LI-языке, как типа данных. Приведен перечень продукций используемых в этом стандарте.

В качестве примера приведен вариант внутреннего преобразования LI-типов данных в типы данных ЯП Паскаль для примитивных типов данных LI-языка (логический, перечислимый, символьный, целый рациональный и др.). Главное замечание касается максимального целого, которое не преобразуется в тип Паскаль и в тип языка MUMPS.

Таким образом, в стандарте ISO/IEC 14044 средства описания типов данных и их методы преобразования являются универсальными, но не имеет общей программной поддержки для всех ЯП.

Рассмотренные программные реализации конкретного преобразования типов данных в ЯП (например, C++ → IDL, IDL→ C++) используются практически.

4.5. ПРОБЛЕМЫ НЕОДНОРОДНОСТИ БД И ПОДХОД К ПРЕОБРАЗОВАНИЮ ДАННЫХ

Одной из важных проблем модернизации информационных систем является перенос данных, хранящихся в БД старых систем, в БД новых систем. При переносе возникают проблемы, связанные с различием логических структур данных, используемых СУБД, изменениями, внесенными в БД [9,10], а также необходимостью сохранения пользовательского интерфейса и большого количества отчетов, созданных на старой СУБД.

Основные проблемы, влияющие на замену данных в БД при модернизации системы, следующие.

1. Многомодельность представления данных в различных БД (иерархические, сетевые, реляционные модели) и СУБД;

2. Различия в логических структурах данных, в справочниках и классификаторах, в системах кодирования информации;

3. Поддержка различных языков для представления текстовой информации;

4. Разные СУБД и постоянное развитие их БД в процессе эксплуатации информационных систем.

Проблема 1 связана с использованием застарелых БД, имеющих иерархические и сетевые модели, и с постепенным переходом к реляционной модели данных и СУБД, обладающие более мощным математическим аппаратом, теории множеств и математической логики. Данные представляются в виде кортежей-таблиц, которые более надежные и простые в употреблении. Реляционная модель данных состоит из структурной, манипуляционной и целостной частей. В структурной части реляционной модели фиксируется структура данных в нормальной *n*-арной форме, а манипуляционная часть базируется на реляционной алгебре и реляционном исчислении и поддерживается SQL-языком. В целостной части формируются требования к целостности, которые поддерживаются любыми реляционными СУБД, а также целостности по ссылкам или внешним ключам. Когда проводится замена иерархических или сетевых моделей, целостность в которых не поддерживается, на реляционные модели данных, возникает угроза нарушения целостности данных и невозможность переноса повторяющихся в столбцах данных.

Проблема 2 связана с тем, что логическая структура данных представляет собой концептуальную схему БД, в которой описаны основные объекты БД и связи между ними. При изменении предметной области, переход на новую СУБД предполагает проектирование новой структуры БД, которая может пересекаться со старой БД, могут появиться новые объекты, требующие ввода новых столбцов или изменения связей между таблицами.

При замене СУБД важную роль играют справочники и классификаторы. Справочники содержат однотипные объекты с уникальными индексами и связывают между собой объекты в различных частях БД. Классификаторы содержат характеристики различных объектов БД и при изменении логической структуры БД меняется справочная информация, а именно, текстовые данные и классификаторы.

При переносе данных могут возникнуть различные искажения из-за различных систем кодировок данных на новых компьютерах в англоязычной кодировке. Для работы с текстовой русской и украинской информацией существует несколько кодировок, которые должны учитываться при переносе данных в новые БД.

Проблема 3 определяется разноязычными текстовыми представлениями информации в БД. В старых БД используется, как правило, один язык, а в новых может быть несколько. В новой

БД необходимо организовать хранение данных таким образом, чтобы обеспечить более простой доступ к текстовым данным с помощью справочников для установления соответствия текстовых данных, записанных на разных языках.

Проблему 4 можно сформулировать как факт хранения и обработки разных данных, вызванных спецификой СУБД иерархического, сетевого и реляционного типов. Наличие явной несовместимости типов и структур моделей данных, различные языки манипулирования данными приводят к тому, что нельзя сгенерировать на языке старой СУБД скрипты по переносу данных с последующим запусков этих скриптов в среде другой СУБД. Каждая СУБД при внесении изменений в БД может менять концептуальную модель, если в нее вносятся новые объекты. Внесенные изменения должны отображаться в справочниках и классификаторах.

4.5.1. Основные этапы преобразования данных и приложений

Учитывая приведенные проблемы, рассмотрим пути их решения. Отметим, что промышленная эксплуатация систем, работающих с БД, может продолжаться достаточно долго. При этом изменяются прикладные программы, работающие с БД, повторно преобразуются данные, если в систему введена новая БД, а часть ранее определенных данных уже перенесены в новую БД. Это влечет за собой доработку прикладных программ доступа к данным, чтобы приспособить их к измененной структуре новой БД или к старой БД. Для переноса данных из старой БД в новую разрабатываются скрипты с приведенной логической структурой БД или DBF-файлы, которые вначале размещаются в транзитной БД, а затем с учетом особенностей новой основной БД переносятся в нею. Может оказаться, что процесс приведения структур транзитной БД к новой окажется нецелесообразным, тогда разработку новой БД нужно проводить «с нуля». При этом справочники и классификаторы дополняются появившимися новыми данными.

В разных СУБД данные имеют различные способы хранения, среди которых могут быть несовместимые типы данных, доступ к данным осуществляется разными языками манипулирования данных, используемых СУБД.

Преобразование данных может проводиться путем создания специальных скриптов и файлов с учетом ранее введенных данных, дублирования данных и корректного приведения несовместимых типов данных При этом устраняются ошибки, связанные с

изменением форматов данных, дополнением старых справочников новыми данными и т.п.

Этапы преобразования данных. Процесс преобразования данных включает перенос данных между СУБД, обработку данных в транзитной базе, перенос данных из транзитной базы в основную базу данных.

Этому процессу соответствуют этапы преобразования данных, которые приведены в табл. 4.2.

ТАБЛИЦА 4.2. Характеристика этапов преобразования данных

Этап преобразования данных	Входные данные	Выходные данные
1. Экспорт данных из старой БД	Старая БД	Транзитные файлы в DBF- или SQL-формате
2. Преобразование данных к кодировке новой БД	Транзитные файлы с данными в старой кодировке	Транзитные файлы с данными в новой кодировке
3. Импорт данных из транзитных файлов в транзитную БД	Транзитные файлы в DBF- или SQL-формате	Заполнение таблиц транзитной БД
4. Приведение соответствия структур старой БД и новой структуры БД	Структуры транзитной БД и основной БД	Скрипты переноса данных из транзитной БД в основную БД
5. Приведение в соответствие кодов справочников	Данные справочников транзитной и основной БД	Соответствие кодов справочников старой и основной БД системы
6. Импорт данных из транзитной БД в основную БД	Данные таблиц транзитной БД	Заполнение таблицы основной БД и справочников новыми значениями
7. Проверка правильности преобразования данных	Данные таблиц и справочников основной БД	Сводная информация, полученная с помощью прикладных программ

В таблице учтены все этапы преобразования данных при переносе данных с одной БД в другую БД, а также соответствующие им входные данные для преобразования и полученные на каждом этапе выходные данные для новой БД.

К задачам модернизации информационной системы, работающей с БД, относятся:

1. Замена СУБД, сохранение структуры БД и пользовательских приложений с небольшой их модернизацией.

2. Замена как СУБД, так и пользовательских приложений с сохранением структуры БД.

3. Замена СУБД, пользовательских приложений и модернизация структуры БД.

Первый вариант замены представляет собой наиболее безболезненный вариант для пользователей и разработчиков.

Второй вариант замены представляет собой создание нового проекта системы на основе имеющейся модели данных. При третьем варианте создается система заново и в новую БД заносятся унаследованные данные из старой БД.

В случае 1 и 2 структура старой БД сохраняется и никакого преобразования данных и соответствия справочников и классификаторов не требуется – они используют единый формат хранения данных.

4.5.2. Унифицированные файлы для передачи данных между БД

Проблема замены, переноса и преобразования данных БД между разными СУБД решается с помощью (рис. 4.6):

1) специального драйвера (две СУБД соединяются друг с другом и напрямую передают данные);

2) транзитных файлов, в которые копируют данные и переносят в новую транзитную БД.

В 1) случае две СУБД соединены напрямую и передают данные, используя интерфейс или драйвер. Иными словами, они применяют специальные программы взаимодействия двух СУБД, при которых вторая СУБД понимает результаты выполнения запросов на языке манипулирования данными первой СУБД, и наоборот. Данные на выходе первой СУБД являются данными на входе второй СУБД в языке манипулирования данными второй СУБД, именно такие данные могут быть внесены в транзитную БД.

Во 2) случае из старой БД данные переносятся в транзитные файлы, SGL-скрипты, DBF-файлы с заранее заданными форматами данных, которые пересылаются через сеть в транзитную БД.

Данные из транзитных файлов с помощью специальных утилит или средств новой СУБД переносятся в транзитную БД для дальнейшей обработки. Если вторая СУБД реляционного типа, то данные в транзитных файлах должны быть представлены к табличному виду. Если первая СУБД не реляционная, то данные должны быть приведены к табличному виду и первой нормальной форме.

Дальнейшая нормализация данных и приведение их к структуре новой БД осуществляется в транзитной БД. Существуют пять нормальных форм задания структур данных, чаще всего используется 3-я или 4-я нормальная форма. Каждая более высокая форма нормализации содержится в качестве подмножества более низкую форму первой нормальной формы, содержащей скаляр-

ные значения. Иными словами, отношение находятся в первой нормальной форме, если они хранятся в табличном виде (все ячейки в строке таблицы расположены в строго определенной последовательности), и каждая ячейка таблицы содержит только атомарные значения.

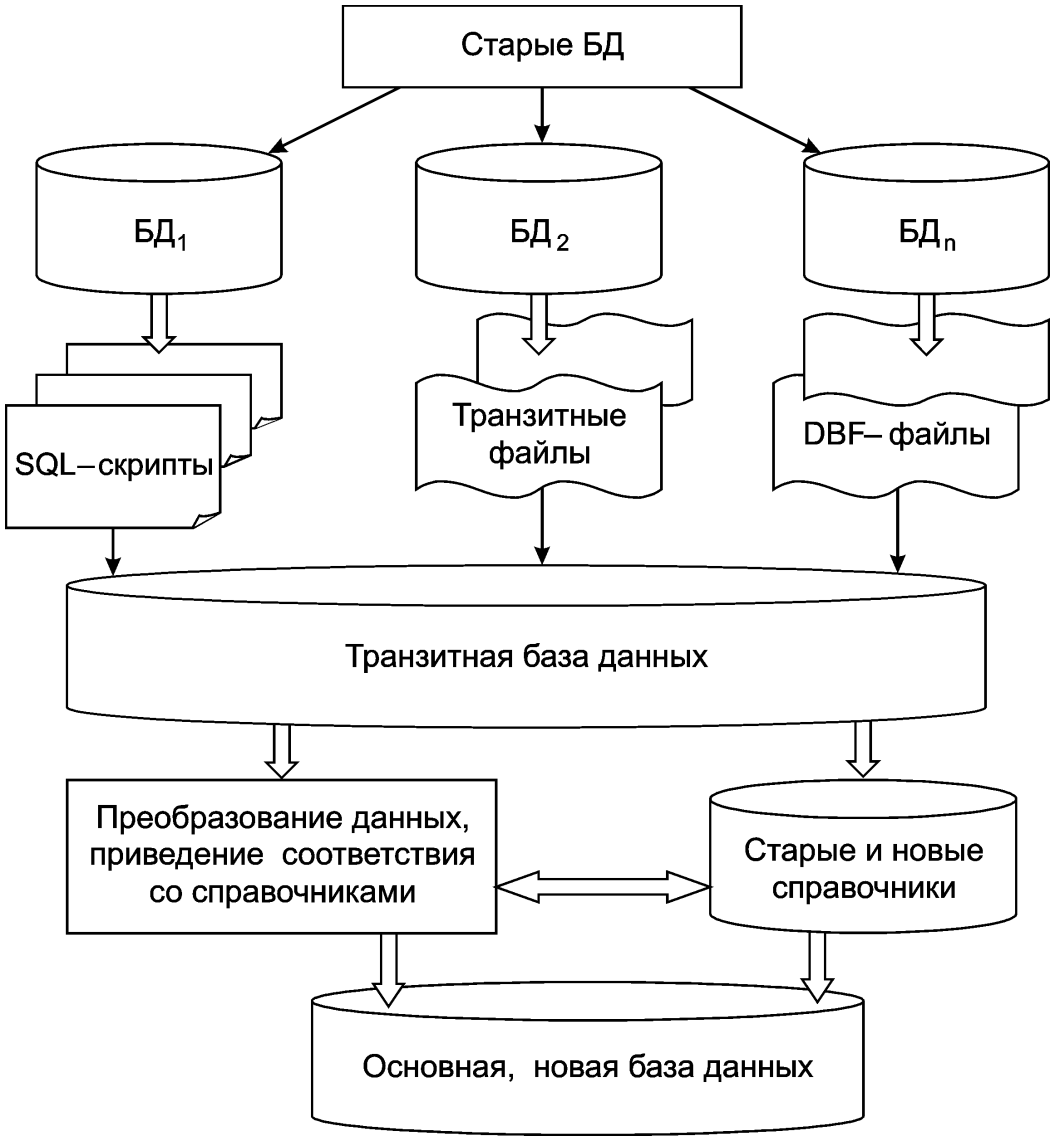

РИС. 4.6. Процесс преобразования и формирования новой БД из старых БД

Отношение находится в *третьей нормальной форме* тогда и только тогда, когда каждый кортеж состоит из значения первичного ключа, которое идентифицирует некоторую сущность, и набора пустых значений или значений независимых атрибутов, описывающих эту сущность. Иными словами, отношение находится в третьей нормальной форме, когда неключевые атрибуты являются взаимно независимыми и зависят от первичного ключа.

Неключевой атрибут – это атрибут, который не задействован первичным ключом рассматриваемого отношения.

Два или несколько атрибутов являются взаимно независимыми, если ни один из них не зависит функционально от какой-либо комбинации остальных атрибутов. Подобная независимость подразумевает, что каждый атрибут может быть обновлен независимо от остальных.

Процесс нормализации отношений позволяет избавиться от проблем, которые могут возникнуть при обновлении, внесении или удалении данных, а также при обеспечении целостности данных.

Структуры старых баз данных, не всегда приводятся к третьей нормальной форме, поэтому требуется, чтобы данные, находящиеся в транзитных файлах, существовали хотя бы в первой нормальной форме и относились к реляционной модели.

В качестве унифицированного формата транзитных файлов используется формат DBF-файлов, поскольку многие СУБД, такие как DB2, FoxPro и некоторые другие хранят данные в таких файлах, тем самым не требуется начальный перенос данных из старой СУБД в транзитные файлы. Большинство СУБД, формат хранения данных которых отличается от формата DBF-файлов, снабжены утилитами или драйверами, которые позволяют перенести данные в такой формат.

4.5.3. Приведение данных транзитной БД к единой системе кодирования

При переносе данных на другую СУБД перевод текстовых данных в другую кодировку можно осуществить путем переноса:

– данных из старой БД в транзитные файлы;
– данных в транзитные файлы;
– данных из транзитных файлов в транзитную БД;
– данных из транзитной БД в основную.

Первый и третий пути приведения текстовых данных к правильной кодовой странице идентичны в реализации. Аналогичны второй и четвертый пути с той лишь разницей, что во втором способе изменения кодов страницы и всех манипуляций над данными осуществляются специальными утилитами, которые распознают формат транзитных файлов и работают с их данными, как с текстовыми данными.

При внесении данных в транзитную базу иногда можно поменять кодовую станицу на страницу, используемую в новой

СУБД. Например, чтобы специальные утилиты, входящие в комплект СУБД Oracle, распознали какие столбцы таблицы, хранящейся в DBF-файле, необходимо их перенести в таблицу БД Oracle, и в специальные DBF-файлы записывается кодировка хранения данных. По этим DBF-файлам утилита переноса данных SQLLoader перенесет данные и автоматически исправит кодировку на кодировку Oracle.

Можно исправить кодировку в транзитной БД на примере той же СУБД Oracle. Для того, чтобы изменить кодировку в Oracle с кодовой страницы 866 (MC-DOS кодировка на русском языке) на 1251 (WINDOWS кодировка, на русском языке), необходимо обновить текстовые данные каждой таблицы, где используется национальный язык.

Преобразование структур данных в транзитной БД. При замене СУБД, пользовательских приложений и одновременного преобразования структуры БД приходится проектировать новую БД, отвечающую требованиям к концептуальному и внешнему уровню представления данных. В старой БД анализируются поля, которые соответствуют полям в новой базе, а затем данные переносятся в новую базу и сопоставляются со справочниками, указывающие соответствие кодов в старой и новой БД. Таким образом, при переносе данных в основную базу некоторые коды, используемые в новой БД, подставляются из этих справочников соответственно.

Структура новой БД должна быть разработана еще до начала преобразования данных. Проектирование БД проводится с учетом правил нормализации, а также обновления, изменения или удаления данных. Кроме того, обязательным условием проектирования БД является обеспечение целостности.

Для новой БД разрабатывается внешний интерфейс, через который происходят запросы к базе от пользовательских клиентских приложений и ответы на данные запросы. Такими интерфейсами, например, в СУБД Oracle является Views – виртуальные таблицы, созданные из частей реальных таблиц. При формировании новой БД применяется также информация, переданная от разных старых БД в транзитную БД в виде транзитных файлов, DBF-файлов и SQL-файлов. На основе справочников эти файлы в транзитной БД преобразуются к структуре новой БД.

В главе были представлены результаты исследований и разработок по форматному, теоретическому и системному преобразованию данных, передаваемых между взаимодействующими объектами в сетевой среде, а также преобразование данных в БД в

связи с переходом на новые БД. Рассмотрены различные методы преобразования типов данных современных ЯП, дополнительные возможности современных систем и сред по обеспечению взаимодействия.

Рассмотрен стандарт ISO/IEC 11404–96 о независимости типов данных от ЯП и средства Li-языка для преобразования типов данных во внешнее и внутреннее представление ЯП и обратно. Практический интерес представляют примеры реализации двух пар ЯП: Li-язык ↔ Паскаль и Li-язык ↔ MUMPS.

Проанализированы проблема замены старой БД на новую и способы преобразования данных.

ГЛАВА 5

МЕТОДЫ ЭВОЛЮЦИИ ПРОГРАММНЫХ СИСТЕМ

В связи с тем, что каждые 8–10 лет появляются новые архитектуры компьютеров, ЯП и операционные среды, возникает необходимость эволюционного развития программ и систем в целях адаптации их к новым компьютерам или к новым средам. Сами по себе работающие ПС на старых компьютерах постоянно эволюционирует из-за исправлений, вносимых после обнаруженных ошибок, или из-за постоянного расширения функциональности, улучшения качественных показателей и т.п. Процесс эволюционного развития ПС составляет, как утверждают некоторые эксперты, 60–80% общей стоимости разработки ПС.

Термин ***эволюция*** в программной инженерии можно сформулировать как постепенное изменение программ и ПС в целях усовершенствования и улучшения качества. Однако после проведения изменений в зависимости от поставленной цели качество может снизиться либо повыситься. Если поставлена абстрактная цель – повысить качество ПО, то эта цель может быть не достигнута, если не были детально выявлены причины низкого качества и не определены пути его повышения [1–7].

Каждое внесенное изменение в разрабатываемую или разработанную систему усложняет ее настолько, что в ней трудно разобраться, особенно, если внесено много изменений в виде отдельных латок, вставок в исходный текст ПС и они не зафиксированы в моделях или документах спецификаций требований, то система становится непонятной самим разработчикам. Эволюция такой системы практически невозможна даже ее разработчикам и она должна быть утилизирована.

Одним из путей снижения сложности изменяемой ПС является отметка специальных мест (точек) в спецификациях требований для дальнейшего внесения предполагаемых изменений на этапах ЖЦ. Этим точкам могут соответствовать предусмотренные заранее разные виды интерфейсов или функций для удовлетворения потребностей нескольких категорий пользователей.

5.1. ОСНОВНЫЕ ЗАДАЧИ ЭВОЛЮЦИИ ПС

К увеличению сложности и снижению качества ПС могут привести ошибки в спецификациях требований, архитектуры и реализации кода программного продукта (рис. 5.1), а также желание к усовершенствованию путем изменения функциональности, добавления некоторых «бантиков» или удаления ненужных функций по просьбе заказчика.

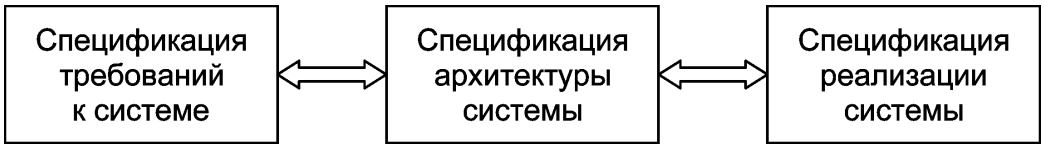

РИС. 5.1. Виды спецификаций программной системы

Обнаружение ошибки в одной из приведенных спецификаций влечет за собой соответствующие изменения в другие спецификации. Если ошибки обнаружены в реализации системы (при тестировании, верификации и др.), то проводится согласование проведенных изменений во все спецификации путем анализа разных видов документации на этапах ЖЦ и принятия решения об окончательной их модификации. Восстановление и обновление спецификаций, как правило, осуществляется с помощью обратной инженерии.

Причиной изменений являются разного рода несогласованные действиями между разработчиками проекта на этапах ЖЦ, недостаточно серьезное отношение к требованиям заказчика, несоблюдение стандартных положений при достижении качества и др.

Кроме того, промежуточные продукты на этапах ЖЦ (артефакты, спецификации требований и др.) связаны между собою так, что изменение любого из них требует пересмотра других для определения влияния проведенного изменения и дальнейшей корректировки системы. Эта взаимосвязь является главной трудностью управления изменениями как во время проектирования ПС, так и при сопровождении.

Во время эксплуатации системы обнаруживаются отказы и дефекты, которые не были выявлены на этапе тестирования, или оказалось, что некоторые требования не отвечают потребностям пользователей или заказчик провел модернизацию оборудования, обновление среды функционирования и др. Учитывая сказанное, необходимо:

– проверка степени реализации сформулированных требований к функциям, интерфейсам и показателям качества в промежуточном продукте на каждом этапе ЖЦ;

– проверка правильности функционирования ПС путем тестирования для обнаружения разных ошибок, дефектов и отказов для последующего их устранения;

– классификация обнаруженных ошибок, разработка требований к процессу их исправления и составление плана проведения изменений компонентов ПС;

– определение метода и стратегии управления процессом внесения изменений в ПС или в отдельные компоненты.

Управление изменениями ПС, как способ эволюционного развития (модернизации), базируется на составленном плане, в котором распределены работы по этапам ЖЦ для замены, добавления или модификации отдельных функций ПС, интерфейсов компонентов и системы в целом, а также для проверки полученных показателей качества после внесения изменений и т.п. Иными словами, процесс управления изменениями ПС – ключевой в достижении необходимого качества разработки ПС и важное направление программной инженерии [1, 2], ориентированное на полную или частичную эволюцию ПС с обеспечением требуемой функциональности и качества

Этот процесс выполняется с помощью реинженерии, рефакторинга и реверсной инженерии. Особенности их применения для эволюции программ излагаются ниже.

5.2. ФОРМАЛЬНЫЕ ОСНОВЫ ЭВОЛЮЦИИ КОМПОНЕНТОВ И ПС

К основам эволюции компонентов относятся операции, которые выполняются при разработке и сопровождении, и включают в себя:

– добавление компонентов (Sum) в ПС;

– объединение компонентов (Link);

– создание экземпляра компонента (Create);

– удаление экземпляра компонента или самого компонента (Del) и др.

Операция добавления компонента – это пополнение состава компонентов новыми реализациями и интерфейсами, что соответствует расширению функциональных возможностей, а также уточнению конфигурации (версии) системы. Операции *создания* и *объединения* компонентов обеспечивают образование интегрированных структур ПС.

Операция *удаления* компонента способствует изменению структуры ПС путем извлечения его из этой структуры и обязательной корректировки ранее установленных связей и отношений в ней.

Для определения особенностей изменения компонента, рассмотрим абстрактную модель компонента (*C*):

$$Mc = (Cn, Ci, Ce, Cr, Cs), \quad (5.1)$$

где *Cn* – уникальное имя компонента;

Ci – интерфейсы компонента;

Ce – интерфейс управления экземплярами;

Cr – реализации компонента;

Cs – системный сервис, необходимый для функционирования и взаимодействия с компонентной средой.

Уникальное имя компонента гарантирует избежание возникающих коллизий не только в пространстве имен компонентной среды, а и применения.

Интерфейс *Ci* состоит из:

– интерфейса, который реализуются в среде данного компонента и его метода;

– интерфейса, реализованного в других компонентах, функциональность которых необходима для выполнения методов данного компонента.

Каждый интерфейс включает в себя имя, функциональность (совокупность методов), описание типов, констант и сигнатур методов.

Реализация компонента – это совокупность методов и типов данных для передачи или приема параметров, которые возвращаются после выполнения метода. Она отображает функциональность, спецификацию, условие выполнения, параметры настройки и т.д. По сигнатуре методов и типам данных происходит сопоставление реализации интерфейсов.

Составным элементом компонента является экземпляр, над которым выполняются операции O_e локализации (*Locate*), образования (*Create*), удаления (*Del*) и перемещения *(Remove*), т.е. $O_e = \{Locate, Create, Del\}$.

Необходимым требованием для проведения изменений компонента является условие целостности и обеспечение его не только необходимым интерфейсом, но и теми интерфейсами, которые связаны с другими компонентами.

Компоненты выполняются в компонентной среде с помощью операций: инсталляции (*ins)*, удаления и замещения: $O_к$ = (ins, Del, Remove, Deput).

Обозначим множество всех операций взаимодействия компонента со средой так:

$$\Omega_1 = \{ \{O_э , O_к \}, C \},$$

где $\{O_э , O_к \}$ – множество операций обработки компонентов и их экземпляров в компонентной среде.

В общем, методы эволюции предназначены для постепенного и систематического изменения компонентов, их интерфейсов и ПС и основываются на проведении [4, 5]:

– анализа исходного кода для внесения в него изменений;

– адаптации компонентов и систем к новым условиям среды и платформы;

– кодирования и декодирование данных при обмене данными между компонентами, расположенными на разных платформах;

– изменения функций путем добавления, удаления или замены новыми;

– расширения возможностей (сервиса, связей и т.п.) компонентов;

– модификации спецификации требований, структуры системы или компонентов.

Содержание всех изменений в компонент и его совокупности состоит в улучшении ПС, в продлении жизни наследуемых, стареющих программ и ПИК. С теоретической точки зрения методы эволюции недостаточно изучены, а с практической – изменения постоянно вносятся в ПС.

Например, проблема 2000 года – изменение даты 2000 года во всех ПС и прикладных системах. В ее изменении приняли участие широкий круг специалистов, в частности из оффшорных зон (Индия, Россия, Украина и др.) путем поиска места расположения этой даты в ПС и замены ее новой датой для сохранения жизнеспособности множества программ или адаптации к новым возможностям ОС, ЯП и платформ современных компьютеров.

Эволюция включает в себя внешние методы, как способы обработки компонентов в распределенной среде, и внутренние методы, обеспечивающие изменение компонентов (Com), интерфейсов (Int) и/или системы. К внутренним методам эволюционного развития ПС отнесены методы реинженерии, рефакторинга и реверсной инженерии (рис. 5.2), с помощью которых проводится целенаправленное изменение программ или систем по разным причинам.

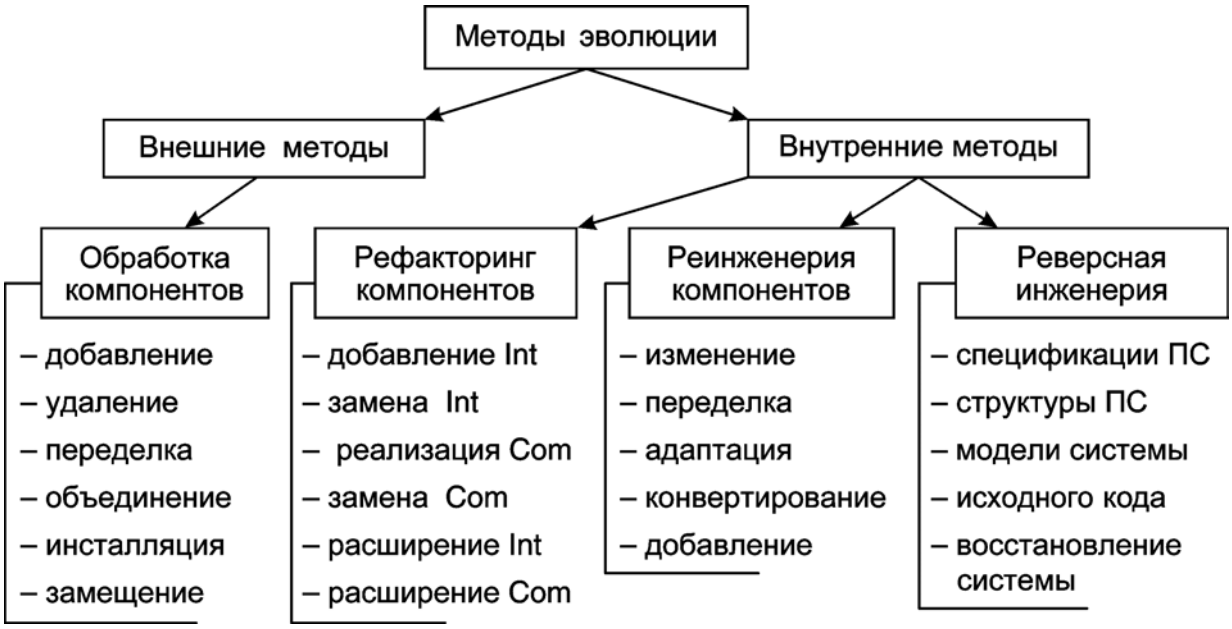

РИС. 5 2. Общая структура эволюционного развития компонентов ПС

5.3. МЕТОД РЕФАКТОРИНГА КОМПОНЕНТОВ

Одним из методов эволюции компонентов является *рефакторинг*, под которым понимается совокупность моделей, методов и средств изменения структурных и качественных характеристик компонентов для получения новых компонентов, наиболее пригодных для работы компонентной системы.

Рефакторинг компонентов получил развитие в ОО-программировании в связи с применением в нем каркасов проектирования, методов улучшения кода и необходимости изменения компонентов и их интерфейсов [5–9]. В результате были разработаны целые библиотеки типичных преобразований объектов (классов), улучшающие разные характеристики компонентной ПС. Метод ориентирован на изменение (замену, замещение, расширение) реализаций и интерфейсов компонентов с учетом требований к системе и к ее конфигурации.

Каждая операция рефакторинга является базовой, атомарной функцией преобразования, которая сохраняет целостность компонента, т.е. совокупность правил, ограничений и зависимостей между составными элементами компонента, задающих компонент как единую и цельную структуру. Цель рефакторинга – сохранить функции и идентичность компонента. Добавление новых функций, характеристик и свойств проводится с помощью реинженерии.

Рефакторинг базируется на типичности структуры, модели компонента, независимости интерфейсов от методов реализации и доступа к экземплярам компонентов, системным сервисам и классам объектов.

Систематическое исследование особенностей применения рефакторинга к компонентам и классам представлено в [5]. Анализ рефакторинга проведен на уровне данных, отдельных методов, реализации классов, интерфейсов классов и программной системы в целом. Для каждого из этих уровней сформулированы основные задачи, которые непосредственно связаны с процессом внесения изменений в компоненты, классы, ПС и приведены в табл. 5.1.

ТАБЛИЦА 5.1. Виды рефакторинга и решаемых задач на уровнях

Виды рефакторинга	**Задачи рефакторинга на уровнях проектирования ПС**
На уровне данных	– замена имени переменной более информативным; – замена выражения на вызов метода; – преобразование элементарного типа данных и кодов в класс; – преобразование массива в класс; – замена традиционной записи на класс данных и др.
На уровне отдельных методов	– извлечение метода из другого метода; – преобразование объектного метода в класс; – замена сложного алгоритма на простой: – отделение операций запроса от операций изменения данных; – объединение похожих методов путем параметризации; – передача в метод объекта отдельных полей; – разделение методов поведения в зависимости от параметров и др.
На уровне реализации классов	– замена объектов-значений на объекты-ссылки, и наоборот; – перемещение специализированного кода в подкласс; – объединение похожих кодов в суперкласс; – замена виртуальных методов инициализацией.
На уровне интерфейсов классов	– перемещение метода в другой класс; – разделение класса на несколько; – удаление класса, посредника и методов установки значений неизменяемых полей; – замена наследования на делегирование и наоборот; – создание внешнего метода и класса-расширения; – скрытие методов и инкапсуляция неиспользованных методов; – объединение суперклассов и подклассов с похожей реализацией и др.
На уровне системы	– изменение связей (однонаправленных в двунаправленные и наоборот) между классами; – использование метода вместо конструктора; – замена кодов ошибок на исключения; – создание стандартного источника данных и др.

Каждой из приведенных задач, выполняемых при эволюционном развитии ПС, соответствует операция рефакторинга, сохраняющая целостность компонента и удовлетворяющая следующим условиям:

1) объект, полученный в результате рефакторинга, является компонентом с соответствующими свойствами, характеристиками и типичной структурой;

2) компонент не теряет своей функциональности и может примениться в компонентных системах;

3) полученные новые компоненты однозначно идентифицируются.

Операци рефакторинга сгруппируем в следующие базовые операции:

$$O_{Refac} = \{AddOImp, ReplImp, AddInt , Cfact\},$$

где AddOImp – добавление операции для изменения существующего интерфейса и получения нового с условием целостности компонента. Эта операция является ассоциативной и коммутативной, доказательство этого вытекает из анализа множеств интерфейсов и реализаций, входящих в состав компонентов с использованием ассоциативных и коммутативных свойств операций над множествами.

ReplImp – операция замены существующей реализации компонента новой с эквивалентной функциональностью и новым интерфейсом, операция – ассоциативная и коммутативная;

AddInt – операция добавления нового интерфейса к старому компоненту и новой его реализации, является частичной операцией;

Cfact – операция расширения интерфейса в связи с новыми системными сервисами компонентной среды, сохраняется структура компонента, а некоторые интерфейсы изменяются в связи с их новой семантикой. Данная операция является комплексной и относится к внутренним операциям.

Используя базовый набор операций, определяем *модель рефакторинга компонентов*:

$$M_{Refac} = \{O_{Refac} , \{Cs = \{NewC\}, \Omega_1\}, \qquad (5.2)$$

где O_{Refac} и $\{Cs = \{C_n\}\}$ – операции и элементы из множества компонентов этой модели, Ω_1 – множество операций обслуживания компонентов из С.

Рефакторинг в широком смысле означает изменение компонентов и интерфейсов, а не их среды. Например, более простым методом для компонента является включение нового сервера в среду, чем переделка этого компонента под действующий сервер.

5.4. МЕТОД РЕИНЖЕНЕРИИ КОМПОНЕНТОВ

Реинженерия компонентов – это совокупность моделей, методов и процессов изменения структуры, функций и возможностей компонентов ПС для получения новых функциональных возможностей ПС в соответствии с функциональными и нефункциональными требованиями к ПС.

Метод реинженерии целесообразно использовать при частичной или полной переделке отдельных функций, компонентов, некоторых фрагментов старых программ на новые при переходе на новые условия ОС, платформу и высокие требования к качеству продукта или процессов ЖЦ.

При неудовлетворительном качестве ПС могут вводиться новые характеристики качества, которые не были заданы в спецификациях требований, что приводит к пересмотру не только спецификаций требований, но и принципов повторной реализации измененных требований на всех этапах ЖЦ. Гарантией требуемого качества на этапах ЖЦ является система обеспечения качества (стандарт ISO/IEC 9126, 12207), в рамках которой выполняются процедуры контроля и оценки заданных показателей качества (например, функциональность, корректность и др.).

Реинженерия может использоваться при обеспечении мобильности ПС, т.е. способности компонента адаптироваться к новой платформе путем настройки параметров или небольших переделок ПС, связанных с новыми условиями среды или архитектуры платформы. При этом изменение компонентов или ПС может быть выполнено одним из способов:

1) реорганизация или изменение спецификаций компонента;

2) замена языка описания компонента новым ЯП;

3) модификация или модернизация функций (методов) компонентов и их данных;

4) изменение связей между компонентами, которые располагаются в разных средах;

5) адаптация компонентов к новым условиям среды функционирования и др.

При этом архитектура компонентной системы может оставаться неизменной, если модификация касается только отдельной функции системы или замены ее другой.

Если архитектура системы не изменяется, а решается задача преобразования централизованной системы в распределенную, то эти аспекты могут потребовать изменения компонентов в целях

внесения в них операций взаимодействия с соответствующей средой. Сложно изменить ЯП старой наследуемой системы новым языком программирования, сохраняя при этом ее структуру и функциональность. Такая задача часто возникает при замене ЯП наследуемой системы на объектно-ориентированный язык (JAVA, PASCAL, C++ и др.).

Если в системе изменяется полностью функции и структура, то этот процесс является дорогостоящим и сопровождается большим риском, что требуемый результат не будет достигнут. Однако с коммерческой точки зрения реинженерия является единым способом сохранения наследуемых систем для продолжения эксплуатации [5].

С помощью реинженерии совершенствуется системная структура, создается новая документация и конфигурация, чем обеспечивается продолжение времени жизни наследуемой системы.

По сравнению с более радикальными подходами к совершенствованию систем реинженерия имеет такие основные преимущества:

1. *Снижение риска* за счет спланированной деятельности по внесению изменений, особенно при разработке ПС из готовых компонентов, когда вероятность внесения ошибок резко снижается.

2. *Снижение затрат* за счет более низкой себестоимости готовых компонентов, которые выбираются из репозитария готовых ресурсов, чем разработка новых (или заново) в создаваемой системе. В соответствии с данными разных коммерческих фирм реинженерия системы в четыре раза дешевле, чем повторная новая разработка такой же функциональности системы.

Процессы реинженерии применяются при смене деловых процессов в целях уменьшения лишних видов деятельности или повышении эффективности отдельных бизнес-процессов. Зависимость делового процесса от наследуемой системы будет минимальной, если заранее спланировать изменения в бизнес-процессах в системе в целом. Количество изменений значительно уменьшится, если система адаптируется только к новой среде функционирования.

Основное различие между реинженерией и новой разработкой ПС состоит в том, что описание системной спецификации начинается не с «нуля», а с готовой спецификации, требующей ее восстановления после внесенных изменений и дополнения для ее использования в качестве основы для разработки новой системы.

К основным этапам этого процесса относятся:

1) перевод исходного кода путем переописания компонентов в старом ЯП в современную версию этого языка или в другой, новый ЯП;

2) анализ документов описания структуры и функциональных возможностей системы для принятия проектных решений;

3) модификация структуры компонентов в целях упрощения, интеграции и наращивания новых свойств;

4) декомпозиция системы на более мелкие компоненты, их группирование для устранения чрезмерной сложности и композиции их компонентной структуры из отдельно сформированных компонентов;

5) изменение системных данных, с которыми работают компоненты системы, для установления их соответствия с проведенными изменениями.

Преобразование исходного кода программ – это наиболее простой способ реинженерии, особенно если оно проводится автоматически или автоматизировано путем перевода исходного кода в одном ЯП в другой, более современный ЯП.

Автоматизированный перевод оказывается невозможным, если структурные конструкции исходного ЯП имеют различия со структурами программ и типами данных в новом языке. Например, язык содержит встроенные условные команды компиляции, не поддерживаемые в новом ЯП. В этом случае преобразование ранее созданной системы проводится вручную и требует значительных технических, человеческих и финансовых ресурсов.

Автоматизированный перевод программы в одном ЯП на другой требуется при:

– изменении аппаратных средств платформы, на которой может не выполняться компилятор исходного переносимого языка;

– недостаточном числе квалифицированного персонала для сопровождения системы, описанной в некоторых специфических языках, которые вышли из употребления;

– изменении организационной структуры ПС и переходе на общий стандартный ЯП, в котором решаются проблемы неоднородности типов данных.

В целом такой автоматизированный перевод снижает затраты на изготовление новой системы и на ее сопровождение.

В процессе разработки системы создается комплексная модель системы для моделирования и выбора вариантов для дальнейшей реализации и модификации с учетом функциональных и нефункциональных требований (к защите, безопасности и др.).

Метод реинженерии – целевой способ эволюционного развития ПС для получения нового компонента или ПС путем:

– анализа исходного кода для определения логики компонента и внесения изменений;

– настройки компонентов системы на изменения спецификаций требований и архитектуры в зависимости от платформы, а также определения стратегии изменений ПС;

– кодирования и декодирования данных компонентов при переходе с одной платформы на другую;

– модификации системы в связи с необходимостью изменения или добавления отдельных функций в систему;

– расширения интерфейсов, сервисов, сценариев работы системы и т.п.;

– частичного или полного изменения отдельных элементов системы методами перепрограммирования.

Реинженерия ПС из компонентов будет более продуктивной, если провести восстановление измененной структуры и логики компонента и произвести новую спецификацию требований и архитектуры системы.

Восстановление программного текста, в который вносились в процессе разработки отдельные исправления, требует установления соответствия с проектными решениями и планами проведения изменений для обновления этого текста.

Операции реинженерии. Среди множества операций реинженерии выделим те, выполнение которых удовлетворяет следующим свойствам:

– объект, полученный в результате реинженерии, это компонент с типичной структурой и со своими функциональными особенностями и характеристиками;

– новый компонент может сохранять свои функции и применяться в построенных системах из отдельных компонентов;

– компонент, полученный в результате реинженерии, отвечает требованиям и ограничениям модели системы.

Операции реинженерии по изменению требований, структуры и адаптации составных элементов системы, выполняются в условиях сохранения существующих функций или их расширения в рамках исходной модели системы, в которой был расположен компонент. К операциям над компонентами относятся следующие:

1) именование компонентов и идентификация компонентов;

2) расширение функций существующей реализации компонента;

3) перевод языка компонента в новый ЯП;

4) изменение структуры компонента или системы;

5) модификация описания компонента и его данных.

Измененные новые компоненты идентифицируются для дальнейшего обращения к ним в целях интеграции в комплексную компонентную систему, создания конфигурации (версии) и каркаса системы.

Модель реинженерии. Будем предполагать, что компоненты разработаны на разных языках и могут взаимодействовать друг с другом через операторы вызова (Call, RMI и др.) или через сообщения, передаваемые по сети. При этом используются общие операции реинженерии:

– переописание текста компонента – rewrite,

– реструктуризация компонента – restruc,

– адаптация (настройка) компонента к новым условиям – adop;

– добавление к множеству компонентов новых функций и компонентов – supp;

– конвертация данных, которые используются компонентами – conver.

Формально множество этих операций обозначим так:

$$O_{Reing} = \{rewrite, restruc, adop, supp, conver\},$$

они изменяют отдельные компоненты системы без изменения их интерфейса, поскольку предполагается, что в качестве интерфейса используется запрос (request) или сообщение к среде, где компоненты располагаются. В запросах задаются параметры и их типы, которые формируются в среде клиента и отправляются через сеть серверу для выполнения соответствующего компонента.

Обозначим *OldC* – старый компонент, который после реинженерии превращается в новый *NewC* с помощью операций множества O_{Reing}, способствующих созданию нового компонента. Все операции реинженерии являются ассоциативными и коммутативными, обеспечивают целостность нового компонента, т.е. $NewC = OldC \cup NewC^n$.

В запросе могут быть заданы условия расширения требований, например, сохранение компонентов в репозитарии компонентов и интерфейсов в репозитарии интерфейсов. Эти условия допустимы в современных средах (CORBA, JAVA и т.п.) [6, 7], в которых операции добавления и расширения интерфейса пополняют репозитарий интерфейсов новыми.

Модель реинженерии компонентов имеет вид

$$M_{Reing} = \{ O_{Reing} , \{C_s = \{NewC\}\},$$

она включает в себя набор операций O_{Reing}, множества C_s процесса реинженерии и обеспечивают изменение компонентов в целях получения нового элемента *NewC*, образованного в процессе реинженерии.

5.5. МЕТОД РЕВЕРСНОЙ ИНЖЕНЕРИИ КОМПОНЕНТОВ

Реверсная инженерия – это процесс восстановления результатов прямой инженерии системы, т.е. отдельных ее элементов – спецификаций требований, архитектуры и реализованного кода в целях построения нового варианта системы на более высоком языковом уровне [1–5].

Этот процесс сводится к восстановлению спецификации ПС, включающей в себя описание структуры и логики компонентов, а также к преобразованию исходного кода путем перепрограммирования системы по измененным спецификациям требований, архитектуры и документации на систему (рис. 5.3).

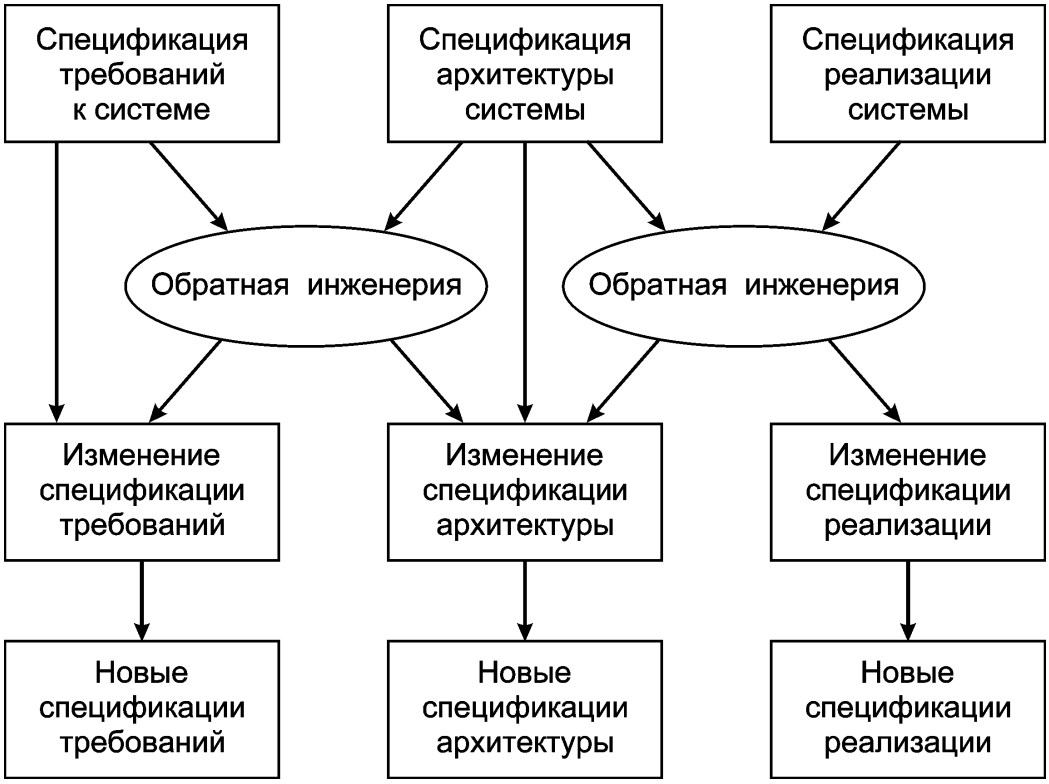

РИС. 5.3. Процесс обратной инженерии описания системы

Поддержка и эволюция действующих устаревших больших ПС зависит от их размера, сложности и зрелости, поскольку в их разработке принимают участие разные специалисты на протяжении продолжительного срока инвестирования их заказчиками.

Такие системы обладают типичными проблемами:

– устаревшие методы реализации наследуемых систем,

– использование ЯП, который вышел из употребления,

– плохо оформленной документацией описания системы.

Эволюция системы, обладающей такими особенностями, технически является очень трудной с точки зрения программирования.

Сложилось два типа задач реверсной инженерии.

Задачи первого типа включают в себя:

– анализ для проведения изменений в структуру кода;

– расширение функциональности ПС;

– замену платформы, ЯП и др.;

– изменение логической структуры, спецификации и кодов;

– применение шаблонов проектирования в качестве инструментов инженерии;

– изменение моделей и структур данных.

Задачи второго типа состоят из использования:

– архитектуры и структуры компонентов с сохранением показателей качества компонентов (надежность, защищенность, совместимость и др.);

– новых современных ЯП, платформ и технологий и методов достижения показателей качества;

– новых подходов к организации взаимодействия пользователей с ПС и средой, а также новых типов интерфейсов, которые обеспечивают оценку показателей качества компонентов и использование в новых условиях среды функционирования;

– новых моделей вычислений для выполнения заданных требований к ПС (экономичность, масштабность и др.);

– новых моделей данных, для которых предъявляются такие требования, как несанкционированный доступ, авторизация, конфиденциальность и т.п.

Подходы к реализации реверсной инженерии. В практике программирования сложились новые подходы к автоматическому или автоматизированному переводу устаревшего исходного кода с одного ЯП на другой. Они выполняются, как правило, с помощью специально созданных программ конвертирования кода с одного языка на другой с учетом разных типов данных, или системы сопоставления с образцом, задаваемом в виде списка команд, необ-

ходимым для перевода из одного языкового представления в другое. Перевод по образцу используется при:

– наличии новых аппаратных средств, на которых не может выполняться компилятор на исходном ЯП наследованной системы, подлежащей изменению;

– недостатке квалификации персонала для внесения изменений в систему или в отдельные компоненты, написанные на ЯП, которые вышли из употребления;

– изменении политики сопровождения ПС в связи с переходом на новую среду и ЯП.

Автоматизированный перевод спецификации системы или отдельных ее частей затруднен, если в описании исходного кода системы нет соответствующих конструкций в новом языке (например, в исходном описании содержатся встроенные условные команды компиляции, отсутствующие в новом языке). В таком случае проводится ручное преобразование системы.

Таким образом, процесс реверсной инженерии ПС состоит из двух этапов:

1) анализ описания компонента в целях получения новой структуры и логики;

2) преобразование действующей логики описания компонентов и ПС в новое описание с некоторыми изменениями.

Первый этап проводится путем нахождения артефактов каждого из компонентов из его программного текста и установления соответствия с проектными решениями для проведения преобразования.

Второй этап – это преобразование программного кода для новой реализации. В случае изменения существующей устаревшей архитектуры выполняется перепроектирование согласно новым требованиям. Осуществляется оценка новой архитектуры для достижения заданных характеристик качества системы и разного вида ограничений. При этом решается проблема перемещения компонентов из предыдущей системы в новую платформу или среду с полной заменой компонентов другими, измененными.

Главным достижением выполненных исследований по методам реверсной инженерии является использование объектно-ориентированного программирования [9–13] с базовыми операциями визуализации (visual), измерения метрик (metric) компонентов ПС и выполнения следующих задач:

– обеспечение высокого качества системы, переоценка сроков, объемов, сложности и структуры разрабатываемой системы;

– определение иерархии классов и атрибутов программных объектов в целях наследования их в новой системе;

– идентификация классов объектов с определением размера и /или сложности классов системы;

– поиск и выбор паттернов, их идентификация, фиксация их места и роли в структуре системы.

Этот подход может использоваться для изменения индустриальных систем (в миллион строк кода) с применением метрик для оценки характеристик системы. Он обеспечивает генерацию тестов проверки кодов, а также проведение метрического анализа системы для получения фактических значений внутренних и внешних характеристик системы.

В результате анализа изменяемой системы, строится модель, содержащая список классов и паттернов системы, которые могут модифицироваться и перепроектироваться в процессе эволюции системы. Если некоторый класс плохо спроектирован (например, много мелких методов) или система не выполняет требуемые функции, то осуществляется сбор информации для уточнения модели ПС.

Действия по визуализации структуры системы средствами UML отражаются на двухразмерном экране в виде иерархического дерева, узлы которого отображают объекты и их свойства, а отношения между ними задаются командами фрагментов программ. При этом в рабочую таблицу заносится информация:

– о метриках классов объектов (атрибутах, подклассах и строках кода),

– о методах объектов (параметрах, вызовах, сообщениях и т.п.),

– об объектах (время, доступ к классу или подклассу и т.п.).

В процессе визуализации ведется сбор метрик ПС, выполняется оценка качества ПС согласно плану переделки унаследованной системы на новую систему.

Модель реверсной инженерии. Во множество операций реверсной инженерии включены:

$$O(Rever)=\{visual, metric, restruc, design, rewrite\},$$

где – *visual* – визуальное представление структуры системы на экране дисплея;

– *metric* – оценка состояния системы по визуальным данным и метрикам, которые используются для реструктуризации системы;

– *restruc* – реорганизация и реструктуризация системы, аналогично операциям реинженерии;

– *design* – построение нового компонента или системы с учетом некоторых положений, унаследованной версии системы, которая не удовлетворяла новым условиям эксплуатации системы;

– *rewrite* – описание языка отдельных компонентов в другой ЯП в соответствии с новыми требованиями.

Модель реверсной инженерии принимает следующий вид:

$$M_{Rever} = \{ O_{Rever}, \{C_s = \{NewO\}\},$$

где O_{Rever}, – операции реверсной инженерии, C_s – множества компонентов для применения к ним операций реверсной инженерии; *NewO* – множество новых объектов после применения операций реверсной инженерии.

Управление всеми видами изменений программных продуктов обеспечивается процедурами слежения.

5.6. МОДЕЛЬ СЛЕЖЕНИЯ ЗА ИЗМЕНЕНИЕМ ПС

Рассмотренные методы эволюции ПС обеспечивают разные виды их изменений, контроль за которыми осуществляет метод слежения, сущность которого состоит в том, чтобы все операции процесса изменения ПС были управляемыми и выполнялись по специальному плану.

Сведения об изменениях и сами изменения должны сохраняться в базе проекта или в репозитарии ПС в виде дистрибутивного файла системы, результатов выполнения формальных процедур оценки внесенных изменений и их влияния на работоспособность ПС. Сохранение результатов изменения в репозитарии выполняется с помощью специальных процедур.

После внесения изменений выполняется техническая оценка и контроль качества изменений, а также их учет и регистрация в специальном файле, в котором фиксируется место изменения компонента, причина изменения, и место хранения каждого изменения в файле.

Если произведено много изменений в ПС, то проводится трассировка выполненных изменений и детальный аудит. Для этих целей используются инструментальные средства, например, Rational's ClearCase, SourceSafe of Microsoft и др. [12, 13].

Причинами внесения значительных изменений в ПС является несоблюдение ограничений на разработку, отклонение от поставленных требований, недостаточное тестирование продукта, переделки в системе без разрешения заказчика, и т.п.

Как свидетельствуют современные данные [13], интенсивность выявления ошибок и их изменений колебаться от 0,002 до 0,005, т.е. в системе оказывается приблизительно одна ошибка, которая фиксируется на протяжении двух месяцев ее эксплуатации.

Пример оценки числа ошибок при опытной эксплуатации OS−360 фирмы IBM по интуитивно подобранной формуле:

$$n = 23m + 2k,$$

где n − общее количество изменений системы, связанных с исправлением выявленных ошибок, m − число модулей с более чем 10 исправлениями в каждом, k − число модулей с 1 или 2 исправлениями. Эта формула характеризует работоспособность OS−360 (370) при каждой новой версии системы на момент ее передачи многочисленным пользователям.

Процессы изменения компонентов и ПС поддерживаются средствами слежения за изменениями компонентов в процессе разработки и сопровождения.

Подход к слежению за изменениями. Поскольку большое количество ошибок связано с изменениями функций, структуры ПС и процессов ЖЦ, подход к слежению за этими изменениями базируется на спецификации структуры, описании функций системы, а также на принятых проектных решениях о целесообразности и направлениях модификации системы по процессам ЖЦ.

Поскольку функции ПС, ее архитектура определяются на этапе требований, то в них должны закладываться решения относительно будущих изменений в виде точек вариантности и методов слежения за изменениями на всех этапах ЖЦ.

Модели слежения за изменениями отображают взаимодействие компонентов ПС с внешней средой и с внутренними операциями функционирования. Они рассматриваются на уровне функций, структуры, процессов разработки и обработки.

Модели слежения на уровне функций содержат функциональные типы решений для конечных пользователей. К основным факторам построения моделей слежения за изменениями на уровне архитектуры относятся:

− внесение новых функций;

− усовершенствование существующих функций за счет изменения данных, фрагментов описания функций и расположения функции в иерархии функций;

− изъятия функций из системы.

К задачам слежения за изменениями относятся:

– изменение конфигурации функций, которые обеспечивают выполнение функциональных требований;

– изменение функций путем подключения дополнительных данных, перераспределение доступа к функциям;

– изменение последовательности реализации функций в связи с изменением условий внешней среды;

– восстановление проектных решений после изменений.

Модели слежения за изменениями процессов обработки моделируют изменения в системе при изменениях ряда нефункциональных требований (например, параллельность, распределенность и т.п.). С точки зрения организации вычислений реинжинерия может обеспечить изменение принципов предоставления приоритетов процессам, перераспределение компонентов по узлам системы в целях оптимизации их нагрузки. Такие модели включают в себя задачи:

– перераспределения процессов (задач) по узлам (процессах) сети;

– перераспределения объектов по узлам сети;

– изменения схем защиты системы от несанкционированного доступа и т.п.

Модели слежения за изменениями системы концентрируются на изменениях организации хранения проектных решений и программных модулей в рамках среды разработки. Изменения касаются:

– репозитарии проектных данных для ПС;

– база данных системы;

– ЖЦ эволюции требований к ПС;

– формализованные сценарии создания и развития системы.

Общая модель слежения за изменениями. Так как разработанное количество функций в архитектуре системы постоянно уточняется в процессе ее создания, то методы слежения за текущими изменениями повышают качество изменений.

Модель слежения за изменениями включает в себя выбор наиболее целесообразных изменений, а также изучение влияния изменений на работу системы.

К параметрам модели слежения относятся:

– *ChF* (Change Funtion) для фиксации изменения функций;

– *ChS* (Change Structure) для фиксации изменений в структуре системы.

Эти переменные принимают значение соответственно 1, если изменение выполнено, 0 – в противном случае.

Модель слежения за изменениями имеет вид

$$Mch = \{M(Ch), N(Ch), \rho, \rho 1\},$$

где $M(Ch)$ – количество измененных функций в ПС, $N(Ch)$ – количество измененных компонентов в структуре ПС, ρ – показатель степени изменчивости функций, ρ_1 – показатель степени изменчивости структуры ПС.

Эта модель выполняется тогда, когда изменения вносятся в функции или в структуру системы и фиксируются в репозитарии системы.

Функция ChF вычисляет степень изменений ρ, а ChS – степень ρ_1 изменчивости структуры системы.

Переменная ρ вычисляется по формуле: $\rho = ChF_{n1} / Ch_n$, где n – исходное количество функций, n_1 – количество функций, которые изменены.

Степень ρ может принимать такие значения:

$$\rho = \begin{cases} < 0{,}5, \text{ если изменения выполнены и последующие изменения не проводятся,} \\ > 0{,}5, \text{ если изменения выполнены успешно.} \end{cases}$$

При первом значении степени ρ может возникнуть дисбаланс между первоначальным текстом ПС и после изменения.

Аналогично определяется и степень переменчивости структуры ПС:

$$\rho_1 = ChS_{m1} / Ch_m,$$

где m – исходное количество компонентов в системе, m_1 – количество компонентов, измененных в данное время.

Степень ρ_1 может принимать такие аналогичные значения:

$$\rho_1 = \begin{cases} < 0{,}5, \text{ если следующие изменения компонентов не вносятся,} \\ > 0{,}5, \text{ если изменения могут выполняться.} \end{cases}$$

Иными словами, степень изменения функций и компонентов колебаться между $0{,}5 < (\rho, \rho_1) < 0{,}5$ и означает, что при значении меньшем 0,5, процесс изменений необходимо остановить.

Поскольку изменения на этапе эксплуатации системы влияют на состояние системы, в дальнейшем принимаются решения о применении методов реинженерии или реверсной инженерии системы.

Системные изменения, которые связаны с адаптацией системы к новым условиям среды, с модификацией конфигурации (или ее инсталляцией) и уточнения средств безопасности, здесь не рассматриваются. Новые модели слежения за изменениями учитывают задачи изменения:

- конфигурации системы,
- системных функций управления,
- нефункциональных требований к системе,
- технологических процессов ЖЦ,
- распределения объектов по узлам сети,
- защиты системы от несанкционированного доступа,
- проектных решений относительно данных репозитария и др.

Рассмотрены методы реинженерии, реверсной инженерии и рефакторинга, которые зафиксированы в инженерии программирования, как методы систематического и последовательного эволюционного развития и изменения унаследованных программ. Слежение за внесением изменений позволяет проводить контроль и управление изменениями.

ЧАСТЬ

3

ПРОГРАММИРОВАНИЕ: методы проверки правильности, тестирвания и оценки надежности программ

ГЛАВА 6

ФОРМАЛЬНЫЕ ОСНОВЫ ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ ПРОГРАММ

Одним из современных направлений в области проверки правильности программ является использование формальных методов для доказательства их правильности, т.е. привлечение математического аппарата для доказательства того, что спецификация алгоритма программы правильно задает решение задачи, для которой она разработана.

В формальных методах нет рутинного написания текста на ЯП, а есть анализ текста и описание поведения программы в стиле, близком математической нотации, путем рассуждений и доказательств, принятых в математике. Формальные методы в программировании появились одновременно с самим программированием, на которое оказало влияние работы по разработке алгоритмов А.А. Маркова [1], А.А. Ляпунова [2], схемы Ю.И.Янова [3], формальные нотации языка описания взаимодействующих процессов К.А. Хоара [4] и др.

В 70-х годах прошлого столетия появились формальные спецификации, которые близки ЯП и предоставляют средства, облегчающие рассуждения о свойствах формальных тестов и сближающие их с математической нотацией. Несмотря на это, исследования формальных методов носили в основном академический, теоре-

тический характер, поскольку извлечь из них практическую пользу в программировании соизмеримо с огромными затратами, связанными с описанием формальных спецификаций и проведением по ним доказательства правильности программ [5–10].

Формальное математическое доказательство программ основывается на спецификациях алгоритмов программ, аксиомах, описаниях некоторых условий, называемых предварительными условиями (предусловиями), утверждениях и постусловиях, определяющих заключительное условие получения правильного результата программой.

Спецификация – это формальное описание функций и данных программы, с которыми эти функции оперируют. Различают видимые данные, т.е. входные и выходные параметры, а также скрытые данные, которые не привязаны к реализации и определяют интерфейс с другими функциями.

Предусловия – это ограничения на совокупность входных параметров и постусловия – ограничения на выходные параметры. Пред- и постусловие задаются предикатом, т.е. функцией, результатом которой будет булева величина (true/false). Предусловие истинно тогда, когда входные параметры входят в область допустимых значений данной функции. Постусловие истинно тогда, когда совокупность значений удовлетворяет требованиям, задающим функциональность. Постусловие фактически задает формальное определение критерия правильности получения результата.

Доказательство проводится с помощью *утверждений*, которые составляются в формальном языке и служат способом проверки правильности программы в заданных ее точках. Создается набор утверждений, каждое из которых использует предусловия и последовательность инструкций, приводящих к проверке промежуточного результата относительно отмеченной точки программы, для которой сформулировано данное утверждение. Если утверждение соответствует конечному оператору программы, где требуется получить окончательный результат, то с помощью заключительного утверждения и постусловия делается окончательный вывод о частичной или полной правильности работы программы.

К традиционным методам проверки правильности программ относятся:

1) методы доказательства правильности программ;

2) формальная верификация и валидация.

Далее рассматриваются особенности этих методов с учетом анализа работ [14–26] применительно к разным объектам ПС.

6.1. МЕТОДЫ ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ ПРОГРАММ

Как отмечалось выше, методы доказательства были разработаны в 70-е годы прошлого столетия и они разделяются на методы доказательства частичной и полной правильности программ [4, 5].

Под доказательством *частичной правильности* понимается проверка выполнения свойств данных программы с помощью утверждений, которые описывают то, что должна получить эта программа, когда закончится ее выполнении в соответствии с условиями заключительного утверждения. *Полностью правильной программой* по отношению к ее описанию и заданным утверждениям будет программа, когда она частично правильная и заканчивается ее выполнение при всех данных, удовлетворяющих ей.

Для доказательства частичной правильности используется метод индуктивных утверждений, сущность которого состоит в следующем. Пусть утверждение А связано с началом программы, В – с конечной точкой программы, и утверждение С отражает некоторые закономерности получения значений переменных, по крайней мере в одной из точек каждого замкнутого пути в программе (например, в циклах). Если при выполнении программа попадает в *i*-ю точку и справедливо утверждение A_i , а затем она проходит от точки *i* к точке *j*, то будет справедливо утверждение A_j.

Теорема 6.1. *Если выполнены все действия метода индуктивных утверждений для программы, то она частично правильна относительно утверждений A, B, C.*

Требуется доказать, что, если выполнение программы закончится, то утверждение В будет справедливым. По индукции, проходя через некоторые *n*-точек программы, в которых утверждение С будет справедливым, то В так же будет справедливым. Таким образом, если программа прошла *n*-точку и утверждения А и В справедливы, то тогда, попадая из *n*-ой точки в *n+1* точку, утверждение A_{n+1} будет справедливым, что и требовалось доказать.

Методы доказательства правильности программ, основанные спецификациях и формальных методах, изложены ниже.

6.1.1. Анализ языков формальной спецификации

Языки спецификаций, используемые для формального описания свойств программ, являются языками более высокого уровня, чем ЯП. Проведем их классификацию по следующим категориям: универсальные языки с общематематической основой

(например, RAISE, Z, VDM и др.) [6–9]; языки спецификации отдельных проблемных областей (например, ЯП, трансляторы, БД и др.) [5, 10, 11]; специализированные языки спецификации (например, языки таблиц, логики, равенств и подстановок и др.) [5, 10]; языки, ориентированные на спецификацию параллельных процессов (например, CIP-L, Concurrent Pascal, Ada-68 и др.) [10] (рис. 6.1).

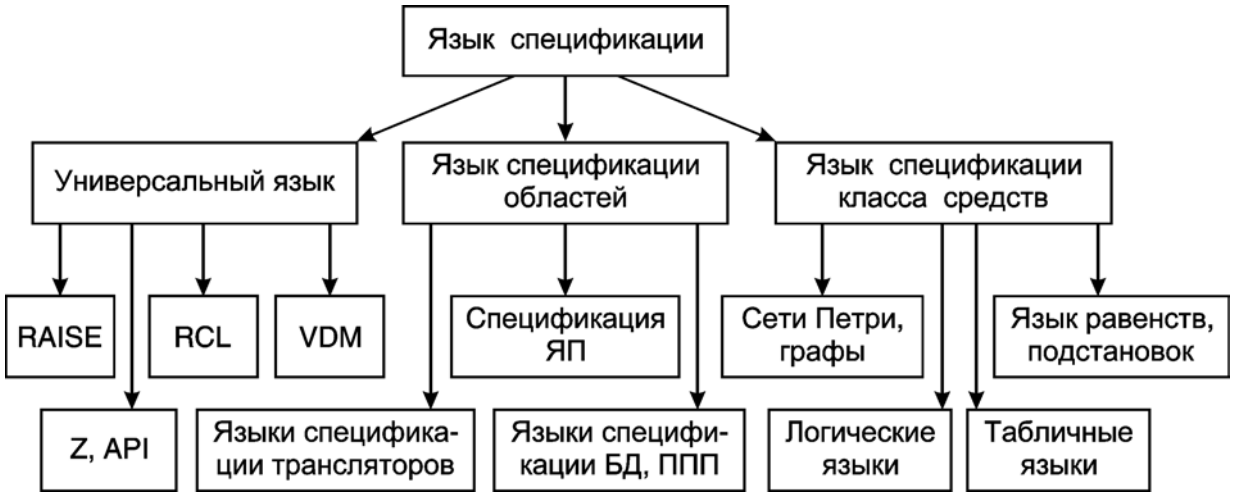

РИС. 6.1. Категории языков спецификации

Спецификация программы – это точное, однозначное и недвусмысленное описание с помощью математических понятий и терминов, а также правил синтаксиса и семантики языка спецификации. В языке спецификаций могут быть понятия и конструкции, которые нельзя выполнить на компьютере, их необходимо представлять последовательностью операций, функций, понятных для интерпретации.

Описание задачи в языке спецификации включает в себя описание общего контекста всех понятий, через которые определяются понятия, участвующие в формулировке задачи или описании модели ПрО. Описание задачи представляется формально в виде аксиом, утверждений, пред- и постусловий и других формализмов, требующих для их реализации не систем программирования, а специального математически ориентированного аппарата для доказательства или верификации описания задач, в частности специальных интерпретаторов или метасистем.

Дадим далее краткую характеристику видов языков спецификации.

Универсальные языки спецификации имеют общематематическую основу, сущность которой поясним на примере языка VDM [7], который имеет следующие математические средства:

1) логические средства логики первого порядка, включая кванторы;

2) арифметические операции;

3) средства образования множеств, в том числе описание множеств с помощью логических формул и операций над множествами;

4) средства описания конечных последовательностей (кортежей, списков) и операции над ними;

5) средства описания конечных функций и операции над ними;

6) средства описания древовидных структур;

7) средства построения областей или множества объектов, включая произведения, объединения и рекурсивные определения;

8) определение функций с помощью выражений и равенств, включая рекурсивные определения;

9) процедурные средства ЯП (операторы присваивания, цикла, выбора, выхода);

10) операции композиции, аргументами и результатами которых могут быть функции, выражения, операторы.

В этом языке не хватает графовых средств, деревьев, средств управления и параллелизма. Однако содержащихся в языке VDM средств достаточно для формальной математической спецификации задач различного типа.

Общая характеристика языков RAISE, RCL и Z [6–9] приведена в 2.3 данной книги.

Языки спецификации областей включают в себя языки:

1) задания процессов управления, взаимодействия и параллельного выполнения;

2) спецификации ЯП и трансляторов;

3) спецификации БД и знаний;

4) спецификации пакетов прикладных программ и др.

Каждый из этих языков имеет специализированные средства, отображающие специфические особенности соответствующей области. Языки типа 1 в отличие от ЯП позволяют специфицировать процессы управления вычислениями, передачей сообщений и взаимодействием объектов в распределенных системах. Лексический и синтаксический анализ трансляторов хорошо описываются языками регулярных выражений КС-грамматик в форме Бэкуса–Наура в виде спецификации контекстных зависимостей синтаксиса ЯП. Такого типа языки называются метаязыками. Для спецификации семантики языков используется фор-

мализм равенств. Техника описания ЯП в целях перевода основывается на атрибутных грамматиках и абстрактных типах данных. Задача описания ЯП и перевода – очень сложная, для ее описания используются денотационный, алгебраический и атрибутный подходы, а также логические термины с ориентацией на верификацию описания [11].

Языки спецификации с ориентацией на средства включают в себя: языки, основанные на равенствах и подстановках с операционной семантикой (Лисп, Рефал); логические языки; языки операций (APL) над последовательностями и матрицами; табличные языки; сети, графы и др. [5, 10].

Язык логики предикатов с набором базисных функций используется для записи пред- и постусловий, инвариантов и верификации. Отдельные операции логики предикатов используются также в языках логического программирования (например, Пролог).

Основой математических объектов являются равенства и подстановки. Определение семантики равенства бывает трех классов: денотационное, операционное и аксиоматическое. Операционная семантика связана с подстановками (замена, продукция) и определяется в терминах операций, приводящих к вычислениям алгоритмов. При этом фиксируется порядок и динамика выполнения операций. Денотационный подход к семантике предпочитает статическое описание в терминах математических свойств объектов, а аксиоматической – специфицирует свойства объектов в рамках некоторой логической системы, содержащей правила вывода формул и/или интерпретаций.

Продукция или правило подстановки общего вида это:

$$\lambda \to \rho,$$

где λ и ρ – произвольные слова в фиксированном алфавите. Нормальный алгоритм Маркова [1] представляет собой упорядоченный набор правил, некоторые из них отмечены как завершающие. Применение правила $\lambda \to \rho$ к слову φ состоит в подстановке слова ρ вместо самого левого слова λ в φ. Вычисление заканчивается, когда применяется завершающее правило, состоящее в порождении одного слова.

Завершая краткую характеристику разных типов языков спецификации, можно сказать, что их средства оказывают влияние на верификацию программ, перевод спецификаций в формы, приближающие их к эффективному выполнению программ, и разработку новых моделей вычислений. Эти языки постепенно станут привычными ЯП.

6.1.2. Общая характеристика формальных методов доказательства

Наиболее известными формальными методами доказательства программ являются метод рекурсивной индукции или индуктивных утверждений Флойда, Наура, метод структурной индукции Хоара и др. [4, 12–17].

Метод Флойда основан на определении условий для входных и выходных данных и в выборе контрольных точек в доказываемой программе так, чтобы путь прохождения по программе пересекал хотя бы одну контрольную точку. Для этих точек формулируются утверждения о состоянии и значениях переменных в них (для циклов эти утверждения должны быть истинными при каждом прохождении цикла-инварианта).

Каждая точка рассматривается для индуктивного утверждения того, что формула остается истинной при возвращении в эту точку программы, и зависит не только от входных и выходных данных, но и от значений промежуточных переменных. На основе индуктивных утверждений и условий на аргументы программы создаются конкретные утверждения для проверки правильности этой программы в отдельных ее точках. Для каждого пути программы между двумя точками задается проверка на соответствие условий правильности и определяется истинность этих условий при успешном завершении программы на данных, удовлетворяющих входным условиям.

Формирование таких утверждений является довольно сложной задачей, особенно для программ с высокой степенью параллельности и взаимодействия с пользователем. Кроме того, трудно проверить достаточность и правильность самих утверждений.

Доказательство корректности применялось для написанных программ в ЯП и тех, которые разрабатываются методом последовательной декомпозиции задачи на несколько подзадач, для каждой из них формулируются утверждения с учетом условий ввода и вывода и точек программы, расположенных между входными и выходными утверждениями. Суть доказательства истинности выполнения условий и утверждений относительно заданной программы и составляет основу доказательства ее правильности.

Данный метод доказательства способствует уменьшению числа ошибок и сокращению времени тестирования программы, а также отработки спецификаций на полноту, однозначность и непротиворечивость.

Метод Хоара – это усовершенствованный метод Флойда, основанный на аксиоматическом описании семантики языка программирования исходных программ. Каждая аксиома описывает изменение значений переменных с помощью операторов этого языка. Основное внимание уделяется формализации операторов перехода и вызовов процедур с помощью правил вывода, каждое из которых задает индуктивное высказывание для каждой метки (точки) и функции исходной программы.

Оператор перехода рассматривается как выход из циклов и аварийных ситуаций. Данный метод имеет некоторые ограничения на параметры процедур. Система правил вывода дополняется механизмом переименования глобальных переменных, условиями на аргументы и результаты, а также на правильность задания данных программы.

Описание с помощью системы правил утверждений об операторах программы является громоздким и отличается неполнотой, поскольку все правила предусмотреть невозможно. Данный метод проверялся экспериментально на ограниченном множестве программ, а средства автоматизации метода не были реализованы.

Метод рекурсивных индукций Маккарти состоит в структурной проверке функций, работающих над структурными типами данных, изменяют структуры данных и диаграммы перехода во время символьного выполнения программ.

Техника символьного выполнения включает в себя моделирование выполнения кода при использовании символов для изменяемых данных. Тестовая программа имеет детерминированное входное состояние при вводе данных и разных условий ее выполнения. Благодаря тому, что каждая строка кода должна выполняться самостоятельно, фиксируются состояния и значения переменных программы, а также их проверка.

Выполняемая программа рассматривается как серия изменения состояний, т.е. каждой логической части программы соответствует упорядоченная серия изменения состояний. Самое последнее состояние программы считается выходным состоянием и, если оно получено, то программа считается правильной. Данный метод обеспечивает высокое качество исходного кода.

Метод Дейкстры включает в себя два подхода к доказательству правильности программ. Первый подход основан на модели вычислений, оперирующей историями результатов вычислений при работе программ, анализе пути прохождения и правил обработки большого объема данных этой программы.

Второй подход базируется на формальном исследовании текста программы с помощью задаваемых предикатов первого порядка, которые применяются к асинхронным программам. В процессе выполнения каждая из этих программ получает некоторое состояние, которое запоминается в так называемом сборщике мусора, который по окончании программы очищается, уничтожая эти состояния.

Метод Дейкстры основывается на перевычислении, математической индукции и абстрактном описании программы.

Перевычисление используется для проверки границ вычислений проверяемой на правильность программы с помощью инвариантных отношений.

Математическая индукция применяется для организации прохождения циклов и рекурсивных процедур с помощью необходимых и достаточных условий утверждений повторных вычислений в программе.

Абстракция позволяет ослабить количественные ограничения, которые накладываются методом перевычислений.

Доказательство программ по данному методу можно рассматривать как доказательство теорем с использованием аппарата математической индукции при формальном доказательстве правильности, а также как систему статической проверки правильности программ за столом с целью обнаружения в них ошибок.

Метод математической индукции позволяет доказать истинность некоторого предположения $P(n)$ в зависимости от параметра n при всех $n \geq n_0$, и тем самым доказать случай $P(n_0)$. Исходя из истинности P(n) для любого значения n, доказывается $P(n+1)$, что достаточно для доказательства истинности $P(n)$ при всех $n \geq n_0$.

Этот путь доказательства используется для утверждения A относительно программы, которая при своем выполнении достигает определенной точки. Проходя через эту точку n раз, можно получить справедливость утверждения $A(n)$, если доказать:

1) что справедливо $A(1)$ при первом проходе через заданную точку,

2) если справедливо $A(n)$ (при n-проходах через заданную точку), то справедливо и $A(n+1)$ при прохождении через заданную точку $n+1$ раз.

Чтобы доказать, что некоторая программа правильная, надо правильно описать работу этой программы, т.е. ее логику. Такое

описание называется *правильным высказыванием* или просто утверждением. Исходя из предположения, что программа в конце концов успешно завершится, утверждение о ее правильности будет справедливым.

Проверка правильности программ в системе автоматизации СПРУТ по методу Хоара, основанной на проверках утверждений для программ на ЯП Паскаль [15].

На вход данной системы поступает программа на языке Паскале с аннотациями, содержащими контрольные точки для организации проверки программы. Генератор системы обрабатывает заданные условия проверки правильности и формирует список предикатных формул для осуществления доказательства арифметических и логических выражений.

В систему вводятся аксиомы и списки подцелей, с помощью которых устанавливается соответствие между аргументами аксиом и параметрами формул. Процесс завершается получением заключения о правильности программы или о найденных ошибках.

6.1.3. Модель формального доказательства корректности программ

Рассматривается формальное доказательство экспериментальной программы, заданной структурной логической схемой. Для нее составляются утверждения с помощью логических операторов, задаваемых комбинациями логических переменных (true/false), логических операций (конъюнкция, дизъюнкция и др.) и кванторов всеобщности и существования (табл. 6.1).

ТАБЛИЦА 6.1. Логические операции.

Название	Пример	Значение
Конъюнкция	$x \,\&\, y$	x и y
Дизъюнкция	$x * y$	x или y
Отрицание	$\neg x$	не x
Импликация	$x \rightarrow y$	если x, то y
Эквивалентность	$x = y$	x равнозначно y
Квантор всеобщности	$\forall x\ P(x)$	для всех x условие истинно
Квантор существования	$\exists x\ P(x)$	существует x, для которого $P(x)$ – истинно

Алгоритм программы состоит в построении для одномерного массива целых чисел T длины N (array $T\ [1:N]$) эквивалентного массива T' той же длины N, что и массив T, и его элементы должны располагаться в порядке возрастания их значений. Иными словами, цель данного алгоритма – сортировка элементов исходного массива T по их возрастанию. Доказательство правильности алгоритма сортировки элементов массива T проведем с использованием ряда утверждений относительно элементов этого алгоритма, который описывается пунктами П1–П6.

П1. Входное условие алгоритма задается в виде начального утверждения:

A_{beg}: ($T\ [1:N]$ – массив целых) & ($T'\ [1:N]$ массив целых).

Выходное утверждение A_{end} запишем как конъюнкции таких условий:

(а) (T– массив целых) & (T' – массив целых)

(б) ($\forall i$, если $i \le N$, то $\exists j\ (T'(\ i\ ) \le T'\ (\ j\ ))$),

(в) ($\forall\ i$, если $i < N$, то $(T'\ (\ i\ ) \le T'\ (i+1)\ )$,

т.е. A_{end} – это

(T – массив целых) & (T' – массив целых)

& $\forall i$, если $i \le N$, то $\exists j\ (T'(\ i\ ) \le T'\ (\ j\ )\ )$,

& $\forall i$, если $i < N$, то $(T'\ (\ i\ ) \le T'\ (i+1)\ )$.

Для расположения элементов массива T в порядке возрастания их величин в массиве T' используется алгоритм пузырьковой сортировки, суть которого заключается в предварительном копировании массива T в массив T', а затем проводится сортировка элементов согласно условию их возрастания. Алгоритм сортировки представлен на блок-схеме (рис. 6.2).

Операторы алгоритма размещены в прямоугольниках. Условия выбора альтернативных путей представлены в параллелограммах, в кружках заданы точки с начальным A_{beg} и конечным A_{end} условиями и состояниями алгоритма, кружок с нулем задает начальное состояние, а с одной звездочкой – состояние после обмена местами двух соседних элементов в массиве T', кружок с двумя звездочками – состояние после обмена местами всех пар за один проход всего массива T'.

Кроме уже известных переменных T, T' и N, в алгоритме использованы еще две переменные: i – целое и M – булева переменная, значением которой являются логические константы true и false.

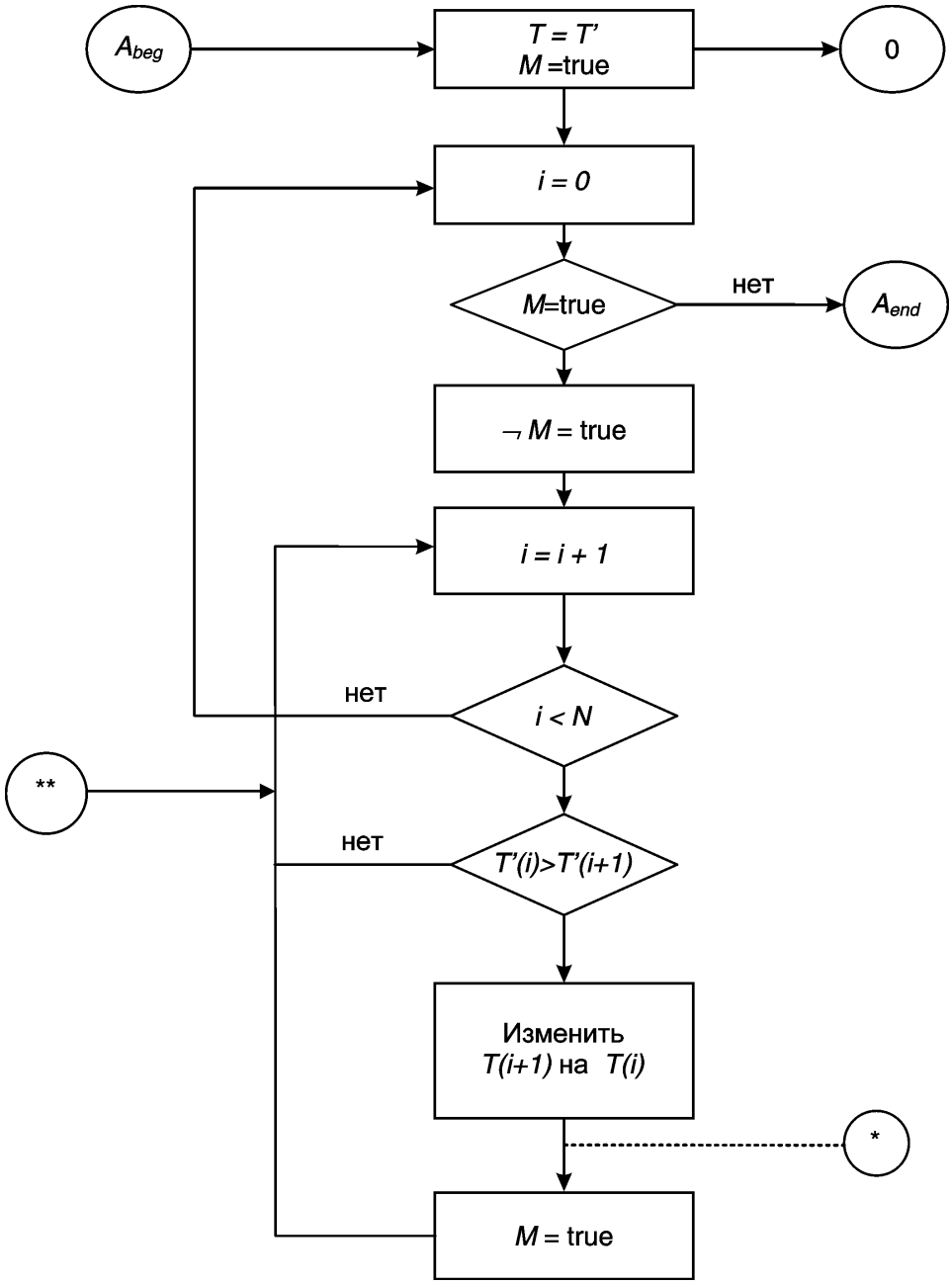

РИС. 6.2. Схема сортировки элементов массива *T*

П2. Для доказательства того, что алгоритм действительно обеспечивает выполнение исходных условий, рассмотрим динамику выполнения этих условий последовательно в определенных точках алгоритма.

Заметим, что указанные точки делят алгоритм на части, правильность любой из которых обосновывается отдельно.

Так, оператор присваивания означает, что для всех i $(i \le N \ \& \ i>0)$ выполняется $(T'[i]:=T[i])$.

Результат выполнения алгоритма в точке с нулем может быть выражен утверждением:

($T[1: N]$ − массив целых) & ($T'[1: N]$ − массив целых)

& ($\forall i$, если $i \le N$ ($T[i] = T[i]$)).

Доказательство очевидно, поскольку за семантикой оператора присваивания (поэлементная пересылка чисел из T в T') сами элементы при этом не изменяются, к тому же в данной точке их порядок в T и T' одинаковый. Итак, получили, что выполняется условие "б" выходного утверждения.

Заметим, что первая строка доказанного утверждения совпадает с условием "а" выходного утверждения A_{end}, которая остается справедливой до конца работы алгоритма, и поэтому в следующих утверждениях приводиться не будет.

В точке с одной звездочкой выполнен оператор, где

$(i < N)$ $(T'(i)) > T'(i + 1) \rightarrow (T'(i)$ и $T'(i + 1)$ − меняются местами элементы.

При работе оператора будет справедливо такое утверждение:

$\exists i$, если $i < N$, то $(T'(i) < T'(i + 1)$,

которое − часть условия "в" утверждения A_{end} (для одной конкретной пары смежных элементов массива T').

Очевидно также, что семантика оператора обмена местами не нарушает условие "б" выходного утверждения A_{end}.

В точке с двумя звездочками выполнены все возможные обмены местами пар смежных элементов массива T' за один проход через T', т.е. оператор обмена работал один или больше раз. Однако пузырьковая сортировка не дает гарантии, что достигнуто упорядочение за один проход по массиву T', поскольку после очередного обмена индекс i увеличивается на единицу независимо от того, как соотносится новый элемент $T'(i)$ с предыдущим элементом $T'(i - 1)$.

В этой точке также справедливо утверждение:

$\exists i$, если $i < N$, то $T'(i) < T'(i + 1)$.

Часть алгоритма, обозначенная точкой с двумя звездочками, выполняется до тех пор, пока не будет упорядочен весь массив, т.е. не будет выполнено условие "в" утверждения A_{end} для всех элементов массива T':

$\forall i$, если $i < N$, то $T'(i) < T'(i+ 1)$.

Итак, выполнение исходных условий обеспечивается соответствующей семантикой операторов преобразования массива и порядком их выполнения.

Доказано, что выполнение программы завершено успешно, что означает ее правильность.

П3. Этот алгоритм можно представить в виде серии теорем, которые доказываются. Начиная с первого утверждения и переходя от одного преобразования к другому, определяем индуктивный путь вывода, т.е. если одно утверждение является истинным, то истинно и другое. Иными словами, если первое утверждение – A_1 и первая точка преобразования – A_2 , то первой теоремой является: $A_1 \rightarrow A_2$. Если A_3 – следующая точка преобразования, то второй теоремой будет: $A_2 \rightarrow A_3$.

Таким образом, формулируется общая теорема: $A_i \rightarrow A_j$, где A_i и A_j – смежные точки преобразования.

Последняя теорема формулируется так, что условие – истинное в последней точке отвечает истинности выходного утверждения: $A_k \rightarrow A_{end}$.

Следовательно, можно возвратиться потом к точке преобразования A_{end} и к предшествующей точке преобразования.

Доказав, что $A_k \rightarrow A_{end}$, значит, имеем что верно $A_j \rightarrow A_{j+1}$ и так далее, пока не получим, что $A_1 \rightarrow A_0$.

П4. Далее специфицируются утверждения *if – then*.

П5. Чтобы доказать, что программа корректная, необходимо последовательно расположить все утверждения, начиная с A_1 и заканчивая A_{end}, что подтверждает истинность входного условия и выходного условий.

П6. Доказательство программы завершено.

6.1.4. Доказательство корректности интеграции компонентов

Для доказательства правильности интеграции компонентов, которые записаны в разных ЯП, используется граф $G\{K, I\}$, где K – множество компонентов, из которых создаются программа P и I – множество описаний интерфейсов в языке IDL. В случае неоднородности интегрируемых компонентов применяется преобразование неэквивалентных типов данных. Программа P – это совокупность компонентов и соответствующих им интерфейсов из множества I. Сформулируем необходимые и достаточные условия корректности интеграции компонентов для получения правильной программы P [19–22].

Утверждение 6.2. *Для корректной интеграции компонентов в программу P необходимо и достаточно*, чтобы

1) существовали операции интеграции такие, что для заданного графа $G\{K, I\}$ процесс построения интегрированной структуры был конечный;

2) находились компоненты во множестве K, т.е. соблюдалось условие принадлежности компонента $\kappa_i \in K$ и каждый из них реализовал некоторую функцию, представленную исходным текстом и интерфейсным модулем;

3) имелись правила преобразования параметров (x, y) компонента κ_i и изменения компонента в распределенной среде.

Доказательство данного утверждения сведем к доказательству лемм.

Лемма 6.1. *Алгоритм построения программы P из компонентов является конечным.*

Рассмотрим схему алгоритма метода интеграции, которая с помощью графа $G\{K, I\}$ создает P из компонентов множества K и представлена следующими шагами.

1. Проверка правильности задания графа $G\{K, I\}$ путем установления соответствия вершин графа элементам множеств K и I, а также количества и порядка передаваемых параметров компонентов, описанных в интерфейсе I. Результатом проверки является переменная ρ, которая может принимать следующие значения:

$$\rho = \begin{cases} 0, \text{ когда граф } G \text{ задан правильно на множестве } K \text{ и } I, \\ 1, \text{ когда порядок параметров или их количество в операторе вызова не имеют соответствия.} \end{cases}$$

2. Проверка согласованности типов формальных и фактических параметров операторов вызовов компонента κ_i. Результатом этого шага являются данные о проверке принадлежности типов данных параметров (x, y) в каждой паре взаимодействующих компонентов (κ_i, κ_j) через интерфейс I_{ij}, отображенный на графе $G\{K, I\}$ и имеющий следующее значения:

$$I_{ij} = \begin{cases} 1, \text{ если передаваемые параметры в интерфейсе релевантные,} \\ 0, \text{ иначе.} \end{cases}$$

3. Выполняется операция интеграции (при $\rho=0$): формируются команды преобразования нерелевантных типов данных (например, целого в плавающее и наоборот) сопоставляемых формальных и фактических параметров взаимодействующих компонентов (κ_i, κ_j) и обратно.

4. Обработка обнаруженной ошибки и формирование диагностического сообщения при несогласовании типов данных или при не сопоставлении количества формальных и фактических параметров.

Лемма 6.2. *Для любой пары компонентов* $(\kappa_i , \kappa_j) \in K$ *созданная программа* $P = \{ \kappa_i , I_i , \kappa_j , I_j \}$ *будет корректной.*

Справедливость этого утверждения вытекает из определения интерфейсного модуля, в соответствии с которым для каждой пары компонентов $\{\kappa_i , \kappa_j \}$ существуют такие интерфейсные модули I_i и I_j , которые входят в программу $P = \{\{\kappa_i , I_i \}, \{\kappa_j , I_j \}\}$. Они обеспечивают проверку корректности каждого параметра при входе в вызываемый компонент и при выходе из него.

Так как количество компонентов графа $G\{K, I\}$ − конечное множество, и каждая пара является корректной, то в результате получаем и корректность программы P, построенной из конечного множества пар взаимодействующих компонентов.

Схема построения любой программы P из компонентов $\kappa_i \in K$ включает в себя:

1) конечное количество операций преобразования данных;

2) программы формирования и выдачи диагностического сообщения в формальных и фактических параметрах операторов вызовов компонентов;

3) конечное множество компонентов K.

Конечность построения программы P из компонентов на основе графа $G\{K, I\}$ показана леммой 1, 2. При этом связь некоторой пары компонентов может быть недопустимой в случае неэквивалентности типов данных в формальных и фактических параметрах интерфейсов модулей при условии:

$\exists\, x_i \in X$ таких, что когда $t(x) \sim t$ (тип параметра, эквивалентный множеству типов T), то $\rho = 0$, иначе $\rho = 1$.

Из выполнения этого условия вытекает, что процесс построения программы P по графу $G\{K, I\}$ является конечным и приводит к созданию корректной программы P или к диагностическому сообщению.

6.1.5. Доказательство свойств моделей распределенных приложений

Доказательство свойств моделей распределенных приложений, основанных на модели проверки (model-checking) конечных автоматов и индуктивных утверждениях, предложено в [17]. Основная

идея метода состоит в редукции системы с бесконечным числом состояний к системе с конечным числом состояний, а также в доказательстве распределенных приложений (РП) с помощью индуктивных рассуждений и системы переходов конечного автомата.

В данном методе используется язык спецификации SDL (Specification Description Language) для описания процессов РП и их верификации. Связь между процессами РП осуществляется через канал, в котором находится сигнал – сообщение с параметрами или без них [15, 16].

Особенностью канала является запись в него другого сигнала после освобождения (пустой канала) или чтения очередного сигнала. Процесс описывается последовательностью действий, приводящих к изменению переменных, чтению сигнала из канала, записи в канал и очистки канала. Проверка спецификации ограничивается условиями справедливости.

Основными типами данных спецификации в SDL являются предопределенные и конструируемые типы данных (массив, последовательность и т.д.). Язык SDL позволяет описывать формулы с помощью предикатов, булевых операций, кванторов, переменных, модальностей. Семантика их определения зависит от момента времени, заданного во множестве всех моментов времени, и возможных последовательностях действий (поведений), выполняемых спецификацией процесса.

В качестве предикатов используются локаторы управляющих состояний процессов, контроллеры заполнения каналов (пусто/заполнен канал) и проверки наличия сигналов в каналах, а также отношения между переменными и параметрами сигналов.

Спецификация процесса состоит из заголовка, контекста, схемы и подспецификации. В заголовке указывается имя и вид процесса, формулы или предикаты.

Контекст включает в себя описание типов, переменных и каналов. Переменная принадлежит процессу, если в ее описании указано место и имя процесса (через точку), которому эта переменная принадлежит. При ее использовании указывается имя этой переменной с расширением. Если указывается параметр, то в расширенное имя входит имя канала, сигнал и имена параметров, разделенных точками.

В логических условиях используются кванторы всеобщности и существования.

Схема спецификации процесса включает в себя описание условий выполнения и диаграммы процессов. Она инициируется

посылкой сообщения во входной канал, который передает сообщение внешней сетевой среде для выполнения.

Диаграмма процесса состоит из описаний переходов, которые включают в себя состояние, набор операций процесса и перехода на следующее состояние. Набор операций – это действия типа: чтение сообщения из входного канала, запись его в выходной канал, очистка входного и выходного каналов, изменение значений переменных программы P_i-процесса.

Каждая операция определяет поведение процесса и создает некоторое событие. Логическая формула задает модальность поведения спецификации и моменты времени.

Процесс, представленный формальной спецификацией, выполняется недетерминировано. Обмен с внешней средой производится через входные и выходные параметры сообщений.

Событие. В каждый момент времени выполнения процесс имеет некоторое состояние, которое может быть отражено в виде снимка, характеризующего некоторое событие. Снимок процесса может включать в себя значения его переменных, которым соответствуют параметры и характеристики состояний процесса.

К событиям процесса относятся:

– отправка сообщения в канал;
– получение сообщения из канала;
– чистка входных и выходных каналов;
– выполнение программ;
– непредвиденное событие (взлом канала и др.)

Конфигурация процесса определяется в виде тройки:

$$КОНФ = (И.О,\ СУ),$$

где $И$ – интерпретация процесса объекта, принадлежащего множеству $(p_1, p_2, \ldots p_к)$; $О$ – отображение текущего состояния канала; $СУ$ – управляющие сигналы канала для запуска процесса.

Семантика выполнения процесса определяется в терминах событий и правил с помощью следующего утверждения:

Любой процесс Pi ∈P вызывает событие при чтении или записи сообщения из/в канал, а также при выполнении процесса в узле распределенной системы.

6.1.6. Методы анализа структур программ

Анализ структуры программ относится к доказательству правильности программ [21–24] и является инспекцией программы независимыми экспертами и разработчиками. На начальном этапе проектирования инспекция – это проверка полноты, целостности,

однозначности, непротиворечивости и совместимости документов с исходными требованиями к ПС. На этапе реализации системы под *инспекцией* понимается анализ текстов программ на соблюдение требований, стандартов и принятых руководящих документов технологии программирования.

Эффективность инспекции заключается в том, что эксперты пытаются взглянуть на проблему "со стороны" и подвергнуть ее всестороннему критическому анализу. Если проводится сквозной контроль, то эксперты воспринимают еще и словесное объяснение задачи и способов ее проектирования разработчиком. Непосредственный просмотр кода позволяет обнаружить ошибки в логике и в описании алгоритма.

Сквозной контроль состоит в ручной имитации выполнения программы. Разработчик программы устно объясняет и обосновывает выбранные подходы и методы реализации, подбирает тесты для ручного прослеживания за выполнением программы.

Эти приемы позволяют на более ранних этапах проектирования обнаружить ошибки или дефекты при многократном просмотре исходного кода.

Методы просмотра не формализованы и определяются степенью квалификации экспертов группы.

Метод простого структурного анализа ориентирован на анализ структуры программы, заданной в виде графовой модели, в которой каждая вершина представляет собой оператор, а дуга – передачу управления между операторами. На ее основе определяется, достижимы ли вершины программы и существует ли выход из всех потоков управления для завершения программы [21].

В целях проведения анализа потоков данных граф расширяется указателями переменных, их значениями и ссылками на каждый оператор программы. При тестировании потоков данных вначале определяются значения предикатов в операторах реализации логических условий, по которым формируются пути выполнения программы. Затем производится проверка вычислений на арифметических операциях. Для прослеживания путей программы устанавливаются точки, в которых имеются ссылки на переменные до присвоения им значений. Ошибкой считается, если переменной присваивается значение без ее описания, либо выполняется повторное описание переменной, к которой нет обращения.

Метод анализа дерева отказов пришел в программную инженерию из техники, как метод анализа отказов и неисправностей в аппаратуре. Для программ суть метода состоит в выборе «ситуа-

ции отказа» в определенном компоненте системы, прослеживании событийных цепочек, которые могли бы привести к ее возникновению, и в построении дерева отказов с использованием связок *и, или*. Данный метод применяется как на модульном уровне, так и на уровне анализа функционирования системы.

Анализ программ методом дерева отказов соответствует доказательству корректности и помогает в описании утверждений для доказательства правильности работы программы.

Метод проверки непротиворечивости применяется при анализе логики программы и выявлении операторов, вычисление по которым не проводится, а также для обнаружения противоречий в логике программы. Этот метод часто называют методом *анализа потоков управления* входных данных, часть из которых представляется в символьном виде [21]. Результатом выполнения являются значения переменных, выраженные формулами над входными данными.

В этом методе решаются две задачи.

1. Построение тестового набора данных для заданного пути, определяющего последовательность операторов при символьном выполнении. В случае отсутствия такого набора делается заключение о нереализуемости (противоречивости) данного пути.

2. Определение пути прохождения при заданных ограничениях на входных данные в виде некоторых областей значений и символьных значений переменных, полученных в конце пути просмотра. Иными словами, строится функция, которая реализует отображение входных данных в выходные.

Опишем особенности метода символьного выполнения.

1. Пусть $P\ (X,\ Y)$ – программа, выполняемая символически на некоторых наборах данных, $D = (d_1,\ d_2,\ ...,\ d_n),\ D \subset X,\ X$ – множество входных данных программы P.

Пронумеруем операторы программы $P=\{P_1,\ P_2,\ ...,\ P_n\}$. Обозначим состояние выполнения программы в виде тройки $< N,\ pc,\ Z >$, где N – номер текущего оператора программы P, pc (part condition) – условие выбора пути в программе (вначале это true), что задается логическим выражением над D; Z – множество пар $\{ < z_i,\ e_i > |\ z_i \in X \cap Y$, в которых z_i – переменная программы, а e_i – ее значение; Y – множество промежуточных и выходных данных.

Семантика символьного выполнения задается базовыми конструкциями ЯП и правилами оперирования символьными значениями, согласно которым арифметические вычисления заменяются алгебраическими.

Для императивных языков базовые конструкции – это операторы присваивания, перехода и условные операторы.

В операторе присваивания $Z = e\ (x, y),\ x \in X,\ y \in Y$, в выражение $e\ (x, y)$ производится подстановка символьных значений переменных x и y, в результате чего получается выражение $e\ (D)$, которое становится значением переменной z $(z \in X \cup У)$. Вхождение в полученное выражение $e\ (D)$ переменных из Y означает, что их значения, а также значение z не определены.

По оператору перехода управление передается оператору, помеченному соответствующей меткой.

В условном операторе «***если*** $\alpha\ (x, y)$, ***то*** B1 ***иначе*** B2» вычисляется выражение $\alpha\ (x, y)$. Если оно определено и равно α' (D), то формируются логические формулы:

$$pc \rightarrow \alpha'\ (D\ ), \qquad (6.1)$$

$$pc \rightarrow \neg\ \alpha'\ (D). \qquad (6.2)$$

Если pc – ложно (*false*), то только одна из последних формул может быть выполнимой, тогда

– если выполнима формула (6.1), то управление передается на оператор B1;

– если выполнима формула (6.2), то управление передается на оператор B2;

– если (6.1), (6.2) не выполнимы (т.е. из pc не следует ни $\alpha'(D)$, ни $\neg\alpha'$ (D)), то по крайней мере имеется один набор данных, который удовлетворяет pc и соответствует части «***то***» условного оператора, а набор данных оператора после «***иначе***» этого условного оператора не определен.

В результате создаются два пути символьного выполнения, которым соответствуют

$$pc_1 = pc \wedge \alpha'\ (D\ ),$$

$$pc_2 = pc \wedge \neg\ \alpha'\ (D\ ).$$

Символьное выполнение, нацеленное на построение тестового набора данных, реализуется следующим алгоритмом:

Пусть pc = *true*, тогда

– для заданного пути формируется pc исходя из семантики условного оператора, задающего преобразование pc в виде (6.1) или (6.2);

– решаются системы уравнений (6.1) и (6.2), если решения нет, то это означает невыполнимость пути.

2. Определение пути при заданных ограничениях на входные данные проводится таким способом:

– полагаем $pc = \beta\ (D)$, где $\beta\ (D)$ – входная спецификация, когда ее нет, то $pc = true$;

– производим символьное выполнение операторов, если встречается ветвление, то запоминается состояние в данной точке или выбирается одна из ветвей; выполняется условный оператор, при котором формируется состояние программы с условием *pc*;

– в конце пути прохождения по программе определяется функция и набор данных, которым удовлетворяет *pc,* покрывающий данный путь;

– для всех промежуточных $\delta\ (x, y)$ выходной спецификации $\gamma\ (x,y)$ проводится доказательство выполнимости следующих логических формул:

$$pc \rightarrow \delta\ (x,y),\ pc \rightarrow \gamma\ (x,y),$$

где *pc* является значением текущего условия в данной точке программы.

Доказательство этих формул проводится путем верификации участка пути программы. Выражение формулы декомпозируется на неравенства для установления некоторых несовместимостей, мешающих прохождению пути.

В целях проведения статического анализа программы используются различные инструменты, позволяющие определить ошибки в программе (например, неинициализированные или не использованные переменные).

Вывод. Техника формального анализа программ еще не означает, что в программе нет ошибок. Эта техника не распознает ошибки в проекте, в интерфейсах с другими компонентами, в интерпретации спецификаций или в документации. При применении международного проекта по верификации ПО [29, 30] недостатки устранятся.

6.2. ВЕРИФИКАЦИЯ И ВАЛИДАЦИЯ ПРОГРАММ

Верификация и валидация – это методы, которые предназначены для анализа, проверки правильности выполнения и соответствия ПО спецификациям и требованиям заказчика [22–26]. Данные методы проверки правильности программ и систем соответственно означают

– верификация – это проверка правильности создания системы в соответствии с ее спецификацией;

– валидация – это проверка правильности выполнения заданных требований к системе.

Метод верификации помогает сделать заключение о корректности созданной системы после завершения ее проектирования и разработки. *Валидация* позволяет установить выполнимость заданных требований и включает в себя ряд действий для получения правильных программ и систем, а именно:

– планирование процедур проверки и контроля проектных решений и требований;

– обеспечение уровня автоматизации проектирования программ CASE-средствами [28];

– проверка правильности функционирования программ методами тестирования на наборах целевых тестов;

– адаптация продукта к операционной среде и др.

Валидация выполняет эти действия путем просмотра и инспекции спецификаций и результатов проектирования на этапах ЖЦ для подтверждения того, что имеется корректная реализация начальных требований и выполнены заданные условия и ограничения. В задачи верификации и валидации входят проверки полноты, непротиворечивости и однозначности спецификации требований и правильности выполнения функций системы.

Верификации и валидации подвергаются:

– основные компоненты системы;

– интерфейсы компонентов (программные, технические и информационные) и взаимодействия объектов (протоколы и сообщения), обеспечивающие функционирование системы в распределенных средах;

– средства доступа к БД и файлам (транзакции и сообщения) и проверка средств защиты от несанкционированного доступа к данным разных пользователей;

– документация к ПО и к системе в целом;

– тесты, тестовые процедуры и входные данные.

Иными словами, основными систематическими методами правильности программ являются.

1. *Верификация* компонентов ПС и валидация спецификации требований.

2. *Инспектирование ПС* для установления соответствия программы заданным спецификациями.

3. *Тестирование* выходного кода ПС на тестовых данных в конкретной операционной среде для выявления ошибок и дефектов, вызванных разными недоработками, аномальными си-

туациями, сбоями оборудования или аварийным прекращением работы системы (см. главу 7).

Верификация и валидация как процессы. Стандарты ISO/IEC 3918–99 и 12207 [27] включают в себя процессы верификации и валидации. Для них определены цели, задачи и действия по проверке правильности создаваемого продукта (включая рабочие, промежуточные продукты) на этапах ЖЦ и соответствия его требованиям.

Основная задача процессов верификации и валидации состоит в том, чтобы *проверить и подтвердить*, что конечный программный продукт отвечает назначению и удовлетворяет требованиям заказчика. Эти процессы позволяют выявить ошибки в рабочих продуктах этапов ЖЦ, без выяснения причин их появления, а также установить правильность программного продукта относительно его спецификации.

Эти процессы взаимосвязанные и определяются одним термином «верификация и валидация» или «Verification and Validation» (V&V).

При верификации осуществляется:

– проверка правильности перевода отдельных компонентов в выходной код, а также описаний интерфейсов путем трассировки взаимосвязей компонентов в соответствии с заданными требованиями заказчика;

– анализ правильности доступа к файлам или БД с учетом принятых в используемых системных средствах процедур манипулирования данными и передачи результатов;

– проверка средств защиты компонентов на соответствие требованиям заказчика и проведение их трассировки.

После проверки отдельных компонентов системы проводятся их интеграция, а также верификация и валидация интегрированной системы. Систему тестируют на множестве наборов тестов для определения адекватности и достаточности этих наборов для завершения тестирования и установления правильности системы.

6.2.1. Валидация требований на основе сценарного подхода

В настоящее время для разработки программ широко используется концепция вариантов использования (use case), основными понятиями которых являются сценарий и актер – внешняя сущность относительно разрабатываемой ПС. [19, 20], который взаимодействует с системой для достижения определенной цели. Сценарий

задает последовательность взаимодействий между одним или несколькими актерами и системой, в результате чего актер выполняет цели сценария. Сценарий задает несколько альтернативных последовательностей событий с помощью формальных средств описания и валидации требований. Данный подход описан в [20] и ориентирован на валидацию требований.

Описание требований к системе задается с помощью сценариев на языке диаграмм взаимодействий UML. Сценарии разделяются на функциональные (системные) и внутренние, определяющие поведение системы.

После составления сценариев требований проводится их валидация. Под *валидацией требований* понимается процесс выявления ошибок в сценариях требований к проектируемой ПС. Описание и валидация требований – это итерационный процесс, состоящий из следующих шагов:

– формализованное описание требований в виде сценариев;
– создание исполняемой модели требований;
– создание сценариев для валидации требований;
– применение валидационных сценариев к модели требований;
– оценивание результатов поведения этой модели;
– проверка условий завершения процесса и при возникновении каких-либо неточностей повторение с создания модели.

При выполнении сценариев возникают штатные или нештатные (ошибочные) ситуации, при которых поведение системы становится недетерминированным. В этих целях проводится контроль покрытия сценариев модели требований валидационными сценариями. Такая проверка связана с возможным возникновением рисков и с последующим обнаружением ошибок. Создается модель ошибок, покрывающая модель требований системы и включающая в себя типичные ошибки, используемые при выводе сценариев. Составной частью валидации требований являются классы эквивалентности входных и выходных данных.

Входной информацией для проведения синтеза сценариев является сценарная модель на языке SDL (рис. 6.3).

Данная модель используется при генерации дополнительных сценариев в целях улучшения валидации и автоматического синтеза всех сценариев модели для получения модели системы. Эта модель проверяется с помощью тестов и модели ошибок, а также позволяет обнаружить неполноту исходных требований или противоречия в требованиях.

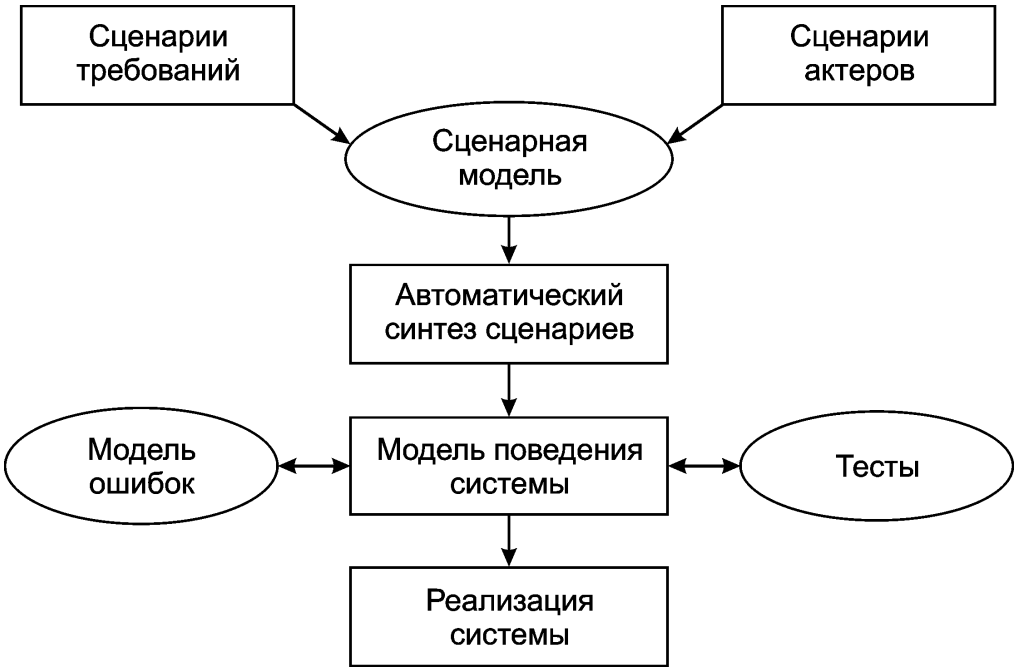

РИС. 6.3. Валидация сценариев требований к системе

Автоматический синтез заключается в следующем:

– валидация требований путем исполнения валидационных сценариев;

– добавление проверенных сценариев к набору валидационных сценариев и их применение в качестве входных данных;

– поиск ошибок и проверка разных композиций сценариев.

Синтез спецификаций по сценариям с помощью языка диаграмм взаимодействия – это генерация кодов системы на основе требований.

6.2.2. Метод верификации объектно-ориентированных программ

Верификация в ООП имеет свою специфику и ее можно рассматривать, исходя из композиционного метода, применяя к объектно-ориентированным программам методы доказательства правильности композиционных программ [21].

Верификация проектирования объектных моделей (ОМ) и программ включает в себя:

1. Базовые объекты в структуре ПрО, атрибутами которых являются данные и внутренние операции объекта – функции над этими данными.

2. Объекты, построенные с помощью операций наследования, агрегации или инкапсуляции, исходя из следующих предположений:

– базовые объекты считаются проверенными, если их операции (функции) используются в качестве теорем;

– доказательство других объектов сводится к этим теоремам;

– все операции, которые применяются над подобъектами, не выводят их из множества состояний.

3. Верификация интерфейсов объектов сводится к доказательству правильности интерфейса связи объектов и достаточности параметров интерфейса.

Метод верификации объектно-ориентированных программ можно использовать как “вниз”, так и “вверх”, но сначала доказывается правильность построения ОМ для некоторой ПрО следующим образом:

– введение дополнительных и (или) удаление лишних атрибутов объекта и его интерфейсов в ОМ, и доказательство правильности объекта после изменений интерфейсов и взаимодействий с другими объектами;

– доказательство правильности выбора типа для атрибутов объекта, т.е. того, что выбранный тип реализует операции и множество его значений включает в себя множество состояний объекта;

– доказательство правильности спецификации объектов ОМ независимо от правильности смежных объектов, верификация параметров интерфейсов, которые передаются другим объектам. Это доказательство является заключительным при проверке правильности ОМ.

6.2.3 Верификация композиции из верифицированных компонентов

В [24] определен подход к верификации композиции компонентов системы, базирующейся на спецификации функций и временных (temporal) свойствах готовых компонентов (типа reuse), для которых проводится проверка этих свойств и их композиций. Свойства составного компонента, состоящего из нескольких подкомпонентов, проверяются с помощью абстракции, включающей в себя проверенные свойства подкомпонентов и условий среды их выполнения. При композиции компонентов создается общая компонентная модель (ОКМ), состоящая из совокупности проверенных компонентов, спецификации их временных свойств и условий функционирования. Проверка компонентов модели ОКМ осуществляется с помощью аппарата асинхронной передачи сообщений (АПС).

Общая компонентная модель. Основу ОКМ составляет модель проверки [16, 17, 24], которая охватывает пространство состояний ПС и предназначена для обнаружения ошибок взаимодействия, возникающих при композиции компонентов. Модель проверки (Model checking) ориентирована на разработку надежных компонентных ПС и включает в себя следующие задачи:

– идентификация правильных компонентов;

– композиция reuse-компонентов по их спецификациям;

– формирование общей спецификации компонентной системы, составленной из "правильных" компонентов и др.

Решение этих задач основано на следующих положениях:

– спецификация компонента задается в языке xUML [17, 24] диалекта UML и включает в себя описание временных свойств для проверки правильности;

– reuse-компоненты представляют функции, спецификацию интерфейса и временные свойства;

– свойства составного компонента из подкомпонентов проверяется композиционным аппаратом.

Модель ОКМ – это структура представления совокупности компонентов и их свойств, а также составных компонентов системы. Модель предназначена для верификации спецификаций компонентов и их выполнения.

Свойство компонента определяется из условий среды компонента. Когда компонент многократно используется в составе составного компонента, эти свойства должны учитывать возможности среды и связи с другими компонентами композиции. ОКМ проверяется на модели вычислений АПС, которая обеспечивает семантику выполнения компонентов и взаимодействие с другими.

Компоненты ОКМ-модели могут быть примитивами и составными. Описание свойств примитива проверяется непосредственно с помощью модели проверки, а свойство составного компонента – на абстракции компонента, составленной из простых компонентов с интегрированным описанием среды и проверенных свойств ее подкомпонентов.

Если абстракция слишком сложная для проверки, то используется композиционный подход для декомпозиции свойств компонентов, проверки этих свойств на отдельных подкомпонентах и включения проверенных свойств в абстракцию.

Данный подход предназначен для использования в распределенных программных системах, функционирующих на платформах типа CORBA, DCOM и EJB.

Формально каждый компонент ОКМ-модели задается в виде:

$$C = (E, I, V, P), \quad (6.1)$$

где E – представление компонента в исходном коде; I – интерфейс компонента C с другими компонентами через механизм передачи сообщений или вызовов процедур; V – множество переменных, определенных в E и связанных со свойствами, определенными на множестве P временных свойств, отражающих особенности среды компонента.

Каждое свойство должно быть проверено на множестве E и является парой $(p, A(p))$, где p – свойство компонента C в E, $A(p)$ – множество временных формул из свойств, определенных на множествах I и V. Свойства компонента C включаются в P только тогда, когда они проверены в среде.

Композиция компонентов представляется совокупностью более простых компонентов:

$$(E_0, I_0, V_0, P_0), \ldots, (E_{n-i}, I_{n-i}, V_{n-i}, P_{n-i}), \quad (6.2)$$

определенных на модели компонента C следующим образом: E создается из множества представлений $E_0, E_1, \ldots, E_{n-1}$, связанных между собой интерфейсами из набора интерфейсов $I = \{I_0, \ldots, I_{n-1}\}$, операций связи Ih $(0 < h < n)$ для обеспечения взаимодействия с другими компонентами;

V – подмножество $\bigcup_{i=0}^{n-1} V_i$ (V_i – ссылка на свойство компонента из C, заданное в P); P – множество временных свойств, определенных на I и V, и проверенных на E с использованием отдельных свойств $P_0, \ldots, P_{n-1}$.

Модель вычислений АПС – это вычислительная модель системы, заданная на конечном множестве взаимодействующих процессов, представленных кортежами:

$$P = (X, \Sigma, Q, \nabla), \quad (6.3)$$

где X – множество переменных, каждая из которых имеет тип; Σ – расширенная модель состояния; Q – очередь сообщений в порядке поступления; ∇ – начальное состояние со значениями для каждой переменной из X, E и пустое состояние для Q.

Расширенная модель состояния определяется кортежами (Φ, M, T), где Φ – множество состояний, каждое из которых связано с ассоциативным действием; M – множество типов сообщений; T – набор переходов, определенных на множествах Φ и M.

Каждое из состояний переходов является кортежем *(r, t, m)*, где *r* и *t* – состояния в *Ф* и *m* – тип сообщения во множестве сообщений *M*. Семантически действие задается сегментом программы, составленным из операторов: пустой оператор, присваивания, передачи сообщений, условный и составной операторы и др.

Выполнение АПС заключается в чередовании переходов состояний и действий процессов, взаимодействующих через асинхронную передачу сообщений. Для двух процессов P_1 и P_2 передача сообщения от P_1 к P_2 включает в себя: тип сообщения *m* из множества *M* для P_2 и параметры. Когда оператор действия выполняется, сообщение *m* с параметрами ставится в очередь к процессу P_2.

Верификация компонентов. Примитивные и составные компоненты проверяются на правильность, исходя из их описания в языке спецификации *xUML,* временными свойствами и соблюдением следующих требований:

– свойства компонентов задаются в xUML;

– описание ОКМ-модели в xUML и свойства компонентов транслируются к свойствам модели АПС;

– при отсутствии свойств у компонента выдаются ошибки.

Свойства компонентов определяются автоматическим или ручным путем с учетом условий среды этих компонентов. Для проверки составных компонентов композиции условия генерируются в виде набора простых и интуитивных предположений.

Верификация примитива. Пространство состояний примитива имеет небольшой размер, необходимый для проверки приложения. Учитывая, что примитив $C = (E, I, V, P)$ и свойство $(p, A(p))$ определены на множествах *I* и *V*, верификация состоит в нахождении условий *p* во множестве *E* и $A(p)$ компонентов из *C* для простых ПС и выполняется следующими шагами:

– создание процесса АПС на множестве входных типов сообщений, модели состояний и выходных сообщений в *I*;

– формирование процессов АПС во множестве *E,* транслирование среды $A(p)$, условий *p* и сопоставление их с моделью.

Верификация составного компонента. Метод для проверки свойства $(p, A(p))$ составного компонента $C = (E, I, V, P)$ из двух компонентов $C_0 = (E_0, I_0, V_0, P_0)$ и $C_1 = (E_1, I_1, V_1, P_1)$ использует свойства, которые были проверены на подкомпонентах C_0 и C_1. Расширим это на случай, когда *C* составлен из *n*-компонентов $C_0, \ldots, C_{n-1}$.

Создается абстракция компонента в виде композиции подкомпонентов, условий среды, интерфейсов передачи сообщений и проверенных свойств. Свойство считается допустимым, если условия среды при проверке содержат свойства входящих в композицию подкомпонентов.

Допустимое свойство – это свойство $(p, A(p))$ компонента C_i для $i \in \{0, 1\}$ и $(p_i, A(p_i)) \in P_i$, может функционировать в композиции C_0 и C_i, если $A(p_i)$ является пустым или для каждого $q \in A(p_i)$, q является предположением $A(p)$ и свойством множества P_{i-1}.

Если свойство $(p, A(p_i))$ компонента C_i, для $i \in \{0, 1\}$ допустимо в композиции C_0 и C_i, то делается вывод, что $A(p)$ содержится в композиции. Для каждого $q \in A(p_i)$ функция первоначально идентифицирует множество P' свойств, которые находятся в P_{i-1} и связаны с q допустимой функции. Подробные сведения о верификации компонентов и их выполнение в АПС приведены в [16].

6.2.4. Верификация сообщений

Верификация сообщений основана на формальном описании функций в ЯП и интерфейса в языке IDL [21]. Для доказательства сообщения в каждую его точку вводятся утверждения, представленные булевыми переменными и значениями. Правильность утверждения в каждой точке вытекает из истинности утверждения во всех предыдущих точках сообщения согласно индуктивности утверждений.

Сообщение состоит из операций обращения к программе, которая посылает данные другой программе. Программе может соответствовать один или несколько процессов, выполняемых на одном компьютере или на разных. Если программе соответствует один процесс, то он содержит входной и выходной порты, которые могут располагаться на разных узлах сети. Сообщение поступает во входной порт и часть информации при пересылке по сети может быть потеряна. Для борьбы с этим вводится понятие таймаута, позволяющего согласовать время и объем переданной по сети информации. При этом усложняется описание утверждений, а следовательно, проверка функционирования сообщений в сети.

Для доказательства правильности сообщения создается набор утверждений, доказывающий, что для любой пары, например, A и B, переход от A к B проходит за один шаг. Действие, выполняемое в промежутке между A и B, приводит к B. При этом часть утверждений проверяет входной сигнал и его поступление на вход про-

цесса с целью подтверждения его на выходе. Если доказано, что процесс, вызванный сообщением, формирует правильный выходный сигнал, то сообщение считается правильным.

Если сообщение отсылается циклически и один из портов находится в тайм-ауте, то ответа может не быть. Для наблюдения за этим явлением, вводится счетчик, который уменьшается на 1 после каждой повторной передачи сообщения. При счетчике равном 0, процесс прекращается, и отправляется ответное сообщение.

Таким образом, полнота доказательства сообщения определяется моделированием процессов и среды, рассмотрением всех входов, прослеживанием путей прохождения протокола по сети и верификации сообщения с помощью набора утверждений.

6.3. ПЕРСПЕКТИВНЫЕ НАПРАВЛЕНИЯ ВЕРИФИКАЦИИ ПРОГРАММ

По данным, опубликованным в [29], ежегодно ошибки в ПО США обходятся в 60 млрд дол. США. Для преодоления этих проблем американские специалисты и специалисты из европейских стран по формальным методам и спецификациям программ приняли решение поставить теоретические достижения в этой области на производственную основу. Этим задан повторный интерес к формальным методам верификации правильности программ. Предпосылки – результаты ряда исследований по формальной верификации моделей ПрО, проверки правильности обращения к функциям в языке API с помощью правил спецификации интерфейсов (проект SDV фирмы Microsoft), проверки безопасности и целостности баз данных и др. Кроме того, в конкретных приложениях начали применяться формальные языки спецификации (RAISE, Z, VDM и др.). Все это послужило основой постановки задач международного проекта по формальной верификации создаваемого ПО [29, 30].

Формальные методы дают возможность экспериментировать с формальными спецификациями и выбирать наилучший ее вариант для практической реализации. Так как формальные методы разрабатывались в университетах и академических учреждениях, практических реализаций было недостаточно, поэтому очень важно поставить их на производственную основу для снижения ошибок в программах.

Идея создания международного проекта по формальной верификации была предложена Т. Хоаром, она обсуждалась на

симпозиуме по верифицированному ПО в феврале 2005 г. в Калифорнии. Затем в октябре этого же года на конференции IFIP в Цюрихе был принят международный проект сроком на 15 лет по разработке «целостного автоматизированного набора инструментов для проверки корректности ПС».

В нем сформулированы следующие основные задачи:

– разработка единой теории построения и анализа программ;

– построение всеобъемлющего интегрированного набора инструментов верификации для всех производственных этапов, включая разработку спецификаций и их проверку, генерацию тестовых примеров, уточнение, анализ и верификацию программ;

– создание репозитария формальных спецификаций и верифицированных программных объектов разных видов и типов.

В данном проекте предполагается, что верификация будет охватывать все аспекты создания и проверки правильности ПО, и таким образом, станет панацеей от всех бед, связанных с постоянным возникновением ошибок в создаваемых программах.

Многие формальные методы доказательства и верификации специфицированных программ прошли практическую апробацию (см. п.6.1.2.). Проделана большая работа международного комитета ISO/IEC в рамках стандарта ISO/IEC 12207:2002 [27] по стандартизации процессов верификации и валидации ПО. Поэтому проблема создания автоматизированного набора инструментов и репозитария для проверки корректности разных объектов программирования является перспективной.

Репозитарий станет хранилищем программ, спецификаций и инструментов, применяемых при разработках и испытаниях, оценках готовых компонентов, инструментов и заготовок методов. На него возлагаются такие общие задачи:

– накопление верифицированных спецификаций, методов доказательства, программных объектов и реализаций кодов для сложных применений;

– накопление всевозможных методов верификации, их оформление в виде, пригодном для поиска и выбора реализованной теоретической идеи для дальнейшего применения;

– разработка стандартных форм для задания и обмена формальными спецификациями разных объектов программирования, а также инструментов и готовых систем;

– разработка механизмов интероперабельности и взаимодействия для переноса готовых верифицированных продуктов из репозитария в новые распределенные и сетевые среды для создания новых ПС.

Данный проект предполагается развивать в течение 50 лет [30]. Более ранние проекты ставили подобные цели: улучшение качества ПО, формализация сервисных моделей, снижение сложности за счет использования ПИК, создание отладочного инструментария для визуальной диагностики ошибок и их устранения и др. [21]. Однако коренного изменения в программировании не произошло ни в смысле визуальной отладки, ни в достижении высокого качества ПО. Процесс продолжается.

Новый международный проект по верификации ПО требует от его участников знаний не только теоретических аспектов спецификации программ, но и высокой квалификации программистов для его реализации в ближайшие годы.

В этой главе рассмотрены методы формального доказательства правильности формального описания отдельных объектов, компонентов и системы в целом. Приведены основные особенности формальных методов проверки правильности компонентов путем доказательства верификации и валидации, включая компонентные программы из верифицированных компонентов. Дана краткая характеристика перспективы развития верификации в рамках международного проекта.

ГЛАВА 7

МЕТОДЫ И ПРОЦЕССЫ ТЕСТИРОВАНИЯ ПС

Тестирование – это процесс обнаружения ошибок в ПС путем исполнения выходного кода на тестовых данных и проверки выходных результатов и рабочих характеристик в конкретной операционной среде [1–3]. С помощью тестирования выявляются различные ошибки, дефекты и изъяны, вызванные недоработками, аномальными ситуациями, сбоями оборудования и аварийным прекращением работы ПО [4–7].

Тестирование составляет от 30 до 50% трудоемкости работ по созданию ПС. В работе [1] дано определение термина *тестирование* как процесса выполнения программы (или ее части) в целях выявления ошибок, а отладка (debugging) программ как способ определения причин найденных ошибок в программе и их исправление. Конечная цель тестирования – получение более надежного и правильного программного продукта [5].

Согласно стандарту [11] тестирование является деятельностью, направленной на анализ выполнения программы, в задачу которого входит проверка разных рабочих продуктов (документов, схем, кодов, таблиц и т.п.) на этапах ЖЦ и на заключительном этапе – полученного программного продукта.

Первым видом тестирования является *отладка* – проверка исходного описания программного объекта на наличие в нем синтаксических и семантических ошибок и их устранение. При этом не исключается, что при устранении ошибок в программу вносятся новые ошибки.

Наряду с отладкой и тестированием используются процессы верификации и валидации для проверки правильности ПС и удовлетворения требований заказчика [ДСТУ 3918–99]. Эти процессы рассмотрены подробно в главе 6.

Тестирование состоит в статической проверке путем инспекции, просмотра программ и документации за столом и в *динамической* проверке путем выполнения программы на *конечном* множестве тестовых данных, специальным образом выбранных из *бесконечного* входного пространства, на соответствие ожидаемому результату [7].

Одна из основных проблем тестирования является неполнота тестирования на множестве тестов, которое можно рассматривать как бесконечное. Это приводит к таким проблемам тестирования, как принятие решений об адекватности тестирования, управлении процессом тестирования, оценки затрат (стоимости, времени, трудозатрат).

В процессе своего развития в тестировании создавалось параллельно несколько направлений:

– исследование и разработка методов тестирования и критериев адекватности тестирования;

– определение метрик тестирования и критериев его завершения;

– формирование моделей оценивания процесса тестирования и др.

Метрики тестирования. В процессе тестирования производится устранение дефектов, ошибок и отказов. В [11, 12] рекомендовано соответствующие им метрики – «количество дефектов», «плотность дефектов» и «интенсивность отказов».

Количество дефектов, выявленных тестированием, служит признаком эффективности тестирования. В то же время небольшое количество дефектов, выявленных во время тестирования, не свидетельствует о плохо выполненном тестировании, а позволяет определить эффективность процесса разработки ПС с помощью отношения:

$$E_e = D_e / (D_e + D_o),$$

где E_e – эффективность тестирования, D_e – количество дефектов, выявленных во время тестирования, D_o – количество дефектов, выявленных при эксплуатации программного продукта.

Проведенный анализ данных о плотности дефектов в зарубежных программных продуктах позволил определить средний показатель плотности дефектов в ПС – это 6 дефектов на 1000 строк у программистов США и Европы, 2 – Японии [12].

Регистрация дефектов – одна из целей процесса тестирования, рассматриваемая согласно требованию базового стандарта по качеству ISO 9000–3, как механизм формирования исторических эталонных данных о ПС.

Для измерения результатов тестирования используются метрики: оценивания набора тестов соответственно критерию покрытия, тенденций дефектов, времени и стоимости тестирования.

Включаемые в процесс тестирования метрики предназначены для оценки состояния выполнения процесса тестирования и самого ПС. Объективными критериями завершения тестирования являются критерии, базирующиеся на:

1) метриках покрытия (требований, функций, кода) структурного и функционального типов, используемых в качестве индикатора полноты выполненного тестирования;

2) серьезности дефектов, которая отражает не процесс выявления дефектов, а процесс их устранения разработчиком;

3) оценке интенсивности отказов ПС (среднее время функционирования без отказа или вероятность безотказной работы), которая не учитывает последствий отказов, а их вероятность;

4) оптимизации времени тестирования и оценки надежности, базирующейся на использовании моделей надежности для поиска «компромисса» между временем тестирования и требованиями к показателю качества – надежности ПС.

7.1. МЕТОДЫ ТЕСТИРОВАНИЯ ПС

Методы тестирования различаются подходами к выбору множества тестовых данных из входного пространства и выполняются согласно приведенным критериям. Выбор наиболее эффективных методов тестирования на разных уровнях тестирования связан с анализом рисков отказов ПС [8, 12].

Различают следующие методы тестирования:

– *статические* методы, используемые во время проведения инспекций и анализа спецификаций компонентов без их выполнения;

– *динамические* методы, применяемые в процессе выполнения программ на тестах, с помощью которых определяется полнота выполнения функциональных задач и их соответствие исходным требованиям;

– *функциональное тестирование*, проводимое для проверки правильности реализации функций в ПС.

Рассмотрим их.

7.1.1. Статические методы тестирования

Техника статического анализа заключается в методическом просмотре (или обзоре) и анализе структуры программ за столом.

Статический анализ направлен на анализ документов, разрабатываемых на всех этапах ЖЦ, инспекции исходного кода и сквозного контроля программы.

Инспекция состоит в совместном рассмотрении документов независимыми экспертами с участием разработчиков. На начальном этапе проектирования инспектирование предполагает проверку полноты, целостности, однозначности, непротиворечивости и совместимости документов с исходными требованиями к ПС. На этапе реализации системы под *инспекцией* понимается анализ текстов программ на соблюдение требований стандартов и принятых руководящих документов технологии программирования.

Инспектирование проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, архитектурных схем или исходного кода программ) на этапах разработки ПО. Исходный код программ и соответствующие документы могут проверяться с помощью средств автоматизации. При оценке реализации нефункциональных характеристик ПО (например, производительности, надежности и др.) используются данные, полученные в результате тестирования. Просмотры и инспекции результатов этапов проектирования и соответствия их требованиям заказчика обеспечивают более высокое качество создаваемых программных компонентов ПО и системы в целом.

Эффективность такой проверки заключается в том, что привлекаемые эксперты пытаются рассмотреть эту проблему «со стороны» и подвергают ее всестороннему критическому анализу. Когда проводится сквозной контроль, то эксперты воспринимают еще и словесное объяснение решаемой задачи, и способы ее проектирования. Это и непосредственный просмотр кода позволяют обнаружить ошибки в логике и в описании алгоритма.

Сквозной контроль состоит в ручной имитации выполнения программы. Разработчик программы объясняет и обосновывает выбранные подходы и методы реализации, подбирает тесты для ручного прослеживания за выполнением программы.

Эти приемы позволяют на более ранних этапах проектирования обнаружить ошибки или дефекты. Методы просмотра не формализованы и определяются степенью квалификации экспертов группы.

Метод простого структурного анализа ориентирован на анализ структуры программы (потоков управления) и потоков данных.

Одним из подходов к анализу структуры программ является использование графа для представления структуры программы, в котором каждая вершина – оператор, а дуга – передача управления между операторами. На основе графа определяется достижи-

мость вершины программы и существование выхода из всех потоков управления при ее завершении. Данный подход позволяет обнаружить логические ошибки [9].

Для проведения анализа потоков данных граф расширяется указателями переменных, их значениями и ссылками на операторы программы. При тестировании потоков данных вначале определяются значения предикатов в операторах логики, по которым формируются пути выполнения программы и проверка вычислений арифметических операций. Для прослеживания путей устанавливаются точки, в которых имеются ссылки на переменную в момент присвоения ей значения, либо переменной присваивается значение без ее описания, либо описывается переменная, к которой нет обращения.

Способы статического тестирования. К ним относятся способы расчета продолжительности выполнения модулей и его характеристик аналитическим путем без выполнения программы на машине. Основу расчета составляет анализ структурных характеристик компонентов программы, требующих значительных затрат времени при их динамическом выполнении. В процессе анализа обнаруживаются ошибки (например, в циклах), которые влияют на продолжительность выполнения составных модулей ПО, а также несоответствия требований ко времени их выполнения в реальной среде.

7.1.2. Динамические методы тестирования

Под динамическим тестированием понимают проверку корректности и надежности разрабатываемой ПС посредством ее выполнения на ЭВМ с применением систематических, статистических (вероятностных) и имитационных методов.

Систематические методы тестирования делятся на методы, в которых программы рассматриваются как «черный ящик», использующий информацию о функциях системы, и методы, в которых программа – «белый ящик», использующий информацию о структуре программы.

Тестирование программ по принципу «черного ящика» называют тестированием с управлением по данным или управлением по входу-выходу. Цель такого тестирования – выяснить обстоятельства, при которых поведение программы не соответствует ее спецификации. При этом количество обнаруженных ошибок в программе является критерием качества тестирования, достигаемого тестовыми наборами входных данных.

Целью динамического тестирования программ по принципу «черного ящика» является выявление с помощью теста максимального числа ошибок [8, 13–15].

Методы «черного ящика» состоят из следующих видов тестирования:

– эквивалентное разбиение;

– анализ граничных значений;

– применение функциональных диаграмм, которые в объединении с реверсивным анализом дают достаточно полную информацию о функционировании тестируемой программы.

Эквивалентное разбиение состоит в разбиении входной области данных программы на конечное число классов эквивалентности так, чтобы каждый тест, являющийся представителем некоторого класса, был эквивалентен любому другому тесту этого класса.

Классы эквивалентности выделяются путем перебора входных условий и разбиения их на правильные, задающие входные данные для программы, и неправильные, основанные на ошибочных входных значениях.

Разработка тестов методом эквивалентного разбиения выходной области данных осуществляется в два этапа: выделение классов эквивалентности и построение тестов. При построении таких тестов проводится символическое (прямое и обратное) выполнение программы, исходя из ограничений на выходные данные.

При обратном символическом выполнении (в начало программы) можно подобрать ограничения на входные данные, при которых программа выполнялась бы в ошибочном направлении, и с учетом ограничений сгенерированных тестов. Данный подход позволяет целенаправленно разбивать входную область данных на эквивалентные классы в соответствии с потенциально допустимыми ошибками на выходе программы.

С точки зрения семантики, методы тестирования по принципу «черного ящика» используются для тестирования функций, реализованных в программе, путем проверки несоответствия между реальным поведением функций и ожидаемым поведением с учетом спецификаций требований. Во время подготовки к этому тестированию строятся таблицы условий, причинно-следственные графы и области разбивки. Кроме того, подготавливаются тестовые наборы, учитывающие параметры и условия среды, которые влияют на поведение функций. Для каждого условия определяется множество значений и ограничений предикатов, с помощью которых программа тестируются.

Методы «белого ящика» позволяют исследовать внутреннюю структуру программы, причем обнаружение всех ошибок в программе является критерием исчерпывающего тестирования маршрутов и потоков передач управления.

Поскольку критерии исчерпывающего тестирования путей соответствуют полному структурному комбинаторному тесту, то на практике используются, как правило, более слабые критерии, а именно:

– покрытие операторов набором тестов, обеспечивающих прохождение каждого оператора не менее одного раза;

– критерий тестирования ветвей (известный как покрытие решений или покрытие переходов), т.е. набор тестов, который должен обеспечить прохождение каждой ветви (каждого выхода оператора), по крайней мере, один раз.

Критерий покрытия соответствует простому структурному тесту и основывается на построении набора путей, охватывающих все ветви программы. Нахождение оптимального покрытия является гарантией более полного тестирования программы. Методы тестирования по принципу «белого ящика» ориентированы на проверку прохождения всех путей программ и включают в себя путевое тестирование, которое применяется на уровне модулей и для небольших программ. В его основе лежит графовая модель программы, используемая для создания тестовых ситуаций, и включает в себя проверку:

– операторов, которые должны быть выполнены хотя бы один раз без учета возможных ошибок (например, передача управления на неправильный оператор) и логических путей, в которые эти операторы входят;

– путей в графе потоков управления, порождающих разные маршруты передач управления и задаваемых с помощью путевых предикатов, в каждый из которых входит набор уравнений, неравенств и тестовых данных для проверки этих путей. Могут остаться некоторые неохваченные пути, которые станут источником ошибок в процессе эксплуатации;

– блоков, разделяющих программы на отдельные части, которые выполняются, по крайней мере, один раз, а в сложных комбинациях – многократно посредством прохождения путей программы, охватывающих этих блоки и реализующих одну функцию.

Иными словами, методы «белого ящика» базируются на структуре исходной программы, позволяющей строить разнообразные тесты для прохождения всех путей выполнения програм-

мы, а методы «черного ящика» скрывают структуру программы, но предоставляют перечень функций, входные и выходные данные.

Способы динамического тестирования. Их можно распределить на два типа. Первый тип непосредственно обеспечивает выполнение программ соответственно тестовым заданиям, второй – вспомогательные способы, которые вычисляют результаты тестирования и проводят необходимые корректировки программ. Ко второму типу относятся способы трансляции задач с языка отладки, выполнение программ по отлаженной программе и регистрация данных о результатах тестирования. Тестовые значения превращаются в форму, пригодную для выполнения отлаживаемой программы. Операторы отлаженной задачи объединяются с программой, которая тестируется, или выполняется в режиме интерпретации.

Способы управления тестированием ПС базируются на отладочных заданиях, определяющих проверку выполнения отдельных участков программы и полученных результатов тестирования для их сопоставления с эталонными значениями.

В процессе выполнения программ в режиме тестирования результаты могут отображаться на дисплей (путь прохождения по графу) в виде последовательности диаграмм UML, а также сведений об отказах или ошибках, или значениях параметров программы. Эти данные анализируются разработчиками для того, чтобы сделать вывод о результатах проверки программы или о завершении ее тестирования.

7.1.3. Функциональное тестирование

Основная цель функционального тестирования – обеспечение полноты и согласованности реализованных в программных компонентах функций и интерфейсов между ними. При тестировании возникают ошибки и ситуации, которые делают невозможной правильную работу компонентов системы, а именно:

– ошибки пользователя при обращении к системе или во время подготовки данных;

– непредвиденные ситуации, возникающие при функционировании системы (не выявленные при проектировании);

– ошибочное выполнение некоторой функции;

– не выполнение некоторой услуги;

– случайные сбои аппаратуры и т.п.

Реакцией системы на исключительные ситуации может быть:

– двойное вычисление и сравнение результатов (в том числе выполненных на разных процессорах) с эталонными;

– задание интервалов времени для фиксации текущего состояния системы;

– проверки корректности данных, передаваемых внешней системе или компонентам этой системы.

– введение заранее неправильных данных и др.

В случае аварийного завершения работы системы, восстанавливается ее предшествующее состояние, для которой применяется новая стратегия:

1) проверки выполнения функций или услуг;

2) корректировка программы и повторное ее выполнение со старыми данными;

3) поиск ошибок, исправление и повторное выполнение;

4) при аварийной ситуации для принятия решения о доработке системы привлекается заказчик.

Для уменьшения вероятности появления исключительных ситуаций в системах повышенного риска (космические и ядерные системы, управление реальными объектами и др.) предусматривается дублирование процессов и дополнительные проверки.

Результаты тестирования функций системы используются при оценке надежности системы с учетом исключительных ситуаций, отказов и восстановления предшествующих состояний и внесенных исправлений.

К функциональному тестированию относится и метод *«черного ящика»* для обнаружения несоответствий между реальным поведением реализованных функций и ожидаемым поведением посредством проверки соответствия спецификаций исходным требованиям.

Для этого метода создаются функциональные тесты, которые должны охватывать проверку всех функции ПС с учетом наиболее вероятных типов ошибок. Тестовые сценарии, объединяющие тесты, проверяют качество выполнения функциональных задач, создаются по внешним спецификациям функций, проектной информации и текстам на ЯП. Тесты, основанные на внешних спецификациях ПС, применяются на этапе комплексного тестирования и испытаний системы для проверки полноты выполнения функциональных задач и их соответствия исходным требованиям.

Функциональному тестированию предшествует функциональный анализ ПС, в задачи которого входят:

– определение функциональных требований;

– спецификация внешних функций для реализации в ПС и выявлении последовательностей функций в соответствии с их использованием;

– описание множества входных данных для каждой функции и определение областей их изменения;

– построение тестовых наборов и сценариев тестирования функций;

– выявление и представление всех функциональных требований с помощью тестовых наборов, проведение поиска ошибок в программе и во взаимодействии со средой функционирования.

Тесты, создаваемые по проектной информации, учитывают структуру данных, алгоритмы, интерфейсы между отдельными компонентами и применяются для тестирования отдельных компонентов и интерфейсов между ними. Основная цель данного тестирования – обеспечение полноты и согласованности реализованных функций и интерфейсов.

Предпосылками функционального тестирования являются:

– корректное формирование требований и ограничений к ПС и к качеству;

– корректное описание модели функционирования ПС в среде эксплуатации;

– адекватность модели ПС заданному классу.

Комбинированный метод «черного ящика» и «белого ящика» основан на разбиении входной области функции на подобласти обнаружения ошибок. Подобласть содержит однородные элементы, которые обрабатываются корректно либо некорректно. Для тестирования подобласти выполняются программы на одном из элементов этой области.

Подобласть характеризуется категориями ошибок, дающих возможность сосредоточить поиск на вероятных ошибках на множестве входов функции. Методология поиска вероятных ошибок использует спецификации и выходной код для разбиения программы по входам в функции, а также по последовательным путям программы.

Если разбиения пересекаются, то создается множество классов эквивалентности, путем выбора одного элемента из каждого класса разбиения с одинаковыми свойствами.

Автоматизация тестирования обеспечивает автоматизированный анализ полноты теста, трассировку путей и контроль набора утверждений, устанавливающих факт проверки правильности работы программы.

Утверждения могут быть локальными и глобальными. Локальное утверждение должно быть истинным в точке его описания, а глобальное – в любой точке программы. Их контроль состоит в оценке выполнения программы и регистрация их истинности или ложности. С помощью анализатора теста обеспечивается учет выполнения каждого логического сегмента программы, которые предварительно разделяется на достаточно малые сегменты программы на уровне логических переходов, являющихся началом двух или более таких сегментов.

После выполнения тестового примера анализатор выдает таблицу сегментов об их выполнении, показывающую число выполненных сегментов для заданного теста. Если после выполнения всех тестовых примеров, окажется, что для одного или более сегментов число выполнения равно 0, то это означает наличие непроверенной части программы.

7.2. ОШИБКИ В ПРОГРАММАХ И ПРИЧИНЫ ИХ ПОЯВЛЕНИЯ

7.2.1. Определение видов ошибок

В процессе тестирования сложились такие общие понятия: ошибка, дефект, отказ, аномалия, сбой, имеющие разные толкования и определения в современной литературе, включая стандарты. Среди разных существующих определений рассмотрим следующие.

Ошибка – это состояние программы, при котором выдаются неверные результаты, причина которых состоит в неправильном описании операторов программы или в ее разработке, что приводит к неправильной интерпретации входной информации или к неверному решению [8, 12–18]. Ошибка может быть следствием недостатка в одном из процессов разработки ПС, который приводит к неправильной интерпретации информации, заданной человеком как неверное решение.

Все ошибки, которые возникают в программах, принято подразделять на несколько классов:

- логические и функциональные;
- вычислений и времени выполнения;
- ввода-вывода и манипулирования данными;
- интерфейсов;
- объема и прочие.

Дефект в программе – следствие ошибок разработчика на любом из этапов разработки и ошибок, которые содержатся во входных или проектных спецификациях, текстах кодов программ, в эксплуатационной документации и т.п. Иными словами, дефект является следствием ошибок разработчика на любом из процессов разработки, в описании спецификаций требований, начальных или проектных спецификациях, эксплуатационной документации и т.п. Дефекты, не выявленные в результате проверок, являются источником потенциальных ошибок и отказов системы в целом. Они могут появиться как отказ, в зависимости от выполнения пути программы и входных данных.

Отказ – это отклонение программы от функционирования или невозможность выполнять функции, определенные требованиями и ограничениями. Иными словами – это событие, способствующее переходу программы в нерабочее состояние по причине ошибок в программе или в среде функционирования. Отказ может быть вызван внешними факторами (изменениями элементов среды эксплуатации) и внутренними (дефектами в самой ПС), а также быть следствием таких причин:

– спецификация не отображена в требованиях заказчика;

– спецификация содержит требования, которое невозможно выполнить на данной аппаратуре и ПО;

– проект программы содержит ошибки, связанные с нефункциональными требованиями (например, база данных спроектирована без защиты от несанкционированного доступа пользователя);

– программа может быть неправильной из-за недоработки алгоритма.

Однако не каждый дефект ПС может вызвать отказ или может быть связан с дефектом в ПС или отказом операционной среды. Любой отказ (как событие) может вызвать *аномалию* из-за внешних ошибок или дефектов.

Если во время тестирования выявлены отклонения результатов выполнения от ожидаемых пользователем, инициируется их поиск и выяснение причин.

Таким образом, появление отказа, как правило, это результат одной или более ошибок в программе, а также наличия в ПС разных дефектов.

Правильно оттранслированная программа может также выдавать ошибки, если для ее выполнения неверно заданы некоторые граничные условия или обращение к компоненту задано неверно.

7.2.2. Классификация ошибок

Классификация ошибок и отказов проведена в международном стандарте AMSİ/İEEE–729–83 и касается определения общих понятий: ошибка, дефект, отказ.

Каждая организация по разработке ПО сталкивается с проблемами нахождения ошибок и поэтому в зависимости от типа и вида ПС ей приходится вводить собственную классификацию типов ошибок и определять пути их устранения. Классификация ошибок позволяет объяснить появление ошибок под разным углом зрения процесса разработки: функций системы, типичных областей и действий при появлении ошибок и др. Рассмотрим некоторые практические подходы к классификации ошибок.

1. Подход к классификации ошибок фирмы IBM называется ортогональной классификацией дефектов ПО и предусматривает разбиение ошибок по категориям с учетом ответственности разработчиков за эти ошибки [13].

Данная классификация ориентирована на программный продукт и организационно не зависит от управления этапами разработки ПО. В табл. 7.1 приведен список типов ошибок и их расположение в разных видах промежуточных продуктов процесса разработки. Каждая пропущенная ошибка, как правило, свидетельствует о получении неточного результата или об его отсутствии, причиной которого может быть либо неинициализированная переменная, либо такой переменной присвоено неправильное значение.

ТАБЛИЦА 7.1. Ортогональная классификация дефектов IBM

Контекст ошибки	Классификация дефекта
Функция	Ошибки интерфейсов конечных пользователей ПО вызваны аппаратурой или обусловлены глобальными структурами данных
Интерфейс	Ошибки во взаимодействии с другими компонентами в вызовах, макросах, управляющих блоках или в списке параметров
Логика	Ошибки в программной логике, неохваченной валидацией, а также использовании значений переменных
Присваивание значений	Ошибки в структуре данных или в инициализации переменных отдельных частей программы
Нерегулярные события (зацикливание, сбой)	Ошибки, вызванные ресурсом времени, реальным временем или разделением времени
Среда хранения или выполнения	Ошибки в репозитарии, в управлении изменениями или в контролируемых версиях проекта
Алгоритм	Ошибки, связанные с обеспечением эффективности, корректности алгоритмов или структур данных системы
Документация	Ошибки в записях документации по запуску или сопровождению

Ортогональность схемы классификации заключается в том, что любой термин классификации относится только к одной категории. Иными словами, прослеживаемая ошибка в системе должна находиться в одном из классов, что дает возможность двум разработчикам классифицировать ошибки одинаковым способом.

2. В подходе *Буча* [13] дана классификация ошибок, исходя из категорий обнаруживаемых ошибок (рис. 7.1), которые связаны с объектами тестирования, и документов, на основании которых делаются выводы (текст курсивом в прямоугольниках второго уровня).

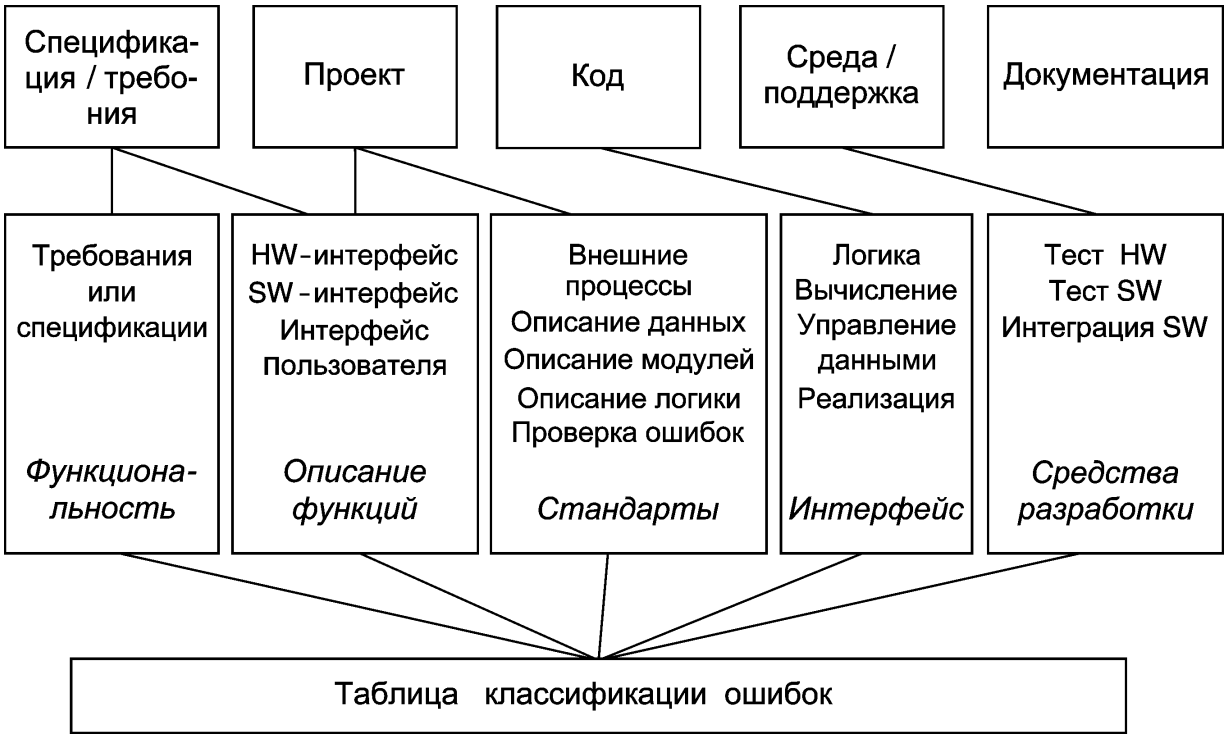

РИС. 7.1. Классификация ошибок, по Бучу: SW – ПО, HW– аппаратура

В классификации Буча модель и схема выбора ошибок задается по трем измерениям:

– место нахождения ошибки;

– тип ошибки;

– характер ошибки (пропуск, неясность, неправильная запись, обмен и др.).

7.2.3. Характеристика ошибок на этапах ЖЦ

Причинами ошибок в ПС на этапах ЖЦ могут быть:

– неумышленное отклонение разработчиков от принятых стандартов или планов реализации системы;

– спецификации требований представлены без соблюдения стандартов разработки;

– недостатки в управлении тестированием со стороны менеджера ресурсов проекта (человеческих, технических, программных и др.) в отдельных элементах проекта.

Рассмотрим виды ошибок, которые могут возникать при разработке ПС на разных этапах ЖЦ (рис. 7.2 – сводка типовых ошибок на этапах ЖЦ).

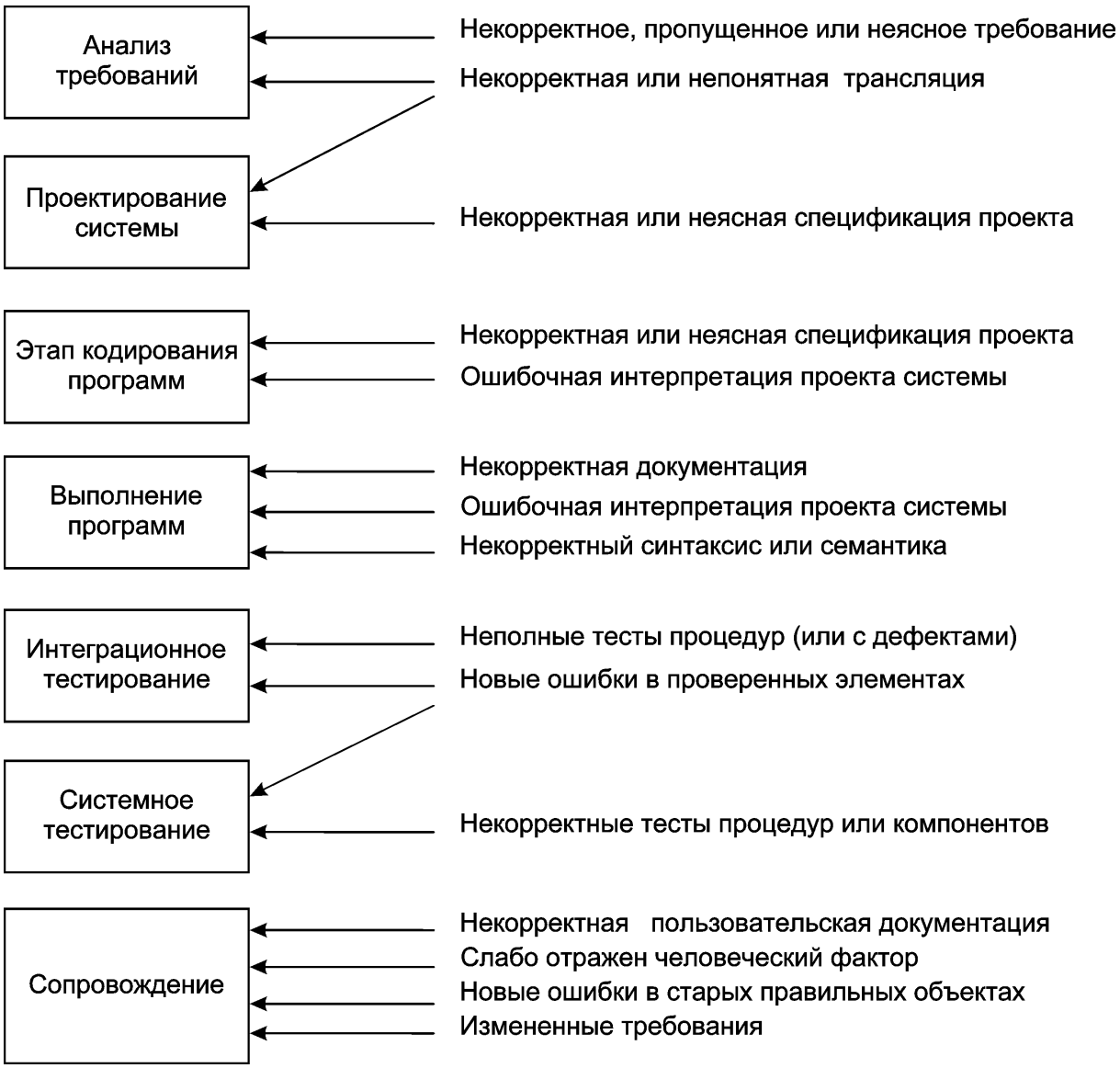

РИС. 7.2. Типы ошибок на основных этапах ЖЦ

1. Этап анализа требований. При определении исходной концепции системы и входных требований заказчика имеются ошибки аналитиков в формулировке спецификации верхнего уровня и концептуальной модели предметной области.

Характерные ошибки:

– неадекватность описания спецификаций требованиям конечных пользователей;

– некорректность описания взаимодействия ПО со средой функционирования или с пользователями;

– несоответствие требований заказчика к отдельным или общим свойствам ПО;

– некорректность в описании функциональных характеристик;

– необеспеченность инструментальными средствами реализации требований заказчика и др.

2. Этап проектирования компонентов. Ошибки во время проектирования компонентов могут возникать при описании алгоритмов, логики управления, структур данных, интерфейсов, логики моделирования потоков данных, форматов ввода-вывода и т.п. В основе этих ошибок лежат дефекты при спецификации программ аналитиками или при проектировании.

Характерные ошибки:

– в определении интерфейса пользователя со средой;

– в описании функций (неадекватности сформулированной в проекте цели и задач отдельных компонентов);

– в определении процесса обработки информации или взаимосвязей компонентов;

– в определении данных и их структур в отдельных компонентах (некорректность);

– некорректное определение структуры модуля;

– нарушение требований, принятых для проекта, без учета стандартов и технологий.

3. Этап кодирования и отладка. На данном этапе возникают ошибки, которые являются результатом дефектов проектирования, ошибок программистов, менеджера процесса разработки и отладки отдельных программ.

Характерные ошибки:

– бесконтрольность значений входных и исходных параметров (например, деление на 0 и др.);

– неправильная обработка нерегулярных ситуаций во время возврата из подпрограмм;

– нарушение стандартов кодирования (плохие комментарии, нерациональное выделение модулей или компонентов);

– использование одного имени для обозначения нескольких объектов или нескольких имен для обозначения одного объекта;

– несогласованное внесение изменений в программу разными разработчиками и т.п.

4. Этап тестирования. На этом этапе ошибки допускают программисты, выполняя технологию сборки и тестирования компонентов, а также тестовики, которые выбирают тестовые наборы и сценарии тестирования и т.п. Отказы в ПО вызываются приведенными выше типами ошибок и отражаются на статистике ошибок и отказов в ПО.

5. Этап сопровождения. При сопровождении ПО причиной ошибок являются дефекты эксплуатационной документации, недостаточные указатели модификации и понятности ПО, а также некомпетентность лиц, ответственных за сопровождение и/или усовершенствование ПО. В зависимости от сущности внесенных изменений на этом этапе могут возникать практически любые ошибки.

7.2.4. Тесты программ и систем

Для проверки правильности программ специально разрабатываются тесты и тестовые данные. Под *тестом* понимается некоторая программа, предназначенная для проверки работоспособности другой программы и обнаружения в ней ошибочных ситуаций. Тестовую проверку можно провести также путем введения в проверяемую программу отладочных операторов, которые будут сигнализировать о ходе ее выполнения и получении результатов.

Тестовые данные служат для проверки работы системы и готовятся разным способом: генератором тестовых данных, проектной группой на основе внемашинных документов или имеющихся файлов, пользователем по спецификациям требований и т.д. Очень часто разрабатываются специальные формы входных документов, в которых отображается процесс выполнения программы с помощью тестовых данных. Тесты способствуют установить:

– полноту набора реализованных функций и их корректность;

– согласованность интерфейсов между программами;

– правильность функционирования системы в заданных условиях;

– надежность выполнения программ;

– защищенность от сбоев аппаратуры или от невыявленных ошибок.

Тестовые данные отражают стадии процесса разработки и виды проверки по разным объектам тестирования (рис. 7.3).

РИС. 7.3. Классификация тестов проверки объектов тестирования

Некоторые тесты зависят от цели и необходимости установления: работает ли система в соответствии с ее проектом, удовлетворены ли требования и участвует ли заказчик в процессе тестирования. Наблюдение за процессом тестирования проводит команда (группа) тестовиков, составляющая тесты для проверки промежуточных результатов проектирования элементов системы на этапах ЖЦ, а также тесты испытаний окончательного программного продукта системы.

Тесты интегрированной системы. Для проверки отдельных объектов системы и интегрированной системы в целом создаются тесты, которые имеют специфические и общие черты. На рис. 7.4 в качестве примера приведены общие действия этапов интеграции системы из готовых верифицированных объектов (объект 1, объект 2 и т.д.) и связи между разными действиями тестирования объектов тестирования.

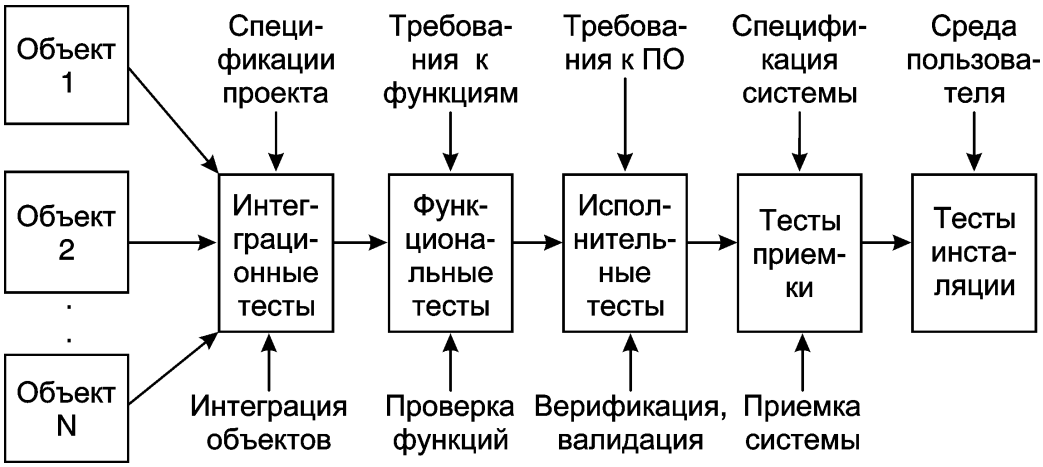

РИС. 7.4. Виды тестов для разных объектов интеграционного тестирования ПС

Для каждого объекта системы команда тестовиков готовит тесты и наборы данных, используемых для проверки разных состояний интегрированной системы в соответствии с заданными требованиями к объектам тестирования.

Проверка функционирования системы проводится с помощью функциональных тестов, включающих в себя тесты проверки функций и требований к ним. Тест испытаний предназначен для проверки системы, который составляется в соответствии с требованиями, ограничениями и условиями заказчика к реальной среде, в которой система будет функционировать.

7.3. ОРГАНИЗАЦИЯ УПРАВЛЕНИЯ ТЕСТИРОВАНИЕМ

На этапе тестирования ПО проводится контроль функционального и технологического соответствия системы стандартам и регламентированным документам. Согласно стандарту создается специальная группа тестовиков, в задачу которой входит всесторонняя проверка созданного программного продукта.

7.3.1. Задачи команды тестовиков

За функциональные и исполнительные тесты несут ответственность разработчики, а заказчик влияет на составление тестов испытаний и инсталляции системы. Как правило, команда тестовиков не зависит от штата разработчиков ПС, некоторые члены этой команды могут быть иметь опыт работы или даже быть профессионалами в этой области. К ним относятся также аналитики и программисты, которые работают вместе с тестовиками, используют спецификации для тестирования, создания тестов и планов их выполнения в системе в заданной среде. Тестовики включаются в процесс разработки с самого начала создания проекта для составления тестовых наборов и сценариев, а также графиков составления тестов.

Члены этой команды проводят анализ требований к системе на полноту, согласованность и тестируемость, а также участвуют в составлении планов тестирования. Они взаимодействуют с заказчиком при проведении опытной эксплуатации системы и регистрации ошибок, а также проводят согласование вопросов устранения обнаруженных дефектов и отказов.

Профессиональные тестовики работают также совместно с группой управления конфигурацией, составителями документации на требования и спецификацию проекта. Они разрабатывают методы и процедуры тестирования. В эту команду включаются

дополнительные специалисты, которые знакомы с требованиями к системе или с подходами к их разработке, а также аналитики, которые проектировали спецификацию и архитектуру системы по требованиям заказчика, проводили декомпозицию системы на подсистемы и функции.

Многие специалисты сравнивают тестирование системы с созданием новой системы, в которой аналитики отражают потребности и цели заказчика, работая совместно с проектировщиками и добиваясь разбора идей и принципов работы системы для решения поставленных задач перед тестированием.

После подготовки тестов в состав команды подключаются проектировщики для анализа возможностей системы и составления тестовых сценариев. Ошибки, обнаруженные при выполнении тестов, фиксируются в документации, отражаются в требованиях к проекту, а также в описаниях входных и выходных данных. Необходимость внесения изменений приводит к модификации тестовых сценариев, конфигурации и соответственно к изменению планов тестирования.

Пользователи, входящие в команду тестовиков системы, оценивают получаемые результаты, удобство использования системы и высказывают мнение о принципах работы системы. Представители от заказчика планируют работы по сопровождению системы и внесению некоторых изменений в проект при условии, что требования не полностью реализованы или были обнаружены ошибки в системе.

7.3.2. Планы тестирования

Для проведения тестирования создается план (Test Plan), в котором описываются стратегии, ресурсы и график тестирования отдельных компонентов и системы в целом с учетом мнения разных членов команды, выполняющих определенные роли в процессе тестирования (рис. 7.5).

В план включаются места расположения тестов, степень покрытия компонента тестами и части тестов, которую выполняют со специальными данными. Тестовики создают множество тестовых сценариев (Test Cases), каждый из которых предназначен для проверки результата взаимодействия актера с системой в соответствии с определенными пред- и постусловиями использования таких сценариев. Сценарии в основном относятся к тестированию по типу «белого ящика» и ориентированы на проверку структуры и операций интеграции компонентов системы.

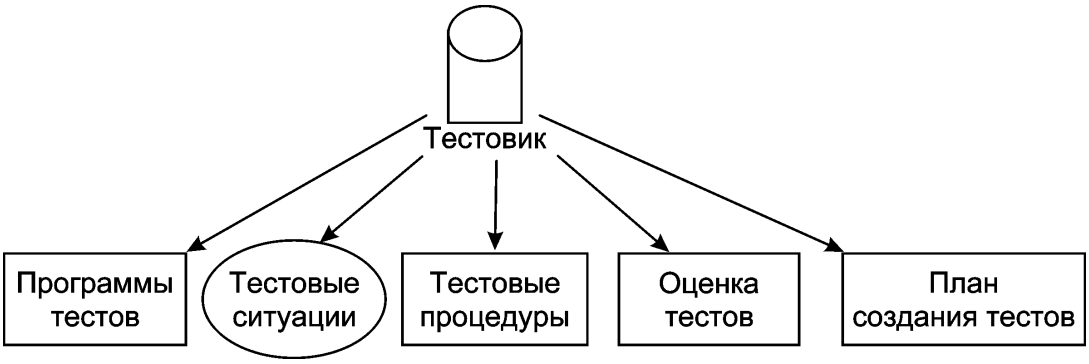

РИС.7.5. Виды тестовых объектов тестовика

Для проведения тестирования тестовики задают процедуры тестирования (Test Procedures), включающие в себя валидацию и верификацию тестовых сценариев для объектов в соответствии с планом графиком.

Оценка тестов (Test Evaluation) заключается в оценке результатов, степени покрытия компонентов сценариями и статуса полученных ошибок.

Тестовик интеграционной системы проводит тестирование интерфейсов и оценку результатов выполнения соответствующих тестов. При выполнении системных тестов, как правило, находятся дефекты, являющиеся следствием глубоко скрытых погрешностей в программах, обнаруживаемых при прогонках системы на тестовых данных и сценариях. При этом тестовики проводят сбор, регистрацию и анализ дефектов, обнаруженных в процессе тестирования.

Планирование процесса тестирования основываются на структуре системы: циклы, пути программы в виде графа, предикаты управления выполнением путей и границы областей изменения параметров и переменных. Планирование путей включает в себя подготовку тестов, критериев покрытия (например, процент проходов, отказов в тестовых требованиях и др.) и входных данных.

План описываются предикатами с условиями выполнения программы, но без циклов. Исходные данные обеспечивают определение вероятного разветвления вершин графа и характеристик путей выполнения. Циклы, как правило, проверяются отдельно на соответствующих входных данных, после чего исключаются из графа выполнения программы. Все перечисленные данные используются при многоразовом прохождении тестов в выполняемой программе.

План комплексного тестирования включает в себя сроки завершения проверки функций и интерфейсов компонентов в ком-

плексной системе, а также выявление дефектов, не обнаруженных при автономном тестировании отдельных компонентов.

План установки системы у заказчика включает в себя план тестирования конфигурации и настройки системы на заданной платформе и операционной среде.

7.3.3. Документирование результатов тестирования

В соответствии с действующим стандартом ANSI/IEEE 829 результаты отладки и тестирования ПС на всех этапах ЖЦ должны собираться в специальных документах, в которых должны содержаться:

– общее описание задач, назначение и содержание ПС, а также описание ее функций в соответствии с требованиями заказчика;

– описание технологии разработки системы;

– описание планов тестирования различных объектов, необходимых ресурсов, в частности соответствующих специалистов для проведения тестирования и технологических операций;

– спецификация тестов, контрольных примеров, критериев и ограничений, накладываемых на оценку процесса тестирования;

– отчет об аномальных событиях, отказах и дефектах с итоговыми результатами тестирования компонентов и системы в целом.

После проведения тестирования проводится анализ данных о результатах тестирования, которые содержатся в журналах, отчетах и др. материалах. По результатам анализа проводится оценка степени достижения установленного заказчиком критерия завершения тестирования.

Группа тестирования готовит отчет о результатах тестирования, содержащий все действия по проведенному тестированию. В него включается выводы по результатам тестирования и оценки критериев прохождения тестов каждым объектом. В заключение приводятся данные об использованных ресурсах, стоимости и времени тестирования, включая разработку тестов и их выполнение.

7.4. ОЦЕНКА СТЕПЕНИ ТЕСТИРУЕМОСТИ ОБЪЕКТОВ ПС

Обязательным условием получения программной системы без ошибок или с минимальным количеством ошибок является тестирование компонентов, интерфейсов, сообщений и системы в целом. При тестировании находятся, устраняются ошибки и

проводится оценка степени тестируемости элементов ПС. Подход к оценке степени тестируемости базируется на встраивании специальных механизмов контроля и наблюдения за объектами ПС для проверки их корректности [13−18] и оценки степени тестирования каждого элемента и ПС в целом.

Оценка степени тестируемости компонентов. Исходя из того, что описание каждого компонента в ЯП включает в себя базовые структуры управления (БСК) − операторы вычислений, предлагается механизм тестирования, который обеспечивает контроль обращений к БСК для поиска ошибок. Для оценки степени тестирования компонентов используются специальные метрики: контролируемость тестирования (КТ) и слежение (наблюдаемость) за тестированием (СТ).

Метрика КТ имеет количественные значения, которые зависят от БСК:

$$КТ_{БСК} = \begin{cases} 1, \text{ если БСК являются независимыми друг от друга;} \\ 0, \text{ если БСК зависят от пути выполнения, или от выполнения выражения.} \end{cases}$$

Степень тестируемости компонента ТК вычисляется по формуле

$$ТК = 1/n \sum_{i=1}^{n} КТ_{бск} ,$$

где $КТ_{бск}$ − контролируемость БСК компонента на области [0, 1]. Для проверенного компонента метрика ТК может иметь такие значения:

$$КТ = \begin{cases} 1, \text{ если тестирование компонента целиком проконтролировано;} \\ 0 \text{ , если тестирование компонента не проконтролировано;} \\ <0<1, \text{ если тестирование компонента частично проконтролировано.} \end{cases}$$

Метрика слежения СТ определяет просмотр операторов и значений, которые принимает любая переменная на пути следования тестирования. Область определения СТ совпадает с областью КТ. Если для анализируемого компонента СТ =1, то это означает, что компонент исследован, а если СТ = 0 − нет.

Очевидно, что чем большее значение КТ и СТ, тем больше степень полноты тестирования компонента. Тестируемость компонента ТК является функцией от двух метрик КТ и СТ, т.е. ТК = *f* (КТ, СТ) = КТ*СТ, где ТК = 1, когда метрики КТ и СТ равны 1, ТК = 0, когда хотя бы одна из метрик равна 0.

Область значений ТК ∈ [0,1] та же, что и у метрики КТ и СТ. При ТК < 1 получаем полную тестируемость компонента, при ус-

ловии, что встроенные механизмы контроля и наблюдения за тестированием выдают соответствующие данные КТ и СТ для окончательной оценки степени тестирования компонентов системы.

Оценка степени тестируемости интерфейсов. Сущность концепции данной оценки состоит в проверке правильности описаний интерфейсов взаимодействующих компонентов ПС и выполнении операций вызовов компонентов без сети, а потом их вызовов в среде сети.

Для проверки корректности данных, которые передаются по сети, разработана контролирующая программная функция – контролер интерфейсов, которая представляет собой удаленный системный компонент повторного использования. Для обращения к этой функции в описание интерфейса включаются специальные операции контроля и наблюдения за процессом вызова удаленного компонента, его выполнение и возврат результата.

Механизм управления тестированием интерфейсов компонентов в процессе взаимодействия с другими компонентами среды обеспечивается контролером интерфейсов и специальной метрикой, которая предназначена для независимой оценки степени проверки интерфейса в режиме тестирования. Фактически встроенный механизм слежения накапливает информацию о результатах проверки правильности переданных и/или полученных данных между взаимодействующими компонентами распределенной среды.

Метрика интерфейса МИ определяется по формуле

$$\text{МИ} = \frac{1}{n}\sum_{i=1}^{n} \text{К}_i ,$$

где К_i – результат контроля *i*-го интерфейса компонента.

Областью определения МИ является [0, 1]. Если МИ = 1, то интерфейс целиком проконтролирован, если МИ = 0, то он не проконтролирован. В других случаях этот объект частично проконтролирован.

Метрика наблюдения за интерфейсом *СИ* – это свойство независимого наблюдения за интерфейсами и просмотра значений выполненного запроса/ответа после взаимодействия компонентов и определяется по формуле

$$\text{СИ} = 1/\text{n}\sum_{i=1}^{n} \text{СИ}_i ,$$

где СИ_{Ii} – значение степени наблюдения за *i*-м интерфейсом компонента.

Количественные значения метрики СИ:

$$\text{СИ} = \begin{cases} 1, \text{ если проведено наблюдение за значениями } \text{СИ}_{Ii} \text{ в интерфейсе;} \\ 0 - \text{в противном случае.} \end{cases}$$

Для определения полной наблюдаемости ПС имеется некоторая трудность, обусловленная ограничениями на тестируемость точек выполнения сообщения.

Очевидно что, чем больше значение МИ и СИ, тем легче провести оценку полного тестирования интерфейсных объектов распределенного приложения. Метрика полной тестируемости интерфейсного объекта ТИО – это функция управления и наблюдения за тестированием интерфейсов объекта:

$$\text{ТИО} = f\,(\text{МИ, СИ}) = \text{МИ*СИ}.$$

Данное выражение позволяет определить значения тестируемости интерфейса объекта:

$$\text{ТИО} = \begin{cases} 1, \text{ если МИ и СИ } = 1, \\ 0, \text{ если МИ и СИ } = 0. \end{cases}$$

Областью значений ТИО является [0,1], при этом МИ$\in$[0,1]. Раскрывая формулы для ТИО, с помощью формул метрик МИ и СИ, получаем

$$\text{ТИО} = \text{МИ}\times \text{СИ} = 1/\text{n}\ (\sum_{i=1}^{n} \text{K}_i \times \sum_{i=1}^{n} \text{СИ}_i\ ).$$

При тестировании посредников важную роль играют указанные метрики, которые позволят получить полное тестирование интерфейса. Тем самым определять тестируемость двух взаимодействующих компонентов и проводить сравнительный анализ интерпретаций компонентов и их интерфейсов средствами системно контролирующей функции.

Оценка степени тестируемости ПС. После тестирования отдельных объектов ПС и установления степени их тестируемости проводится комплексное тестирование. Для определения тестируемости ПС–$\text{Т}_{\text{ПС}}$ используются метрики ТК и ТИО:

$$\text{Тпс} = 1/\text{n} \sum_{i=1}^{m} (\text{ТК}_\text{j} \times \text{ТИО}_\text{j}\ ),$$

где ТК_j – тестируемость компонентов (j = 1, m), ТИО – тестируемость интерфейса, m – количество всех объектов в системе.

Подставив в соответствующие формулы для тестируемости ТК и ТИО, получим

$$\text{Тпс} = 1/m \sum_{j=1}^{n} (1/n_j \sum_{j=1}^{n_i} \text{КТ}_{\text{бск}} \times 1/k_j \sum_{j=1}^{ki} K_i \times 1/k_j \sum_{j=1}^{ki} \text{СИ}_i),$$

где n_j – количество БСК j-го компонента ПС, k_j – количество интерфейсных посредников.

По этой формуле рассчитывается степень тестируемости ПС. Если для всех объектов ПС метрики ТК и ТИО равны 1, то степень тестируемости системы принимает значения 1, означающую ее полную тестируемость. Базируясь на метрических моделях, тестируемость ПС на правильность управляющих структур БСК и наблюдений за тестированием объектов и интерфейсов системы, делаем следующий вывод:

распределенная ПС – полностью тестированная, если

$$\text{Тпс} = 1 \text{ или } \text{ТК}_j = 1, \text{ при } i = 1, 2, \ldots, m.$$

Это свидетельствует о том, что степень тестируемости распределенной системы определяется правильными компонентами, интерфейсами и данными, передаваемыми в сообщениях.

7.5. ПОДХОД К ОПРЕДЕЛЕНИЮ ВРЕМЕНИ ТЕСТИРОВАНИЯ ПС

Для определения оптимального времени тестирования отдельных компонентов ПС разработаны математическая модель, метод и процедуры оценивания риска отказов компонентов в процессе тестирования, базирующейся на распределении времени тестирования между компонентами системы и учете риска их отказов во время эксплуатации.

Модель определения оптимального времени тестирования разработана в [17, 18] после тщательного изучения структурных особенностей СОД, анализа типичных областей риска и уровней угроз при отказах ПС, а также учета неодинакового экономического вклада отказов модулей в убыток пользователей, получающих отказы в ПС.

Под *угрозой* понимается потенциально возможное событие при работе ПС, которое может привести к нанесению убытков пользователям (например, потеря времени, финансовые расходы и др.). Под *риском* возникновения угрозы понимается вероятность отказа ПС и величина связанного с ней ущерба.

Для класса ПС СОД угрозы обычно касаются нарушения целостности, доступности и конфиденциальности данных, а также недостаточно проведенного тестирования.

Стратегия тестирования состоит в выборе компонентов, которые нуждаются в более продолжительном тестировании. Время тестирования каждого компонента выделяется таким образом, чтобы стоимость (трудоемкость) тестирования отвечала величине использования риска при работе ПС. Такая стратегия тестирования включает в себя решение следующих задач:

– анализ риска отказов ПС;

– определение вклада каждого компонента в угрозы ПС;

– оценки ожидаемого количества отказов, которые могут возникнуть в компонентах при их использовании;

– оценки интенсивности отказов компонентов.

Относительно тестирования каждого компонента ПС вводится понятие его *полезности* (т.е. прибыльности). Тестирование считается прибыльным, если его стоимость, а также стоимость устранения дефектов в ПС, не превышают ожидаемых потерь пользователя.

Задача определения оптимального времени тестирования использует следующие данные:

$H = \{H_i,\ i = 1, 2,..., r\}$ – множество всех потенциальных угроз ПС, обусловленных отказами из-за дефектов в компонентах;

$S = \{S_i,\ i = 1, 2,..., n\}$ – множество всех возможных сценариев функционирования ПС, где каждый сценарий S_i является кортежем модулей, которые используются при реализации сценария $S_i = \{M_1, M_2,..., M_k\}$;

t_e – время тестирования;

t_0 – время выполнения модуля в период эксплуатации ПС;

$C_м$ – вклад модуля в риск отказов ПС (стоимость возможного ущерба из-за отказов модуля за время его выполнения t_0 при эксплуатации ПС);

$R(t_0) = C_м \mu\ (t_0)$ – величина риска отказа модуля за время t_0;

$\mu\ (t)$ – функция роста надежности;

$\lambda\ (t) = d\mu\ (t)/\ dt$ – интенсивность отказов модуля;

N – количество дефектов в модуле в начале тестирования;

$P(S_i)$ – вероятность того, что при реализации сценария S_i $(i = 1, 2,..., n)$ будет выполняться данный модуль;

$P(H_j|S_i)$ – условная вероятность того, что при реализации сценария S_i причиной возникновения угрозы H_j будет отказ именно данного модуля;

C_j – стоимость последствий реализации угрозы H_j (j=1, 2,...,r);

$C(t_e)$ – полная стоимость тестирования на протяжении времени t_e;

c_1 – стоимость единицы времени тестирования;

c_2 – стоимость устранения дефекта, который привел к отказу в процессе тестирования.

$\Delta R(t_0|t_e)$ – функция снижения риска отказа модуля за время его выполнения t_0, при условии, что модуль тестировался за время t_e;

$K(t_0|t_e)$ – прибыль от тестирования.

Ставится задача найти такое время тестирования t_e^*, чтобы прибыль была максимальной, т.е. $t_e^* = \{t_e: K(t_0|t_e^*) \geq K(t_0|t_e), 0 \leq t_e \leq t_{доп}\}$, где $t_{доп}$ – максимально допустимое время тестирования.

Утверждение 7.1. Пусть t_0 – время выполнения модуля при эксплуатации ПС, то функция снижения риска

$$\Delta R(t_0|t_e) = C_м[\mu(t_0) - \mu(t_0 + t_e) + \mu(t_e)],$$

где $C_м = \sum_{i=1}^{n} P(S_i) \sum_{j=1}^{r} P(H_j | S_i) \cdot C_j |$ – вклад модуля в общий риск ПС.

Предполагается, что $C(t_e) = c_1 t_e + c_2 \mu(t_e)$ прибыль от тестирования такова:

$$K(t_0 |t_e) = \Delta R (t_0 |t_e) - C(t_e) = C_м [\mu(t_0) - \mu(t_0 + t_e) + \mu(t_e)] - c_1 t_e - c_2 \mu(t_e).$$

Производная функции $K(t_0 |t_e)$ по t_e имеет вид

$$K'(t_0 |t_e) = C_м[\lambda(t_e) - \lambda(t_0 + t_e)] - c_1 - c_2 \lambda(t_e)$$

Решая уравнение $K'(t_0|t_e) = 0$ в зависимости от значения конкретной функции $\lambda(t)$, получаем значение оптимального времени t_e^*.

Решение уравнения $K'(t_0|t_e) = 0$ с учетом применения экспоненциальной модели надежности и при использовании функции $\lambda(t)$ интенсивности отказов для этой модели, имеем

$$t_e^* = -\frac{1}{\beta} \ln\left[\frac{c_1}{\beta\nu[C_м(1 - \exp(-\beta t_0)) - c_2]}\right],$$

$K'' (t_e^*) = - \beta c_1$. Исходя из предположений экспоненциальной модели $\beta > 0$, оптимальное время t_e^* является максимумом функции $K(t_e)$.

Оптимальное время t_e^* существует тогда и лишь тогда, когда выполняются условия:

$C_м\,(1 - \exp\,(-\beta t_0)) > c_2\,,$

$\beta v\,(C_м\,(1 - \exp\,(-\beta t_0)) - c_2\,) > c_1\,.$

Условия получения оптимального времени t_e^* выполнены для четырех моделей роста надежности (см. главу 8).

Оценивание риска отказов. В процессе тестирования возникают риски, присущие проекту в целом, но наиболее важными среди них – это риски отказов. Для анализа причин отказов и их последствий используются формальные методы верификации, основанные на анализе деревьев событий и отказов, а также режимов и последствий отказов. Общая таксономия риска предложена институтом SEI и используется при идентификации общих источников рисков проекта с учетом конкретных условий [12].

При оценке риска отказов программных модулей выполняются следующие задачи:

1) оценка угроз, риска их реализации и возможных потерь $C_j\,,\ j = 1,\ 2,\ldots\ r;$

2) определение вклада каждого модуля $C_м;$

3) оценка функции роста надежности модуля за время выполнения $\mu(t_0);$

4) оценка риска отказов модуля $R(t_0)$.

Задача 1 – это идентификация возможных угроз ПС и оценка стоимости последствий при появлении угрозы, анализ которой проводится при непосредственном участии пользователя ПС с учетом:

– сценариев работы пользователей с ПС;

– возможных сценариев событий, которые могут привести к отказам ПС;

– типов и серьезности отказов ПС для пользователя, которые соответствуют событиям в этих сценариях;

– возможные последствия от произошедших отказов в ПС.

Серьезность последствий отказов – это ожидаемые финансовые убытки пользователя при отказах ПС в процессе эксплуатации.

Задача 2 – это определение вклада каждого компонента в идентифицированные угрозы ПС, который связывает возможные отказы модулей с внешними угрозами и их последствиями. При этом создаются операционные сценарии работы деловых функций ПС, реализованных программными модулями, и обеспечивают определение внешних угроз, влияющих на возможные отказы модуля в каждом операционном сценарии (S_i). Далее устанавливается соотношение угроз с конкретными модулями и их использованием в сценарии S_i. Если установлено, что модуль относится к угрозе H_j при i-ом использовании, то осуществляется распределе-

ние вероятностей $P(H_j|S_i)$ для всех угроз при заданном сценарии использования S_i.

Задача 3 – это оценка ожидаемого количества отказов модуля, возникающих при его выполнении за время t_0, и выполнении функции $\mu(t_0)$ модели надежности.

Задача 4 – это вычисление интенсивности отказов для каждого модуля.

Данные задачи оценивания риска модулей и модель определения оптимального времени тестирования послужили базисом стратегии тестирования ПС по сценариям определения видов и объемов системного тестирования [18].

7.5.1 Определение базового процесса тестирования

В программной инженерии *процесс* рассматривается как упорядоченная совокупность составных элементов процесса – действий (задач), входов (входных данных для выполнения процесса) и выходов (результатов процесса). Эти принципы стандартов положены в основу определения *базового процесса* тестирования, в котором заданы основные элементы, интерфейсы с другими процессами, критерии начала и завершения задач процесса тестирования, роли и ответственности при выполнении этих задач.

При построении базового процесса были исследованы процессы ЖЦ ДСТУ 3918 и ISO/IEEC 12207, выявлены все задачи тестирования и распределены по процессам ЖЦ, им сопоставлены методы, критерии, метрики и правила поиска ошибок в рабочих продуктах процессов ЖЦ, а также планирование их исправления и тестирования.

Характеристика процессов ЖЦ стандарта ISO/IEEC 12207. Все процессы в данном стандарте разделены на три категории:

– основные процессы;

– обеспечивающие (поддерживающие) процессы;

– организационные процессы.

Для каждого из процессов определены виды деятельности (действия), задачи и совокупность результатов (выходов) видов деятельности и задач, а также некоторые специфические требования. Стандарт дает перечень работ для основных, обеспечивающих и организационных процессов.

К *основным процессам* относятся:

– процесс приобретения инициирует ЖЦ и определяет действия организации-покупателя (или заказчика), приобретающей программный продукт (сервис), а именно: инициация; подготов-

ка контракта и его актуализация; мониторинг поставщиков; приемка и завершение;

– процесс поставки определяет действия предприятия-поставщика, которое снабжает покупателя программным продуктом (сервисом), и включает в себя подготовку контракта, планирование, контроль и оценку поставки продукта. В зависимости от условий договора процесс поставки может включать в себя процесс разработки, сопровождения и улучшения продукта;

– процесс разработки определяет действия предприятия-разработчика программного продукта и включает в себя: анализ требований к системе и ПО; проектирование архитектуры системы; детальное проектирование, кодирование и тестирование ПО; интеграция компонентов, квалификационное тестирование и обеспечение приемки ПО;

– процесс эксплуатации определяет действия предприятия-оператора, обслуживающего систему, и включает в себя функциональное и системное тестирование, запуск системы согласно документации и наблюдение за ее эксплуатацией;

– процесс сопровождения определяет действия организации, выполняющей сопровождение программного продукта (модификация, поддержка текущего состояния, инсталляция и др.), а также анализ ошибок сопровождения и их устранение.

К обеспечивающим процессам создания ПС относятся: документирование, управление версиями, верификация и валидация, просмотры, аудиты, оценивание продукта и др. Эти процессы направлены на проверку правильности реализации целей проекта и соответствия его требованиям заказчика.

К организационным процессам относятся процессы управления проектом (менеджмент разработки), качеством, риском и др., а также специальные службы планирования и управления проверками, тестированием, измерением, оценкой качества, стоимости и др.

Все процессы данного стандарта образуют достаточное их множество. Пользователь стандарта может выбрать соответствующее подмножество для достижения своей конкретной цели в программном проекте. В зависимости от целей проекта процессы выбираются, упорядочиваются в виде модели ЖЦ и выполняются последовательно, итерационно или рекурсивно.

Данный стандарт – это документ, определяющий содержание деятельности в сфере технологии разработки ПС, а знания, которыми необходимо владеть для ее выполнения на ЖЦ, – это методы и средства ядра знаний SWEBOK [8, Приложение 1].

Базовый процесс тестирования представлен подпроцессами, включающими в себя подготовку, проведение и оценивание результатов тестирования. Каждый из них состоит из:

– распределения обязанностей между участниками процесса,

– требований к профессиональной подготовке исполнителей процесса,

– стандартов для представления документов промежуточных продуктов,

– метрик процесса,

– методов решения задач тестирования,

– критериев начала и завершения задач тестирования и перехода к следующему подпроцессу.

Иными словами, базовый процесс тестирования – это объединение задач программирования ПС и регламентированных стандартом задач процесса тестирования в единую упорядоченную систему действий по реализации качественной ПС СОД. Результат каждого подпроцесса базового процесса тестирования фиксируется в специальных шаблонах, которые используются при анализе результатов тестирования.

7.5.2. Совершенствование процесса тестирования на основе CMM

Рост популярности моделей оценивания процессов ЖЦ ПО, в частности, *модели зрелости возможностей* (Capability Maturity Model (CMM)) [19], послужил поводом для разработки моделей зрелости процесса тестирования в направлении отображения в нем средств повышения зрелости.

Концепция *зрелости* процесса была предложена американским институтом SEI (Software Engineering Institute) и нашла свое отображение в пятиуровневой модели CMM, с помощью которой определяются характеристики технологических процессов (ТП) на соответствие уровня зрелости и путей их улучшения до требуемого уровня.

Модель CMM – это описание стадий эволюции развития организации разработчика по мере того, как эта модель определяет, реализует, измеряет, контролирует и совершенствует ТП создания ПО. CMM позволяет выбрать адекватную стратегию усовершенствования процессов, предоставляя методическую основу для определения текущего уровня их совершенства и выявления проблем, критичных для качества разрабатываемого ПО.

В основу CMM положены следующие фундаментальные понятия:

– технология проектирования ПС;

– технологический процесс в виде последовательности действий, определяющих выполнение задач процесса и целей проектирования ПС [IEEE–Std–610];

– совершенствование процесса (Process Capability) в зависимости от диапазона результатов, которые ожидают от организации, соблюдающей ТП.

Эти понятия имеют отношение к будущим проектам, базирующимся на фактических характеристиках технологии, достигнутых на предыдущих проектах.

К характеристикам процесса (Process Performance) относятся фактические результаты, достигнутые организацией при соблюдении процесса, имеющего отношение к последнему выполненному проекту.

Зрелость процесса/технологии (Process Maturity) – это степень определенности, управляемости, наблюдаемости, контролируемости и эффективности выбранных процессов технологии проектирования ПС. Фактически зрелость – это индикатор полноты технологии, применения ее во всех проектах и потенциал для дальнейшего развития и совершенства.

По сравнению с другими моделями и стандартами аналогичного назначения модель CMM позволяет провести последовательно-ступенчатую или поэтапную (staged) эволюцию процессов и технологии. При этом организация осуществляет постепенное продвижениек более высокому уровню производства, шаг за шагом последовательно поднимаясь на новую ступень «лестницы» технологической зрелости. Для каждого уровня зрелости в CMM определены ключевые области процессов создания ПС (Key Process Areas).

Инфраструктура организации-разработчика ПС должна иметь:

1) средства оценивания выполнения ТП разработки ПС;

2) методические руководства ПС по улучшению возможностей ТП.

При этом в инфраструктуре организации-разработчика ПС создается группа, осуществляющая:

– аналитическую оценку сильных и слабых сторон в деятельности организации, ориентированной на создание программного продукта;

– экспертную оценку рисков выбора исполнителей проектов и управления работами по реализации программного продукта;

– улучшение и совершенствование ТП;

– планирование и реализацию процесса улучшения ТП.

Модель CMM получила развитие в ряде работ, направленных на совершенствование самых главных процессов тестирования:

– модель зрелости тестирования (Testing Maturity Model – TMM) [20];

– модель обеспечения тестируемости (Testability Support Model – TSM) [21];

– модель усовершенствования тестирования (Test Improvement Model – TIM) [22];

– модель усовершенствования тестирования (Test Process Improvement Model) [23];

– модель зрелости возможностей тестирования (Testing Capability Maturity Model – TCMM) [24].

– программа оценивания тестирования (TAP) [25].

Эти модели ориентированы на упорядочивание и оценивание процессов тестирования с помощью уровней зрелости CMM, начиная от начального к оптимизированному процессу.

Результаты анализа этих моделей показывают, что они имеют как много общего в концепции зрелости процесса тестирования с учетом структурных различий, так и в составе ключевых направлений и задач оценивания риска и времени тестирования. Эти модели можно адаптировать к процессам тестирования других типов ПС и операционным условиям новой среды.

В главе были рассмотрены современные методы и процессы тестирования ПС, основанные на понятии программы – «белый ящик» и «черный ящик», а также на анализе реализованных в ПС функций. Определены критерии тестирования, типы ошибок, обнаруживаемых в программах, а также интенсивность отказов и их учет при определении оптимального времени тестирования. Дано определение базового процесса тестирования, объединяющего в себе критерии, методы проектирования и тестирования, оценки результатов процесса и самого процесса. Изложены пути совершенствования процесса тестирования на основе модели зрелости CMM.

ГЛАВА 8

МОДЕЛИ И МЕТОДЫ ОЦЕНКИ НАДЕЖНОСТИ ПС

8.1. ОПРЕДЕЛЕНИЕ НАДЕЖНОСТИ ПС

Из всех областей знаний программной инженерии надежность ПС является самой исследованной областью. Ей предшествовала разработка теории надежности технических средств, оказавшая влияние на развитие надежности ПС. Вопросами надежности ПС занимались разработчики ПС, пытаясь разными средствами обеспечить надежность, удовлетворяющую заказчика, а также теоретики, которые, изучая природу возникновения ошибок при функционировании ПС, создавали большое количество (более 100) математических моделей надежности, учитывающих разные аспекты работы ПС (возникновение ошибок, сбоев, отказов и др.) и обеспечивающие оценку реальной надежности ПС. В результате надежность ПС сформировалась как самостоятельная теоретическая и прикладная наука [1–5].

Надежность ПС существенным образом отличается от надежности аппаратуры. Носители данных (файлы, сервер и т.п.) обладают высокой надежностью, записи на них могут храниться длительное время без разрушения, поскольку они не подвергаются износу.

С точки зрения прикладной науки *надежность* – это способность ПС сохранять свои свойства (безотказность, устойчивость и др.) преобразовывать исходные данные в результаты в течение определенного промежутка времени при заданных условиях эксплуатации. Снижение надежности ПС происходит из-за ошибок в требованиях, проектировании и выполнении.

Для многих систем надежность является главной целевой функцией реализации. К некоторым типам систем (реального времени, радарные системы, системы безопасности, медицинское оборудование со встроенными программами и др.) предъявляются высокие требования к надежности (недопустимость ошибок, достоверность, защищенность и др.).

Таким образом, надежность ПС зависит от числа оставшихся и не устраненных ошибок на этапах ЖЦ. В ходе эксплуатации ПС ошибки обнаруживаются и устраняются. Если при их исправлении

не вносятся новые, или, по крайней мере, вносится меньшее количество, чем устраняется, то в ходе эксплуатации надежность ПС непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность системы и соответственно ее качество.

Надежность является функцией от ошибок, оставшихся в ПС после ввода его в эксплуатацию. ПС без ошибок можно считать абсолютно надежным. Но для больших программ абсолютная надежность практически недостижима. Оставшиеся необнаруженные ошибки проявляют себя время от времени при определенных условиях (например, при некоторой совокупности данных) сопровождения и эксплуатации системы.

Для оценки надежности ПС используются такие собранные статистические данные как вероятность и время безотказной работы, возможность отказа и частота (интенсивность) отказов. Поскольку в качестве причин отказов рассматриваются только ошибки в программе, которые не могут самоустраниться, то ПС следует относить к классу невосстанавливаемых систем.

При каждом проявлении новой ошибки, как правило, проводится ее локализация и исправление. Строго говоря, набранная до этого статистика об отказах теряет свое значение, так как после внесения изменений программа, по существу, является новой, и отличается от той, которая до этого испытывалась.

В связи с исправлением ошибок надежность, т.е. отдельные ее атрибуты, будут все время изменяться, как правило, в сторону улучшения. Следовательно, их оценка будет носить временный и приближенный характер. Поэтому возникает необходимость в использовании новых свойств, адекватных реальному процессу измерения надежности, таких, как зависимость интенсивности обнаруженных ошибок от числа прогонов программы и зависимость отказов от времени функционирования ПС и т.п.

На надежность ПС влияют факторы:

– риск через угрозы, приводящие к неблагоприятным последствиям и ущербу системы или среды;

– угрозы как проявление неустойчивости системы и нарушения ее безопасности;

– причины возникновения угроз или рисков, их частота и последствия;

– способность системы сохранять устойчивость работы и др.

Риск уменьшает надежность, а обнаруженные ошибки могут привести к угрозе, если отказы носят частотный характер.

8.2. ФОРМАЛЬНОЕ ОПРЕДЕЛЕНИЕ МОДЕЛЕЙ НАДЕЖНОСТИ ПС

С теоретической точки зрения под *надежностью* ПС понимается способность системы сохранять свои свойства (безотказность, восстанавливаемость) на заданном уровне в течение фиксированного промежутка времени при определенных условиях эксплуатации. Иными словами, имеется вероятность того, что функционирующая ПС в заданной среде не даст отказов в интервале времени $(t,.\ t+k)$ при условии, что последний отказ ПС и устранение соответствующей ошибки произошли до момента времени t. Чем больше времени работает система без отказов, тем надежнее она становится

Исследование надежности проводится с помощью методов теории вероятностей, математической статистики и др. Формально, надежность можно определить как вероятность

$P\ (i) = P$, если нет отказов в i прогонах программы на тестах,

$P(t) = P$, если нет отказов в интервале времени выполнения программы $(0,\ t\ )$.

Если при отказе работоспособность системы подлежит восстановлению, то такая система называется восстанавливаемой. Главным показателем такой системы является время безотказной работы ПС и время ее восстановления после отказа. Если эти времена распределены по показательному закону, то для расчета надежности можно использовать однородные марковские процессы, но если они распределены по произвольному закону, то расчет надежности усложняется и проводится по приближенным формулам. Вероятность безотказной работы системы определяется приближенно по формуле: $P(t) \approx \exp\ (-t\ /\ T)$ [1].

Вероятность того, что в системе не произойдет отказ в течение фиксированного времени, определяет рост надежности ПС.

Если имеется поток отказов, то работа системы подчиняется пуассоновскому закону, и вероятность ее безотказной работы определяется по формуле: $P(t\ ) = K_r \exp\ (-t\ /T)$,
где $K_r = T\ (T\ + T_n)^{-1}$ и обозначает, что поток отказов постоянный на заданном интервале времени работы системы.

Исследование надежности ПС с разных точек зрения и теорий привело к тому, что разработано большое количество моделей надежности ПС, которые ориентированы на математическую оценку надежности ПС. Многие модели надежности дают приближенные оценки и исходят из того, что возникшая ошибка

или отказ немедленно устраняются, а новые ошибки при этом не вносятся. Такие модели надежности определяются математической зависимостью между временем выполнения программы и общим числом обнаруженных ошибок.

8.2.1. Базовые понятия моделей надежности ПС

Формально модели оценки надежности ПС базируются на теории надежности и математическом аппарате с допущением некоторых ограничений, влияющих на эту оценку. Главным источником информации, используемой в моделях надежности, является процесс тестирования, эксплуатации ПС и разного вида ситуации, возникающие в них. Ситуации порождаются возникновением ошибок в ПС, требуют их устранения и продолжения тестирования.

К базовым понятиям, которые используются в моделях надежности ПС, относятся следующие [5–9].

Отказ ПС (failure) – это переход ПС из рабочего состояния в нерабочее, или полученные результаты не соответствуют заданным допустимым значениям. Отказ может быть вызван внешними факторами (изменениями элементов среды эксплуатации) и внутренними – дефектами в самой ПС.

Дефект (fault) в ПС – это последствие выполнения элемента программы, приводящее к некоторому событию, например, в результате неверной интерпретации его компьютером или человеком. Дефект является следствием ошибок разработчика на любом из процессов разработки – в описании спецификаций требований, проектных спецификациях, эксплуатационной документации и т.п. Дефекты в программе, не выявленные в результате проверок, является источником потенциальных ошибок и отказов ПС. Проявление дефекта в виде отказа зависит от того, какой путь будет выполняться для нахождения ошибки в коде или во входных данных. Не каждый дефект ПС может вызвать отказ или любой отказ может вызвать аномалию от проявления внешних ошибок и дефектов.

Ошибка (error) может быть следствием недостатка в одном из процессов разработки ПС, который приводит к неправильной интерпретации промежуточной информации, заданной разработчиком или при принятии им неверных решений.

Интенсивность отказов – это частота появления отказов или дефектов в ПС при ее тестировании или эксплуатации.

При выявлении отклонения результатов от ожидаемых во время тестирования или сопровождения осуществляется поиск и выяснение причин этих отклонений для исправления связанных с этим ошибок.

В качестве входных параметров модели оценки надежности ПС используются сведения об ошибках, отказах, их интенсивности, собранных в процессе тестирования и эксплуатации.

8.2.2. Случайный характер возникновения ошибок в ПС

Процесс возникновения ошибок и отказов в ПС является случайным и в основном определяется временем их возникновения или частотой, числом и интенсивностью их появления в ПС. В связи с этим все модели надежности основываются именно на нахождении случайной величины, принимающей значение по определенному закону распределения. Например, в рамках ПС, продолжительность работы программы до первого отказа, количество дефектов и т.п. [1].

Если случайная величина дискретна, т.е. принимает конечное число значений в виде последовательности $x_1, x_2, \ldots, x_n \ldots$, то закон распределения ξ описывается заданием вероятностей $P(\xi = x_i)$ и в общем случае $F(x) = P(\xi < x_i)$ называется функцией распределения случайной величины [1, 2, 10–13].

Эта функция определяет вероятность попадания случайной величины в любой интервал отрезка времени на прямой линии системы координат, при этом $t \in T$ – это момент времени. Случайная величина определяется постоянной величиной t и ей соответствует случайная функция. Областью определения процесса на множестве T является последовательность времен $t_k < t_{k+1}$, тогда случайный процесс является процессом *дискретным* во времени. Случайный процесс с непрерывным временем, описывающийся числом однородных событий, называется *пуассоновским* процессом.

Если характеристика случайной функции оказывается неслучайной величиной, то вычисляется математическое ожидание или дисперсия, как среднее отклонение от средней реализации случайной функции.

Поиск случайных величин осуществляется стохастическими методами, процесс соответственно является стохастическим, вероятностным.

Если на множестве T определяется случайный процесс, то для всех его точек определяется случайная величина $\xi(t)$, которая и называется ее значением.

С точки зрения теории случайных процессов процесс возникновения отказов в ПС является стохастическим, модели надежности также являются стохастическими.

К категории случайных процессов относится марковский процесс, который обладает тем свойством, что его поведение после момента времени t зависит только от его значения, а не зависит от поведения процесса до момента t.

Так как тестирование является динамическим процессом, обеспечивающим поиск дефектов и отказов, которые возникают случайно в ПС, то его можно объяснить с помощью функции $p(t, x, s)$ для $t < s$ и $x \in X$, т.е. точки, определенной в момент времени s, если ее положение было x. Эта функция удовлетворяет соотношению

$$p(t, x, u) = p\,(s, p\,(t, x, s)\,, u)$$

при условии $t < s < u$ и означает, что в момент времени t в точке x система, находясь в состоянии $p(t, x, u)$, переходит в состояние $p(t, x, s)$.

При этом область T марковского процесса может быть последовательностью моментов времени, и тогда процесс будет дискретным, а T – конечным или бесконечным. Когда x – конечное множество, то процесс является процессом с конечным числом состояний. Марковский процесс с дискретным временем и конечным числом состояний называется марковской цепью.

В общем случае вероятность определяется функцией $p_{ij}(t, x, s)$ при условии, что система в момент t переходит из i-состояния в j-состояние момента s, и эта функция находится решением системы уравнений

$$\frac{d}{ds}\, p_{ij}(t, s) = \sum P_{ij}\,(t, s)\; a_{kj}(s),$$

разработанной Колмогоровым.

Другим подходом к исследованию надежности на основе отказов в ПС является классическая теория вероятностей, согласно которой отказы в ПС (в отличие от отказов технических средств) считаются случайными и зависят от дефектов, внесенных при разработке ПС. Рассмотрение их в качестве случайной величины основано на следующих предположениях:

– количество дефектов в ПС неизвестно;

– время выявления и местонахождения каждого дефекта не является предвиденным.

Зависимость количества отказов от времени выполнения связано с тем, что чем дольше используется ПС, тем больше вероятность прохождения по фрагменту кода, который содержит дефект.

Все модели оценки надежности основываются на статистике отказов и распределении интенсивности выявленных отказов в ПС.

Большинство моделей надежности исходят из предположения, что найденные дефекты устраняются немедленно (или временем их устранения можно пренебречь) и при этом новые дефекты не вносятся. В результате количество дефектов в ПС уменьшается, а надежность возрастает, такие модели получили название *моделей роста надежности.*

8.3. КЛАССИФИКАЦИЯ МОДЕЛЕЙ НАДЕЖНОСТИ

На данный момент времени разработано большое количество моделей надежности ПС и их модификаций. Каждая из этих моделей определяет функцию надежности, которую можно вычислить при задании ей соответствующих данных, собранных во время функционирования ПС. Основными данными являются отказы и время. Другие дополнительные параметры связаны с типом ПС, условиями среды и данных.

В виду большого разнообразия моделей надежности, разработано несколько подходов к классификации этих моделей. Эти подходы в целом основываются на истории ошибок проверяемой и тестируемой ПС. Одной из классификаций моделей надежности ПО является классификация Хетча [5, 7] (рис. 8.1)

Прогнозирующие модели надежности основаны на измерении технических характеристик создаваемой программы: длина, сложность, число циклов и степень их вложенности, количество ошибок на страницу операторов программы и др. Например, модель Мотли–Брукса основывается на длине и сложности структуры программы (количество ветвей и циклов, вложенность циклов), количестве и типах переменных, а также интерфейсов. В этих моделях длина программы служит для прогнозирования количества ошибок, например, для 100 операторов программы можно смоделировать интенсивность отказов.

Модель Холстеда дает прогнозирование количества ошибок в программе в зависимости от ее объема и таких данных, как число операций (n_1) и операндов (n_2), а также их общее число (N_1, N_2).

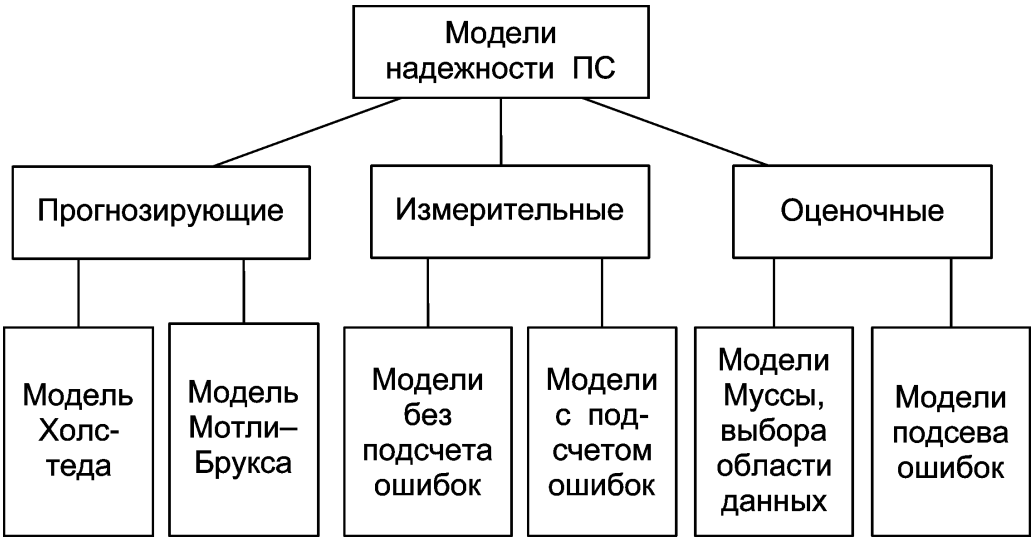

РИС. 8.1. Классификация моделей надежности на основе ошибок

Время программирования программы предлагается вычислять по следующей формуле:

$$T = (n_1 N_2 (n_1 \log_2 n_1 + n_2 \log_2 n_2) \log_2 n_1 / 2 n_2 S,$$

где S – число Страуда (число умственных операций в единицу времени Холстед принял равным 18).

Объем вычисляется по формуле

$$V = (2 + n_2^*) \log_2 ((2 + n_2^*),$$

где n_2^* – максимальное число различных операций.

Измерительные модели предназначены для измерения надежности ПО, работающего с заданной внешней средой и следующими ограничениями:

– ПО не модифицируется во время периода измерений свойств надежности;

– обнаруженные ошибки не исправляются;

– измерение надежности проводится для зафиксированной конфигурации ПО.

Типичным примером таких моделей является модель Нельсона и Рамамурти–Бастани и др. [5, 12].

Модель оценки надежности Нельсона основывается на выполнении k-прогонов программы при тестировании и позволяет определить надежность по формуле

$$R(k) = \exp [- \sum \nabla t_j \lambda (t)],$$

где t_j – время выполнения j-прогона, $\lambda (t) = - [\ln (1 - q_i) \nabla j]$ и при $q_i \le 1$ она интерпретируется как интенсивность отказов.

В процессе испытаний программы на тестовых n_l прогонах оценка надежности вычисляется по формуле

$$R(l) = 1 - n_l / k,$$

где k – число прогонов программы.

Иными словами, данная модель использует полученные количественные данные о проведенных прогонах.

Оценочные модели основываются на серии тестовых прогонов и проводятся на этапах тестирования ПС. В тестовой среде определяется вероятность отказа программы при ее выполнении или тестировании.

Эти типы моделей могут применяться на этапах ЖЦ. Кроме того, результаты прогнозирующих моделей могут использоваться как входные данные для оценочной модели. Имеются модели (например, модель Мусы), которые можно рассматривать и как оценочную, и как измерительную модель [20, 23].

Еще один вид классификации моделей предложил Гоэл [15, 16], согласно которой модели надежности базируются на отказах и разбиваются на четыре класса моделей:

– без подсчета ошибок,
– с подсчетом отказов,
– с подсевом ошибок,
– модели с выбором областей входных значений.

Модели без подсчета ошибок основаны на измерении интервала времени между отказами и позволяют спрогнозировать количество ошибок, оставшихся в программе. После каждого отказа оценивается надежность и определяется среднее время до следующего отказа. К такой модели относятся модели Джелински и Моранды, Шика Вулвертона и Литвуда–Верала [5, 25, 17, 18].

Модели с подсчетом отказов базируются на количестве ошибок, обнаруженных на заданных интервалах времени. Возникновение отказов в зависимости от времени является стохастическим процессом с непрерывной интенсивностью, а количество отказов является случайной величиной. Обнаруженные ошибки устраняются и поэтому количество ошибок в единицу времени уменьшается. К этому классу моделей относятся модели Шика–Вулвертона, Шумана, пуассоновская модель и др.[17, 24].

Модели с подсевом ошибок основаны на количестве устраненных ошибок и подсеве, внесенном в программу искусственных ошибок, тип и количество которых заранее известны. Затем строиться соотношение числа оставшихся прогнозируемых ошибок к числу искусственных ошибок, которое сравнивается с со-

отношением числа обнаруженных действительных ошибок к числу обнаруженных искусственных ошибок. Результат сравнения используется для оценки надежности и качества программы. При внесении изменений в программу проводится повторное тестирование и оценка надежности. Этот подход к организации тестирования отличается громоздкостью и редко используется из-за дополнительного объема работ, связанных с подбором, выполнением и устранением искусственных ошибок.

Модели с выбором области входных значений основываются на генерации множества тестовых выборок из входного распределения. Оценка надежности проводится по полученным отказам на основе тестовых выборок из входной области. К этому типу моделей относится модель Нельсона и др. [5, 12].

Кроме рассмотренной классификации моделей надежности, имеются и другие классификации, например, которые определяются процессами выявления отказов, их интенсивностью и условно их можно разделить на следующие группы:

1) модели, рассматривающие интенсивность отказов как марковский процесс;

2) модели, рассматривающие интенсивность отказов как пуассоновский процесс;

3) модели роста надежности.

Четкой границы между этими моделями провести нельзя, однако по фактору распределения интенсивности отказов и их поведению эти модели можно еще разделить на экспоненциальные, логарифмические, геометрические, байесовские и др.

8.3.1. Модели надежности марковского типа

Марковский процесс характеризуется дискретным временем и конечным множеством состояний. Временной параметр пробегает неотрицательные числа, а процесс (цепочка) определяется набором вероятностей перехода $p_{ig}\ (n)$, т.е. вероятностью на n-шаге перейти из состояния I в состояние j. Процесс называется однородным, если он не зависит от n.

В моделях, базирующихся на процессе Маркова, предполагается, что количество дефектов, обнаруженных в ПС, в любой момент времени зависит от поведения системы и представляется в виде стационарной цепи Маркова. При этом количество дефектов конечное, но является неизвестной величиной, которая задается константой. Интенсивность отказов в ПС или скорость прохода по цепи зависит лишь от количества дефектов, которые остались в ПС.

К этой группе относятся модели Шика–Вулвертона [17], Шантикумера [18], Джелинського–Моранды [25] и др.

Модели надежности, которые обеспечивают рост надежности ПО (так называемые модели роста надежности), находят широкое применение при тестировании и описывают процесс обнаружения отказов при следующих основных предположениях [5]:

– все ошибки в ПС не зависят друг от друга с точки зрения локализации отказов;

– интенсивность отказов пропорциональна текущему числу ошибок в ПС (убывает при тестировании ПО);

– вероятность локализации отказов остается постоянной;

– локализованные ошибки устраняются до того, как тестирование будет продолжено;

– при устранении ошибок новые ошибки не вносятся.

При описании моделей роста надежности используются следующие величины:

m – число обнаруженных отказов ПО за время тестирования;

X_i – интервалы времени между отказами $i-1$ и i, при $i = 1,..., m$;

S_i – моменты времени отказов (длительность тестирования до i– отказа), $S_i = X_k$ при $i = 1,..., m$;

T – продолжительность тестирования программного обеспечения (время, для которого определяется надежность);

N – оценка числа ошибок в ПО в начале тестирования;

M – оценка числа прогнозированных ошибок;

MT – оценка среднего времени до следующего отказа;

$E(T_p)$ – оценка среднего времени до завершения тестирования;

Var (T_p) – оценка дисперсии;

$R(t)$ – функция надежности ПО;

$Z_i(t)$ – функция риска в момент времени t между $i-1$ и i отказами;

c – коэффициент пропорциональности;

b – частота обнаружения ошибок.

Далее приведены модели роста надежности, основанные на этих предположениях и результатах тестирования программ при отказах и интервалах времени между ними.

Модель Джелински–Моранды. В этой модели используются исходные данные, приведенные выше, а также:

m – число обнаруженных отказов за время тестирования,

X_i – интервалы времени между отказами,

T – продолжительность тестирования.

Функция риска $Z_i(t)$ в момент времени t расположена между $i-1$ и i имеет вид

$$Z_i(t) = c\,(N - n_{i-1})\,,$$

где $i = 1,.., m;\ T_{i-1} < t < T_i$.

Эта функция считается ступенчатой кусочно-постоянной функцией с постоянным коэффициентом пропорциональности и величиной ступени c.

Оценка параметров c и N производится с помощью системы уравнений:

$$\sum_{i=1}^{m} 1/\left(N - n_{i-1}\right) - \sum_{i=1}^{m} cX_i = 0\,,$$

$$n/c - NT - \sum_{i=1}^{m} X_i n_i = 0\,.$$

При этом суммарное время тестирования вычисляется так: $T = \sum_{i=1}^{m} X_i$.

Выходные показатели для оценки надежности относительно указанного времени T включают в себя:

− число оставшихся ошибок $M_T = N-m;$

− среднее время до текущего отказа $MT_T = 1/(\ N-m)\ c$;

− среднее время до завершения тестирования и его дисперсия

$$E(T_p) = \sum_{i=1}^{N-n} \left(1/ic\right),$$

$$\mathrm{Var}(T_p) = \sum_{i=1}^{N-n} \left(1/(ic)\right)^2.$$

При этом функция надежности вычисляется по формуле

$$R_T(t) = \exp\,(-\,(N-m)\,ct),$$

при $t>0$ и числе ошибок, найденных и исправленных на каждом интервале тестирования, равно единице.

Модель Шика−Волвертона используется, когда интенсивность отказов пропорциональна не только текущему числу ошибок, но и времени, прошедшему с момента последнего отказа. Исходные данные для этой модели аналогичны для выше рассмотренной модели Джелински−Моранды.

Функции риска $Z_i\,(t)$ в момент времени между $i-1$ и i отказами определяются следующим образом:

$$Z_i(t) = c\ t\ (N - n_{\,i\,-1}),\ \text{где}\ i = 1,..., m;\ T_{i-1} < t < T_{i\,,}$$

$$T = \sum_{i=1}^{m} X_i \ .$$

Эта функция является линейной внутри каждого интервала времени между отказами, возрастает с меньшим углом наклона.

Оценка c и N вычисляется из системы уравнений:

$$\sum_{i=1}^{m} 1/\left(N - n_{i-1}\right) - \sum_{i=1}^{m} X_i^2 /2 = 0 \ ,$$

$$n/c - \sum_{i=1}^{m} \left(N - n_{i-1}\right) X_i^2 /2 = 0 \ .$$

К выходным показателям надежности относительно продолжительности T относятся:

– число оставшихся ошибок $M_T = N - m$;

– среднее время до следующего отказа $MT_T = (p/2(N-m)\ c))^{1/2}$;

– среднее время до завершения тестирования и его дисперсия

$$E(T_p) = \sum_{i=1}^{N-m} \left(\pi/(2ic)\right)^{1/2},$$

$$\mathrm{Var}(T_p) = \sum_{i=1}^{N-m} \left((2 - \pi/2)/ic\right),$$

когда число ошибок, удаленных на каждом интервале тестирования после периода T, равно единице.

Функция надежности вычисляется по формуле

$$R_T(t) = exp\ (-(N-m)\ ct^2/2),\ t \geq 0.$$

8.3.2. Модели надежности пуассоновского типа

В моделях надежности, базирующихся на этом процессе, выявление отказов моделируется неоднородным процессом, который задает $\{M(t),\ t \geq 0\}$ – неоднородный пуассоновский процесс с функцией интенсивности $\lambda(t)$, что соответствует общему количеству отказов ПС за время его использования t. Тогда количество отказов ПС в интервале $[0,\ t]$ имеет распределение Пуассона:

$$P\{M(t) = k\} = \frac{[\mu(t)]^k}{k!} \exp(-\mu(t)),\ k = 0,\ 1,...,$$

где $\mu(t) = \int_0^t \lambda(u)du$ – среднее количество отказов за время t.

Поскольку функция $\mu(t)$ характеризует степень повышения надежности ПС, является функцией роста надежности и описывает взаимосвязь между количеством выявленных отказов ПС и временем, израсходованным на тестирование и устранение дефектов. Скорость роста функции $\mu(t)$ зависит от:

– количества скрытых дефектов в начале тестирования из-за ошибок в спецификациях и программном коде;

– скорости выявления дефектов во время тестирования;

– эффективности устранения дефектов.

Исследования, проведенные в работах [5, 8, 12, 16–26], показывают, что измерение роста надежности ПС по результатам тестирования требует выбора и объединения разных моделей оценки надежности в течение ЖЦ, которые задают изменение надежности ПС во времени, а также адекватного процесса тестирования, как главного условия роста надежности. Оно состоит в том, что тестовые данные и функции ПС выбираются соответственно частоте их ожидаемого использования, а среда тестирования максимально приближена к условиям эксплуатации.

Экспоненциальная модель роста надежности

В основе этой модели лежат такие дополнительные допущения:

– все дефекты в ПС взаимно независимы с точки зрения выявления отказов;

– интенсивность отказов в любой момент времени пропорциональна текущему количеству дефектов в ПС и является постоянной величиной;

– время устранения отказа очень мало и им можно пренебречь.

Эта модель имеет кривую э*кспоненциального роста* надежности, заданную в виде

$$\mu(t) = N\,(1 - \exp\,(-\beta\, t)),\ N, \beta > 0,$$

где N – количество дефектов, которые имеет ПС в начале тестирования; β – коэффициент пропорциональности, который равен скорости выявления одного дефекта.

Функция интенсивности отказов имеет вид

$$\lambda(t) = \frac{d\mu(t)}{d(t)} = N\beta \exp(-\beta t),\ t > 0$$

и строго снижается при $t > 0$.

Определение 8.1. Функция $\mu(t)$ является возрастающей функцией интенсивности обнаружения ошибок, если функция интенсивности $\lambda\,(t)$ не убывает во времени t, $t \geq 0$.

Определение 8.2. Функция $\mu(t)$ является убывающей функцией интенсивности обнаружения ошибок, если функция интенсивности $\lambda(t)$ не возрастает во времени t, $t \geq 0$.

Основными экспоненциальными моделями является базовая модель времени выполнения программ Мусы [23, 27, 28], Шнейдевинда [21], модель Мусы–Окумото [28], а также их модификации.

Модель Мусы имеет параметр β, которому можно дать такую физическую интерпретацию: $\beta = \lambda_0 / N$, где λ_0 – начальная интенсивность отказов.

В этой модели используется не календарное время, а время выполнения программы

$$\mu(t) = N\left(1 - \exp\left(-\frac{\lambda_0}{N}t\right)\right).$$

Оценка параметров экспоненциальной модели выполняется методом максимального правдоподобия и сводится к решению следующей системы уравнений [10]:

$$\frac{m_e}{\beta} - \frac{m_e t_e}{\exp(\beta t_e) - 1} - \sum_{i=1}^{m_e} t_i = 0,$$

$$N = \frac{m_e}{1 - \exp(-\beta t_e)},$$

где m_e – количество выявленных и устраненных дефектов во время тестирования, t_e – время тестирования, t_i – кумулятивные моменты отказов, $i=1,2, \ldots, m_e$.

Экспоненциальная модель имеет ряд преимуществ: проста в применении, четко интерпретированы параметры, использовано фактическое время выполнения ПС. С помощью этой модели можно описывать процесс возникновения отказов на этапе системного тестирования и эксплуатации. В качестве недостатка можно отметить то, что функция интенсивности отказов строго падает, что не всегда подтверждается на начальных этапах тестирования.

Логарифмическая модель

Модель Муса–Окумото основывается на допущениях, что некоторые дефекты имеют большую вероятность проявления в виде отказов, чем другие, и среднее снижение интенсивности отказов с каждым устраненным дефектом получает экспоненциальное распределение [5, 28]. С учетом этого функция $\mu(t)$ является логарифмической, зависит от времени и имеет вид

$$\mu(t) = \frac{1}{\theta} \ln (\lambda_0 \theta t + 1),$$

где λ_0 – начальная интенсивность отказов с той же интерпретацией, что и для модели Мусы, θ – характеризует экспоненциальный спад интенсивности отказов с каждым устраненным дефектом.

Функцию интенсивности отказов $\lambda(t)$ можно записать так:

$$\lambda(t) = \lambda_0 / \lambda_0 \theta t + 1.$$

Если $N = \frac{1}{\theta}$, $\beta = \lambda_0$, то оценку параметров логарифмической модели можно выполнить с помощью метода максимального правдоподобия, который сводится к решению следующей системы уравнений:

$$N = \frac{m_e}{\ln(\beta t_e + 1)},$$

$$\frac{1}{\beta} \sum_{i=1}^{me} \frac{t_i}{(1+\beta t_i)} - \frac{m_e t_e}{(1+\beta t_e)\ln(1+\beta t_e)} = 0,$$

где m_e – количество выявленных и устраненных дефектов за время тестирования, t_e – время тестирования, t_i – кумулятивные моменты отказов, $i=1,2, \ldots, m_e$.

Модель Гоело–Окумото. В основе этой модели лежит описание процесса обнаружения ошибок с помощью неоднородного пуассоновского процесса, ее можно рассматривать как модель экспоненциального роста. В этой модели интенсивность отказов также зависит от времени, а количество выявленных ошибок трактуется как случайная величина, наблюдаемое значение которой зависит от теста и других условных факторов [5, 28].

Исходные данные m, X_i и T аналогичны предыдущим моделям.

Функция среднего числа отказов, обнаруженных к моменту t, имеет вид

$$m(t) = N(1 - e^{-bt}),$$

где b – интенсивность обнаружения отказов и показатель роста надежности $q(t) = b$.

Функцию интенсивности $\lambda(t)$ в зависимости от времени работы ПС до отказа можно представит так:

$$\lambda(t) = Nbe^{-bt}, \; t \geq 0.$$

Оценка b и N получаются из решения уравнений

$$m/N - 1 + \exp(-bT) = 0,$$

$$m/b - \sum_{i=1}^{m} t_i - N_m \exp(-bT) = 0.$$

К выходным показателям надежности относительно времени T относятся:

1) среднее число ошибок, которые были обнаружены в интервале [0, T], $E(N_T) = N\,exp(-bT)$,

2) функция надежности определяется так

$$R_T(t) = exp(-N(e^{-bt} - e^{-bt(t+m)})),\ t \geq 0.$$

В этой модели обнаружение ошибки, трактуется как случайная величина, значение которой зависит от теста и операционной среды. В других моделях количество обнаруженных ошибок рассматривается как константа.

В моделях роста надежности исходной информацией для расчета надежности являются интервалы времени между отказами тестируемой программы, число отказов и время, для которого определяется надежность программы при отказе. На основании этой информации по моделям находятся следующие показатели надежности:

– вероятность безотказной работы;

– среднее время до следующего отказа;

– число необнаруженных отказов (ошибок);

– среднее время для дополнительного тестирования программы.

Модель анализа результатов прогона тестов использует в своих расчетах общее число экспериментов тестирования и число отказов. Эта модель определяет только вероятность безотказной работы программы и выбрана для случаев, когда предыдущие модели нельзя использовать (мало данных, некорректность вычислений). Формула определения вероятности безотказной работы по числу проведенных экспериментов имеет вид

$$P = 1 - Nex/N,$$

где Nex – число ошибочных экспериментов; N – число проведенных экспериментов для проверки работы ПС.

8.3.3. Модели роста надежности

Модели, предложенные Ямадою–Охбою–Осаки (Yamada–Ohba–Osaki), являются модификацией неоднородного процесса

Пуассона для получения кривой S-подобной формы. Эти модели можно рассматривать как обобщение экспоненциальной модели, когда интенсивность отказов сначала возрастает, а потом, с течением времени тестирования, экспоненциально снижается [24].

Существует две модели, базирующиеся на данном допущении – модель S-образного замедленного роста надежности и модель S-образного роста с перегибами [5, 12].

Модель S-образного замедленного роста надежности отображает тот факт, что процесс тестирования, который в экспоненциальной модели надежности отождествляется с процессом выявления отказов, в действительности является двухфазовым процессом. Одна фазы – выявление отказа, и другая – поиск и устранение дефекта ПС, где этот отказ произошел.

Данная модель базируется на таких допущениях:

– все дефекты в ПС взаимно независимы с точки зрения обнаружения отказов;

– время между выявлением отказа и устранением дефекта определяет поведение функции надежности (им пренебречь нельзя);

– отношение вероятности выявления отказа в единицу времени к текущему количеству дефектов является постоянным;

– выявленный дефект можно полностью устранить и при этом новые дефекты не вносятся.

Эта модель графически имеет S-подобную кривую роста надежности:

$$\mu(t) = N(1 - (1 + \beta t)\exp(-\beta t)),\ N,\ \beta > 0,$$

где N – количество дефектов, которые имеются в ПС в начале тестирования; β – коэффициент пропорциональности, равный скорости выявления и устранения дефекта.

Функция интенсивности отказов $\lambda(t)$ имеет вид

$$\lambda(t) = N\beta^2 t \exp(-\beta t).$$

С увеличением времени тестирования t различие между S-подобной формой замедленного роста и кривой экспоненциального роста уменьшается. Эта модель дает лучшую оценку параметра N, если для оценивания используются данные о моментах времени и количестве устраненных дефектов, а не выявленных отказов.

Оценка параметров такой модели вычисляется методом максимального правдоподобия путем решения следующей системы уравнений [26, 28]:

$$2m_e / \beta = \sum_{i=1}^{m_e} (t - N \beta t^2 \exp(-\beta t_e),$$

$$N = m_e / (1-(1 + \beta t_e) \exp(-\beta t_e)),$$

где m_e – количество выявленных и устраненных дефектов за время тестирования, t_e – время тестирования, t_i – кумулятивные моменты отказов, $i=1, 2,..., m_e$.

Функция надежности определяется так:

$$R_T(t) = exp[-[\sum_{i=1}^{z} Np_i ((1+b_i T) exp(-b_i T) -$$

$$- (1+ b_i(t+T)) exp(-b_i(t +T))]].$$

Модель S-образного роста надежности с перегибами используется для анализа отказов ПО и оценки роста надежности системного ПО. Основополагающей концепцией является S-образный рост надежности, если некоторые ошибки нельзя обнаружить до того, когда будут устранены другие ошибки. Эта модель базируется на таких дополнительных допущениях:

– дефекты могут быть взаимозависимыми, т.е. некоторые из них невозможно обнаружить для устранения других;

– вероятность выявления отказа в единицу времени пропорционально текущему количеству выявленных дефектов и эта пропорциональность – величина постоянная;

– частота выявления отказов возрастает на протяжении всего периода тестирования;

– все выявленные дефекты устраняются и при этом новые дефекты не вносятся.

Модель характеризуется кривой S-образной формы и ее уравнения имеет вид

$$\mu(t) = N (1- \exp(-\beta t) / (1+ \psi \exp(-\beta t)),$$

где N– количество дефектов, которые имеет ПС в начале тестирования; β – скорость выявления отказа и устранения дефекта; ψ – параметр перегиба.

Параметр перегиба определяется так:

$$\psi(r) =(1- r) / r, \ 0 \le r \le 1,$$

где r – частота перегибов, представляющая собой отношение выявленных дефектов к общему их количеству.

При r=1 эта модель становится эквивалентной экспоненциальной модели, и только при $r<0,5$ кривая приобретает S-подобную форму.

Функция интенсивности отказов имеет вид

$$\lambda(t) = N\beta(1+\psi)\exp(-\beta t) / (1+\psi\exp(-\beta t))^2,$$

а ее максимум является точкой перегиба функции $\mu(t)$, достигается в точке $t = \ln(\psi)/\beta$ при $\psi > 1$.

Иными словами, данная модель основывается на предположении, что частота обнаружения ошибок повышается на протяжении всего периода тестирования.

8.4. ПРИМЕНЕНИЕ МОДЕЛЕЙ ДЛЯ ПРОГНОЗИРОВАНИЯ НАДЕЖНОСТИ ПС

Существование различных моделей надежности ПС означает, что для выбора среди них конкретной модели для оценки надежности определенного вида ПС должно происходить по критериям, учитывающим контекст, в котором эта модель будет применяться. К критериям относятся: достоверность предсказания, полезность и применимость.

Достоверность – это способность модели предсказывать характеристики будущих отказов в процессе испытаний и в зависимости от этого прогнозировать характеристики эксплуатационных отказов. Достоверность может зависеть также от вида модели и от процедуры вывода результата.

Полезность зависит от модели, обеспечивающей получение требуемой оценки риска по ее результатам и способности оценить количественные характеристики для их использовании при планировании и управлении проектом. Степень полезности проверяется путем рассмотрения относительной важности модели и числом оцениваемых такого рода величин:

– средняя наработка на отказ, интенсивность отказов и и их интевальное распределение;

– количество оставшихся ошибок;

– ресурсы и стоимость достижения показателей надежности;

– оценка четкости и ясности допущений для определения возможности применения модели надежности и др.

Применимость той или иной модели вытекает из ее ориентации на определенный класс ПС, задач, решаемых на этапах ЖЦ при проектировании системы, внешних условий разработки, возможностей среды функционирования системы и критериев достоверности и полезности.

Выше рассмотренные модели Муссы и Мусы–Окумоты рекомендованы для практического применения стандартами в об-

ласти инженерии надежности [30–32] при тестировании и эксплуатации ПС.

Для определения возможности практического применения этих моделей до начала тестирования вводятся дополнительные параметры:

$N(t)$ – ожидаемое количество дефектов, присутствующих в ПС в момент времени t;

T_s – среднее время, необходимое для обработки одного прогона при работе ПС на компьютере;

k_s – ожидаемая часть дефектов, которые проявятся за время T_s.

По этим показателям эти две модели удовлетворяют возможности их применения для прогнозирования надежности ПС. В частности наиболее эффективной является рассмотренная *базовая* экспоненциальная *модель Мусы* [28], в которой функция среднего числа отказов определяется по формуле:

$$m(t) = \beta_0 (1 - e^{-\beta_1 t}), \tag{8.1}$$

Кроме того, не менее эффективно применять и логарифмическую пуассоновскую *модель Мусы–Окумоты* [23], для которой эта функция определяется так:

$$m(t) = \beta_0 \ln(1 + \beta_1 t). \tag{8.2}$$

В этих моделях процесс отказов рассматривается как неоднородный пуассоновский процесс $\{N(t),\ t \geq 0\}$ с функцией интенсивности $\lambda(t)$, которая задает количество отказов за время t функционирования ПС. Допустив, что каждый отказ может быть вызван одним дефектом в ПС [27], получаем, что количество отказов в интервале времени [0, t] распределено по закону Пуассона, т.е.

$$P\{N(t) = n\} = \frac{[m(t)]^n e^{-m(t)}}{n!},\quad t \geq 0,\quad n = 0, 1, 2, \ldots$$

где $m(t) = m(\beta_1, \ldots, \beta_k) = \int_0^t \lambda(u)du$ – среднее количество дефектов, выявленных в ПС при его функционировании на протяжении времени t; $\beta_1, \ldots, \beta_k$ – параметры модели.

8.4.1. Анализ применения модели надежности Мусы при тестировании

Модель Мусы, как было показано выше, позволяет определить число отказов $m(t)$, а также среднее количество дефектов,

выявленных за единицу времени, которое равно $N(t) \cdot \frac{k_s}{T_s}$, при этом функция пуассоновского процесса $N(t)$ не возрастает.

Тогда количество дефектов, которые будут выявлены на протяжении некоторого времени Δt, можно вычислить по формуле:

$$N(t + \Delta t) - N(t) = -N(t) \frac{k_s}{T_s} \Delta t \,.$$

При $\Delta t \to 0$, получим дифференциальное уравнение в виде

$$\frac{dN(t)}{dt} \cdot T_s = -k_s N(t) \,. \tag{8.3}$$

Значения T_i вычисляется по формуле [24]:

$$T_i = I \, \varphi / \rho$$

где I – число инструкций исходного кода ПС, φ – коэффициент числа инструкций выполняемого кода, ρ – интенсивность выполнения кода (скорость процессора).

Ожидаемую часть дефектов k_s можно определить так:

$$k_s = K \frac{T_s}{T_L} \,, \tag{8.4}$$

где K – коэффициент выявления дефектов в ПС, а отношения $\frac{T_s}{T_L}$ зависит от структуры ПС. Тогда из (8.3) и (8.4) имеем

$$\frac{dN(t)}{dt} = -\frac{K}{T_L} \cdot N(t) \,. \tag{8.5}$$

Здесь $\frac{K}{T_L}$ – определяет интенсивность проявления дефектов.

Поскольку в модели Мусы предполагается, что K не зависит от времени, из (8.5) получим

$$N(t) = N_0 \cdot e^{-\frac{K}{T_L} t}$$

или с учетом (8.1) имеем

$$m\,(t) = N_0 - N(t) = N_0 (1 - e^{-\frac{K}{T_L} t}) \,,$$

где N_0 – количество скрытых дефектов в ПС в начале тестирования (при $t = 0$).

Таким образом, параметры модели Мусы $\beta_0 = N_0$, $\beta_1 = \frac{K}{T_L}$ обозначают соответственно: β_0 – общее количество дефектов, которые нужно обнаружить, β_1 – коэффициент интенсивности отказов, которое приходится на один дефект.

8.4.2. Анализ применения модели надежности Мусы–Окумоты при тестировании

Модель Мусы–Окумоты, как было показано выше, позволяет определить число отказов *m(t)*, а также функцию среднего числа отказов (8.2), которая записывается в виде

$$e^{\frac{m(t)}{\beta_0}} = (1 + \beta_1 t).$$

Тогда интенсивность отказа вычисляется так:

$$\lambda(t) = m'(t) = \frac{\beta_0 \beta_1}{1 + \beta_1 t} = \beta_0 \beta_1 e^{-\frac{m(t)}{\beta_0}} =$$

$$= \beta_0 \beta_1 e^{-\frac{N_0 - N(t)}{\beta_0}} = \beta_0 \beta_1 e^{-\frac{N_0 - I \cdot D(t)}{\beta_0}}, \qquad (8.6)$$

где *D(t)* – плотность дефектов в момент времени *t*.

В отличие от модели Мусы, в данной модели коэффициент выявления дефектов *K* не является постоянным и зависит от момента времени *t*.

Поэтому из (8.5) можно получить $K(t) = T_L \frac{\lambda(t)}{N(t)}$ и с учетом (8.6) имеем

$$K(t) = \frac{T_L}{ID(t)} \beta_1 e^{-\frac{N_0 - I \cdot D(t)}{\beta_0}} = \left(\frac{T_L}{ID(t)} \beta_0 \beta_1 e^{-\frac{N_0}{\beta_0}} \right) e^{\frac{I \cdot D(t)}{\beta_0}}.$$

Определим коэффициент *K* как функцию плотности дефектов *D*, а не времени *t*, т.е.

$$K(D) = \frac{\alpha_0}{D} e^{\alpha_1 \cdot D}, \qquad (8.7)$$

где $\alpha_0 = \frac{\beta_0 \beta_1 \varphi}{\rho} e^{-\frac{N_0}{\beta_0}}$ (8.8)

$$\alpha_1 = \frac{I}{\beta_0}. \qquad (8.9)$$

Из выражений (8.8) и (8.9) находим

$$\beta_0 = \frac{I}{\alpha_1},\ \beta_1 = \frac{\alpha_0 \rho \alpha_1}{\varphi I} e^{\frac{N_0 \cdot \alpha_1}{I}}. \qquad (8.10)$$

Для интерпретации α_0 и α_1 во время процесса тестирования возьмем производную по D в формуле (8.7) и приравняем ее нулю

$$-\frac{\alpha_0}{D^2} e^{\alpha_1 \cdot D} + \frac{\alpha_0}{D} e^{\alpha_1 \mathrm{D}} \alpha_1 = 0,$$

в результате плотность $D_{\min}$ дефектов в ПС, для которой коэффициент K имеет минимальное значение $K_{\min}$, равно $D_{\min} = \frac{1}{\alpha_1}$.

Тогда из (8.7) имеем $K_{\min} = \frac{\alpha_0 \mathrm{e}^{\alpha_1 \cdot D}}{D_{\min}} = \frac{\alpha_0 e}{D_{\min}}$,

$$\alpha_0 = \frac{K_{\min} \mathrm{D}_{\min}}{e} \text{ и } \alpha_1 = \frac{1}{D_{\min}}, \qquad (8.11)$$

которые зависят от характеристик процесса тестирования.

Из уравнений (8.10) и (8.11) получаем

$$\beta_0 = I D_{\min},\ \beta_1 = \frac{K_{\min} \rho}{\varphi I} e^{\frac{D_0}{D_{\min}}},$$

где D_0 – плотность дефектов в начале тестирования ПС $D_0 = \frac{N_0}{I}$.

Таким образом, параметр β_0 пропорционален размеру ПС и зависит от того, как изменяется эффективность тестирования с изменением плотности дефектов. Параметр β_1 зависит от минимального значения коэффициента выявления дефектов $K_{\min}$, а также от отношения $\frac{D_0}{D_{\min}}$.

Модель Мусы может использоваться на ранних этапах разработки ПС до начала системного тестирования и для раннего прогнозирования эксплуатационной надежности ПС.

Известно, что $K / T_i = \lambda_0 / N_0$, где $\lambda_0 = \lambda(0)$ интенсивность отказов ПС в начале системного тестирования [24].

Тогда $\lambda_0 = N_0 K / T_i = N_0 \rho K / I\varphi$ и из уравнения (8.1) $m(t)$ будет иметь вид

$$m(t) = N_0(1 - \exp(-\frac{\lambda_0}{N_0} t)) \cdot$$

Параметры K, ρ, φ, I – определяются до *начала тестирования* и означают:

K – постоянный коэффициент выявления дефектов $K = 4 \cdot 10^{-7}$;

ρ – скорость процессора компьютера, на котором будет установлена ПС в среде эксплуатации;

φ – определяется примененным языком программирования ПС и изменяется от 1,5 до 10 и больше.

Таким образом, результаты исследований моделей на процессе тестирования показывают, что логарифмическую модель надежности Муси–Окумоти не целесообразно применять на этапе раннего прогнозирования, поскольку, в отличие от модели Мусы, определение ее параметров основывается на данных, полученных при тестировании ПС.

8.4.3. Определение надежности функциональных компонентов

Надежность выполнения функциональных компонентов ПС, как правило, оценивают во время системного тестирования и эксплуатации, поэтому решается задача априорного нахождения оптимального уровня надежности каждого модуля ПС, реализующего некоторую прикладную функцию, а затем уровня надежности ПС [6, 33]. Введем следующие обозначения:

u_i – коэффициент веса функции F_i, $u = 1, ..., k$;

v_{ij} – коэффициент веса j-го ПС при выполнении i-функции, $u = 1,..., k$; $j = 1,...,l$;

v^*_j – общий коэффициент веса Z_j ПС;

q_j – надежность работы ПС Z_j в период t эксплуатации;

w_{js} – коэффициент веса s-го модуля при выполнении j-го программного приложения, $s = 1,...,m$; $j = 1,...,l$;

w^*_s – общий коэффициент веса модуля M_s;

r_s – надежность модуля M_s в период эксплуатации t;

E_j – множество номеров всех модулей, которые необходимы для выполнения j-го элемента ПС;

α_s – нижняя граница надежности модуля M_s;

β_s – верхняя граница надежности модуля M_s;

Модель распределения надежности модулей базируется на системе соответствия весовых коэффициентов на разных уровнях иерархии, т.е.

$$Q_{nc} = \sum_{j=1}^{l} v_j^* q_j = \sum_{s=1}^{m} w_s^* r_s.$$

При условии независимости модулей надежность q_j каждого j-го элемента ПС $(j = 1, ...,l)$, которое в своей работе использует модули M_s, $(s = 1,..., m)$, вычисляется по формуле

$$q_j = \prod_{n \in E_j} r_n,$$

где $r_n \in (0,1]$ для всех $n \in E_j$.

Поскольку на практике во время испытаний и эксплуатации ПС оценивается надежность ПС, соответствующее распределение надежности для q_j, $j = 1, ..., l$, оценивается по приведенной выше формуле.

Контроль достижимости уровня надежности базируется на построении *прогнозов* для каждого j-го элемента ПС $(j=1,...,l)$ в контрольных точках. Прогнозирование надежности каждого элемента ПС проводится с помощью моделей Мусы и Мусы–Окумоты.

8.5. ОЦЕНКА НАДЕЖНОСТИ НА ЭТАПАХ ЖЦ

Некоторые типы систем реального времени, обеспечения безопасности и другие требуют высокой надежности (недопустимость ошибок, точность, достоверность и др.), которая в значительной степени зависит от количества оставшихся и не устраненных ошибок в процессе ее разработки на этапах ЖЦ. В ходе эксплуатации ошибки также могут обнаруживаться и устраняться. Если при их исправлении не вносятся новые либо вносится их меньше, чем устраняется, то в ходе эксплуатации надежность системы непрерывно растает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность.

На надежность ПО влияют, с одной стороны, угрозы, приводящие к неблагоприятным последствиям, риску нарушения безопасности системы, и с другой стороны, способность совокупности компонентов системы сохранять устойчивость в процессе ее эксплуатации. Риск уменьшает свойства надежности, особенно если обнаруженные ошибки могут быть результатом проявления угрозы извне.

Методы и модели постоянно развиваются, поскольку надежность является одной из ключевых проблем современных ПС. Появилось новое направление – инженерия надежности ПО (Software reliability engineering –SRE), которое ориентировано на количественное изучение операционного поведения компонентов ПС по отношению к пользователю, ожидающему надежную работу системы [30], и включает в себя:

1) измерение надежности, т.е. проведение количественной оценки методами предсказаний, сбора данных о поведении системы в процессе тестирования и эксплуатации ПС;

2) оценку стратегии и метрик конструирования и выбора готовых компонентов в процессе разработки компонентов системы, а также среды функционирования, влияющую на надежность работы системы;

3) современные методы инспектирования, верификации, валидации и тестирования при разработке отдельных компонентов и системы в целом.

В инженерии надежности определен новый термин dependability, означающий пригодноспособность, т.е. надежность в широком смысле [31]. По сравнению с термином reliability новый термин обозначает способность системы обладать свойствами, желательными для пользователя и дающими ему уверенность в качественном выполнении функций, заданных в требованиях к системе. Dependability добавляет дополнительные атрибуты, которыми должна обладать ПС, а именно:

– готовность к использованию (availability);

– готовностью к непрерывному функционированию (reliability);

– безопасность для окружающей среды, т.е. способность системы не вызывать катастрофических последствий в случае отказа (safety);

– секретность и сохранность информации (confidential);

– способность к сохранению системы и устойчивости к самопроизвольному ее изменению (integrity);

– способность к эксплуатации ПО, простоту выполнения операций обслуживания, а также устранения ошибок, восстановление системы после их устранения (maintainability) и т.п.;

– готовность и сохранность информации (security) и др.

Достижение требуемой надежности системы обеспечивается путем предотвращения отказа (fault prevention), его устранения (removal fault), возможного выполнения системы при его наличии и оценки возможности появления новых отказов и мер

борьбы с ними. Отказы носят случайный характер, анализ которых основывается на методах теории вероятностей и случайных процессов. С учетом этого и разработано огромное количество моделей надежности, самые важные из них приведены выше.

Каждый программный компонент, его команды и данные обрабатываются в дискретные моменты времени t_1, t_2,..., t_n и это может быть осуществлено «удачно» или нет. Пусть за время T после первого неудачно обработанного компонента системы появился отказ и q_b – вероятность этой неудачи, тогда $P\{ T > t_n \} = (1 - q\ t_n)^n$, а среднее время ожидания $T = t / q\ t_n$.

Положим, что t убывает так, что время T остается фиксированным, тогда имеем $P\{ T > t \} = (1 - t / T)^{ft\ t} \approx e^{-t/T}$, т.е. время до отказа ПС в данном случае является непрерывной величиной *c...l* распределенной экспоненциально с параметром *T...1/T*.

Обеспечение надежности на этапах ЖЦ. Для получения высокой надежности ПС требуется наблюдать за достижением этого показателя качества на всех этапах ЖЦ, о чем свидетельствуют рекомендации стандарта ISO/IEC 12207 [34]. В нем управление качеством (а значит, и управление основным показателем качества – надежность), определен как новый обязательный процесс ЖЦ в организации выполнения основных процессов реализации ПС. В связи с этим рассмотрим цели и задачи обеспечения надежности, состоящие в возможности предусмотреть возникновение отказов и ошибок в ПС, и собрать статистику их появления и исправления на основных этапах ЖЦ:

– спецификация требований,
– проектирование,
– реализация,
– тестирование,
– испытание,
– сопровождение.

Исходя из организационного характера процесса управления качеством, обеспечение надежности предполагает составление плана-графика, в котором отражаются следующие виды действий:

– выделение управляемых и неуправляемых факторов процесса разработки, влияющих на надежность (управляемые факторы – решения об инспекциях, объемах всех видов ресурсов при тестировании, неуправляемые факторы – параметры среды функционирования, опыт обслуживающего персонала, объем продукта, возможность изменения исходных требований к ПС и др.);

– выбор необходимых значений управляемых факторов для оценки достижения целевых требований к интенсивности отказов и принятия необходимых ограничений;

– анализ факторов, влияющих на интенсивность отказов;

– разработка планов тестирования и испытания продукта для оценки надежности ПС, в том числе при спецификации требований, соответствии их требованиям стандарта и проведение работ по проверке и аттестации готового продукта [7, 34].

На этапе спецификации требований определяются задачи и внешние спецификации основных (целевых) требований к системе и ПС с заданием количественных метрик для оценки надежности, выраженных в терминах интенсивности отказов или вероятности безотказного его функционирования.

Определение этих задач проводится путем интервью разработчика с пользователем для формирования:

– приоритетов функций по критерию важности их реализации в ПС;

– сценариев выполнения функций;

– параметров среды и интенсивности использования функций программ и их отказов;

– характеристик модели (входные и выходные данные) и входного пространства для каждой функции;

– категорий отказов и их интенсивности при выполнении функции в заданном сценарии с числом отказов в единицу календарного времени.

Результатом работы с пользователем по анализу проекта является:

– классификация отказов программного обеспечения (по степени серьезности);

– обоснование требований к интенсивности отказов на основе компромисса между надежностью, стоимостью и усилиями, затрачиваемыми на выпуск продукта;

– определение отношения заказчика к отказам разной степени серьезности и его готовности оплачивать снижение интенсивности отказов по каждой категории серьезности;

– построение функциональной конфигурации программного обеспечения с указанием частоты использования каждой функции;

– разбиение входного пространства каждой функции на категории данных для последующего функционального тестирования программ на этапе испытаний;

– оценка календарного времени работы с процессором ПС.

На этапе проектирования определяются:

– размеры, т.е. информационная и алгоритмическая сложность всех типов проектируемых компонентов;

– категории дефектов, свойственные всем типам компонентов ПС;

– стратегии функционального тестирования компонентов по принципу «черного ящика» с помощью тестов для тестирования и выявления дефектов в классе категории данных.

Для компонентов ПС, выполняющих разные функции, проводится:

– классификация возможных дефектов и степени их распространения в ПС;

– определение среды, режимов и интенсивности использования каждого компонента;

– оценка риска использования некоторого компонента в заданном контексте системы;

– оценка воздействий отказов компонентов на устойчивость всей системы и др.

Для достижения надежного продукта анализируются:

– варианты архитектуры ПС на соответствие поставленным требованиям к надежности;

– виды технологий анализа риска, режимов отказов, деревьев ошибок и перечень критических компонентов для обеспечения свойств отказоустойчивости и восстанавливаемости ПС;

– прогнозирование показателей размера ПС, чувствительности к ошибкам, степени тестируемости, оценки риска и сложности системы.

При необходимости для улучшения надежности ПС проводится перераспределение некоторых задач между исполнителями для перепроектирования, подбора новых повторно используемых компонентов и др.

На этапе реализации ПС проектные спецификации переводятся в тексты ЯП и подготавливаются наборы тестов для автономного и комплексного его тестирования.

При проведении автономного тестирования обеспечение надежности состоит в предупреждении появления дефектов в отдельных компонентах и создание эффективных методов защиты от них. Все последующие этапы разработки не могут обеспечить надежность ПС, а лишь способствуют повышению уровня надежности за счет обнаружения оставшихся ошибок с помощью тестов различных категорий.

Обеспечение надежности на этом этапе достигается за счет:

– применяемой методологии сбора для анализа информации об аномалиях, дефектах и отказах;

– методологии обнаружения и локализации дефектов разных категорий;

– формирования критериев завершения тестирования, установления сроков завершения тестирования и стоимости ресурсов тестирования.

На этапе испытаний создается план испытаний ПС, по которому проводится тестирование на соответствие внешним спецификациям функций и целям проекта.

Испытания программного продукта должна проводить группа специалистов или пользователей в реальной среде функционирования или на испытательном стенде для имитации функций компонентов по планам испытаний. При подготовке к испытаниям изучается "история" тестирования на стадиях ЖЦ в целях непосредственного использования ранее разработанных тестов, а также составления специальных тестов испытаний.

На этапе испытаний в соответствии с планом осуществляется:

– оценка надежности по результатам системного тестирования и полевых испытаний по соответствующим *моделям надежности*, принятым в этих целях;

– управление ростом надежности путем неоднократного исправления и регрессионного тестирования ПС;

– принятие решения о степени готовности ПС и возможности его передачи в эксплуатацию.

На этапе сопровождения оценка надежности ПС осуществляется по моделям надежности, соответствующим типу ПС. Если обнаружены ошибки и внесены необходимые изменения в ПС, проводят такие мероприятия:

– протоколирование отказов в ходе функционирования ПС и измерение надежности функционирования, а также использование результатов измерений при определении потерь надежности в период времени эксплуатации;

– анализ частоты и серьезности отказов для определения порядка устранения соответствующих ошибок;

– оценка влияния функционирования ПС на надежность в условиях усовершенствования технологии или использования новых инструментов разработки ПС.

Таким образом, показано, что надежность является одной и главных характеристик ПС, для которой разработано большое количество моделей для разных видов и типов ПС. Рассмотрены основные базовые понятия надежности, обеспечивающие оценку надежности по соответствующим моделям надежности ПС, основанным на времени функционирования и/или количестве отказов (ошибок), полученных в программах в процессе их тестирования или эксплуатации. Предложены виды классификации моделей надежности. Одна из классификаций учитывает случайный марковский и пуассоновский характер процессов обнаружения ошибок в программах, другая – характер и интенсивность отказов. Некоторые модели позволяют прогнозировать число ошибок в процессе тестирования, другие оценивать надежность с помощью функций надежности и по данным, собранных при испытании. Проведен анализ применения моделей надежности Мусы и Мусы–Окумоты при тестировании ПС. Дана краткая характеристика методов достижения надежности ПС на этапах ЖЦ.

ЧАСТЬ

4

ИНЖЕНЕРИЯ ПРОГРАММИРОВАНИЯ: управление качеством, проектом и конфигурацией

ГЛАВА 9

ОСНОВЫ ИНЖЕНЕРИИ. МЕТОДЫ УПРАВЛЕНИЯ И ОЦЕНКИ КАЧЕСТВА ПС

9.1. ОСНОВЫ ИНЖЕНЕРИИ

Одной из характерных черт инженерии в промышленности является технология управления созданием изделий на основе использования готовых проектных решений и деталей. В программировании промышленное использование готовых решений и программных продуктов еще не стало повседневной практикой, а сформировались главные признаки инженерной деятельности [1–3] и стандарт ISO/IEC 12207 [4] на технологический цикл работ по созданию ПС, начиная с начальной стадии разработки системных требований, реализации ПС и кончая применением продукта пользователем.

В этом стандарте (Приложение 2) регламентированы процессы управления проектом, качеством и «формой» (конфигурацией) продукта, а также задачи организации коллективной разработки. Кроме стандартов, международным сообществом специалистов в области программирования сформировано ядро знаний программной инженерии SWEBOK [1], в котором определены методы и средства для разработки продукта соответственно на каждом процессе ЖЦ, а также менеджмент, система качества и оценка продукта.

Инженерная деятельность согласно стандарту должна планировать и контролировать выполнение разных работ на процессах. *Планирование*, как и в промышленности, охватывает такие главные составляющие разработки ПС – время, ресурсы и стоимость, которые задает заказчик в контракте на разработку. Иными словами, инженерия программирования – это технология, регламентирующая процессы изготовления продукта с применением *методов управления, планирования, контроля и оценки* результатов *процессов*.

В основе инженерной деятельности в программировании лежат следующие базовые понятия [5].

Управление – это метод упорядочения системы (проекта), состоящий в управлении ее целостности, регламентации деятельности людей для достижения поставленной цели проекта со связями между элементами системы.

Планирование – это научно обоснованное распределение целей, задач и работ на проекте с согласованием сроков, времени и стоимости его реализации в плане-графике.

Контроль – это проверка продукта на соответствие заданным целям и свойствам, а также составление плана изменения этого продукта при обнаруженном несоответствии требований и функций.

Эти базовые понятия применяются в программных проектах и направлены на повышение уровня качества проектирования проектов.

В создании программного продукта, кроме программистов, принимают участие новые категории инженеров – управленцы (менеджеры), плановики, контролеры, тестировщики, верификаторы, сертификаторы и др. Каждый из них выполняет соответствующую для данной категории работу на процессах ЖЦ с соблюдением условий и ограничений заключенного контракта на этот продукт.

Одним из главных источников инженерии программирования является специализация, типизация проектных решений и продуктов, а также их классификация и каталогизация для обеспечения производственного характера его изготовления путем сборки из готовых, накопленных ПИК [1, 6, 7].

На данный момент времени принципы сборочного производства ПС из готовых ПИК задействованы на технологической линии выпуска продукции (Framework for Product Line Practice) [8], предназначенной для удовлетворения потребностей рынка в

некотором виде продукта. Аналогичные задачи производства программного продукта выполняет *инженерия приложений* (application engineering) из ПИК в виде самостоятельных продуктов, в том числе многоразового применения, в разных предметных областях.

Более высоким уровнем этой инженерии, включающей в себя классификацию и типизацию объектов предметной области, а также методы поиска готовых ресурсов для покрытия задач отдельных объектов, частей систем или целого семейства, является *инженерия домена* (domain engineering), которая базируется на инструментальных средствах, поддерживающих методы накопления ПИК в репозитариях и технологию встраивания их в новые элементы семейства ПС с идентичными типовыми задачами. Иными словами, доменная инженерия ориентирована на изготовление типовых классов ПС или их составных частей, выполняющих в домене типовые функции и задачи, которые могут применяться в других доменах этого же семейства или нового.

Главное, что объединяет приведенные виды производства – это планирование и управление выпуском продукта, его сроками и затратами, а также получением дохода за счет экономии расходов от использования готовых ПИК.

9.1.1 Инженерия повторно используемых компонентов – ПИК

Прикладная инженерия (Engineering Application) из ПИК – это систематическая и целенаправленная деятельность по выделению реализованных программных артефактов, анализу их функций для классификации и каталогизации готовых ПИК в целях дальнейшего применения в проектируемых системах. Согласно стандарту ISO/IEC 12207 эта деятельность является планируемой в части определения сроков поиска ПИК, принятия решений о применении, оценки стоимости для приобретения и адаптации к новым условиям среды в целях получения прибыли за счет экономии трудозатрат на повторную разработку [1, 6, 7].

В рамках компонентного программирования ПИК – это самостоятельный компонент, который удовлетворяет определенным функциональным и архитектурным требованиям и организации взаимодействия с другими в заданной среде, имеет спецификацию, помогающую объединять его с другими компонентами в интегрированной ПС.

Модель спецификации компонента имеет вид

$M_{ПИК} = (T, I, F, R, S)$,

где T – тип компонента, I – множество интерфейсов, F – функциональность компонента, R – реализация, S – схема взаимодействия со средой (шаблон развертывания).

ПИК разделяются на следующие группы:

– простые компоненты (функция, модуль, класс и др.);

– компонент, интерфейс, функциональность и реализация на ЯП, а также спецификация шаблона развертывания;

– готовые к применению ПИК (например, библиотека классов, AWT компоненты и др.);

– сложные ПИК (каркасы, паттерны и контейнеры).

Систематическое применение ПИК основывается на двух процессах.

Первый процесс – это создание ПИК путем:

– изучения спектра реализуемых задач ПрО, выявления типовых решений и общих функций, которые могут быть повторно используемые;

– реализации типовой функции компонентом и определение интерфейсов;

– спецификации компонента и его параметров для обеспечения связи с другими;

– каталогизации готового компонента и его интерфейса в соответствующих хранилищах для обеспечения их поиска заинтересованными лицами.

Реализация данного процесса предполагает наличие определенного опыта в принятии решения о нескольких подобных между собой задач, обладающих общими чертами и различиями для нахождения типового решения при реализации, сохранении в каталогах и задания приемов настройки на характерные для каждой отдельной задачи особенности.

Второй процесс – конструирование из ПИК новой системы следующими действиями:

– определение специфики ПрО, ее функций и требований к ним;

– определение типовых решений и ЯП описания функций;

– поиск в каталоге готовых ПИК для типовых функций ПрО, которые могут быть использованы в проектируемой системе;

– сопоставление целей новой разработки с возможностями найденных ПИК, апробация каждого компонента и принятие решения о целесообразности приобретения (в смысле функциональности и стоимости);

– определение схемы интеграции приобретенного ПИК в новую разработку и обеспечение его связи с другими элементами ПС.

Построение ПС с помощью готовых ПИК дает экономию трудозатрат вместо повторной разработки аналогичного продукта путем настройки его на новые условия среды функционирования.

9.1.2. Инфраструктура инженерии приложений

В инженерии приложений (Domain Engineering) в качестве ПИК могут быть формализованные артефакты деятельности разработчиков ПС, отражающие отдельные функции, промежуточные продукты процессов разработки ПС (требования, постановки задач, архитектура и др.); спецификации (модели, интерфейсов и т.п.); готовые компоненты или отдельные приложения и т.п. Ими могут быть также прикладные, общесистемные компоненты и универсальные общесистемные средства. *Прикладные компоненты* выполняют отдельные задачи и функции в некоторой прикладной области (бизнесе, коммерции, экономики и т.п.). *Общесистемные компоненты* – это трансляторы, редакторы тестов, системы генерации, интеграции, загрузчики и др., которые массово используются в практике программирования. *Универсальные общесистемные* средства (ОС, СУБД, сетевое обеспечение, электронная почта и др.) предоставляют сервис, обмен данными и сообщениями.

ПИК как инженерный компонент имеет типовую структуру, состоящую из таких основных разделов [7]:

– интерфейс (interfaces),
– реализация (implementation),
– схема развертки (deployment).

Интерфейс – это обращение к другим компонентам с помощью параметров, описанных в IDL или APL, а также описание типов данных и связей. Он включает в себя определение сигнатуры и методов взаимодействия.

Реализация – это код компонента для обращения к операциям интерфейса. Реализаций может быть несколько, которые зависят от количества интерфейсов, операционной среды, платформы и т.п.

Развертка – это физический файл, готовый к выполнению и содержит все необходимые операции для инсталляции, настройки и функционирования компонента.

В инфраструктуру ПИК входят методы накопления и применения их в процессе изготовления новых программных систем.

Репозитарий ПИК. В инфраструктуре разработки ПС репозитарий – это система средств для хранения, пополнения и изменения ПИК, а также для организации поиска необходимого ПИК некоторым пользователем. Каждый ПИК представлен в репозитарии поисковым образом с аннотацией функций, параметров, интерфейсов и др. Репозитарий по запросу пользователя выдает сведения о видах ПИК и их характеристиках, является инструментом, упрощающим и сокращающим сроки разработки новой ПС за счет:

– отображения в нем множества базовых функций и понятий ПрО;

– предоставления операций обновления ПИК и доступа к ним;

– средств обработки непредвиденных ситуаций, возникающих в процессе выполнения запроса поиска ПИК в репозитарии и др.

Поисковый образа ПИК – это информационная модель, которая определяет систему хранения, поиска и сопоставления этого образа с заданным в запросе. Репозитарий разбит на разделы, соответствующие ПрО, перечень которых составляет классификационный каталог первого уровня. Классификаторы следующих уровней – это понятия ПрО и ПИК из разделов каталога.

Информационная модель ПИК содержит: паспортные данные (имя и адрес разработчика, цену и т.п.), название среды (ОС, ЯП, СУБД и т.п.), требуемые ресурсы, имена ПИК в классификационном каталоге, нефункциональные требования (к безопасности, защите, качеству и прочее) и др.

Классификация компонентов проводится по следующим группам:

– компонент типа класс, модуль и т.п.;

– компоненты, интерфейсы (входные и выходные параметры, пред–и постусловия), функциональность и реализация с шаблоном развертывания;

– готовые ПИК;

– каркасы и паттерны как средства взаимодействия отдельных ПИК.

Каталогизация обеспечивает размещение информационной модели и кодов ПИК в репозитарии для их извлечения по запросам в соответствии с заданными потребностями и принятия решения о приобретении ПИК для дальнейшего применения с настройкой на условия новой среды.

9.1.3. Инженерия предметной области

Базисом инженерии программирования, основанного на использовании ПИК, является, как было сказано выше, прикладная инженерия и инженерия ПрО, которые используют накопленные и готовых ПИК, программы, а также отдельные части ПС многоразового применения.

Инженерия ПрО ориентирована на создание архитектуры – каркаса (фреймворка), представленного с помощью ПИК, компонентов многоразового применения из семейства программ ПрО и их интерфейсов.

Основными этапами инженерии ПрО являются:

– анализ ПрО, выявление объектов и отношений между ними;

– определение области действий объектов ПрО;

– определение общих функциональных и изменяемых характеристик, построение модели характеристик, устанавливающей зависимость между различными членами семейства, а также в пределах членов семейства системы;

– создание базиса для производства конкретных программных членов семейства с механизмами изменчивости независимо от средств их реализации;

– подбор и подготовка компонентов многократного применения, описание аспектов выполнения задач ПрО;

– генерация отдельного домена, члена семейства и ПС в целом.

Генерация продукта для ПрО выполняется с помощью модели характеристик и выбранного набора компонентов, реализующих задачи этой области. Используя данную модель, знания о конфигурациях и спецификации компонентов, участвующих в этом процессе, генерируются отдельные представители семейства, а также ПС для всей ПрО.

Инженерия ПрО использует следующие вспомогательные процессы:

– корректировка процессов для разработки новых решений на основе ПИК;

– моделирование изменчивости и зависимостей компонентов многоразового использования, фиксации их в модели характеристик и в справочнике информации об изменении моделей (объектных, Use Case и др.). Фиксация зависимостей между характеристиками модели избавляет разработчиков от некоторых конфигурационных операций, выполняемых, как правило, вручную;

– разработка инфраструктуры ПИК – описание, хранение, поиск, оценивание и объединение готовых ПИК;

– создание репозитария ПИК и компонентов многоразового использования в классе задач ПрО;

– обеспечение безопасности, защиты данных, изменений;

– обеспечение синхронизации и взаимодействия компонентов и ПИК.

Стандартизация процессов доменной инженерии. В стандарте ISO/IEC 12207 (расширение 2002 года) приведено описание процесса доменной инженерии (Domain engineering process), как нового процесса ЖЦ. Согласно стандарту процесс доменной инженерии охватывает ряд видов деятельности.

При *анализе домена* определяются:

– границы домена и его связи с другими доменами;

– общие и отличительные особенности домена (постоянные и переменные требования) и их включение в модель домена;

– словари описания основных понятий домена и взаимосвязи между активами/ресурсами в домене;

– классы моделей ПрО;

– методы оценки моделей и словарей домена.

Проектирование домена (Domain design) – это определение архитектуры домена с помощью программных компонентов – специфичных активов/ресурсов. Архитектура домена – каркас из ПИК, активов и формально определенных интерфейсов, которая согласовывается с моделью домена, стандартами организации и оценивается на соответствие выбранной методологии проектирования.

Технология доменной инженерии включает в себя стандартизированные подпроцессы:

– формирование ресурсов (Asset provision) или их приобретение для их применения при компоновке новых ПС или подсистем;

– разработка базы ресурсов (asset-based development) с помощью ПИК (software reuse) и методов компоновки ПС домена;

– сопровождение ресурсов (Asset maintenance) – модификация и эволюция модели, архитектуры и продуктов домена за счет готовых ресурсов типа ПИК.

Данной технологии требуются методики и инструменты для эффективной генерации систем из ПИК и компонентов многоразового применения.

В результате применения технологии доменной инженерии в софтверной организации создается, поддерживается и развивает-

ся **архитектурный базис** из множества ПИК, хранящихся в репозитарии, который учитывает общие и специфические особенности разных сторон деятельности в доменах.

Таким образом, инженерия ПрО – это обеспечение многоразового применения используемых решений для семейств ПС, а инженерия приложений – это производство одиночного приложения из ПИК.

9.2. УПРАВЛЕНИЕ И ОЦЕНКА КАЧЕСТВА ПС

В условиях постоянного увеличения роста программной продукции остро стоит проблема качества. Проверка количественных и качественных показателей качества в процессе разработки ПС и на заключительном этапе не всегда завершается созданием безошибочного продукта. Поэтому инженерия качества предназначена для наблюдения и контроля достижения показателей качества ПС, начиная с ранних этапов разработки, как отдельных компонентов, так и ПС и кончая полным изготовлением программного продукта. Стандарт ISO/IEC 12207:2002 [4] рекомендует новый процесс ЖЦ – управление качеством, который вместе с процессами верификации и тестирования должен гарантировать достижение качества ПС на основе моделей и методов инженерии программирования, применяемых при реализации задач и действий процессов ЖЦ стандарта.

Качество по определению стандартов ISO/IEC 9126 (1, 2) и государственных стандартов [9–14] – это «совокупность свойств ПС, обеспечивающих ее способность удовлетворять установленным или предполагаемым потребностям в соответствии с назначением».

Проблеме качества посвящено много публикаций [15–22], в которых изложены методы достижения и оценки качества программ. Главная их особенность – модель качества [11] и множество мероприятий по ее реализации на практике для разных типов программных продуктов: планирование, управление, контроль, сбор данных об ошибках и дефектах при тестировании и подходы к оценке показателей качества.

Новым является детальное исследование одного из главных показателей надежности – завершенность и определение мер ее зависимости от вероятности отказов и наличия скрытых дефектов в ПС [17–19], что дает гарантию более высокого качество за счет разработки инженерии наблюдения, контроля и оценивания результатов проектирования компонентов ПС, начиная с самых ранних процессов ЖЦ.

9.2.1. Управление качеством ПС

Под *управлением качества* понимается совокупность организационной структуры и ответственных лиц, а также процедур, процессов и ресурсов проектирования для достижения требуемого качества ПС.

Согласно стандарту ISO/IEC 9126−2 [11] управление качеством ПС − SQM (Software Quality Management) состоит в планировании и систематическом контроле качества проектируемого продукта на процессах ЖЦ для гарантии, что продукт будет удовлетворять потребителя не только назначением, но и качеством. Процесс планирования начинается с выявления требований к ПС, определения требований к показателям качества, управления задачами реализации этих требований на процессах ЖЦ для достижения необходимого качества.

Задачи управления качеством ПС состоят в следующем:

− достижение требуемого качества путем анализа всех рабочих продуктов ЖЦ в заданных контрольных точках;

− достижение качества, базирующегося на метриках, критериях и приемах измерения показателей качества и др.;

− контроль и верификация качества (SQA, V&V) на процессах ЖЦ в целях определения степени достижения заданных показателей, и если они не достигнуты, то регулирования процессов с привлечением специальных методов, позволяющих обеспечить продукты необходимыми свойствами.

Эти задачи выполняются группой качества, которая проводит планирование, оперативное управление, контроль процессов разработки ПО и оценку качества продукта и процессов ЖЦ.

Планирование качества − это деятельность, направленная на определение целей и требований к качеству, а также составление календарного план-графика для проведения анализа реализации требований с учетом принятых критериев, последовательное измерение свойств и технических показателей промежуточных продуктов ЖЦ, а также рассмотрение найденных ошибок, отказов и дефектов, обнаруженных на более поздних процессах ЖЦ.

Оперативное управление − это методы и виды деятельности оперативного характера для текущего управления и проведения контроля результатов проектирования, а также для выявления и устранения недостатков и причин неудовлетворительного обеспечения свойств качества процессом проектирования ПС.

Обеспечение качества − это проверка объекта разработки на разных процессах ЖЦ на удовлетворение внутренних и внешних

целей требований к качеству. Внутренние цели – создание уверенности у руководителя проекта, что качество постепенно обеспечивается в процессе реализации продукта. Внешние цели – это создание уверенности у заказчика и пользователей, что требуемое качество будет достигнуто.

Процесс анализа качества ПС (Software Quality Assurance – SQA) состоит в гарантии того, что продукты и процессы согласовываются с заданными требованиями и поддерживаются следующими видами деятельности [16]:

– внедрение стандартов, методик, соответствующих процедур разработки ПС, методик контроля свойств и отдельных показателей качества на процессах ЖЦ;

– оценивание соблюдения положений стандартов, процедур и методов контроля результатов разработки рабочих продуктов ПС;

– обеспечение качества компонентов и ПС в целом.

Стандарты [9–11] регламентируют два процесса обеспечения качества:

– гарантия (подтверждение) качества ПС, как результат определенной деятельности на каждом процессе ЖЦ, состоящей в проверке соответствия стандартам и процедурам достижения качества;

– инженерия качества, как процесс предоставления продуктам ПО свойств функциональности, надежности, сопровождения и других характеристик с учетом требований к системе.

При достижении качества выполняются такие действия:

– оценка правильности применения стандартов и процедур при разработке программ;

– ревизия управления, разработки и гарантии качества ПС, включая проектную документацию (отчеты, планы-графики и др.);

– контроль проведения формальных инспекций и просмотров рабочих продуктов;

– анализ результатов проведения приемочного тестирования (испытания) ПС;

– оценивание полученных результатов.

Гарантия качества зависит от методов проверки промежуточных продуктов ЖЦ, учета критериев и рекомендаций, а также оценки разных свойств и показателей качества на этапах ЖЦ. Гарантия качества – это проверка непротиворечивости и выполнимости планов создания качественного продукта, сопоставление полученных свойств промежуточных рабочих продуктов с плановыми показателями, а также измерение метрик продуктов и про-

цессов в соответствии с утвержденными стандартом и процедурами измерения.

Если требования к качеству на процессах ЖЦ не удовлетворены и в продукте обнаружены ошибки при его выполнении, то они устраняются, а затем проводится повторный контроль, верификация и валидация ПС и оценка показателей качества.

9.2.2. Инженерия качества ПС

Инженерия качества ПС – это методы планирования, определения требований к качеству, подходы к выбору и усовершенствованию моделей качества, методы управления и оценки показателей качества на этапах ЖЦ и конечного продукта. Инженерия качества процессов – это регламентация процессов, методики и процедуры оперативного контроля промежуточных продуктов на выполнение требований к качеству и снабжения их характеристиками, предусмотренными в этих требованиях.

Для организации, которая занимается разработкой ПС, инженерия качества ПС предусматривает систему качества и группу обеспечения качества.

Система качества (Quality systems) [9–11] – это набор организационных структур, методик, мероприятий, процессов и ресурсов для организации управления качеством. Система качества ставит своей целью обеспечить:

1) требуемый уровень качества ПС, при котором оценка качества проводится после испытания ПС и при предположении, что чем больше обнаружено и устранено ошибок в продукте при испытаниях, тем выше становится его качество;

2) качество процессов ЖЦ, которое зависит от методик и процедур, выполняемых задач на этих процессах, структуры промежуточных рабочих продуктов на каждом из них и способов их контроля для определения достижения качественных показателей.

При ориентации разработки на качество продукта возможны следующие недостатки:

– устранение ошибок в готовом продукте на этапе испытаний обходится в десятки раз дороже, чем их устранение на ранних процессах ЖЦ;

– отсутствие методов и средств испытания ПС с гарантированным выявлением ошибок и разного рода отказов в испытываемых программах.

В случае п.2 предусматриваются меры по предотвращению, оперативному выявлению и устранению недостатков в процессах ЖЦ. Создается план усовершенствования процессов и разработки таких процедур, которые способствуют обеспечению качества ПС. Процессный подход отражен в серии международных стандартов ISO 9000 и 9000-1,2,3 и является эталоном для разработки и ввода в действие внутренних стандартов, учитывающих особенности процессов для достижения качества разных типов ПС.

Соблюдение стандартных положений при выпуске программной продукции способствует значительному повышению качества и соответственно его конкурентоспособности. В задачи системы качества входит мониторинг спроса выпускаемого вида продукции, контроль всех процессов производства ПС и реклама продукции потенциальным покупателям. При отсутствии службы качества контроль продукции проводят сами разработчики по созданным ими нормативным и методическим документам в организации.

В эталонной модели процессов ЖЦ имеются процессы, которые непосредственно направлены на повышение качества ПС, интегрируются с традиционными процессами разработки – верификации и контроля (гарантирования) качества, измерения и оценивания, а также управление проектом, качеством, риском и конфигурацией. Их внедрение в процесс разработки ПС требует методической и программно-технологической поддержки в целях закрепления соответствующих проверочных действий за процессами без их выполнения.

Принципы организации системы качества зависят от вида деятельности предприятия, типа ПО, а также степени использования передовых методов программирования и инструментальных средств их поддержки.

Стандарт ISO/IEC 15504–98 [23] регламентирует процесс реорганизации стандартного процесса разработка ПС в направлении оценивания полноты процессов и их усовершенствования. Оценка промежуточных результатов процесса дает возможность принять решение относительно того, действительно ли реализованы действия на процессах обеспечения характеристик качества в соответствии с заданными требованиями.

Процесс оценивания продуктов в этом стандарте состоит в том, чтобы гарантировать, что продукт удовлетворяет установленным требованиям заказчика и пользователей. Стандарт предусматривает прогнозирование, контроль и управление качеством

конечного продукта, а также подход для принятия решений о возможности приема этого продукта в эксплуатацию и формулировки выводов о положительных и отрицательных результатах его создания [15]. Важное место отводится измерению характеристик процессов ЖЦ, ресурсов и процедур создания на нем рабочих продуктов.

На процессах ЖЦ проводится анализ достижения отдельных атрибутов характеристик качества ПС посредством:

– постепенной поэтапной проверки этих атрибутов с учетом заданных требований заказчика и критериев работоспособности системы;

– обеспечения завершенности системы, как надежно работающей без дефектов и отказов;

– верификации и аттестации (валидации) промежуточных продуктов на процессах ЖЦ и выходного продукта в целях проверки правильности реализации требований и функций продукта.

Инженерия качества включает в себя оценивание качества ПС в целях определения степени удовлетворения системы потребностям и назначению, которое состоит:

– в принятии решения о степени соответствия достигнутого уровня качества разработанной ПС в соответствии с заданными требованиями;

– в аттестации и сертификации программного продукта.

9.2.3. Метрики качества ПС и их измерение

Характеристики качества ПС описываются в функциональных и нефункциональных требованиях к ПС и определяют выполнение функций системы в среде функционирования. Такие характеристики как производительность, результативность, безопасность и т.п. также относятся к характеристикам качества ПС, но проявляются при ее эксплуатации и называются характеристиками эксплуатационного качества.

При определении требований к качеству указываются внешние характеристики качества, отображающие функциональность и надежность программного продукта. Для количественного определения критериев качества, по которым будет осуществляться проверка и подтверждение соответствия ПС требованиям, метрики, специфицируют соответствующие внешние, внутренние характеристики и атрибуты ПС.

Внутренние характеристики используются для планирования достижения необходимых внешних характеристик качества ко-

нечного программного продукта и используются при оценке промежуточных (рабочих) продуктов ПС в процессе разработки на процессах ЖЦ.

В общем внешние и внутренние характеристики качества, которые описаны в требованиях к ПС, касаются свойств будущего продукта и отображают взгляд заказчика и разработчика на функции, назначение и выполнение ПС в заданной среде. Конечный пользователь ожидает достижения максимального эффекта от применения продукта, а именно, эксплуатационного качества. Поэтому для эффективного выполнения требований заказчика к ПС устанавливается взаимосвязь мер и соответствующих метрик внутреннего, внешнего и эксплуатационного качества ПС, так, чтобы объединять взгляды на качество как разработчиков системы, так и непосредственных заказчиков и пользователей.

Характеристики качества ПС измеряются с помощью метрик качества. Согласно стандарту ISO/IEC 9126-2 метрики качества задают «модель измерения» атрибутов, которые связаны с одной из характеристик качества, и представляют комбинацию конкретного метода измерения атрибута и шкалы значений атрибутов [15, 18, 22]. Метрики также делятся на внутренние, внешние и эксплуатационные [4, 29–31].

Внутренние метрики предназначены для оценивания свойств продукта, которые определяет его способность удовлетворять заданным требованиям к качеству при использовании ПС в определенных условиях. Эти метрики оценивают качество промежуточных и конечных продуктов ПС в соответствии с их свойствами и данными, собранными при анализе и обзоре результатов процессов, без выполнения на компьютере.

Внешние метрики предназначены для измерения работающего на компьютере программного продукта и зависят от его поведения в процессе работы, а именно, от выполнения функций системы без ошибок и дефектов. Они предназначены для оценивания внешнего качества, т.е. меры, которой отвечает продукт согласно установленным требованиям.

Метрики эксплуатационного качества измеряются для того, чтобы установить к какой мере программный продукт, используемый в заданной среде эксплуатации, удовлетворяет потребностям пользователей в плане реализации функций ПС. Они помогают оценить эксплуатационное качество на основе данных, которые получены и собраны в процессе тестирования и эксплуатации ПС.

Метрики продукта также подразделяются на внешние, внутренние и эксплуатационные.

Внешние метрики продукта отображают:

– метрики надежности, которые оцениваются по числу дефектов в продукте;

– функциональные метрики, с помощью которых устанавливается правильность реализации функций в продукте;

– метрики сопровождения, с помощью которых измеряются ресурсы продукта (скорость, память, среда);

– метрики применимости продукта, которые способствуют определению степени доступности его для изучения и использования;

– метрики стоимости, которыми определяется стоимость созданного продукта.

К внутренним метрикам продукта относятся:

– метрики размера для измерения продукта (в операторах, командах и т.п.);

– метрики сложности, необходимые для определения уровня сложности продукта;

– метрики стиля, которые служат для определения подходов и технологий создания отдельных компонентов продукта и его документов.

Метрики продукта описываются комплексом моделей для задания различных свойств и значений показателей модели качества или для прогнозирования. Измерение метрик проводятся, как правило, после их калибровки на ранних этапах проекта включают в себя:

– число требований;

– число сценариев и действующих лиц;

– число объектов сценария и требования к каждому сценарию;

– число параметров и операций объекта и др.

Стандарт ISO/IEC 9126-2 определяет меры:

– размера ПС в разных единицах измерения (число функций, строк в программе, размер дисковой памяти и др.);

– времени функционирования системы, выполнения компонента и др.;

– усилий, затраченных на выполнение операций реализации продукта и др.;

– количества ошибок, числа отказов, ответов системы и др.

При оценках качества часто используются средне статистические метрики (например, среднее число операций в классе объектов, среднее число наследников класса или операций класса и др.).

Как правило, меры являются субъективными и зависят от знаний экспертов, производящих оценки качественных атрибутов компонентов программного продукта.

В качестве примера можно привести метрики Холстеда, выявляемые на основе статической структуры программы на конкретном языке программирования: число вхождений операндов и операторов; длина описания программы как суммы вхождений всех операндов и операторов и др. С их помощью вычисляется время программирования и качество ПС.

При компонентном подходе к разработке ПС определены следующие меры:

– размер компонентов и системы (количество операторов, компонентов и др.);

– трудоемкость разработки компонента и системы в целом;

– время выполнения системы (по каждому компоненту отдельно, по передаче данных и др.);

– количество ошибок и отказов в компонентах, попыток исправления обнаруженных ошибок и др.

Специальной мерой может быть уровень использования повторных компонентов, измеряемый отношением размера продукта из готовых компонентов к размеру системы в целом. При сопоставлении двух компонентов, реализующих одну прикладную задачу, предпочтение отдается более короткому (по числу строк) компоненту, так как его создает более квалифицированный программист, в нем меньше скрытых ошибок и его легче модифицировать. По стоимости он дороже, хотя времени на отладку и модификацию уходит больше. Иными словами, длину программы можно использовать в качестве вспомогательного свойства при сравнении программных компонентов с учетом одинаковой квалификации разработчиков, единого стиля разработки и общей среды.

Метрики процессов включают в себя:

– метрики затрат на процессах ЖЦ с учетом оригинальности, средств поддержки, документации разработки и др.;

– метрики надежной работы процесса при проектировании и учете метрик, приводящих к сбоям инструментальных средств и среды.

К метрике процесса относятся:

– общее время разработки и отдельно время каждого процесса;

– время модификации моделей процесса;

– время контроля работ на процессе;

– стоимость проверки качества процессов разработки ПС.

На этапе сопровождения метрики служат для измерения степени удовлетворения потребностей пользователя при решении

его задач. Они помогают оценить не свойства самой программы, а эксплуатационное качество. Примером может служить точность и полнота реализованных задач пользователя, а также ресурсы (эффективность, производительность и др.), потраченные на решение этих задач.

9.3. ОБОБЩЕННАЯ МОДЕЛЬ КАЧЕСТВА ПС

Качество ПС – это относительное понятие, которое имеет смысл только при учете реальных условий его применения, в зависимости от предъявляемых требований и среды выполнения ПС.

Для класса ПС выбираются наиболее важные характеристики качества и их приоритеты, которые отражаются в требованиях на разработку системы. Если приоритет характеристик качества не указан, используется эталон, к которому данный тип системы относится.

Оценка показателей качества проводиться после испытания готового программного продукта или проверки его создания на процессах ЖЦ путем оперативного выявления и устранения ошибок в ПС с помощью процедур и мероприятий стандартов ISO/IE–12207:2002 и ГОСТ 3918–1999 [23] под наблюдением службы качества.

Для оценки качества программного продукта используется стандартная модель качества [11], которая для всех видов и типов ПС имеет такой общий вид:

$$M_{кач} = \{Q, A, M, W\}, \qquad (9.1)$$

где $Q = \{q_1, q_2, ..., q_i\}$ $i = 1,..., 6$, – множество характеристик качества *(Quality – Q)*; $A = \{a_1, a_2,..., a_j\}$ $j = 1,..., J$, – множество атрибутов (*Attributes – A*), каждый из которых фиксирует отдельное свойство q_i – характеристики качества; $M = \{m_1, m_2,..., m_k\}$ $k = 1,..., K$, – множество метрик (*Metrics – M*) каждого элемента a_j атрибута для проведения измерения этого атрибута; $W = \{w_1, w_2,..., w_n\}$ $n = 1,..., N$, – множество весовых коэффициентов (*Weights – W*) для метрик множества *M*.

Эта модель – четырехуровневая, иерархическая. На первом уровне находятся характеристики, каждая из которых определяется набором атрибутов – свойств второго уровня, детализирует разные аспекты этой характеристики в готовом продукте (например, для надежности – свойство безотказной работы, восстанавливаемость и др.).

На третьем уровне находятся метрики атрибутов каждой характеристики. При оценке конкретного атрибута заданной характеристики на процессах ЖЦ (при экспертизе документации, программ и результатов тестирования, т.е. количество ошибок, отказов и др.) метрика уточняется с помощью весовой меры (весового коэффициента) для получения конечного количественного значения этого атрибута.

Общая формальная модель (9.1) используется для выбора конкретной модели создаваемой ПС, в которую включаются необходимые характеристики (не менее одной) из множества Q, задаваемые заказчиком в требованиях к системе. Эти характеристики определяют соответствующие подмножества из представителей элементов множеств A, M, W. По конкретной модели проводится оценивание атрибутов компонентов в соответствии со значениями метрик и их весовых коэффициентов, отражающих атрибуты данной системы. При измерении значений атрибутов, входящих в эту модель, используются экспертные и аналитические методы их оценки.

Оценка качества ПС, состоящей из компонентов, согласно модели качества (9.1), начинается с нижнего уровня, т.е. с атрибутов множества A для каждой характеристики качества в соответствии с установленными для них мерами. На этапе проектирования компонентов определяются значения весовых мер для каждого атрибута, включенного в требования к ним. Сначала измеряется значение атрибута для отдельного компонента, а затем для их совокупности.

9.3.1. Прикладные методы оценки характеристик качества ПС

В стандартах [9–14] и в ядре знаний SWEBOK [1, 17, 20, 25] определено шесть базовых характеристик качества ПО:

q_1: функциональность (functionality),

q_2: надежность (realibility),

q_3: удобство применения (usability),

q_4: эффективность (efficiency),

q_5: сопровождаемость (maitainnability),

q_6: переносимость (portability).

Каждая из характеристик (показателей) определяется свойствами (табл. 9.1), имеет атрибуты, определяющие разные сторо-

ны ее представления, и ориентированные на пользователя и среду функционирования ПО (рис. 9.1).

Далее излагаются характеристики качества модели $M_{\text{кач}}$, соответствующие им атрибуты и их оценки.

ТАБЛИЦА 9.1. Свойства характеристик качества ПО

Характеристика	**Определение свойств характеристик ПО**
Функциональность (functionality)	Группа свойств ПО, определяющая его способность выполнять функции в соответствии с назначением в заданных условиях
Надежность (realibility)	Группа свойств, обусловливающая способность ПО сохранять работоспособность преобразования исходных данных в результаты за установленный период времени в условиях его применения
Удобство применения (usability)	Совокупность свойств ПО отражать легкость его освоения и адаптации к изменяющимся условиям эксплуатации, понимаемость результатов, удобства внесения изменений в программную документацию и в программы
Сопровождаемость (maitainnability)	Группа свойств, определяющая усилия, необходимые для выполнения и способность к внесению изменений и усовершенствованию в связи с изменениям среды функционирования ПО
Эффективность (efficiency)	Группа свойств, характеризующаяся степенью соответствия используемых ресурсов среды функционирования уровню качества (надежности) ПО при заданных условиях применения
Переносимость (portability)	Группа свойств ПО, обеспечивающих его перенос из одной среды функционирования в другую путем адаптации.

Функциональность – совокупность свойств, определяющих способность системы предоставлять требуемое множество функций для решения задач в соответствии с требованиями. В модели качества эта характеристика задается набором атрибутов $q_1 = \{a_{11}, a_{12}, a_{13}, a_{14}, a_{15}, a_{16}\}$, семантика и оценка которых приведена ниже.

a_{11}: функциональная полнота – свойство компонента, которое показывает степень достаточности реализованных в нем функций для решения задач в соответствии с его назначением. Данный атрибут можно представить в виде отношения всех реализованных функций F^c в компонентной системе (c), к функциям F^m, заданным в требованиях (m):

$$a_{11} = \sum_{i=1}^{N} F^c \Big/ \sum_{j=1}^{K} F^m ;$$

a_{12} : корректность – атрибут, который указывает на степень соответствия каждой функции F^m, заданной в требовании, и каждой функции F^c, реализованной в компонентной системе. При этом система обладает свойством полной корректности, если $F^m = F^c$, и частичной корректности, если $F^m \subset F^c$. Для большинства систем достаточно частичной корректности. Степень корректности компонента можно определить, как степень функциональной корректности, по формуле: $\sqrt{} = 1 - (\text{card}\ (F^m / F^c)\ / \text{card}\ F^m$.

a_{13}: точность – свойство, определяющее получение системой согласованных результатов с необходимой степенью точности. Данный атрибут может оцениваться отношением ∇ разности значения функции $F^c_i\ (D_i)$ компонента и значения функции $F^m_i\ (D_i)$, заданной требованиями на D_i входном наборе к значению функции в соответствии с выражением:

$$\nabla = \sum_{i=1}^{card\ F^m} \left(F_i^c(D_i) - F_i^m(D_i)\right) / \left(F_i^m(D_i)\right) / card\ F^m ;$$

a_{14} : интероперабельность – свойство компонента системы взаимодействовать с другими компонентами и операционной средой;

a_{15} : защищенность – атрибут, который показывает возможность компонента (системы) фиксировать дефекты, которые являются следствием субъективных ошибок в его реализации или вызваны программными или аппаратными средствами при выполнении, а также ошибок, связанных с данными. Оценку степени защищенности можно привести с помощью выражения $a_{15} = fal^k / fal$, где fal^k – количество дефектов, от которых компонент защищен; fal – общее количество дефектов в компонентах или ПС;

a_{16}: согласованность – атрибут, который показывает степень соблюдения стандартов, правил и других соглашений процесса разработки, и оценивается экспертно. Затем полученные качественные оценки нивелируются соответствующими весовыми коэффициентами.

Таким образом, характеристика функциональность q_1 вычисляется суммированием ее атрибутов с учетом метрик и их весовых коэффициентов:

$$q_1 = \sum_{j=1}^{6} a_{1j} m_{1j} w_{1j} . \qquad (9.2)$$

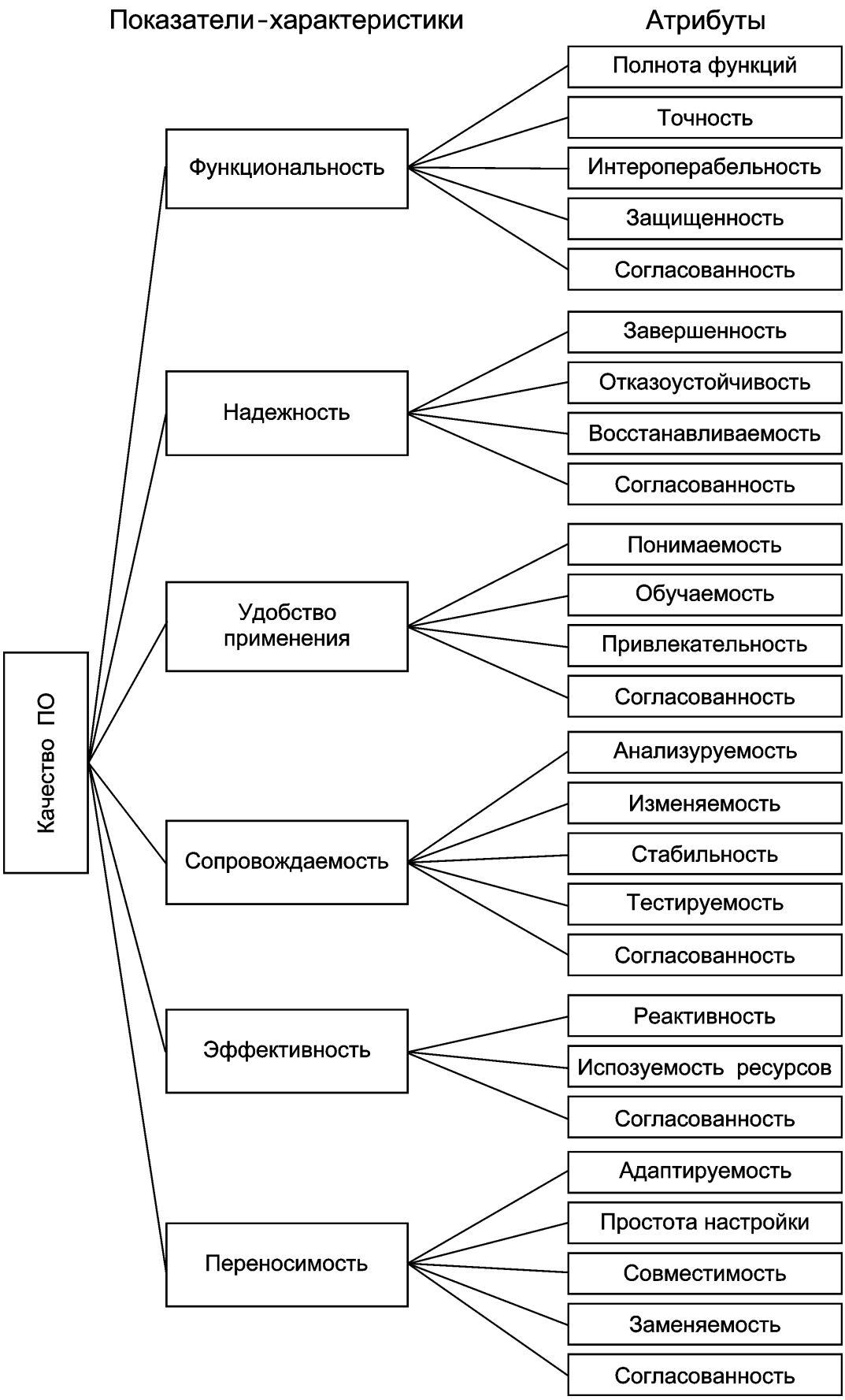

РИС. 9.1. Модель характеристик и атрибутов качества ПО

Надежность ПО будем определять как вероятность того, что компоненты системы или сама система функционируют безотказно в течение фиксированного периода времени в заданных условиях операционного окружения/среды. В модели качества надежность задается на множестве атрибутов $q_2 = \{a_{21}, a_{22}, a_{23}, a_{24}\}$, которые определяют способность системы преобразовывать исходные данные в результаты при условиях, зависящих от периода времени жизни системы (износ и старение не учитываются).

Снижение надежности компонентов происходит из-за ошибок в требованиях, проектировании и выполнении. Отказы и ошибки в программных компонентах могут появляться на заданном промежутке времени функционирования компонента/системы [26−30]. Рассмотрим свойства каждого атрибута надежности.

a_{21}: безотказность − свойство системы, которое определяет функционирование системы без отказов (программных компонентов или оборудования). Если компонент содержит дефект, вызванный субъективными ошибками при разработке, то во множестве $D=\{De/e \in L\}$ всех дефектов, можно выделить подмножество $E \subseteq D$, для которого результаты не соответствуют функции F^m, заданной в требованиях на разработку. Вероятность p безотказного выполнения компонента на De, случайно выбранном из D среди равновероятных, равна:

$$p = 1 - (card\ (E)\ /\ card\ (D)).$$

Отказ (failure) показывает отклонение поведения системы от предписанного и система перестает выполнять предписанные ей функции. Кроме того, появление отказа может быть причиной ошибки (fault/ error), вызывающей его.

Если ошибка сделана человеком, то используется термин mistake. Когда различие между fault и failure не является критическим, используется термин defect, который означает либо fault (причина), либо failure (действие). Связь между этими понятиями можно представить так: fault → error → failure.

Существует большое разнообразие видов отказов ПО, типичные из них: внезапные, постепенные, перемещающиеся (сбои). Причины отказов могут быть физические, структурные, отказы взаимодействия и др. Они могут возникать естественным путём, вноситься человеком или внешней операционной средой в период создания или эксплуатации системы, а также быть постоянными или носить временный характер.

Наработка на отказ как атрибут надежности определяет среднее время между появлением угроз, нарушающих безопасность, и

обеспечивает трудно измеримую оценку ущерба, которая наносится соответствующими угрозами.

Для вычисления среднего времени T наработки на отказ применяется формула

$$T = \sum_{i=1}^{De} \nabla t_i^E / N ,$$

где ∇t_i^E – интервал времени безотказной работы компонента i-го отказа; N – количество отказов в системе.

a_{22}: устойчивость к ошибкам – свойство компонентов системы, которое показывает на способность программной системы выполнять функции при аномальных условиях (сбоях аппаратуры, ошибках в данных и интерфейсах, нарушениях в действиях оператора и др.). Оценку устойчивости можно получить по формуле

$$\Upsilon = N^v/N,$$

где N^v – количество разных типов отказов, для которых предусмотрены средства восстановления; N – общее количество всех отказов в системе.

a_{23}: восстанавливаемость – свойство системы, которое указывает на способность возобновлять функционирование системы после отказов и восстанавливать в ней поврежденные компоненты и/или данные для повторного исполнения. Среднее время восстановления компонента можно определить по формуле

$$T = \sum_{i=1}^{De} \nabla t_i^b / D ,$$

где ∇t_i^b – время восстановления работоспособности компонента после i-го отказа; De – количество дефектов и отказов в системе.

a_{24}: согласованность – атрибут, который отражает степень соблюдения стандартов, технологии, правил и других соглашений на стадиях разработки и тестирования системы для поиска разного рода ошибок разработки. Этот атрибут оценивается экспертами, они фиксируют ошибки в специальных картах и дают экспертную оценку надежности системы.

Некоторые типы систем реального времени, безопасности и другие требуют высокой надежности (недопустимость ошибок, точность, достоверность и др.), которая в значительной степени зависит от количества оставшихся и не устраненных ошибок в процессе ее разработки на этапах жизненного цикла. В ходе эксплуатации ошибки также могут обнаруживаться и устраняться.

Если при их исправлении не вносятся новые либо вносится их меньше, чем устраняется, то в ходе эксплуатации надежность системы непрерывно возрастает. Чем интенсивнее проводится эксплуатация, тем интенсивнее выявляются ошибки и быстрее растет надежность. К факторам, влияющим на надежность ПО, относятся:

– угроза нарушения безопасности системы;

– совокупность угроз, приводящих к нарушению системы или среды и др.

Надежность постоянно изучается и развивается. Новое ее направление – инженерия надежности ПО (Software reliability engineering – SRE), которая ориентирована на решение следующих задач [20, 28]:

1) измерение надежности, т.е. проведение ее количественной оценки с помощью предсказаний, сбора данных о поведении системы в процессе эксплуатации и современных моделей надежности;

2) разработка стратегии, метрик конструирования готовых компонентов и компонентной системы в целом, а также определение среды функционирования, обеспечивающей надежность работы системы;

3) применение современных методов инспектирования, верификации, валидации и тестировании ПС в процессе разработки, а также при эксплуатации.

Практически оценка надежности ПО является трудоемким процессом, в нем важное место занимает метод определения устойчивости системы к отказам ПО, т.е. вероятности того, что система восстановится самопроизвольно в некоторой точке после возникновения в ней отказа (fault).

Количественная оценка характеристики надежности системы по всем ее рассмотренным количественным атрибутам и соответствующим метрикам имеет вид

$$q_2 = \sum_{j=1}^{4} a_{2j} m_{2j} w_{2j} . \qquad (9.3)$$

Удобство применения – это множество свойств программной системы, которые показывают на необходимые и пригодные условия ее использования (в диалоге или без) кругом пользователей для получение результатов выполнения. Эта характеристика в модели качества определяется на множестве атрибутов, которые удовлетворяют эргономичности, и включают в себя атрибуты:

a_{31}: понимаемость означает усилие, затрачиваемое на распознавание логических концепций и условий применения ПС;

a_{32}: изучаемость означает усилия пользователей при определении применимости ПО посредством операционного контроля ввода, вывода, диагностики, а также процедур анализа документации;

a_{33}: оперативность – реакция системы при выполнении операторов и операционного контроля;

a_{34}: согласованность – соответствие ПС требованиям стандартов, соглашений, правил, законов и предписаний.

Все атрибуты оцениваются экспертами, которые в зависимости от их уровня знаний, дают соответствующие качественные значения.

Количественная оценка данной характеристики зависит от оценок экспертов, количественного атрибута a_{33} и имеет вид:

$$q_3 = \sum_{j=1}^{4} a_{3j} m_{3j} w_{3j} . \qquad (9.4)$$

Эффективность – множество атрибутов – свойств системы, которые показывают взаимосвязь между уровнем ее выполнения, количеством используемых ресурсов (аппаратуры, расходных материалов и др.), услуг штатного обслуживающего персонала и др.

К ним относятся:

a_{41}: реактивность – время отклика, обработки и выполнения функций компонента/системы;

a_{42}: эффективность – количество используемых ресурсов при выполнении функций ПС и продолжительность их вычислений;

a_{43}: согласованность – соответствие данного атрибута заданным стандартам, правилам и предписаниям.

Количественная оценка данной характеристики по всем рассмотренным качественным и количественно измеряемым атрибутам оценивается экспертно и аналитически с учетом соответствующих метрик и имеет вид

$$q_4 = \sum_{j=1}^{3} a_{4j} m_{4j} w_{4j} . \qquad (9.5)$$

Сопровождаемость – множество свойств, которые отражают усилия, затрачиваемые на проведение модификаций (корректировка, усовершенствование и адаптация) при изменении среды

выполнения, требований или спецификаций. Данная характеристика в модели качества состоит из следующих атрибутов:

a_{52} : анализируемость – необходимые усилия для диагностики отказов в ПС или идентификации частей, которые будут модифицироваться;

a_{53} : изменяемость – усилия, которые затрачиваются на модификацию компонента, удаление ошибок или внесение изменений, дополнение новых возможностей в систему или среду функционирования;

a_{54} : стабильность – риск проведения модификации компонента /системы;

a_{53} : тестируемость – усилия при проведении верификации в целях обнаружения ошибок и несоответствий требованиям (валидация), а также на необходимость исправления обнаруженных ошибок и проведения сертификации системы;

a_{54} : согласованность – соответствие данного атрибута определенным стандартам, соглашениям, правилам и предписаниям.

Количественная оценка данной характеристики по всем ее атрибутам, измеряемым экспертно и аналитически с учетом соответствующих метрик, имеет вид

$$q_5 = \sum_{j=1}^{3} a_{5j} m_{5j} w_{5j} . \qquad (9.6)$$

Переносимость – множество атрибутов, указывающих на возможность компонентов системы приспосабливаться к работе в новых условиях среды выполнения: организационной, аппаратной и программной. Перенос компонентов или всей системы на другую платформу или среду связан с совокупностью действий, направленных на обеспечение возможности функционирования в новой среде, отличной от той, в которой система создавалось. К атрибутам данной характеристики согласно модели качества относятся:

a_{61} *:* адаптивность – усилия, затрачиваемые на адаптацию системы к различным операционным средам. Этот атрибут можно представить в виде $a_{61} = Za / Zd$, где Za – затраты на адаптацию к новой операционной среде; Zd – затраты на разработку новой системы для новой операционной среды;

a_{62} : настраиваемость определяет необходимые усилия для запуска или инсталляции программного продукта в другой среде;

a_{63} : сосуществование – возможность использования специального ПО в среде действующей системы;

a_{64} : заменяемость – возможность взаимодействия (интероперабельности) с другими программами при совместной их работе и инсталляции или адаптации системы;

a_{65} : согласованность – соответствие стандартам или соглашениям и правилам переноса программной системы в другую среду.

Количественная оценка данной характеристики по атрибутам, измеряемым экспертным и аналитическим путем с учетом соответствующих метрик, имеет вид

$$q_6 = \sum_{j=1}^{5} a_{6j} m_{6j} w_{6j} . \qquad (9.7)$$

Комплексная оценка. На основе измеренных количественных характеристик и проведения экспертизы качественных показателей с применением весовых коэффициентов, нивелирующих разные показатели, вычисляется итоговая оценка качества продукта путем суммирования результатов по отдельным показателям и сравнения их с эталонными показателями ПС.

Если в требованиях к ПС было установлено несколько показателей, то просчитанный каждый показатель умножается на соответствующий весовой коэффициент, а затем все суммируется по всем показателям. В результате получается интегральная оценка уровня качества ПС.

В формулах (9.2)–(9.7) приведены оценки показателей качества отдельного компонента с использованием метрик $a_i \subset A$ и весовых коэффициентов $wi \subset W$ для каждого атрибута. Полученные значения q_j с помощью весовых коэффициентов $wi \subset W$ приведены к единой системе измерения. Используя полученные оценки q_j характеристик качества применительно к отдельному компоненту (com), получаем интегральную оценку качества одного компонента в виде

$$Q_{com} = \sum_{j=1}^{6} q_j .$$

Если программная система содержит N-компонентов и для них проведена количественная оценка, то имеем окончательную комплексную оценку качества системы (sys):

$$Q_{sys} = \sum_{l=1}^{N} Q_{com}^{l} .$$

Таким образом, данный подход к аналитической оценке атрибутов показателей качества программных компонентных систем позволяет получить количественную оценку качества ПС.

9.3.2. Стандартные методы оценки качества и его уровня

Стандарты [11–14] рекомендуют следующие типы мер:

– меры размера в разных единицах измерения (количество функций, размер программы, объем ресурсов и др.);

– меры времени – периоды реального, процессорного или календарного времени функционирования системы;

– меры времени, затраченного на реализацию проекта (производительность труда отдельных участников проекта, коллективная трудоемкость и др.);

– меры интервалов времени между последовательными отказами;

– счетчики для определения количества обнаруженных ошибок, числа несовместимых элементов, числа изменений и др.

Проведение оценки отдельного показателя качества стандарт рекомендует с помощью оценочных элементов: весового коэффициента *k*-метрика, *j*-показателя, *i*-атрибута. Например, если в качестве *j*-показателя взять переносимость, то он будет вычисляться по пяти атрибутам ($i = 1,...,5$), причем каждый из них будет умножаться на соответствующий коэффициент k_i.

Все метрики *j*-атрибута суммируются и образуют *j*-показатель качества. Когда все атрибуты оценены по каждому из показателей качества, производится суммарная оценка отдельного показателя и интегральная оценка качества с учетом весовых коэффициентов всех показателей ПС.

В рассматриваемом стандарте представлены следующие методы измерений:

– измерительный,

– регистрационный,

– расчетный,

– экспертный.

Допускается комбинации этих методов.

Измерительный метод базируется на использовании измерительных и специальных программных средств для получения информации о характеристиках ПС, например, определение объема, числа строк кода, операторов, количества ветвей, число точек входа (выхода), реактивность и др.

Регистрационный метод используется при подсчете времени, числа сбоев или отказов, начала и конца работы ПС в процессе выполнения.

Расчетный метод базируется на статистических данных, собранных при проведении испытаний, эксплуатации и сопровождении ПС. Расчетными методами оцениваются показатели надежности, точности, устойчивости, реактивности и др.

Экспертный метод осуществляется группой экспертов, компетентных в решении данной задачи или типа ПС. Их оценка базируются на опыте и интуиции, а не на непосредственных результатах расчетов или экспериментов. Этот метод проводится просмотром программ, кодов, сопроводительных документов и обеспечивает качественную оценку продукта. Для этого устанавливаются контролируемые признаки, коррелируемые с одним или несколькими показателями качества и включаемые в опросные карты экспертов. Метод применяется при оценке таких показателей, как анализируемость, документируемость и др.

Для оценки значений показателей качества в зависимости от особенностей используемых ими свойств, назначения, способов их определения стандарт рекомендует следующие виды оценок:

– метрическая (1.1 – абсолютная, 1.2 – относительная, 1.3 – интегральная);

– порядковая (ранговая), позволяющая ранжировать характеристики путем сравнения с опорными;

– классификационная, характеризующая наличие или отсутствие рассматриваемого свойства у оцениваемого ПО.

Показатели, вычисляемые с помощью приведенных видов оценок, являются количественными, а с помощью порядковых и классификационных – качественными.

Для правильного использования результатов измерений каждая мера идентифицируется специальной шкалой измерения, которая в стандарте ISO/IES 9126-2 имеет 5 видов измерений, упорядоченных от менее строгой к более строгой:

– *номинальная шкала* отражает категории свойств оцениваемого объекта без их упорядочения;

– *порядковая шкала* упорядочивает характеристики по возрастанию или убыванию путем сравнения их с базовыми значениями;

– *интервальная шкала* задает существенные свойства объекта (например, календарная дата);

– *относительная шкала* задает некоторое значение относительно выбранной единицы;

– *абсолютная шкала* указывает на фактическое значение величины (например, число ошибок в программе равно 10).

Оценка уровня качества ПО. Для оценки уровня качества ПО стандарт [6] рекомендует или дифференциальный, или комплексный, или смешанный метод.

Дифференциальным называется метод, основанный на использовании единичных показателей (атрибутов) качества, относительные значения отдельных показателей вычисляются по формулам

$$Q_i = P_i / P_{ib}, \tag{9.9}$$

$$Q_i = P_{ib} / P_i, \; i = 1, 2, ..., N, \tag{9.10}$$

где P_i – значение i-го показателя качества оцениваемого ПО; P_{ib} – базовое значение i-го показателя; N – количество показателей качества для данного ПО.

Формула (9.9) используется в случае соблюдения условия «лучше, если значение показателя больше». В противном случае применяется формула (9.10).

Для дифференциального метода оценки уровня качества оцениваемая программная продукция считается высшей или равной уровню базовых значений, если все значения относительных показателей большие или равны единице. В противном случае уровень качества оцениваемой программной продукции ниже уровня базовых значений. Если часть показателей $q_i > 1$, а часть значений $q_i < 1$, то следует применять комплексный или смешанный метод оценки.

Комплексный метод оценки уровня качества программной продукции основан на применении обобщенного показателя, выступающего в качестве функции от нескольких количественных (групповых) показателей и коэффициентов их весомости.

Установление коэффициентов весомости показателя качества, как правило, проводят экспертным путем, оно включает в себя шкалы:

5 – чересчур важно, чтобы данный показатель имел высокое значение;

4 – важно, чтобы данный показатель имел высокое значение;

3 – хорошо бы иметь высокое значение данного показателя;

2 – показатель имеет среднее значение;

1 – при низких значениях данного показателя ощутимых потерь нет.

В этом случае обобщенное значение коэффициента весомости m (параметр весомости M) вычисляется по формуле

$$m_i = M_i \Big/ \sum_{l=1}^{N} M_i ,$$

где M_i – значение i-го коэффициента весомости в упомянутых (или других) шкалах.

Значение параметров весомости определяются во время написания технического задания (технических условий) на разрабатываемое ПО. Если к техническому заданию добавляется план обеспечения качества, то значение параметров отмечают в этом плане. Установленные значения параметров весомости пересматриваются только в случае коррекции этих документов.

Если $\sum_{l=1}^{N} M_i = 1$, то коэффициенты весомости называют параметрами весомости.

Обобщенный показатель качества может быть представлен средним взвешенным арифметическим или геометрическим, или иным показателем качества.

Средний взвешенный арифметический показатель вычисляется по формуле

$$U = \sum_{l=1}^{N} Q_i M_i ,$$

где Q_i - относительный i-й показатель качества, который вычисляется по формулам (9.9), (9.10); M_i - параметр весомости i-го показателя, входящего в обобщенный показатель, $i = 1,2,..., N$; N – количество показателей, которые составляют средний взвешенный показатель.

Если к моменту опробования и оценки уровня качества ПО значения параметров весомости не были установлены, то эти значения устанавливаются специалистами, которые проводят оценку качества ПО. Во время повторных опробований, как правило, должны использоваться одни и те же ранее установленные показатели.

При многоуровневой иерархической структуре номенклатура показателей качества увеличивает уровень качества ПО и проводится снизу (от единичных показателей) верх (для получения обобщенного показателя качества).

Смешанный метод оценки уровня качества основывается на совместимом применении единичных и комплексных (групповых) показателей. При этом часть единичных показателей объединяется

в группы. После такого объединения вычисляются относительные значения групповых показателей по формулам (9.9) и (9.10).

Смешанный метод применяется в таких случаях:

– совокупность единичных показателей качества является достаточно широкой и затрудняет получение обобщенных выводов;

– обобщенный показатель качества в комплексном методе не позволяет получить выводы о соответствии значений некоторых существенных групповых показателей нужному уровню.

Сравнение уровня качества оцениваемого ПО с базовыми показателями качества проводятся при смешанном и дифференциальном методах.

9.4. ПОДХОД К ОБЕСПЕЧЕНИЮ ХАРАКТЕРИСТИКИ ЗАВЕРШЕННОСТИ ПС

Отличительной особенностью подхода для обеспечения показателя качества ПС – завершенность на процессах ЖЦ, является выявление и устранение дефектов в ПС и процессах ЖЦ, а также усовершенствование их качества. Характеристика завершенность ПС определяется на этапах ЖЦ с учетом возникающих угроз и риска, плотности дефектов, а также установления взаимосвязи внутренних и внешних метрик качества.

Данный подход к обеспечению показателей качества представлен в [17–19, 27] и решает три группы задач инженерии качества на ранних этапах ЖЦ:

– обоснование целевых требований к качеству ПС;

– определение стратегии прогнозирования достижимости этих требований на каждом этапе работ по проекту ПС;

– обеспечение измеримости рабочих продуктов, процессов и ресурсов проекта в целях их учета и совершенствования.

Основной смысл характеристики завершенности в модели качества ПС состоит в том, чтобы каждый компонент системы мог избегать отказов при наличии скрытых дефектов.

На начальных этапах ЖЦ разработки ПС определяется область значений внешних метрик как критериев достижения необходимого уровня качества при испытании ПС, после чего определяются наиболее пригодные для значений внешних метрик внутренние метрики и планируется поэтапное достижение внешних требований к качеству [9].

Для завершенности ПС главным показателем внутреннего качества являются дефекты, для внешнего – отказы, а для экс-

плуатационного – общая оценка работоспособности системы пользователями.

Взаимосвязь мер внутреннего, внешнего и эксплуатационного качества отражается через следующие данные:

D_0 – количество (плотность) дефектов в каждом компоненте ПС (внутренняя мера);

$R(t)$ – безотказность функционирования каждого компонента ПС в течение заданного времени t, т.е. вероятность того, что за время t эксплуатации компонента ПС при определенных условиях не возникнет последовательность входных данных, которая приведет к отказу (внешняя мера);

$Q_{пс}$ – мера эксплуатационного качества ПС, связываемая с безотказным функционированием ПС и характеристикой эксплуатационного качества ПС – удовлетворенность (satisfaction).

Данные метрики показателя надежности ПС уточняют эксплуатационное качество атрибутами безотказной работы, плотностью дефектов и удовлетворенностью.

При этом уточняются и задачи инженерии качества:

1. Определение значения целевого уровня качества как критерия соответствия ПС потребностям пользователей.

2. Прогнозирование достижимости уровня качества при условии устранения «слабых мест» в процессах ЖЦ.

3. Выбор наилучшего решения по достижению уровня качества с учетом ограниченных ресурсов.

4. Пересмотр или уточнение суждений о важности тех или иных факторов успешного достижения установленного уровня качества и коррекция процессов ЖЦ.

9.4.1. Модель требований к завершенности компонентов ПС

В зависимости от класса системы, частоты использования отдельных компонентов в деловых процессах, последствий отказов тех или иных компонентов ПС, а также других факторов (стоимости разработки, цены потерь и т.п.) требования к завершенности отдельных компонентов ПС являются разными.

Для ПС СОД построена модель требований к завершенности компонентов ПС с учетом дифференцированного подхода к ним и необходимости максимизировать уровень общей удовлетворенности заказчика поставленным программным продуктом через достижение целевых значений завершенности компонентов ПС.

Мера эксплуатационного качества ПС (на верхнем уровне модели качества) определена как функция полезности вида [18, 32, 33]:

$$Q_{nc} = \sum_{i=1}^{k} a_i \cdot R_i ,$$

где a_i – мера важности i-й функции ПС, R_i – безотказность выполнения этой функции в заданном периоде t эксплуатации системы.

Безотказность выполнения функций ПС оценивается во время системного тестирования и эксплуатации ПС, что с позиций управления качеством является запоздалым процессом. Поэтому проблема нахождения оптимального уровня завершенности компонентов ПС декомпозируется по четырехуровневой иерархии системы:

ПС → функции ПС → приложение → программные модули.

Данная иерархия обеспечивает определение параметров модели требований с учетом важности безотказной работы каждого компонента ПС в иерархической структуре.

Предполагается, что оценки важности отдельных функций в ПС задают пользователи системы, а оценки важности отдельных программных приложений для выполнения функций ПС – разработчики ПС. Оценивание важности отдельных модулей выполняется с учетом риска отказов модулей в работе системы.

Решена задача нахождения оптимального целевого уровня *завершенности* каждого компонента (модуля) системы, который обеспечивает максимизацию функции полезности Q_{nc} с учетом технических и ресурсных ограничений проекта ПС.

Исходными данными этой задачи являются:

u_i – коэффициент относительного веса функции F_i в достижении эксплуатационного качества Q_{nc}, $i = 1,..., k$;

v_{ij} – коэффициент относительного веса j-го компонента ПС при выполнении i-й функции, $u = 1,..., k; j = 1,..., l;$

w_{js} – коэффициент относительного веса s-го модуля в обеспечении выполнения j-го программного приложения, $s = 1,..., m$; $j = 1,..., l$;

r_s – безотказность модуля M_s в период эксплуатации t;

E_j – множество номеров всех модулей, которые необходимы для выполнения j-го компонента ПС;

α_s – нижняя граница безотказности модуля M_s;

β_s – верхняя граница безотказности модуля M_s;

G – общая цена ПС;

C – себестоимость создания ПС организацией-разработчиком;

c_s – накладные расходы, связанные с разработкой модуля M_s;

d_s – расходы, необходимые для достижения единичного уровня безотказности модуля M_s (с учетом или без учета затрат на тестирование, а также косвенным учетом времени на разработку);

δ – доля прибыли в цене ПС.

Целевые значения безотказности модулей $r_1 ... r_m$, при которых функция полезности Q_{nc} достигает максимума, определяются так:

$$Q_{nc}(r_1,...,r_m) = \sum_{j=1}^{l}(\sum_{i=1}^{k} u_i v_{ij} \cdot \prod_{n \in E_j} r_n) \to \max \qquad (9.11)$$

при следующих ограничениях

$$0 < \alpha_s \le r_s \le \beta_s \le 1,\ s = 1,...,\ m, \qquad (9.12)$$

$$c_s + d_s \cdot r_s \le (1-\delta) \cdot G \cdot \sum_{j=1}^{l}\sum_{i=1}^{k} u_i v_{ij} w_{js}, \qquad (9.13)$$

$$\sum_{s=1}^{m}(c_s + d_s \cdot r_s) \le C. \qquad (9.14)$$

Данная задача является задачей нелинейной оптимизации с линейными ограничениями (9.12)–(9.14) и может быть решена средствами математического пакета MATLAB.

Параметры u_i, v_{ij}, w_{js} ($u = 1,...,\ k$; $j = 1,...,\ l$; $s = 1,...,\ m$) находятся методом анализа иерархий путем парного сравнения и последовательного определения локальных приоритетов компонентов ПС в пределах каждого уровня иерархии относительно компонентов предыдущего (высшего) уровня.

Ограничения (9.12) задают допустимые нижние α_s и верхние β_s границы безотказности модулей, исходя из оценок важности каждого модуля.

Ограничения (9.13) позволяют установить взаимосвязь общих расходов на разработку модуля, с одной стороны, и части цены системы G, которая приходится на этот модуль с учетом его вклада в надежность работы системы, и доли прибыли δ разработчика, с другой стороны. Допускается линейная зависимость между стоимостью модуля и уровнем его безотказности.

Ограничение (9.14) устанавливает взаимосвязь суммарных расходов на разработку всех модулей и себестоимости создания ПС. Расходы на разработку модуля включают в себя накладные (косвенные) и прямые расходы непосредственно на его разработку.

Таким образом, с помощью данной модели устанавливаются требования к завершенности каждого модуля (r_i), а затем и каждого приложения (q_i) (с учетом независимости модулей), т.е. определяются целевые уровни завершенности всех компонентов и приложений ПС, достижение которых контролируется в ходе выполнения проекта на этапах ЖЦ.

Контроль достижимости целевого уровня завершенности q_i, установленного для каждого i-го компонента ПС, базируется на построении прогнозов двух видов: поискового прогноза, назначение которого – определение возможности достичь поставленных целей в существующих условиях выполнения проекта, и нормативного, предназначенного для определения условий, при которых поставленные цели могут быть достигнуты.

Для выполнения прогнозов разработаны:

1) модель раннего прогнозирования безотказности компонентов ПС (внешняя метрика качества ПС как завершенность);

2) модель раннего прогнозирования скрытых дефектов в компонентах ПС (внутренняя метрика качества ПС как завершенность);

3) метод анализа вариантов достижения целевого уровня качества ПС;

4) модель определения оптимального времени тестирования компонентов ПС с учетом риска их отказов.

9.4.2. Раннее прогнозирование безотказности ПС

Раннее прогнозирование безотказности ПС заключается в построении проекции значений измерений безотказности, полученных по внутренним метрикам на определенном этапе ЖЦ, на значения, вычисленные по внешним метрикам на любом следующем этапе и в конце разработки ПС [26, 29–31]. Цель данного прогнозирования – определение методов усовершенствования процессов разработки ПС для того, чтобы обеспечить минимальную плотность дефектов к моменту начала системного тестирования.

Позднее прогнозирование безотказности ПС связано с применением аналитических моделей роста надежности на этапе системного тестирования после сбора определенного количества данных об отказах ПС (см. глава 8). Все известные модели роста надежности ПС фактически охватывают период, когда надежность повышается в результате тестирования и устранения дефектов [26, 33–37].

При прогнозировании безотказности ПС процесс отказов целесообразно моделировать с помощью пуассоновских моделей надежности, в которых условная функция надежности $R(t \mid T)$ определяется по формуле

$$R(t \mid T) = \exp(-(m(T+t) - m(T))),$$

где $R(t \mid T)$ – условная вероятность того, что в течение заданного времени *t* эксплуатации ПС в определенных условиях среды ее функционирования не возникнет отказа, если ПС тестировалось в течение времени *T*; *m(t)* – функция роста надежности, как среднее количество дефектов в ПС, выявленных во время его функционирования в течение времени *t*.

Анализ ряда пуассоновских моделей применительно к раннему прогнозированию безотказности компонентов ПС показывает целесообразность использования экспоненциальной модели Дж. Мусы [27], поскольку она не требует данных об отказах.

Функция роста надежности *m(t)* в этой модели определяется формулой

$$m(t) = N_0\left(1 - \exp\left(-\frac{\lambda_0}{N_0} \cdot t\right)\right),$$

где N_0 – количество скрытых дефектов в ПС в начале системного тестирования; λ_0 - интенсивность отказов ПС, определяемая по формуле $\lambda_0 = N_0 \cdot \frac{\rho \cdot K}{I \cdot \varphi}$, в которой ρ – интенсивность выполнения кода; $K = 4 \cdot 10^{-7}$ – постоянный коэффициент выявления дефектов (в модели Дж. Муссы [33]); I – количество инструкций исходного кода; φ – коэффициент расширения кода (число команд кода на одну инструкцию исходного).

Исходя из этих данных, условная функция надежности эксплуатации ПС имеет вид

$$R(t \mid T) = \exp(-(m(T+t) - m(T))) =$$
$$= \exp\left(-N_0 \exp\left(-\frac{\lambda_0}{N_0} T\right)\left(1 - \exp\left(\frac{\lambda_0}{N_0} t\right)\right)\right). \qquad (9.15)$$

В частности, при $T = 0$ ПС не проходит этап системного тестирования,

$$R(t) = R(t|0) = \exp\left(- D_0 \cdot I \cdot \left(1 - \exp\left(\frac{\lambda_0}{D_0 \cdot I} \cdot t \right) \right) \right), \qquad (9.16)$$

где $D_0 = N_0/I$ – плотность скрытых дефектов в начале тестирования.

Формулы (9.15) и (9.16) – это метрики раннего прогнозирования безотказности компонентов ПС с учетом (или без) времени его тестирования.

Для прогнозирования вероятной плотности дефектов D_0 (или количества дефектов N_0), которую будет иметь ПС в момент начала системного тестирования, могут использоваться существующие мультипликативные модели [26].

9.4.3. Прогнозирование плотности дефектов ПС

В связи с разнообразием методов создания ПС возникает неопределенность относительно влияния одних показателей качества на другие и на качество конечного программного продукта. Поэтому целесообразно применять в инженерии качества механизмы корректировки суждений по факторам качества при накоплении опыта.

Средства для построения логически непротиворечивой схемы суждений с возможностью их пересмотра в свете новых данных предоставляет байесовский подход, который может быть положен в основу не только управления качеством, но и программными проектами в целом. Однако его непосредственное применение для разрешения задач программной инженерии долгое время усложнялось очень большим объемом расчетов условных вероятностей. Фактически, только с появлением байесовских сетей доверия он обрел практический смысл не только в этой, но и в других отраслях знаний [35].

Основным направлением использования байесовских моделей в инженерии качества является прогнозирование дефектов и диагностика наиболее вероятных причин их возникновения, а также простота модификации посредством действующих эффективных алгоритмов и доступных инструментов. Пример модели прогнозирования дефектов показан на рис. 9.2, полученной путем определения множества факторов качества, анализа причинно-следственных связей между ними, комбинирования экспертных и количественных оценок влияния на плотность дефектов [36].

Семантическое описание вершин данной байесовской сети представлено в табл. 9.2.

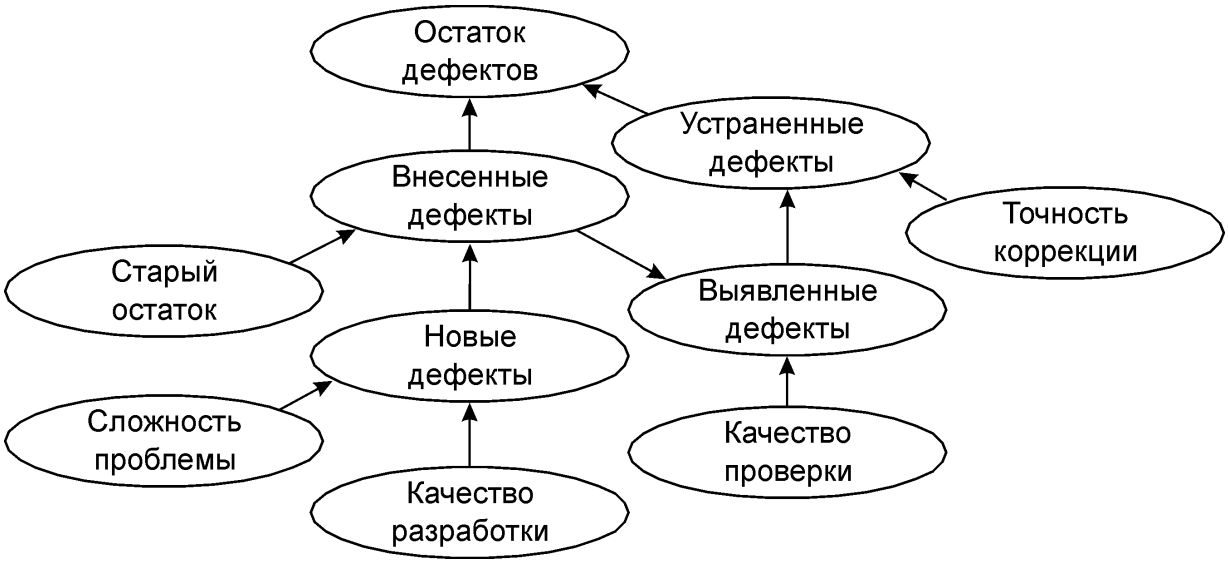

РИС. 9.2. Модель прогнозирования плотности дефектов в ПС

ТАБЛИЦА 9.2. Описание вершин модели прогнозирования плотности дефектов

Название вершины	**Описание вершин сети и зависимостей переменных**
Остаток дефектов	Разница между плотностью внесенных дефектов в текущую версию ПС и плотностью устраненных (откорректированных) дефектов
Внесенные дефекты	Сумма плотности новых внесенных дефектов и плотности остатка дефектов от предыдущего этапа
Внесенные дефекты (новые)	Плотность внесенных новых дефектов, зависит от сложности решаемых задач в ПрО и качества процесса разработки
Остаток старых дефектов	Плотность дефектов, которые, согласно прогнозу, остались не выявленными в ПС на предыдущем этапе разработки (остаток минус фактически устраненные дефекты)
Устраненные дефекты	Плотность устраненных дефектов, зависит от плотности выявленных дефектов в ПС, а также точности их устранения на отрезке [0,1]. Биномиальный закон распределения значений переменной в вершине
Выявленные дефекты	Плотность выявленных дефектов, зависит от плотности внесенных дефектов и качества проверки на отрезке [0,1]. Биномиальный закон распределения значений
Точность коррекции	Способность разработчика точно устранить дефект при коррекции. Чем больше значение, тем выше способность
Качество проверки	Способность контролирующего лица найти дефекты в ПС. Чем больше значение, тем выше способность
Качество разработки	Способность разработчиков предотвратить внесение дефектов при разработке. Чем больше значение, тем выше способность
Сложность проблемы	Ассоциируется с риском проекта в категории «сложность реализации требований» к ПС. Чем выше оценка риска, тем больше сложность проблемы

Модель позволяет в начале каждого текущего этапа разработки проекта прогнозировать (с определенной степенью уверенности) плотность дефектов D_{0i} в i-м компонент ПС, если не изменятся условия его разработки. Прогнозное значение плотности дефектов используется для нахождения вероятного прогнозного значения безотказности $R_i(t)$ ПС.

9.4.4. Достижение целевого уровня завершенности ПС

Анализ достижения целевого уровня завершенности i-го приложения базируется на сравнении прогнозируемого значения безотказности $R_i(t)$ с фактически установленным значением q_i.[18, 19]. При этом для принятия решений по управлению качеством i-го компонента ПС используются такие критерии:

– q_i – установленный целевой уровень безотказности i-го компонента ПС;

– D_i^* – максимальное значение плотности дефектов в ПС, для которого целевой уровень q_i определяется из уравнения

$$q_i = \exp\left(-D_i^* I_i\left(1-\exp\left(\frac{\lambda_{0i}}{D_i^* I_i}t\right)\right)\right);$$

L_0 – минимально допустимая вероятность прогнозного значения плотности дефектов;

– максимально допустимые отклонения: σ_r-прогнозного значения безотказности $R_i(t)$ от целевого значения q_i; σ_d-прогнозного значения плотности дефектов D_{0i} от значения D_i^*;

– σ_d – полученной наибольшей вероятности прогнозного значения плотности дефектов $L(D_{0i})$ от приемлемой L_0.

При предположении, что на ранних стадиях ЖЦ $q_i \geq R_i(t)$ и $D_{0i} \geq D_i^*$, применяются такие варианты последующих действий по обеспечению качества i-го компонента ПС:

1. Если прогнозирование выполняется с приемлемым уровнем уверенности $L(D_{0i}) - L_0 \geq \sigma_p$ в полученном значении, то:

а) при $|q_i - R_i(t)| \leq \sigma_r$, продолжается выполнение проекта,

б) при $|q_i - R_i(t)| > \sigma_r$, принимается решение о времени завершения разработки и трудоемкости системного тестирования, а также о возможности совершенствования процессов ЖЦ ПС.

2. Если при прогнозировании уровень уверенности σ_η $|L(D_{0i}) - L_0| < \sigma_p$, то принимается решение о вычислении $R_i(t)$ в диапазоне близких значений D_{0i}. Затем оценивается относительные отклонения $R_i(t)$ от q_i для каждого из них, учитывая то, что D_{0i} – это плотность дефектов в ПС, размер которого вычислен в условных единицах функциональности, т.е. диапазон значений $R_i(t)$ может быть достаточно широким.

3. Если при прогнозировании уровень уверенности σ_η $|L_0 - L(D_{0i})| > \sigma_p$, то вероятность распределения значения D_{0i} может быть ниже приемлемого уровня уверенности, или на ранних стадиях ЖЦ нарушились предположения, что свидетельствует о несовершенстве применимой модели и необходимости ее уточнения.

Если в ходе выполнения проекта после стабилизации кода i-го компонента ПС (при приближении разработки к завершению) не наблюдается тенденция к снижению плотности дефектов (по результатам прогнозирований в контрольных точках проекта t_s, t_{s+1}..., получаются значения D_{0i}, для которых $D_{0i} - D_i^* > \sigma_d$), то это свидетельствует о несовершенстве процессов разработки. Одно из возможных решений – передача ПС в группу тестирования для повторного выполнения процесса тестирования и определения ресурсов (времени и затрат), адекватных важности ПС, его модулей системы и решения задачи оптимального выпуска ПС.

Таким образом, практика определения характеристики завершенности показала возможность установления обоснованных количественных требований к безотказности компонентов ПС и систематический контроль их достижимости путем прогнозирования уровня скрытых дефектов и принятия адекватных организационных решений в ходе выполнения проекта.

Результаты раннего прогнозирования безотказности ПС незначительно отличаются от оценок по данным об отказах, полученным во время тестирования и опытной эксплуатации ПС. Качество прогнозов удовлетворяет общепризнанному критерию в инженерии качества $PR(0{,}25) = 0{,}75$, где PR – показатель качества прогноза: $PR(\varepsilon) = \frac{k}{n}$, ε – допустимая относительная погрешность прогнозирования, n – количество компонентов ПС, k – количество модулей компонентов ПС, для которых погрешность прогнозирования не превышает ε.

По мере усовершенствования моделей раннего прогнозирования дефектов и повышения точности оценок получаем преимущества их использования наравне с оценками параметров моделей позднего прогнозирования (моделей роста надежности).

9.5. СЕРТИФИКАЦИЯ ПРОГРАММНОГО ПРОДУКТА

Под *сертификацией программного продукта* понимается процесс (выполняемый третьей стороной) для удостоверения его специальным знаком или свидетельством идентифицированной программной продукции на соответствие конкретному стандарту, техническим условиям или требованиям.

Сертификат на программную продукцию свидетельствует о соответствии проверенных показателей качества этой продукции заданным требованиям. В условиях рыночных отношений наличие такого сертификата повышает конкурентоспособность, является средством завоевания рынка и защиты потребителей от недоброкачественной продукции.

Предусматривается два вида сертификации создаваемой продукции: обязательная и добровольная.

Обязательная сертификация ориентирована на проведение в Государственной системе сертификации проверки соответствия реальных свойств сертифицируемой продукции требованиям, определенным государственными нормативными документами. К обязательной сертификации отнесены потенциально опасные и вредные продукты, изделия, процессы. В этом перечне не указана программная продукция, хотя ошибки в ней могут привести к опасным последствиям как для безопасности людей, так и для экономики. Примером опасных последствий могут служить аварии при запуске космических кораблей «Челенджер» (США, 1995) и «Зенит-2» (СНГ, 1998), причиной которых явились ошибки в программах управления полетом.

В связи с этим ясно, что необходима сертификация программной продукции как механизма управления качеством, обеспечение ее безопасности и конкурентоспособности отечественных программных продуктов, защита пользователей от недоброкачественной продукции необходима. Однако многие организации, производящие программные продукты, не проводят мероприятий по обеспечению их качества и сертификации по следующим причинам:

– нежеланием подвергать программные продукты сертификации из-за затрат дополнительных ресурсов;

– непониманием заказчиком ПС преимущества получения сертифицированного программного продукта;

– отсутствием в организациях систем обеспечения качества и др.;

– отсутствием рынка отечественной программной продукции.

Системы обеспечения качества ПС, нормативно-методические документы, а также системы сертификации программных продуктов направлены на решение следующих задач:

1) создание нормативной базы инженерии качества ПО, соответствующей требованиям международных и государственных стандартов;

2) разработка типовых элементов систем обеспечения качества в организациях, разрабатывающих программные продукты;

3) освоение и совершенствование методов оценивания качества продуктов и процессов их производства;

4) создание нормативно-методической и инструментальной базы системы сертификации программных продуктов.

К этим задачам примыкают задачи оценки зрелости организации и процессов производства программных продуктов на основе модели CMM [38].

Представлены результаты исследований по проблематике качества ПС, включающие в себя методы инженерии качества, метрики, оценки атрибутов показателей качества и качества в целом. Отмечается, что на оценку качества ПС влияют методы инженерии требований к ПС и методы, гарантирующие достижение заданных характеристик на ранних этапах ЖЦ, в частности, основанные на прогнозировании плотности дефектов, безотказной работы ПС, и используемых при оценке атрибута качества – завершенность ПС.

ГЛАВА 10

МЕТОДЫ УПРАВЛЕНИЯ ПРОГРАММНЫМИ ПРОЕКТАМИ

Генри Гант впервые предложил диаграммную схему учета времени выполнения проекта по созданию лайнера для перевозки пассажиров из Европы в Америку (1900 г.). Это был первый шаг, после которого постепенно стали формироваться задачи и методы управления проектом, в частности планирование времени выполнения работ и отображение его в плане-графике.

Управление проектом – это руководство ресурсами (человеческими, техническими и финансовыми), временем и затратами на реализацию целей и задач проекта. Особенность программного проекта в отличие от технического проекта состоит в том, что он не материален, быстро стареет, компьютерная техника постоянно обновляется, а стандарты еще не в полной мере регламентируют их изготовление.

Комитетом мирового сообщество специалистов разработано общее ядро знаний программного проекта – PMBOK (Project Management Body of Knowledge [1–3]), в котором сформулированы цели и задачи:

– выбор метода управления;

– планирование и составление графика работ,

– подбор стандартов, процедур и технологии ведения реализации программного проекта;

– распределение работ между исполнителями с учетом их квалификации, оплаты, исходя из общей стоимости проекта;

– координация работ исполнителей проектной группы;

– управление рисками проекта;

– обеспечение безопасности и защиты в проекте;

– оценка выполненной работы каждым исполнителем и ее приемка;

– использование инструментов управления проектом для ведения планов-графиков и слежения за их выполнением.

Основными вопросами управления проектом являются:

1. Методы управления проектами;
2. Планирование и выполнение проекта.
3. Управление рисками.
4. Стандартные положения по управлению проектом.
5. Особенности управления практическим проектом документооборота для управляющих информационных систем.

10.1. МЕТОДЫ УПРАВЛЕНИЯ ПРОЕКТАМИ

Метод управления CPM (Critical Path Method) был разработан (в 50-х годах прошлого столетия) Уолкером и Келли (M.Walker, D.Kelly), а позднее получил название метода критического пути CPM. Метод основан на графическом представлении задач (работ) и действий по реализации проекта с ориентировочным временем их выполнения. В вершинах графа располагаются работы, время их выполнения указывается под вершинами графа либо на дугах графа.

Критический путь – наиболее полный путь продолжительности работ на графе (от начальной работы к конечной). Работы, которые лежат на этом пути, также называются критическими. Продолжительность критического пути определяет наименьшую общую продолжительность работ в проекте в целом. Критический путь включает в себя вершины: начало пути, промежуточные (например, *A*1, *A*3, *A*6 на рис. 10.1) и конечные вершины.

Время выполнения работ может изменяться в сторону увеличения или уменьшения и на путях, не вошедших а критический (например, начало пути – *A*1, *A*4 – конец пути). Увеличение времени может повлиять на стоимость завершения работ.

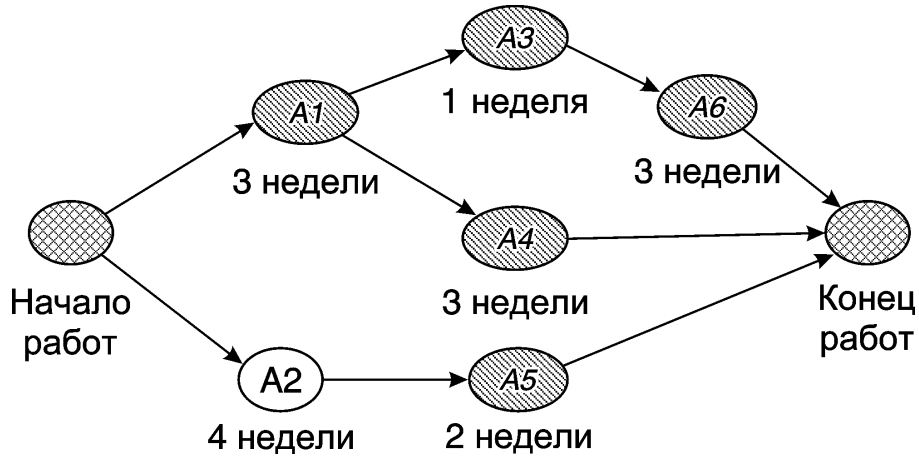

РИС. 10.1. Пример сетевой модели CPM

Основное преимущество этого метода заключается в управлении сроками реализации задач, которые не лежат на критическом пути и не влияют на время выполнения независимых работ проекта. Таким образом, метод CPM позволяет создавать календарные графики выполнения комплекса работ на проекте, представлять их в виде логической сетевой структуры и оценивать фактическое время реализации каждой работы [4].

Метод PERT (Program Evaluation and Review Technique)**.** Параллельно с разработкой метода CPM корпорацией "Lokhid" и консалтинговой фирмой "Буз, Аллен & Гамільтон" в рамках реализации проекта создания ракетной системы "Polaris" военно-морских сил США был создан метод PERT анализа и оценки программ, который объединял около 3800 основных подрядчиков и составлял более 60 тыс. операций. Метод PERT позволил руководству проекта точно определить виды работ, время их выполнения и подобрать необходимых исполнителей, а также определить вероятность своевременного завершения отдельных работ. Благодаря этим особенностям управление данным проектом по методу PERT руководство рассматриваемого проекта завершило его реализацию на два года раньше запланированного срока.

У*правление* проектом с помощью метода PERT заключается в задании сетевой диаграммы, в которой отображается полный комплекс видов работ, время их выполнения и установленные зависимости между работами. В отличие от метода CPM метод PERT учитывает возникающие неопределенности в длительности выполнения каждой работы и их взаимосвязанности.

Сетевая диаграмма – это граф, в вершинах которого располагаются работы, а дуги задают взаимные связи между этими работами. Принципиальным его отличием от блок-схемы является то, что сетевая диаграмма моделирует логические зависимости между элементарными работами. Она не отображает входы и выходы, и не допускает повторяемых циклов или петель [3].

Существует другой тип сетевой диаграммы, когда в вершине указывается событие, а работа задается линией между двумя узлами-событиями. Есть событие, соответствующее началу и концу данной работы. PERT-диаграмму можно отнести к этому типу диаграмм. В целом расхождение между двумя методами представления сети является незначительным, а представление более сложных связей между работами приводит к усложнению структуры сети. Такая сеть редко используется на практике.

Ожидаемое время выполнения работы определяется с помощью таких оценок:

– оптимистической (О),

– пессимистической (П),

– вероятностной (В)

и вычисляется по формуле (О+4В+П)/6. Полученное время указывается в сетевом графике.

10.1.1. Планирование программных проектов

Планирование представляет собой процесс распределения и назначения ресурсов (материальных и человеческих) в графике с учетом заданной стоимости и времени выполнения проекта. Неадекватное планирование может привести к нарушению сроков и некачественному изготовлению продукта проекта, а также к невыполнению отдельных требований заказчика.

План проекта состоит из технического и ресурсного планов и ориентирован на выполнение таких задач [15, 16]:

– определение видов продуктов, их содержания, формы и критериев качества;

– формирование модели ЖЦ разработки проекта с учетом стандартов и методов программирования;

– определение видов инженерной деятельности, связанных с проектированием и тестированием продуктов проекта;

– определение последовательности работ и их взаимозависимостей;

– распределение человеческих ресурсов для процессов ЖЦ;

– определение интервалов времени, календарных сроков и видов контроля;

– распределение ответственности членов группы разработчиков по работам на проекте.

Элементы этого плана – *продукты, задачи, действия* и *ресурсы*.

Продукты – это компоненты, реализующие функциональные требования и заданные показатели качества. Продукты идентифицируются в соответствии с их делением на группы и использованию.

Задачи и действия. План должен отображать все задачи и действия на процессах ЖЦ, необходимые для получения продукта проекта заданного качества. Действия должны быть отображены в сетевом графике с помощью последовательности взаимосвязанных работ, в том числе и параллельно выполняемых.

Ресурсы. Каждому действию в сетевом графике соответствует работа, требующая некоторое количество ресурсов, которые должны согласовываться с имеющимися в распоряжении проекта, а также со сроками и количеством персонала, участвующего в выполнении работ.

Все ресурсные требования агрегируются в общем, ресурсном плане и с соответствующей стоимостью. К техническим ресурсам относятся компьютеры, устройства и программные средства, а также персонал и уровень его квалификации.

Виды планов. Планирование программного проекта учитывает:

1) инструкции проекта, составленные советом проекта;

2) бизнес план и границы проекта;

3) список задач проектирования и их контроль на процессах ЖЦ;

4) методы выполнения проекта.

При подготовке общего плана проекта выполняются следующие основные задачи:

– анализ исходных данных и получение на их основе функций продуктов, которые должны быть созданы в процессе выполнения работ проекта;

– определение технической стратегии и стандартов для проекта;

– идентификация проектных действий на основе преобразований в рамках диаграммы потоков на соответствующем уровне детализации, всех зависимостей между этими действиями и ограничений на них;

– принятие адекватных мер по проверке качества и контрольных точек управления;

– выбор границ этапов и действий работ;

– подготовка плановых проектных документов и согласование их с руководством проекта.

План проекта может состоять из планов этапов, детализированного плана и индивидуального рабочего плана.

План этапа содержит перечень работ, предусмотренных в технических и ресурсных планах и механизмы их отслеживания. Эти планы регламентируют виды изготавливаемых продуктов, сроки, ресурсы и стоимость.

Детализированный план – детализирует задачи и действия на этапах проекта. Если этот план превышает некоторые ограничения на этапе, то разрабатывается новый план, который повторно согласуется с руководством проекта.

Рассмотренные планы позволяют регулировать выполнение проекта в заданных границах времени и стоимости, и в случае необходимости изменять план или заменять его другим откорректированным.

Окончательный план включает в себя разделы:

– график плана и его описание;

– описание внешних факторов и условий выполнения плана;

– риски в процессе выполнении плана и др.

Описание плана содержит распределение обязанностей, контроль плана, учет отклонений, мониторинг и порядок отчетности.

Внешние факторы используется для выявления зависимостей проекта от внешних ресурсов или событий, а также для отображения взаимосвязей с другими проектами.

Риски плана включают в себя описание ситуаций, связанных с болезнями исполнителей, недостаточным количеством ресурсов, сбоев аппаратуры и др.

10.1.2. Методы разработки графиков работ

До момента появления программной инженерии и соответствующих стандартов, регламентирующих управление проектом, планы проектов в основном не составлялись, а также не проводилась количественная и качественная оценка их выполнения. Программный продукт разрабатывался согласно договору с заказчиком и утвержденному техническому заданию, в котором оговаривались сроки начала и конца работ. Современные методики предлагают средства, методы и инструменты, которые позволяют спланировать работы с учетом реальных ресурсов и работ с учетом плановых сроков.

При планировании используются средства автоматизированного создания планов-графиков (например, в системе Project Management 2002 [5, 6]), диаграммы Ганта, сетевые графики, календарные планы и др.

Диаграмма Ганта – горизонтальная линейная диаграмма, на которой задачи проекта представляются сроками в виде отрезков времени, имеют даты начальной и конечной точек с задержками и другими временными параметрами.

Процесс составления графика работ включает в себя действия, которые приведены на рис. 10.2.

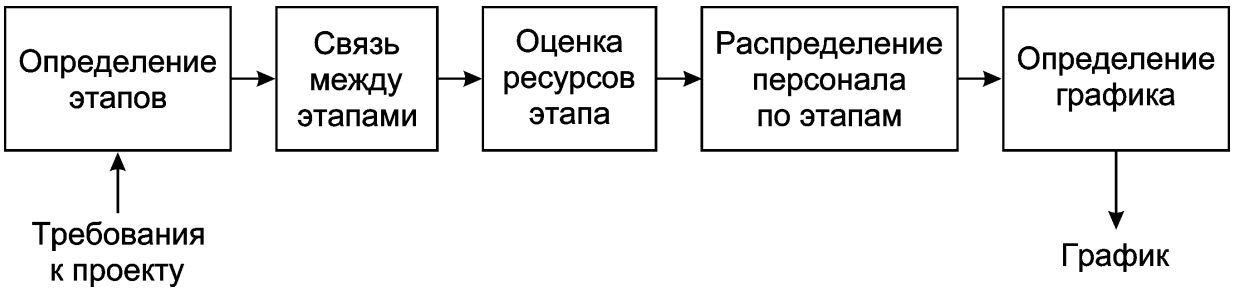

РИС. 10.2. Процесс подготовки графика работ на проекте

Результат этого процесса – сетевой график, состоящий из вершин – точек проекта и дуг, задающих номера видов работ и время их выполнения, как это показано на рис. 10.3 [4].

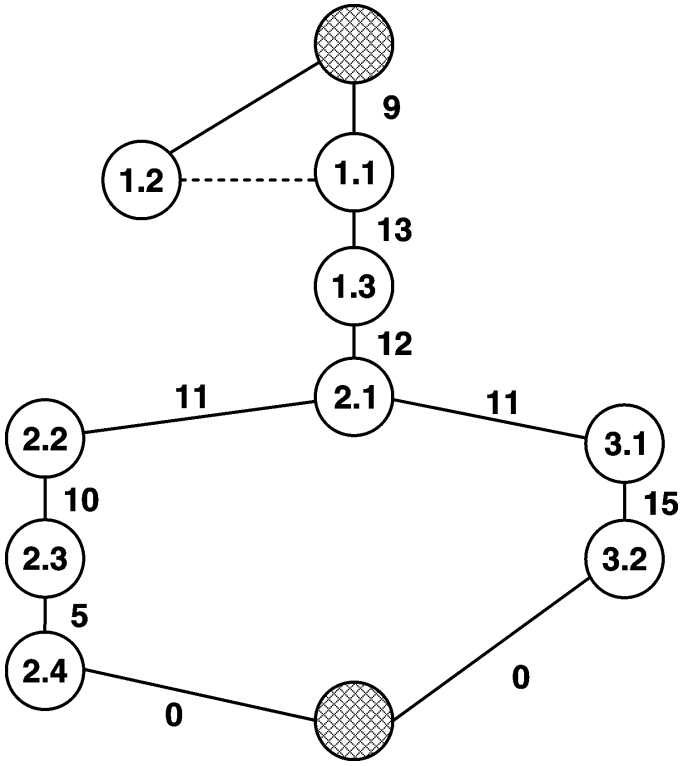

РИС. 10.3. График работ и сроков выполнения (на дугах)

Дуге, выходящей из начальной вершины и входящей в заключительную вершину, соответствует временная метка 0. С помощью этих меток задается время выполнения работ. В графе могут присутствовать циклические пути. На основании такого графа можно проводить анализ критических путей, т.е. определять данные о продолжительности и полученных промежуточных продуктах.

Проектирование проекта разделяется на проектирование верхнего уровня, пользовательского интерфейса и различных компонентов проекта.

На верхнем уровне определяется концептуальная модель проектируемой системы, ее составные части и объекты. На основе концептуальной модели проектируется архитектура системы и набор компонентов, входящих в нее. Каждый компонент кодируется в ЯП, тестируется и оценивается на качество.

Одновременно с этим на основе графика работ определяется критический путь и наблюдение за выполнением каждой работы в нем.

На этапах ЖЦ проекта согласно стандарту реализуются:

– требования заказчика к системе;

– архитектура системы;

– задачи и функции проектируемой системы;

– программные компоненты и их тестирование, а также проверка проекта в целом.

Выбор структуры ЖЦ проекта предполагает исходить из следующих основных положений:

– назначение системы,

– сроки разработки и виды результата,

– распределение ролей за реализацию функций,

– стратегия управления работами проекта и техника их выполнения,

– наличие всех видов ресурсов и др.

Для ведения проектом Боэм [8] предлагает подход, основанный на спиральной модели ЖЦ разработки программного обеспечения, при которой на каждом витке спирали выносится решение о выполненных задачах, условиях выбора и ограничениях. Однако для простых проектов могут применяться модели ЖЦ – водопадные, итерационные и др.

10.1.3. Управление персоналом и коммуникациями

К задачам управления проектом относится подбор и формирование команды исполнителей, управления их работой, а также установление коммуникаций между ними. Распределение обязанностей исполнителей проекта проводится в зависимости от квалификации и их личных качеств, процедур принятия решений и делегирования полномочий в команде, а также от целей заинтересованных сторон (stakeholders), менеджеров проекта и главного программиста.

Пример. Разработка известной операционной системы ОС–360 фирмы IBM, в которой принимало участие более 1000 сотрудников и на протяжении 12 лет, имела 21 версию с критической оценкой – 200 программных модулей в год. Используя опыт разработки этой ОС и организацию работ на фирме, Брукс предложил следующую структуру команды разработчиков ПО (рис. 10.4) [21].

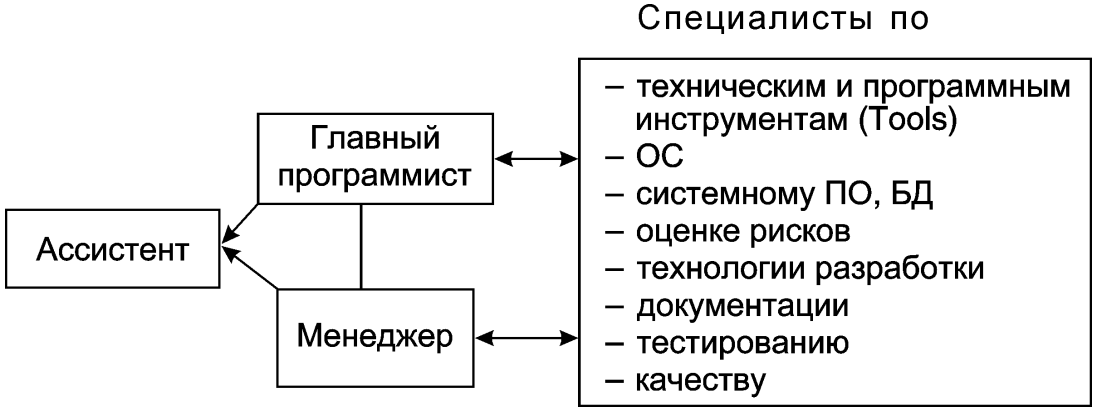

Рис. 10.4. Структура организации группы главного программиста

В ней управление проектированием и разработкой проекта выполняет главный программист. За каждым видом деятельности на проекте закреплен соответствующий член группы (программист, библиотекарь, тестировщик и т.п.). Устанавливается его ответственность за свойственный участок работы и подчинение вышестоящему руководителю. Так, библиотекари, ассистенты, программисты и другие исполнители непосредственно подчиняются менеджеру с правом принятия решений. Главный программист руководит подгруппой программистов, знающих детали проекта, методы и процессы разработки, а также членами группы сопровождения, обученных методам, средствам и инструментам, принятым в организации.

Ассистент главного программиста дублирует и замещает главного программиста. Библиотекарь ведет документацию проекта: компилирование, тестирование модулей библиотеки и поиск ошибок при создании необходимых материалов.

В группу входит администратор и группа тестировщиков. Структура такой рабочей группы иерархическая и каждый член группы может общаться непосредственно с главным программистом или с другими сотрудниками. Главный программист держит под контролем реализацию всех частей основного проекта и программ.

Определение количества членов группы и их ролей зависит от масштаба работ и их опыта. Исполнители проекта должны быть квалифицированными и уметь выявить ошибки и разные неточности в проекте на самых ранних стадиях ведения разработки.

Специалисты группы различаются между собой:

– способностью обучаться и выполнять работу;

– интересом к работе;

– опытом работы с подобным проектом и используемыми языками, технологиями и ОС;

– коммуникабельностью;

– способностью разделить ответственность с другими;

– профессиональным навыком как в менеджменте, так и в программировании.

Менеджер проекта учитывает способности членов группы и ориентирует их на выполнение соответствующих работ по проектированию, тестированию и реализации системы. Группа должна пользоваться одним и тем же стилем программирования.

Альтернативная структура ведения проекта предложена Вейнбергом (Weinberg) [20]. Это – так называемое обезличенное программирование, при котором все несут одинаковую ответственность за качество продукта. В проекте не концентрируются на персоналиях, критике подвергается программный продукт, но не члены группы. Такая структура подходит для маленьких групп программистов.

10.1.4. Управление рисками проекта

Риск – это нежелательное событие, которое может иметь непредвиденные негативные последствия. Если в проекте идентифицировано множество возможных рисков, которые могут повлечь за собой негативные последствия, то такой проект склонен к риску. Вероятность риска может меняться во времени, как и последствия риска. Поэтому проводятся мероприятия по оценке риска и его последствий в разных точках проекта, например, в случае возникновения того или иного непредвиденного события [1, 13].

Риски могут быть спрогнозированы, оценены, а также спланировано управление ими на этапах ЖЦ.

Существуют два основных типа риска: общий риск для всех типов проектов и риск, специфический для конкретного проекта.

К первому типу риска относится риск, возникающий при недопонимании требований, при нехватке профессионалов или

недостатке времени на тестирование. Риск второго типа выражается недостатками проекта (незавершенность проекта к обещанному сроку, несоблюдением всех требований и др.).

Американский Институт управления проектами (PMI), разрабатывающий и публикующий стандарты в области управления проектами, переработал разделы, регламентирующие процедуры управления рисками. В новой версии PMBOK описано шесть процедур управления рисками, которые обеспечивают идентификацию, анализ рисков и принятием решений по минимизации отрицательных последствий рисковых событий.

Процесс управления рисками проекта включает в себя оценку и контроль процедур (рис. 10.5):

– планирования управления рисками – выбор подходов и планирование деятельности по управлению рисками проекта;

– идентификации рисков, способных повлиять на проект, и документирование их характеристик;

РИС. 10.5. Схема управления рисками проекта

– оценки вероятности возникновения и влияния последствий рисков на проект;

– оценки приоритетов исправления рисков;

– ослабление отрицательных последствий рисковых событий и их минимизация;

– мониторинг и контроль рисков.

После идентификации рисков, строится схема приоритетов для рисков, исходя из показателей ущерба риска, оценок последствий риска и вероятности риска. При постепенном наращивании функций проекта и проведении регрессионного тестирования новая функция тестируется вместе со старыми функциями, что снижает риск. Для каждого возможного риска определяется показатель степени вероятности появления и потерь от риска. Во время проведения регрессионного тестирования отыскиваются критические ошибки. В зависимости от того, насколько эта ошибка критична и от того, какие показатели риска действуют, вычисляется ущерб риска.

Далее рассмотрены основные задачи процедур управления рисками.

Планирование управления рисками – это запланированный процесс принятия решений по применению и управлению рисками в конкретном проекте. Он может включать в себя мероприятия по организации, кадровому обеспечению процедур управления рисками проекта, выбор предпочтительной методологии, источников данных для идентификации риска, временного интервала для анализа ситуации, а также управление рисками, адекватное уровню и типам риска и важности проекта.

Идентификация рисков определяет виды рисков, влияющих на проект, и документирует характеристики этих рисков. Идентификация рисков должна проводиться регулярно на протяжении реализации проекта. В ней принимают участие: менеджеры проекта, заказчики, пользователи и др. Вначале эта идентификация может быть выполнена менеджером проекта или группой аналитиков рисков. Далее, ею может заниматься основная группа, управляющая проектом. Для формирования объективной оценки в завершающей стадии разработки проекта в ней могут участвовать независимые специалисты.

Оценка рисков – процесс качественного анализа идентификации рисков, определения возникающих рисков, требующих быстрого реагирования, и оценки влияния рисков на проект. Для разных категорий рисков расставляются приоритеты. Основная

задача – это оценка условий возникновения рисков и определение их влияния на проект стандартными методами и средствами, имеющимися на проекте. На этапах ЖЦ проекта может происходить переоценка рисков.

Количественная оценка рисков проводится для определения вероятной оценки возникновения рисков и влияние последствий рисков на проект, что помогает правильно принимать решения и избегать разных неопределенностей. В процессе проведения количественной оценки рисков рассматриваются:

– вероятность достижения конечной цели проекта;

– степень воздействия риска на проект и объемы непредвиденных затрат и материалов, которые могут понадобиться;

– риски, требующие скорейшего устранения и оценки их влияния на последствия реализации проекта;

– фактические затраты и предполагаемые сроки окончания работ на проекте.

Количественная оценка рисков связана с качественной оценкой и идентификацией рисков и зависит от времени и бюджета, необходимого для этой оценки.

Планирование реагирования на риски – это разработка плана для снижения отрицательного воздействия рисков на проект и определения ответственности за эффективную защиту проекта от влияния на него рисков. Планирование включает в себя идентификацию и распределение каждого риска по категориям. Эффективность разработки зависит от последствий влияния рисков на проект положительным или отрицательным образом. Стратегия планирования должна соответствовать типам рисков, рентабельности ресурсов и временными параметрами. Вопросы, обсуждаемые во время встреч, должны быть адекватны задачам на каждом этапе проекта, и согласованы со всеми членами группы проекта. Обычно требуются несколько вариантов стратегий реагирования на риски.

Мониторинг и контроль – это процессы слежения за идентификацией рисков, определения остаточных рисков, выполнения плана рисков и оценки понижения риска. Показатели рисков, связанные с осуществлением условий выполнения плана, фиксируются. Мониторинг и контроль сопровождают процесс внедрения проекта в практическую жизнь. Контроль выполнения проекта собирает информацию, помогающую принимать своевременные решения по предотвращению возникших рисков.

В процессе мониторинга и контроля могут выясниться вопросы:

– внедрения плана реагирования на риски системой и его эффективность,

– влияния изменений на выполнение проекта,

– определения влияния рисков, принятых мер и причин их появления.

Контроль может повлечь за собой выбор альтернативных стратегий, принятие корректив, перепланировку проекта для достижения базового плана. Между менеджерами проекта и группой риска должно быть постоянное взаимодействие для фиксации всех изменений и явлений, а также отчета по выполнению проекта.

Определяется результат управления риском проекта в виде отношения:

$$\frac{\text{Ущерб до минимизации} - \text{ущерб после минимизации}}{\text{Цена минимизации риска}}$$

Боэм идентифицировал 10 наиболее часто возникающих причин риска в проекте [8]:

1. Сокращение штата или набор неквалифицированных сотрудников.
2. Нереалистические в проекте планы и бюджеты.
3. Разработка функционально неправильных программных элементов.
4. Разработка неудачного пользовательского интерфейса.
5. Неудачная постановка требований.
6. Постоянное изменение требований.
7. Недостатки во внутренней организации работ.
8. Недостатки взаимосвязи с заказчиком.
9. Неумение работать в реальном времени.
10. Ограниченные компьютерные ресурсы.

10.1.5. Оценка затрат и стоимости проекта

При выполнении проекта проводится анализ затрат и оценка текущей длительности проекта с помощью CPM, PERT и моделей, приведенных в [8, 14, 21].

Модель оценки программного обеспечения предложена Боэмом – COCOMO (Constructive Cost Model). Она позволяет рассчитать затраты и стоимость создаваемого проекта.

Оценка затрат на разработку проекта проводится по формулам:

для простого проекта $PM=2.4 \times (K)^{1.06} \times M$,

для среднего проекта $PM=3.0 \times (K)^{1.12} \times M$,

где K – это количество инструкций (в тыс. команд), M – показатель, зависящий от коэффициента повторного использования компонентов, сложности и др.

Общий расчет затрат определяется по формуле: $PM= A \times K^B \times \times M + Pмав$.

Оценка стоимости. Общая стоимость проекта зависит от стоимости его отдельных частей, условий работы, штата сотрудников, используемых методов и инструментов. В стоимость проекта входит стоимость аппаратуры и объектов окружения исполнителей (мебель, телефоны, канцелярские товары и др.), где сотрудники работают.

Модель стоимости программного продукта включает в себя стоимость аппаратуры, ОС, БД И СПО (ОС+СПО+БД), а также затрат *SE* на разработку проекта:

$SE = PM \times \text{Real}\ (Над.) \times T \times \text{Store} \times ИС\ (\text{Tools}) \times \$15000$.

Время оценивается по формуле $T = 3 \times PM^{0.33+0.2B-1.01}$. Если проект не укладывается в заданное директивное время, то добавляется процент для увеличения длительности выполнения проекта.

Иногда требуются дополнительные расходы на тестирование системы, кодирование или использование других автоматизированных средств – CASE системы. Главной оценкой в проекте является оценка усилий по ведению проекта в человеко-днях сотрудников. Эти оценки проводятся на ранней стадии в процессе составления плана проекта, их погрешность должна быть меньше 10%.

Правильность оценки зависит от компетентности, опыта, объективности и восприятия экспертом модулей и системы в целом.

В оценке принимают участие эксперты, они проводят опрос всех членов рабочей группы, просматривают различные проектные документы и в дальнейшем осуществляют коррекцию каждой оценки, выводя наиболее правдоподобную с помощью оценок: пессимистической (x), оптимистической (y) и реальной (r) и формулы $(x+4y+r)/6$.

Разные методы приближенного обобщения оценок относительно реалистичных рассмотрены в [8]. Во всех приведенных методах имеются недостатки, связанные с трудностью определения степени отличия старой системы от новой системы. В некоторых рабочих группах пессимистическая и оптимистическая

оценки могут сильно различаться. Иногда система оценок, которая успешно работает в одной организации, не работает в другой.

Алгоритмические методы оценки базируются на модели, в которой отображаются связи между затратами в проекте и факторами, влияющими на них. В этой модели затраты – зависимая переменная, а влияющие факторы – независимые переменные. Например, стоимость проекта $E = (a+bS^c)\ m(X)$, где S – оценка размера системы, *а, в, с* – эмпирические константы, X – вектор факторов стоимости размерностью *n, m* – регулирующий множитель, основанный на затратных факторах.

В [8] предложена модель оценки проекта, полученная экспериментальным путем: $E = 5{,}25S^{0.91}$ в проекте, в котором были построены ПС от 4000 до 467000 строк кода на 28 различных языках программирования высокого уровня для 66 компьютеров. На ее разработку было затрачено от 12 до 11758 человеко-месяцев. Предложенная техника моделирования затрат дала более точную оценку: $E = 5{,}5+0{,}73S^{1.16}$.

В большинстве случаев модели оценки зависят от размера системы в строках кода. Модель COCOMO Боэма объединила три техники измерения проекта с использованием показателей цены, персонала, свойств проекта и среды. Эта модель дает оценку трех стадий ведения проекта.

На первой стадии строится прототип для задач повышенного риска (интерфейс пользователя, ПО, взаимодействие, реализация и др.) и проводится оценка затрат (например, по числу таблиц в БД клиента, повторному использованию отчетных форм др.). На второй стадии проводится оценка затрат на проектирование и реализацию функциональных точек проекта, отраженных в требованиях к проекту. На третьей стадии оценка относится к завершенному проектированию, когда размер системы может быть определен по числу готовых строк программы и других факторов.

Базовой моделью служит уравнение вида: $E=bS^c\ m(X)$. В нем первичная оценка bS^c корректируется с помощью вектора стоимости $m(X)$. Параметр *с* изменяется от 0 до 1,0 на первой стадии и от 1,01 до 1,26 для остальных.

Расчет усилий на разработку. Модель COCOMO основывается на предположении, что размер системы *Size* и величина усилий, потраченных на разработку проекта, выражаются в человеко-месяцах по формуле: $\log PM = A + \log Size + b$. Это предположение было проверено на примере 161 проектов [32].

Оценка усилий на выполнение проекта (в человеко-месяцах) по методу COCOMO проводится по формуле

$$PM_{NS} = A \times Size^{e} \times \prod_{i=1}^{E} EM_i$$

где $E = B + 0{,}01 \times \sum_{j=1}^{E} SF_j$, индекс NS – значение для бюджета PM означает усилия номинального расписания графика работ, $Size$ – размер системы, EM_i равно 16, SF_j – экспоненциальные факторы, константы A и B равны, соответственно, 2,94 и 0,91 [33].

Продолжительность разработки $TDEV_{NS}$ оценивается по формуле

$$TDEV_{\text{NS}} = C \times (PM_{NS})^{F},$$

где $F = D + 0{,}2 \times 0{,}01 \times \sum_{j=1}^{5} SF_j = D + 0{,}2 \times (E - B)$, N – количество мультипликативных факторов, $EM_i = 16$, SF_j – экспоненциальные факторы, константы C и D равны, соответственно, 3,67 и 0,28 [33].

10.2. СИСТЕМА УПРАВЛЕНИЯ ПРОЕКТОМ

Систему MS Project 2003 по данным корпорации Microsoft использует более 8 миллионов для целей управления проектом. Эта система применяется для планирования малых и средних организаций [5, 6] и включает в себя набор базовых функций, а также большое количество специальных функций, свойственных пакетам профессионального уровня. Базовые функции на стадии планирования позволяют ответить на такие вопросы:

1) возможно ли реализовать данный проект,

2) какие работы надо выполнить, чтоб достичь цели проекта,

3) какой состав исполнителей и видов материальных ресурсов потребуется для реализации проекта,

4) какая стоимость проекта и как наиболее выгодно распределить финансовые затраты на реализацию проекта,

5) кто должен отвечать за те или иные виды работ,

6) возникнет ли риск и какие мероприятия надо сделать, чтобы избавиться от вреда, наносимого риском на той или иной стадии планирования.

Для ответа на *первый вопрос* проводится полный анализ проекта по методу CPM поиска критического пути ресурсным пла-

нированием, без детализации, но с использованием шаблонов, которые входят в состав пакета. Каждый шаблон относится к определенной области и может считаться некоторым стандартным планом проекта. Внося необходимые поправки, в соответствии с особенностями конкретного пакета можно получить реальную оценку возможного развития событий и нужных затрат.

Ответ на *второй вопрос* также можно получить с помощью одного из стандартных решений. Если соответствующий шаблон отсутствует, то структура проекта создается вручную. При этом MS Project оказывает помощь за счет средств построения сетевого графика, технология построения которого практически не отличается от рисования на бумаге, но это занимает больше времени. Для работ проекта автоматически задаются параметры (протяжность, календарные даты начала и окончания работ и т.д.). На основе сетевого графика автоматически формируется календарный план в виде диаграммы Ганта со сроками, которые относятся к реальным датам.

Для получения ответа на *третий вопрос*, необходимо распределить ресурсы (например, на уровне текущего представления менеджера о составе и характере работ, которые входят в состав проекта). В качестве ресурсов проекта могут быть заданные или уникальные для него исполнители и материалы, или унаследованные виды ресурсов, которые использовались в предшествующих продуктах (или взятые из шаблонов). Обобщающую информацию об используемых в проекте ресурсах можно получить с помощью таблицы ресурсов, а более детальную – на основе анализа назначений ресурсов. Для каждого ресурса могут быть построены гистограммы его загрузки и стоимости.

После назначения нового ресурса (с указанием его стоимости и объема) проводится автоматический перерасчет стоимости проекта (ответ на *четвертый вопрос*), благодаря чему легко получить сравнительную характеристику разных вариантов назначений. Для проведения стоимостного анализа проекта в этом пакете используется “анализ освоенного значения” (Earned Value Analysis), который позволяет получить затраты, а также текущую дату.

В плане рассмотрения *шестого вопроса*, можно сказать, что даже очень хороший план проекта не застрахован от разных случайностей. Чтобы адекватно проанализировать риск, необходимо иметь детализированный план проекта. Как правило, при анализе рисков рекомендуется использовать многие из тех средств и форм представления проекта, о которых было сказано выше. Кроме них,

могут применяться дополнительные методы и средства, выбор которых зависит от специфики проекта и уровня подготовки пользователя. Например, простым и вместе с тем эффективным средством является сравнение нескольких версий (сценариев) оценок проекта. Как правило, таких оценок может быть три: пессимистическая, оптимистичная и наиболее вероятностная. Для сравнительной оценки длительности проекта по этим сценариям в составе MS Project имеется процедура анализа проекта с помощью метода PERT, учитывающая в оценке проекта возникающие риски.

Стандартизация процесса управления проектом. Процесс управления проектом входит в ЖЦ стандартов ISO/IEC 12207−96 и ISO 15504 (части 1−9) 2002г.[12, 15]. В первом стандарте он представлен как самостоятельный процесс и вошел в организационную категорию процессов ЖЦ. Этот процесс рекомендует создавать структурную группу для планирования и управления проектом, учитывающую задачи и действия, выполняемые на процессах ЖЦ. Согласно стандарту ISO 15504 процесс *управления проектом* состоит в идентификации, координации и контроля действий, задач и ресурсов, используемых для изготовления продукта и/или услуги в соответствии с заданными требованиями к проекту.

Главными факторами осуществления задач программного проекта являются:

− объем работ, имеющиеся ресурсы и ограничения;

− стоимость выполнения задач и необходимых ресурсов;

− компетентность специалистов;

− методы, средства и стандарты, соответствующие выполнению задач проекта;

− планы проекта и контроль их выполнения;

− формирование модели ЖЦ, удовлетворяющей проекту;

− получение целевых показателей проекта;

− оценка стоимости, трудоемкости и продолжительности выполнения проекта;

− измерение размера и оценка сложности продуктов, полученных на процессах ЖЦ.

Методы и средства выполнения процесса управления проектом ориентированы на методическую и инструментально-технологическую поддержку выполнения процессов ЖЦ и служат созданию соответствующего продукта согласно модели ПрО.

Модель ЖЦ программного проекта. Независимо от конкретной модели ЖЦ, которая используется в проекте, основным понятием модели является *проектная работа* процесса ЖЦ (напри-

мер, кодирование, проектирование и т.п.), которая требуют руководящих действий. Другими понятиями проекта являются план проекта, метод его проектирования и управления, а также контроль и аудит результатов процессов.

Каждая работа на процессе ЖЦ обеспечивает преобразование некоторого промежуточного продукта с привлечением необходимых ресурсов, правил и действий ЖЦ.

Концепция управления проектом характеризуется тем, что его разработка выполняется согласно разным уровням планирования (проект, работы, процессы ЖЦ) и контроля их выполнения.

Для реализации проектов разработаны еще специализированные программные средства Adaptable Process Model, а также методы PERT или CPM, позволяющие спланировать работы с распределением времени и специалистов, руководить этим планом, и получать статистическую оценку характеристик проекта (например, длительность процесса). Более глубокий анализ характеристик проекта и процесса его выполнения обеспечивают методы моделирования процессов (конечные автоматы, сети Петре, байесовские сети и т.п.).

10.2.1 Инфраструктура и контроль на процессах ЖЦ

Соответственно стандарту для проекта разрабатываются планы управления качеством работ на процессах ЖЦ, управление специалистами на основе плана-графика, управление рисками и конфигурацией, V&V, тестирование, измерение, обучение и т.п.

Процесс управления проектом базируется на выборе модели ЖЦ, адекватной типу, размеру и сложности проекта и привязке к задачам процессов ЖЦ плана графика и ресурсов, необходимых для выполнения работ, а также методов программирования.

Созданный план проекта, включающий в себя календарный план, отображает объем и график работ, риски проекта и применяемую модель ЖЦ. Он должен объединяться с процессами ЖЦ, потребовать изменений календарного плана, связанных с проверками промежуточных продуктов процессов в контрольных точках проекта.

Контроль и учет в проекте направлен на отслеживание выполнения плана путем просмотра результатов, полученных в ходе выполнения задач процессов, разработку плана внесения изменений в план проекта, а также усовершенствование методов реализации работ по созданию проекта.

Мониторинг проекта – это слежение за планами разработки ПО, своевременное устранение возникающих проблем и недостатков, а также удержание этого плана в рамках последовательного его выполнения рабочими группами исполнителей отдельных работ.

Для оценки результатов работ на процессах ЖЦ проекта применяются методы измерения и оценивания, основанные на сборе разных данных о сроках, затратах (материальных, трудовых), размерах документов и программ, а также о тестировании каждого продукта проекта, количестве сбойных ситуаций, выправленных ошибках в рабочих продуктах. Контроль выполняют менеджеры проекта, они принимают меры по корректировке планов в связи с разными ситуациями, возникшими в проекте.

Управление качеством и оценка качества на процессах реализуется посредством встраивания средств достижения качества в процессы ЖЦ, к которым относятся анализ и контроль продуктов ЖЦ и оценки качества исполнения задач проекта. Если оцененное качество не удовлетворяет менеджера проекта, он занимается вопросами усовершенствования как процесса управления проектом, так и методов реализации задач проектирования ПО [19].

Инфраструктура управления проектом включает в себя интегрированный набор общедоступных технических, технологических, методологических и организационных средств, которые необходимы для выполнения проекта.

Однозначной инфраструктуры для разных видов проектов, нет, поэтому стандарт дает общие рекомендации по включению разных видов компонентов в инфраструктуру проекта:

– техника и коммуникации – компьютеры, файлы, серверы, сети и др.;

– офисная техника (сканеры, принтеры, проекторы) и т.п.;

– общесистемное ПО, инструменты клиент/серверных технологий, ОС, CASE-инструменты, системы программирования (4GL), защита информации и т.п.;

– информационные и стандартные ресурсы, включая методы и инструменты управления проектами, ресурсы Интернета, разные нормативные документы и т.п.;

– группы качества, контроля, сопровождения и т.п.;

– стандарты процессов ЖЦ, унифицированные методы работы в организации, обучение исполнителей и молодых специалистов;

– методы управления всеми ресурсами в том числе финансовыми организации-разработчика программного проекта.

10.2.2. Организационная структура проекта

Организационная структура проекта – это группы исполнителей, распределенных по задачам проекта, планирования, контроля и оценивания результатов. Как правило, допускается совмещение задач и ролей специалистов в этих структурных группах. Приведем права и обязанности разных видов исполнителей проекта с учетом организационных процессов стандартов ISO/IEC 12207 и ISO/IEC 15504.

Менеджер программного проекта несет ответственность перед организацией-разработчиком и заказчиком ПО за успешное выполнение качественного программного продукта, а именно:

– разрабатывает модель ЖЦ ПО и согласовывает ее с руководителем проекта системы;

– подключает к проекту специалистов по группам, действующих на уровне организации (группы качества, группы V&V и др.);

– вырабатывает стратегию действий на процессах ЖЦ и несет ответственность за целостность проекта

– руководит разработкой основных документов, верификацией и валидацией на процессах ЖЦ и т.п.

Контролер качества владеет стандартами разработки проекта, устанавливает несоответствия в исполнении работ принятым стандартам, нормативным документами и стандартам уровня организации.

Верификатор знает рекомендации к требованиям в стандартах и к содержанию документов ПО, сопоставляет решения, отображенные в разных рабочих продуктах процессов, и проверяет правильность преобразования входных данных в выходные результаты.

Валидатор проверяет и подтверждает соответствие рабочих продуктов требованиям заказчика, осуществляет проверку документов, которые прошли контроль качества и верификацию, утверждает документы у менеджера и тестирует систему в соответствии с требованиями заказчика.

Тестировщик владеет методами тестирования и выполняет исследовательское тестирование, автономное тестирование ПО на соответствие постановкам задач, тестирование соответствия ПО требованиям к качеству, системное и приемочное тестирование.

Группа управления конфигурацией регистрирует версии рабочих продуктов, которые прошли контроль, обеспечивает хранение этих версий и разграничивает доступ к ним в соответствии с принятым порядком в организации.

Группа сопровождения устанавливает порядок выполнения ПО, диагностирует его, подает запросы на изменение версии продукта, восстанавливает среду тестирования и пробует устранить дефекты самостоятельно. Выправленная версия продукта верифицируется повторно в установленном порядке.

10.3. УПРАВЛЕНИЕ ПРОЕКТОМ «ДОКУМЕНТООБОРОТ В ИНФОРМАЦИОННЫХ СИСТЕМАХ»

Во многих информационных системах (ИС) главными элементами обработки являются документы и их массивы, представленные в виде упорядоченной их совокупности, которые реализуются процессами сбора, передачи, обработки, хранения и использования [25–27].

В связи с ростом потока документов и увеличением делопроизводства в ИС эффективность их проектирования достигается за счет методов управления проектом, планирования сетевого графика работ, учитывающего все виды работ и время их выполнения, а также координации процессов ЖЦ, методов ведения БД и оценки стоимости работ.

10.3.1. Документооборот в ИС

Базовыми процессами в ИС является документооборот и делопроизводство. *Документооборот* включает в себя создание и передачу документов, распределение задач между участниками ИС, оценку количества документов и эффективность их обработки при распределении их по разным маршрутам прохождения по системе от одного узла до другого.

Делопроизводство – это комплекс мероприятий документирования (регистрация, учет, рассылка) и организация работы с документами с учетом состояния нормативно-правовой базы хранения документов и регламентации документооборота.

Созданные стандарты об электронных документах, документообороте и электронной цифровой подписи регламентируют вопросы документооборота, а ни методов проектирования, учитывающих ресурсы, время и затраты. Управление проектированием ИС должно базироваться на современных методах менеджмента проектов [9–12] в части планирования работ и их управлением на этапах ЖЦ проекта.

В данном пункте излагается методология управления проектированием документооборота в ИС, включающая в себя решение следующих взаимосвязанных задач [29–31]:

– распределение заданных ресурсов, времени и стоимости;

– моделирование и оптимизацию задач управления технологическими процессами (ТП);

– определение информационных характеристик документов – объем и время использования;

– функционирование документооборота на множестве ТП работ проекта и оценок их выполнения соответственно плану.

Характеристика документооборота в ИС. К главным задачам разработки ИС относится автоматизация базовых процессов путем создания систем, делопроизводства, электронного документооборота и автоматизации деловых процессов (*workflow*) обработки информации. В их основе – модели управления документооборотом, совокупность маршрутов и шаблонов типовых документов.

Автоматизация деловых процессов *Workflow* включает в себя процесс документооборота по передачи документов, распределение задач между участниками для достижения целей их обработки и поддержку трех главных функций: построение бизнес-процессов, выполнение бизнес-процессов, анализ бизнес-процессов.

Модель управления документооборотом рассматривает множество маршрутов, которые проходят документы от одного узла системы к другому, а также шаблоны типовых документов и сами документы.

Маршрут – это направленный граф, вершинами которого являются узлы процесса обработки документов, а ребрами – переходы документов от одного узла к другому. Каждый документ имеет ЖЦ, включающий в себя ведение, генерацию входного и выходного документа до сдачи в архив. Маршрутизация определяет путь движения документа между участниками процесса, которые знакомятся и обрабатывают поступившие документы. Каждый путь состоит из набора действий, которые выполняются перед/после поступления документа в соответствующую вершину графа.

Выполнение процесса состоит из отдельных шагов, связей между процессом и моделью, которая выполняется фактически при взаимодействии с ней участников проекта. Распределение работ и информации между участниками проекта выполняется через процессы *workflow* с использованием разных механизмов коммуникации (электронная почта, сообщения и т.п.). Шаблоны типовых документов имеют общую структуру и атрибуты (назначение, содержание, адрес, исходный номер документа и пр.).

10.3.2. Моделирование документооборота в ИС

Процесс моделирования и проектирования документооборота базируется на вычислении количественных оценок информационных характеристик документов [11, 29, 31]:

– объем, т.е. размер документа (средний и максимальный) и количество документов, которые поступают за определенный промежуток времени на обработку;

– время обработки документа в разных узлах нахождения документов на пути прохождения данных, передачи их по сети, выполнения операций над документами и т.п.

Объем вычисляется с помощью последовательности повторяемых групп полей документа, называемой регулярной частью документа, и нерегулярной – без повторяемых структур данных, к которым относятся шапки и заголовки документов.

Расчеты характеристик объема документов включают в себя:

– средний объем $V=l_h+n_s k_s\, l_s^{\max}$;

– максимальный объем $V_{max}=l_h+ n_s^{\max} k_s\, l_s^{\max}$,

где l_h – размер нерегулярной части документа; n_s – количество строк, которые заполняются для данного типа документов; k_s – коэффициент заполнения; $l_s^{\max}$ – максимальный размер регулярной части документа.

Вычисление характеристики – время выполняется в динамике работы и содержит:

– суммарное время обработки разных типов документов в соответствии с их маршрутом;

– время выполнения отдельных операций над документами в разных P_i и P_j узлах системы;

– время передачи документов между разными узлами обработки в ИС.

Расчеты характеристик объема и времени выполняются в два этапа.

На первом этапе их значения вычисляются статически с предположением, что документы обрабатываются автономно и не используются вычислительные ресурсы для других работ.

На втором этапе предполагается, что существуют потоки документов разного типа и назначения, и расчеты времени прохождения и обработки документов выполняются по формулам

$T_i^c = V_i^g (1/R_i^g + 1/R_i^c + 1/R_{i+1}^g)$ есть время, необходимое на перемещения документа из узла P_i в узел P_{i+1};

$T_i^d = t_i^1 + t_i^d + t_i^2$ есть время обработки документа в узле P_i; $T = \sum_{i-1}^{r1} T_i^c + \sum_{i=1}^{r} T_i^d$ есть общее время обработки документа в соответствии с прохождением им разных маршрутов между узлами.

После этих вычислений проводится моделирование документооборота ИС с использованием построенных моделей информационных потоков документов и вывода аналитических зависимостей между значениями величины интенсивности потоков документов в ИС и общим временем их обработки на заданных вычислительных ресурсах и средствах передачи данных. Эти зависимости используются также для оценки численных результатов моделирования документооборота.

Документы проходят по маршрутам, могут создать очереди, которые увеличивают время, как производную от размера очереди и количества загруженных документами узлов. Для управления процессами прохождения и распределения их совокупностей по разным рабочим станциям (РС) − аналог *Workstation* − созданы три модели РС обработки документов. Каждой модели соответствует однородный или неоднородный поток документов. Характерным свойством однородного потока является обработка независимых друг от другого документов одного типа с интенсивностью h. Порядок поступления документов со случайным характером потока описывается распределением Пуассона [28, 29].

Локальная модель РС имеет один компьютер обслуживания, в котором поток заявок отвечает потоку документов, которые поступают в него и ждут очереди для их обработки. Для оценки возможности обработки потока документов и оценки нижней границы коэффициента использования мощности r выполняется условие: $h*t_s \le r \le 1$, где t_s − среднее время обработки одного документа.

С помощью формулы Хинчина−Поллачека вычисляется среднее значение размера очереди: $w=r/2*(1-r)*(1+G_t^s/t_s)^2$, где G_t^s − стандартное отклонение от t_s. Откуда определяется среднее значение: $r=2w/(2w-x^2)$, где $x=(1+G_t^s/t_s)$.

Распределенная модель РС ориентирована на обработку неоднородных потоков заявок, объединенных по группами операций обработки документов или по их типам. Модель ориентирована на один компьютер обслуживания и несколько входных потоков, упорядоченных по приоритетам. Очередь документов является общей и первым из очереди на обработку подается документ с высоким приоритетом.

Общая модель РС включает в себя другие типы РС и несколько компьютеров обслуживания входных потоков документов без учета приоритетов. Время обработки документов является случайной величиной и зависит от экспоненциального их распределения. Отдельные РС данной модели моделируются с одинаковым распределением значений времени обслуживания.

Моделирование продвижения документов в ИС осуществляется по *модели маршрута*, созданной на основании сходства этапов обработки и применения приведенных трех типов моделей РС.

Методика автоматизированной поддержки процесса моделирования документооборота включает в себя процедуры и алгоритмы расчетов исходных данных и параметров информационных потоков. Значения характеристик моделирования документооборота используются при расчетах общих объемов баз данных ИС, времени, суммарной интенсивности их поступления и оценки необходимых ресурсов управления проектом документооборота в ИС. Она является базисом обследования документооборота в организации-разработчике ИС, обеспечивает выбор критериев принятия проектных решений и их оценку.

Полученные оценки используются в качестве исходных данных для построения формальной модели управления проектированием ИС.

10.3.3. Принципы и методы управления проектом документооборота в ИС

В настоящее время планирование и управление программным проектом, а именно его ресурсами (человеческими и материальными), бюджетом, временем и рисками – это главные задачи менеджмента проекта [24, 25]. Управление проектированием ИС выполняется с использованием принципов и методов математического программирования, стохастических сетевых моделей и моделей, построенных на собранных статистических данных.

Общая схема процесса управления проектированием (рис. 10.6) представлена схематически в виде двух процессов – исследование предметной области ИС и управление разработкой ИС, на которых осуществляется контроль и коррекция планов работ на проекте.

С учетом структуры и назначения процессов стандарта ISO/IEC 12207–96, разработан технологический процесс (ТП) для построения ПО документооборота. Из стандарта были выбраны такие процессы, действия и задачи, которые удовлетворяют рассматриваемой ПрО и каскадной модели ЖЦ.

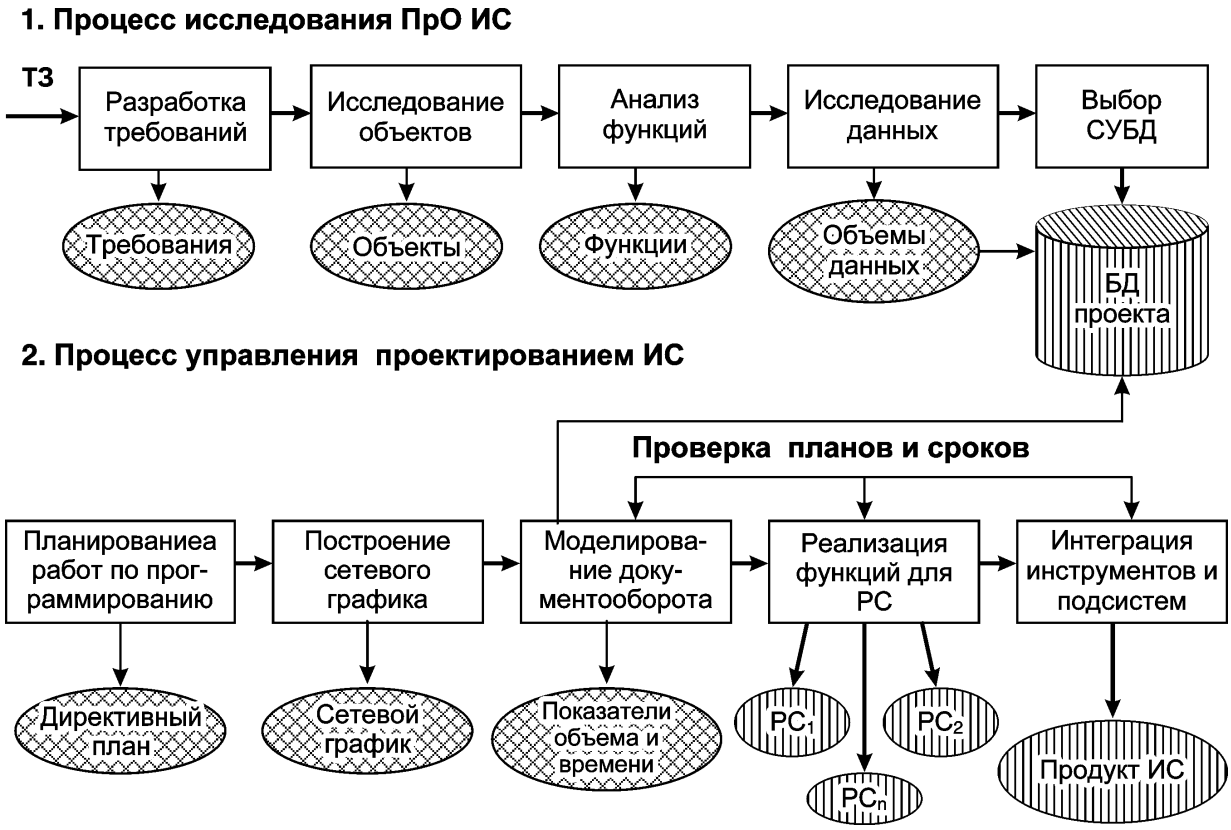

РИС. 10.6. Общая схема управления проектированием ИС

В результате сформирована модель ТП, которая объединяет выбранные процессы ЖЦ этого стандарта, целевой план проекта, методики оценки и сопоставления полученных результатов на процессах с фактически заданными базовыми данными проекта: ресурсы, время и стоимость [11, 31].

Данная модель построена по принципу итерационной модели, которая обеспечивает возвращение на предыдущие процессы для внесения изменений после нахождения ошибок или добавления/изменения требований к системе. В процессе выполнения плана работ B выполняется контроль, коррекция плана, а также параметров ТП.

Модель ТП задается в виде графа $G=\{Z_{ij}, l_j\}$, $ij = 0,..., n$, где l – дуга, Z_o – начало работ, Z_i – текущая работа, Z_n – конечная работа. Эта модель определена на следующих множествах параметров:

– набор элементарных работ $W = \{W_1 ,..., W_{n1}\}$;

– технические средства $S = \{S_1 ,..., S_{n2}\}$;

– признаки квалификации исполнителей $L = \{L_1 ,.., L_{n3}\}$ и вероятность $P = \{P_{ij}\}$, $ij = 1,..., n$, P_{ij} – вероятность возвращения к работе W_i, в вершине Z_j.

Вероятность переработки отдельных работ ИС, начиная с события в вершине Z_j, зависит от выявления ошибок, отказа технического средства S_i, изменения квалификации L_i и совокупности переходов, которые обусловлены состоянием технических средств, изменениями требований к ИС в процессе выполнения работы W_i.

Задача управления проектированием ИС определяется на таких данных:

– вариант плана X работ для проектирования компонентов ИС;

– укрупненный сетевой график $У$ выполнения работ, рассчитанный на последовательное их выполнение ($l_i \in L$);

– характеристики каждой l_i – работы относительно ее объема q_i и вида W_i;

– совокупность ресурсов $R =< R_L, R_S >$, трудовых R_L и материальных R_S, их виды и количество;

– нормы потребления ресурсов по видам работ $NR_i \in NR$;

– распределение случайных величин $F=\{F_1,..., F_r\}$ в зависимости от ошибок при выполнении работ, отказов в программах, сбоев технических средств и т.п.

Величина Y вычисляется исходя из планового периода $[t_0, T]$, вероятности P и таких характеристик ТП:

– вероятность окончания работы в заданный срок;

– объем необходимых ресурсов и оценка объема работ на ТП по формуле

$$Y = Y(X(B, R, L, NR), F, t_0, T). \qquad (10.1)$$

Предполагается, что вариант плана X принадлежит области D $(X \in D)$, а величина $K(X)$ – критерий оптимального варианта плана.

Задача нахождения оптимального варианта плана $X^* \in D$ выполняется при минимизации критерия

$$K(X^*) = \min K(X),\ X \in D \qquad (10.2)$$

Основные задачи плана X используют формулы (10.1), (10.2) и выполняются с помощью следующих шагов:

1) рассматривается вариант плана X, для которого исходные параметры находятся в области Y^D и удовлетворяют соотношению:

$$Y = Y(X) \in Y^D, \qquad (10.3);$$

2) выбирается комплекс работ B, оптимальный относительно заданного критерия K.

При выполнении плана работ на основе ТП проводится оперативный контроль и оценка расхождения между фактическим состоянием величин на ТП и значением параметров плана X в момент t. При расхождении, анализируется возможность возникновения разных нерегулярных ситуаций (сбой, болезнь исполнителя и т.п.), которые требуют возвращения к предыдущему этапу ТП для внесения изменений в процесс обработки документов или для корректировки плана с учетом значения X^* и соотношений (10.2) и (10.3).

Выбор оптимальных параметров ТП проводится исходя из заданных ресурсов, сроков и стоимости таким образом, чтобы проект был выполнен в директивный срок при условии, что реальные функции ИС моделируются, собирается статистика, оцениваются полученные выходные данные и сопоставляются с бюджетом для реализации проекта с минимальным риском.

Как показывает результат внедрения метода и технологии управления проектированием документооборота ИС, процесс создания ИС становится регламентированным, в нем обобщенна и упорядочена деятельность ее разработчиков, а также повышается технологическое и эксплуатационное качество документооборота ИС.

10.3.4. Пример управления проектом документооборота

Метод управления проектом использован при разработке ИС: система1 – Учет документов и контроль их выполнения и система2 – Автоматизированный банк данных нормативно-правового и методического обеспечения организации учебного процесса в общеобразовательных учебных заведениях через Интернет [11].

В системе1 применена методика моделирования документооборота и оценки информационных характеристик документов, вычислены технические параметры корпоративной сети и размер распределенной БД. С учетом этих расчетов определены параметры конфигурации сервера БД: размер пространства под БД для регулярной и нерегулярной частей документов и резервное копирование (экспорт БД, поддержка архива и др.).

В системе2 применены и апробированы принципы управления проектом. Был задан вариант плана X для выполнения комплекса работ с определением укрупненного сетевого графику b, и осуществлялось построение модели плана проекта на исходных данных.

Управление календарным планом проекта системы2 проводилось с применением Microsoft Office Project 2003 [3] (рис. 10.7).

Для оптимизации сетевого графика работ была рассчитана вероятность P наступления конечного события путем определения математического ожидания и дисперсии на исходных данных проекта. Получены значения достоверности P=0,47 в интервале [0,35; 0,65] и апробирован сетевой график, который не потребовалось оптимизировать Конечный срок разработки проекта отвечал плану, заданному в модели проекта.

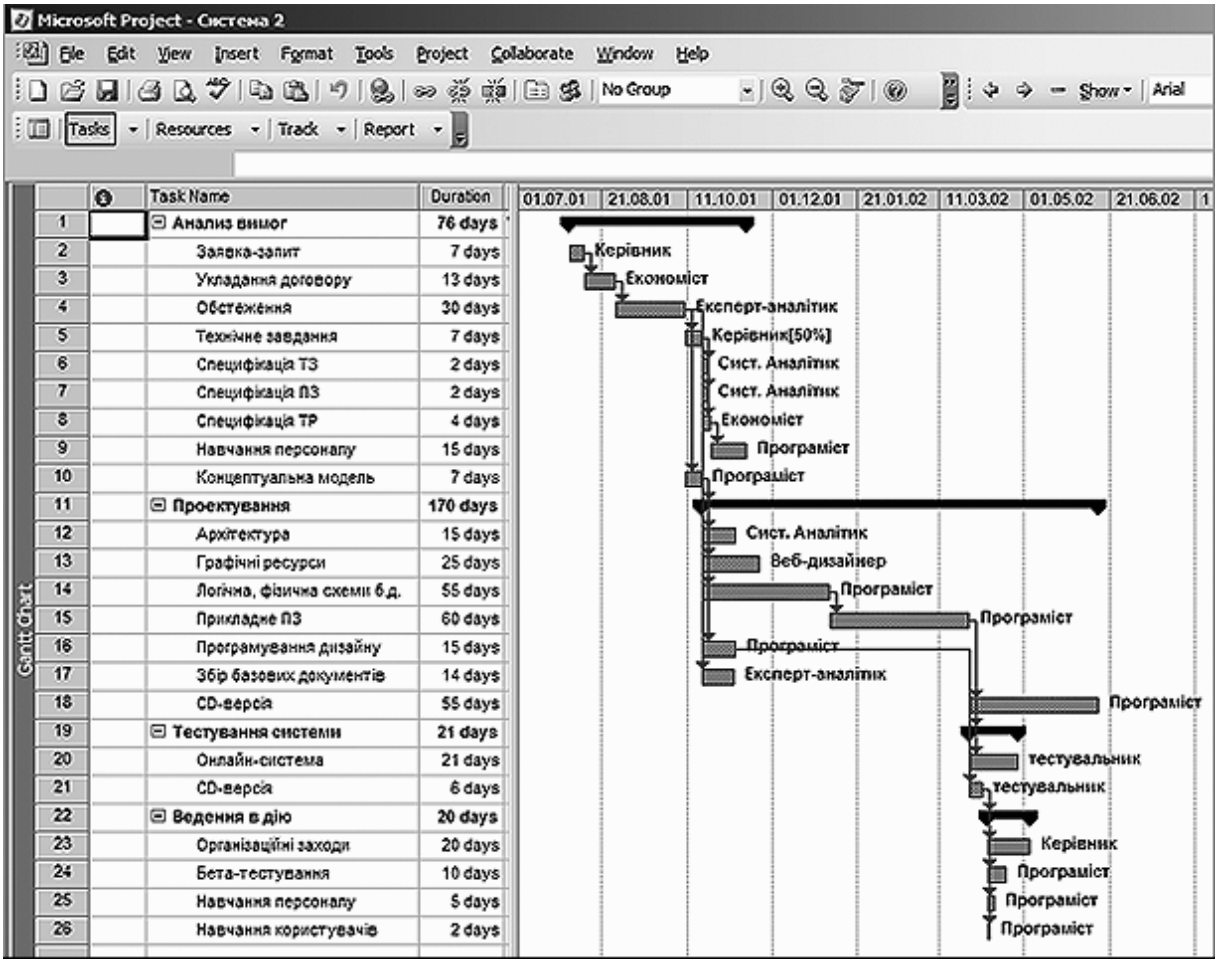

РИС. 10.7. Экранная форма процессов в управлении проектом

В главе изложена инженерия программных проектов, включающая в себя процессы планирования и управления проектом, а также процессы распределения сроков, стоимости и затрат так, чтобы проект был успешно выполнен. Рассмотрены методы выявления и устранения рисков как в плане подбора исполнителей, так и при использовании инструментальных средств. Приведены расчеты стоимости, затрат и времени выполнения проекта.

ГЛАВА 11

УПРАВЛЕНИЕ КОНФИГУРАЦИЕЙ

Под *конфигурацией* понимается конкретная версия программной системы для ОС, включающая в себя функции, объединенные между собой процедурами связи (или развертывания) и параметрами, задающими режимы функционирования системы [1–6]. Выпуск версии разных вариантов системы делается в целях поставки заказчику. Процесс получения конкретной версии системы можно представить в виде схемы (рис. 11.1). Выбираются компоненты системы из базы данных конфигурации, реализующие заданные функции, которые собираются компоновщиком системы, а система управления версиями создает исходную версию для компиляции и получения объектного кода, редактируемого в готовую версию.

Изготовленная версия системы проверяется на все виды связей для последовательного выполнения компонентов и получения контрольного результата.

Среди собираемых компонентов могут быть такие, которые изменялись и отдельно проверялись. Поэтому изготовленная версия системы должна обязательно тестироваться на тестовых наборах данных, а результаты сравниваться с ожидаемыми.

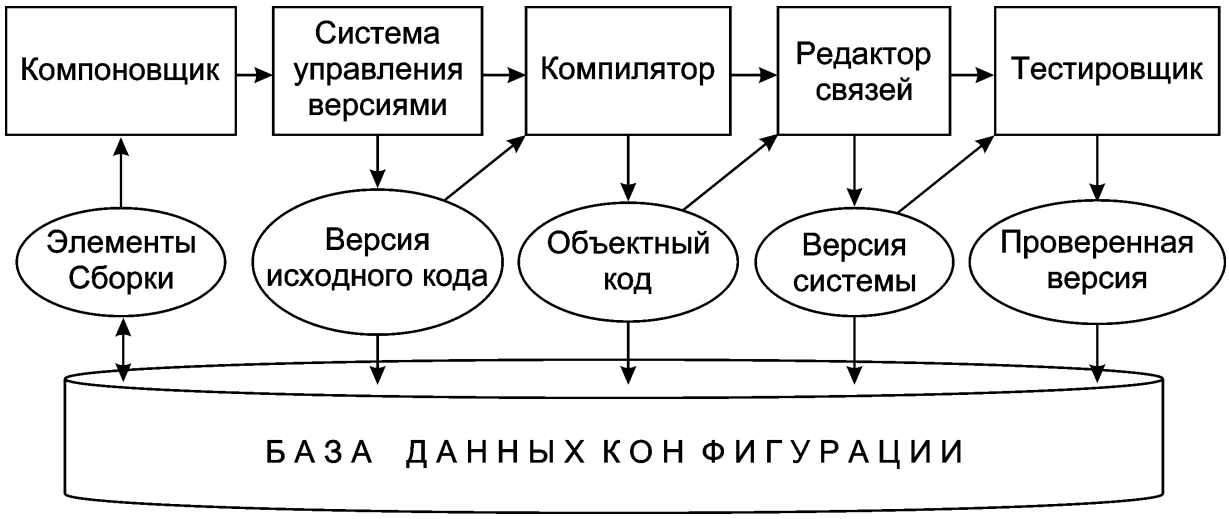

РИС. 11.1 Схема получения версии системы

Версия или конфигурация системы состоит из:

– базиса конфигурации – БК (Configuration Baseline) – формально созданной основы (версии) системы из отдельных компонентов и документации, позволяющей проводить дальнейшее развитие системы;

– элементов конфигурации (Configuration Item), выделенных для управления или обработки функций системы на процессорах компьютеров системы;

– программных компонентов, выполняющих задачи в сформированной версии системы.

Базис конфигурации определяет технические решения, перечень главных элементов конфигурации, значения параметров и специализированные процедуры связи и развертывания компонентов для функционирования версии системы в заданной последовательности. Чем больше в системе компонентов, тем больше вероятность того, что некоторые из них могут иметь ошибки. Это приводит к необходимости исправления обнаруженных ошибок, уточнений или дополнений как функций, так и технических средств (компьютеров, оборудования и др.).

*Управление конфигурацией (*Configuration Management*)* – состоит в наблюдении за модификацией параметров конфигурации и компонентов системы, а также в проведении систематического контроля, учета и аудита внесенных изменений, поддержки целостности и работоспособности системы.

Идентификация конфигурации – это именование всех элементов системы на основе схемы классификации и кодирования элементов, а также методов представления и ведения версий конфигурации с использованием входящих в нее элементов.

К элементам управления конфигурацией относятся технические и функциональные характеристики, схема развертывания и версия конфигурации. Цель управления конфигурацией – обеспечение целостности системы, контроль функционирования системы, наблюдение за производимыми изменениями структуры и элементов конфигурации.

Под *целостностью конфигурации* понимается способность системы воспроизводить и выполнять заданные функции после технических и функциональных изменений отдельных элементов и их повторного объединения в новую версию системы. *Наблюдение* – это процесс получения в любой момент сведений о структуре системы, времени ее работы, выполнении заданных функций и формирование ответов на поставленные запросы, касающиеся состояния текущей версии конфигурации системы.

11.1 УПРАВЛЕНИЕ КОНФИГУРАЦИЕЙ СИСТЕМЫ

Согласно действующему стандарту IEEE Std.828–90 управление конфигурацией обеспечивается следующими основными задачами:

1. Идентификация конфигурации (Configuration Identification).
2. Контроль конфигурации (Configuration Control).
3. Учет статуса конфигурации (Configuration Status Accounting).
4. Аудит конфигурации аудит (Configuration Audit).

Управление конфигурацией (УК) для больших систем создается с помощью методов и средств, обеспечивающих идентификацию элементов этой системы, контроль вносимых изменений и возможность определения фактического состояния системы при разработке и эксплуатации в любой момент времени. Управление конфигурацией базируется на точной и достоверной информации о состоянии системы и планах проведения изменений.

С формальной точки зрения управление конфигурацией – это дисциплина управления и методов наблюдения за документированными функциональными и техническими характеристиками элементов системы, а также методы управления изменениями, подготовка отчетов по выполненным изменениям и их проверка на соответствие поставленным требованиям.

Работы по управлению конфигурацией, как правило, выполняет специальная служба, которая определяет возможные ограничения на функционирование системы в заданных условиях операционной среды, планирование внесения изменений, проверку разных частей системы, сбор данных и учет внесенных изменений в систему и конфигурацию. К деятельности этой службы относится также управление проектом, контроль качества и целостности конфигурации системы и ее сопровождение.

Структура службы зависит от сложности системы, этапов развития проекта и от специалистов организации-разработчика системы и заказчика. От хорошей организации работы службы зависит эффективность управления конфигурацией.

Взаимосвязь видов деятельности по управлению конфигурацией (УК) представлена рис. 11.2

Результатом управления конфигурацией является отчет о проведенных изменениях версии системы и документации, а также документ о передаче измененной версии пользователю.

Достижение целей управление конфигурацией должно планироваться и выполняться с учетом возникающих ограничений новой ОС и аппаратных возможностей компьютеров у заказчика.

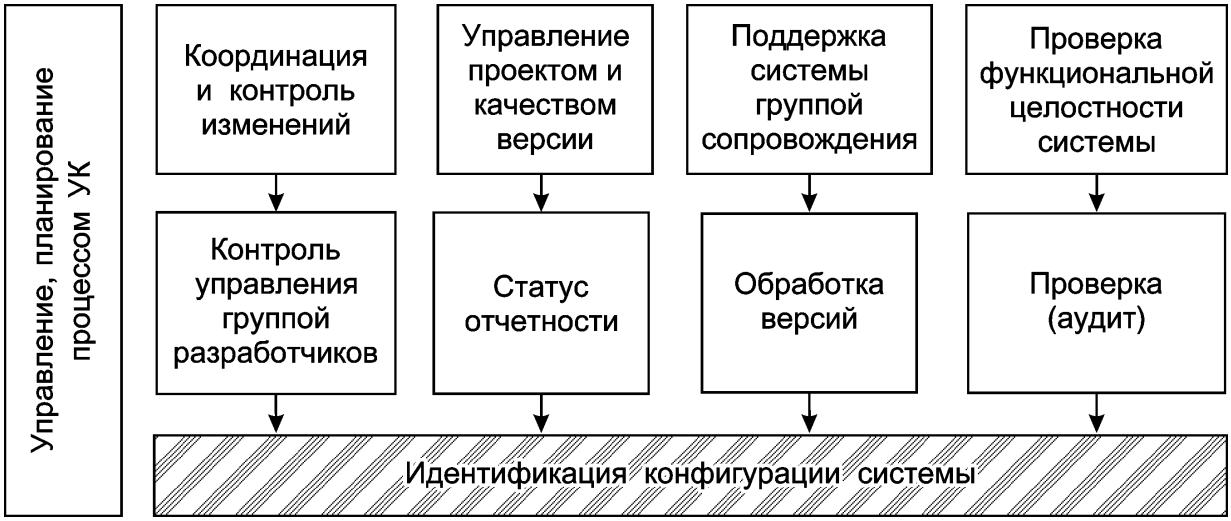

РИС. 11.2 Виды деятельности управления конфигурацией

Процессом планирования занимаются менеджеры службы управления проектом. Предложения на изменение компонентов системы подаются в эту службу для проведения анализа и определения целесообразности внесения изменений в версию системы и ее конфигурацию, оценку стоимости этих работ, разработку предложений в виде утвержденного перечня изменений для их реализации.

Процесс изменений включает в себя определение типов изменений, организацию их проведения и формирование концепции допуска отклонений и отказов относительно требований проекта системы.

Результатом внесения изменений является новая версия системы, документация по проведению на ней испытаний и пользовательская документация на систему.

Заказчик оценивает предложения на внесение изменений и дает разрешение на проведение наиболее важных изменений, влияющих на ее технические характеристики или стоимость. Анализ и контроль проведения изменений конфигурации системы проводит специальная группа службы управления. Она выполняет систематический учёт и контроль внесения изменений на всех этапах ЖЦ.

План изменений в конфигурацию системы утверждается формальными процедурами, расчетами оценок влияния изменений на стоимость, принятием решений об изменениях или отказа от них. Запросы на внесение изменений выполняются в соответствии с процедурами разработки системы на этапах ЖЦ или на этапе сопровождения системы. Поскольку требуемые изменения

могут проводиться одновременно с разработкой, предусматривается трассирование изменений при построении новых версий. Каждое проведенное изменение подвергается детальному аудиту.

За внесением изменений проводится контроль текущей версии системы, т.е. выходного кода полученной версии с помощью инструментальных средств типа Rational's ClearCase, SourceSafe of Microsoft системы Unix и др.

После завершения изменений и испытания системы проводится тиражирование системы и документации для передачи системы и ее конфигурации заказчику.

В конфигурацию системы входят сведения об аппаратных и программных элементах системы. При этом на систему могут накладываться ограничения с учетом контрактов с заказчиком, аудитами, информации из разных источников (спецификации требований, описаний, отчетов и др.), а также состава инструментальных средств и рекомендаций государственных или межведомственных стандартов.

11.2. ПЛАНИРОВАНИЕ УПРАВЛЕНИЕМ КОНФИГУРАЦИЕЙ

Планирование зависит от типа проекта, организационных мероприятий, ограничений и общих рекомендаций по руководству конфигурацией. К видам планирования управления конфигурацией системы относятся: идентификация, определение статуса и аудита конфигурации, управление изменениями конфигурации (рис. 11.3).

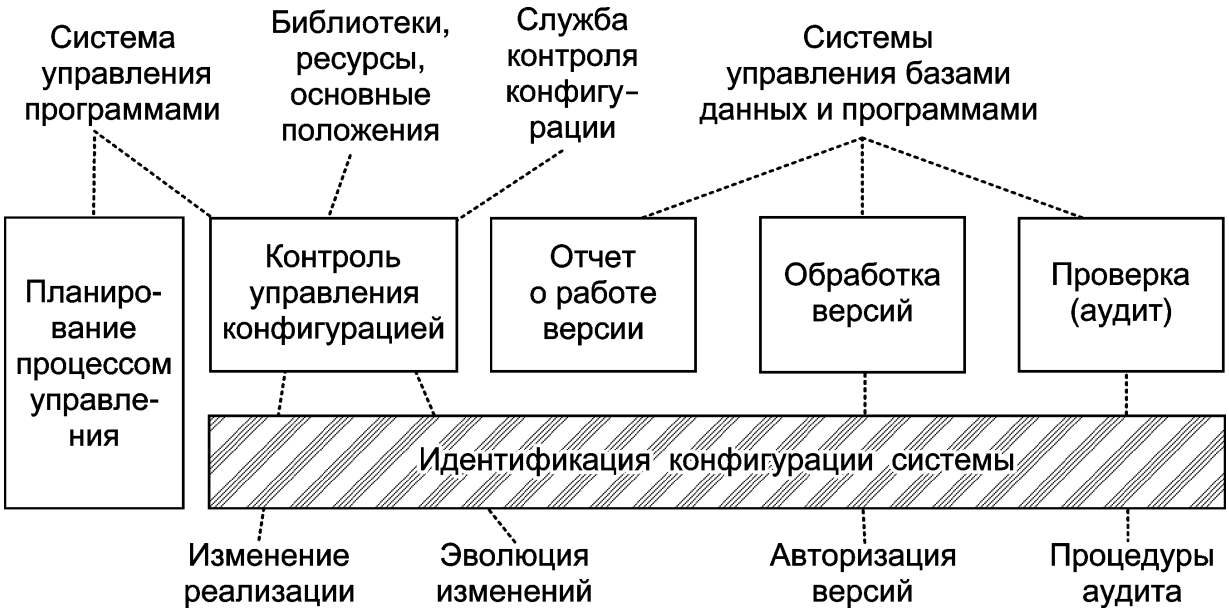

Рис. 11.3. Инструменты и процедуры управления конфигурацией

При планировании составляются планы, выбираются инструменты, анализируются требования проекта, интерфейсы компонентов и т.п. К средствам планирования относятся:

– система управления кодами компонентов, их переводом и объединением в конфигурацию системы;

– базовые библиотеки и ресурсы;

– специальные группы контроля системы и ее конфигурации;

– СУБД для ведения проекта и хранения изменений в систему.

К основным задачам планирования относятся:

– фиксация разных заданий на изменения и выбор инструментария для их выполнения;

– определение человеко-часов и инструментальных ресурсов, стандартов, затрат на внесение изменений и др.;

– установление связей с заказчиком для проведения контроля системы и конфигурации, а также проведение оценки системы;

– определение последовательности работ управлением конфигурацией.

Результаты планирования отмечаются в плане управления конфигурацией проекта, а также в документе внесения изменений в версию, конфигурацию или систему.

11.3. ИДЕНТИФИКАЦИЯ ЭЛЕМЕНТОВ КОНФИГУРАЦИИ

Идентификация элементов конфигурации выполняется с помощью методов структуризации, классификации и именования элементов системы и ее версий. При проведении идентификации проводится:

– определение стратегии идентификации для получения учтенной версии системы;

– именование составных элементов частей и всей конфигурации системы;

– установление соотношения между количеством выполняемых задач и количеством пунктов конфигурации;

– ведение версии системы (или ее частей) и документирование;

– выбор элементов базиса конфигурации и его формальное обозначение.

При идентификации используется библиотека элементов, версий и изменений системы. Основу идентификации составляет конфигурационный базис – набор формально рассмотренной и утвержденной конфигурационной документации, как основы для дальнейшего развития или разработки системы.

Выделение в продукте контролируемых единиц конфигурации – сложная задача и, как правило, является составной частью процесса высокоуровневого или архитектурного проектирования и выполняется системными архитекторами. Построение адекватной схемы классификации и идентификации объектов конфигурационного управления выполняется одновременно со структуризацией продукта и заключается в определении правил уникальной идентификации (кодирования, маркирования):

– конфигурации продукта и ее версий;

– контролируемых единиц конфигурации и их версий;

– составляющих конфигурационного базиса и их редакций.

Результат создания и применения схемы идентификации дает возможность отличать разные продукты друг от друга и версии продуктов между собой.

11.4. УПРАВЛЕНИЕ ВЕРСИЯМИ

Версия (конфигурация) системы включает в себя элементы конфигурации и коды системы для передачи получателю [6, 7]. Управление версиями состоит в выполнении действий:

– *интеграции* или композиции корректной и окончательной структуры системы из элементов конфигурации, которые реализованы на этапах ЖЦ;

– *выбора инструментария* построения версии, оценки возможностей среды и средств процесса построения отдельных версий программного обеспечения и данных;

– *управления вариантами версий* из идентифицированных элементов системы, удовлетворяющих заданным требованиям заказчика на систему.

При формировании версий системы учитываются ограничения и требования на разработку элементов конфигурации системы. Например, решений об изменении конфигурации способом, который не совпадает с предложенным и согласованным заказчиком. Когда новая версия системы получена, тогда заказчику передаются версия, документация и средства управления версиями для внесения изменений в элементы системы в процессе ее сопровождения.

Пример. Формирование 21 текущих версий ОС 360 (1965-1980 гг.) осуществлялось на фирме IBM. В ОС постоянно и поэтапно добавлялись новые функциональные возможности и вно-

сились изменения в предыдущую версию при ее эксплуатации. Над развитием дополнительных возможностей данной ОС и внесением изменений в предыдущую версию постоянно работал коллектив фирмы. Трудоемкость разработки очередной версии ОС считалась пропорциональной интервалу времени между регистрациями очередных версий и принималась за единицу измерения сложности создания новой версии [7].

В качестве меры трудоемкости сопровождения и создания очередной версии использовалось число модулей (ограниченных размеров), подвергающихся изменениям и дополнениям. Кроме того, оценивалась интенсивность работ по созданию версии, которая измерялась числом измененных модулей в единицу времени. После 12 лет постоянных изменений в ОС 21 версия работала более стабильно, в нее почти не вносились изменения, так как претензий со стороны пользователей в основном не поступало.

Метрический анализ процесса развития ОС 360 позволил установить, что объем среднего прироста системы на каждую версию соответствовал примерно 200 модулям. При этом общий объем увеличивался от 1 тыс. модулей в первых версиях до 5 тыс. модулей в последних версиях. Когда уровень прироста сложности был большим, тогда для устранения ошибок или дополнительных корректировок иногда создавались промежуточные версии с меньшим числом изменений.

В результате появилось понятие «критической» массы или «критической» сложности модифицируемой системы. Если при модернизации и выпуске очередной версии системы объем доработок превышает «критический», то возрастает вероятность ухудшения характеристик системы или необходимость введения промежуточной версии с внесением некоторых изменений. «Критический» объем доработок ОС–360 около 200 модулей оставался постоянным, несмотря на рост квалификации коллектива, совершенствование технических и программных средств и др. В первых версиях объем доработок составлял 20% модулей, а в последних версиях снизился до 5%.

11.5. КОНТРОЛЬ КОНФИГУРАЦИИ

Под контролем конфигурации принято понимать управление изменениями в ходе эксплуатации системы, при котором вносятся непрерывные корректировки, которые имеют отношение к согласованному и/или утвержденному конфигурационному базису (КБ). Предметом контроля конфигурации являются:

– изменения в утвержденном КБ и связанные с ними корректировки в конфигурации и/или в ее элементах;

– ошибки и отклонения в конфигурации относительно утвержденного КБ.

Процедуры инициализации, анализа, принятия и контроля исполнения управленческих решений по предложенным изменениям, обнаруженным ошибкам и отклонениям в конфигурации выполняются формально.

Формальная обработка запросов на изменение КБ. После того, как заинтересованные участники проекта достигли взаимопонимания по требованиям, архитектуре и другим техническим решениям, соответствующие проектные документы считаются утвержденными и не могут произвольно модифицироваться. Иными словами, любая потребность в изменении, исходящая от любого участника проекта, должна пройти формальную процедуру, включающую в себя такие шаги:

1. Регистрация предложения /запроса на изменение.
2. Анализ влияния предложенного изменения на имеющийся задел, объем, трудоемкость, график и стоимость работ по проекту.
3. Принятие решения по запросу на изменение (удовлетворить, отказать или отложить).
4. Реализация утвержденного изменения и верификация системы.

Управление ошибками и отклонениями от утвержденного КБ. Второй важнейшей составляющей контроля конфигурации является проверка несоответствия конфигурации или элементов продукта конфигурационному базису. С точки зрения управления все несоответствия принято делить на ошибки и отклонения. К ошибкам относят несоответствия, которые имеют непосредственное отношение к целевому использованию продукта по его назначению. Все остальное относится к отклонениям. Если дефекты в продукте носят негативный характер, то они подлежат устранению.

Для устранения дефектов и выявленных отклонений проводится:

– регистрация информации о полученном дефекте /отклонении;

– анализ и диагностика места и причины дефекта /отклонения, оценка трудоемкости, сроков и стоимости переделок;

– принятие решения по устранению этих недостатков и их верификация.

Подобного рода действия являются управленческими, принятие решения по изменению программного продукта должен быть принят в проекте на уровне согласованных или утвержденных документов КБ. Формой реализации такого управленческого решения являются руководящий совет по конфигурационному контролю (Configuration Control Board).

11.6. УЧЕТ СТАТУСА КОНФИГУРАЦИИ

Суть этого учета состоит в регистрации и предоставлении информации управления конфигурацией. Предметом учета является информация о текущем статусе идентифицированных объектов конфигурационного управления, предложенных изменениях, а также о выявленных дефектах и отклонениях от утвержденного конфигурационного базиса.

Отчетность по статусу конфигурации является ключевым фактором принятия управленческих решений по проекту информационной системы или ПО. Более того, оперативно регистрируемые и регулярно обновляемые данные учета статуса конфигурации являются исходным материалом для формирования количественных оценок или метрик производительности и качества работ на проекте. Применение этих метрик позволяет принимать эффективные управленческие решения по созданию программного проекта.

В системе учета статуса конфигурации накапливаются сводные отчеты о количестве обнаруженных и исправленных дефектов, поступивших и реализованных запросов на изменения, динамике внесения изменений в конфигурацию продукта во времени и др. Отчетами практически пользуются все участники проекта: заказчики, аналитики, разработчики, тестировщики, внедренцы, служба качества и руководство проекта. На их основе проводится количественная оценка производительности и качества работ по проекту.

11.7. АУДИТ КОНФИГУРАЦИИ

Аудит – это ревизия или проверка системы перед выпуском очередной ее версии или перед ее сдачей очередному заказчику. Кроме того, в обоих случаях аудиторская работа большей частью связана с рассмотрением и оценкой документации – данных, сводок, отчетов.

Конфигурационный аудит производится непосредственно перед выходом новой версии продукта, его части и включает в себя:

– функциональный аудит конфигурации, т.е. подтверждение соответствия функциональных характеристик конфигурации предъявляемым требованиям;

– физический аудит конфигурации, т.е. подтверждение взаимного соответствия документации и фактической версии продукта.

Функциональный аудит – это не верификация или валидация продукта, а проверка того, что тестирование проведено в установленном объеме, результаты документированы и подтверждают соответствие функций и характеристик продукта предъявляемым требованиям. При этом все изменения реализованы, критичные дефекты устранены, а по всем выявленным отклонениям в конфигурационном базисе приняты адекватные решения.

Физический аудит – это сверка выпущенного продукта документам конфигурационного базиса, а также проверка того, что данная конфигурация построена в соответствии с установленными процедурами и из корректных компонентов. Конфигурационный аудит проводится независимыми экспертами, например, представителями службы качества.

Таким образом, эффективное управление конфигурацией – это системная работа на проекте, требующая определенной инструментальной поддержки для проведения идентификации, изменений, сбора сведений о дефектах, ошибках и отклонениях.

ЗАКЛЮЧЕНИЕ

ПЕРСПЕКТИВЫ РАЗВИТИЯ ПРОГРАММИРОВАНИЯ

Постоянное использование языков и методов программирования, инженерных методов управления проектом, качеством и выходной конфигурационной структурой, а также инструментальных средств их поддержки естественно приводит к необходимости их совершенствования и поиска новых путей их развития. Ключевыми направлениями программной инженерии являются инженерия приложений и доменов, базис которых – готовые ПИК, многоразовые системы и инструменты их поддержки. В них воплощаются многие теоретические, прикладные и инженерные аспекты программирования, что в значительной степени оказывает влияние на рост производства и повышение качества ПС. Вместе с тем появляется много новых перспективных идей и проектов, которые должны изменить характер технологии разработки программных проектов на ближайшие десятилетия. Приведем некоторые из них.

1. В разработке архитектур систем новыми концепциями будут:

– независимое от среды и платформы описание моделей MDA (model driven architecture) и PIM (Platform independent model) и их трансформация к платформо-зависимой модели среды .Net, J2EE; описание среды и доступа к сервисам любых приложений в языках типа XML – базис сервисно-компонентной архитектуры веб-сервисов в среде Интернета;

– общий двухфазовый процесс производства больших и сложных систем из одиночных программ, сгенерированных на первой фазе производства – инженерии приложений, и многоразового применения проектных решений, ПИК, членов семейства ПС. На второй фазе производства – инженерия ПрО, будет формироваться сервисно- и компонентно-ориентированный **архитектурный базис** во множестве сервисов, ПИК, учитывающих общие и специфические особенности разных доменов и накапливаемых в репозитарии общего пользования;

– генерирующая модель домена (generative domain model) для производства программных приложений и ПрО по принципу

конвейера со всеми атрибутами планирования, управления, измерения и оценивания продуктов и процессов. Система качества (Quality systems – QS) при этом займет ведущую роль.

2. В программировании новыми перспективами развития станут:

– специальные функции и компоненты изменчивости, синхронизации и безопасности и интерфейсы ПрО для связи с разными средами;

– языки описания аспектов, независимые от компонентов и предназначенные для решения задач защиты, безопасности и взаимодействия компонентов и др.;

– повышение компетенции агентов для применения на процессах ЖЦ в целях управления тестированием, оцениванием показателей качество и разных видов затрат др.;

– новые языки описания специфики доменов (Domain Specific Languages, Language Workbench) и расширенное применение для этих целей языков UML и XML;

– языково-ориентированные средства, анализаторы или интерпретаторы, которые будут инструментом трансформации DSL описания компонентов многоразового использования в XML;

– пользователи, активно принимающие участие в описании известных им частей создаваемых систем, будут приносить значительную выгоду при их внедрении.

3. Проблемы доказательства и верификации станут главными в программировании в 15-летнем международном проекте (идея Т. Хоара в статье журнала «Открытые системы», 2006, №6) представлены следующими перспективными задачами:

– разработка единой теории построения и анализа программ;

– построение многостороннего интегрированного набора инструментов верификации на всех производственных процессах – разработка формальных спецификаций, их доказательство или проверка правильности, генерация программ и тестовых примеров, уточнение, анализ и оценка;

– создание репозитария формальных спецификаций, верифицированных программных объектов разных типов и видов.

Формальные методы верификации будут охватывать все аспекты создания и проверки правильности программ, что приведет к созданию мощной верификационной производственной основы и значительному сокращению ошибок.

На репозитарий возлагаются такие глобальные задачи:

– накопление верифицированных спецификаций, методов доказательства, программ и реализаций кодов для сложных приложений;

– хранение всевозможных методов верификации, их формализация и стандартизация для поиска и выбора необходимого теоретического базиса для применения;

– разработка стандартных форм для задания и обмена формальными спецификациями между разными объектами и инструментами программирования;

– разработка механизмов интероперабельности и взаимодействия для переноса готовых верифицированных продуктов из репозитария в современные среды производства верифицированных систем.

Авторы международного проекта предполагают, что его развитие будет продолжаться в течение 50 лет, что приведет к коренным изменениям в программировании, особенно при достижении высокого качества верифицированных программных систем.

ПРИЛОЖЕНИЕ 1

КРАТКОЕ ОПИСАНИЕ ОБЛАСТЕЙ ЗНАНИЙ – SWEBOK

Специальный комитет, организованный ACM (Association for Computing Machinery) и компьютерным союзом института инженеров по электронике и электротехнике (IEEE Computer Society), создал (2001–2004 г.) ядро знаний SWEBOK (Software Engineering Body Knowledge) программной инженерии из 10 областей и дал следующее определение [1–3]:

программная инженерия – это система методов, способов и дисциплины планирования, разработки, эксплуатации и сопровождения программного обеспечения, которая способна к массовому воспроизводству. Таким образом, программная инженерия – это инженерная дисциплина, которая охватывает все аспекты создания ПО от разработки требований до его использования. В нее включены как ключевые разделы – планирование и сопровождение. Планирование как анализ задач разработки, их распределение и выделение необходимых ресурсов, а сопровождение как устранение найденных недостатков в системе и внесение необходимых изменений.

Каждая область в ядре знаний представлена по единой схеме: понятийный аппарат, методы, средства и инструменты. В рамках этой схемы дается краткая характеристика зафиксированных 10 областей SWEBOK.

1.1. РАЗРАБОТКА ТРЕБОВАНИЙ К ПО

Требования – это свойства, которыми должно обладать ПО для адекватного выполнения предписанных функций, а также условия и ограничения на ПО, данные, ОС и техническое обеспечение. Требования отражают потребности заказчиков, пользователей и разработчиков, заинтересованных в создании ПО.

Область знаний «Разработка требований к ПО (Software Requirements)» состоит из следующих разделов:

- инженерия требований (Requirement Engineering),
- выявление требований (Requirement Elicitation),
- анализ требований (Requirement Analysis),
- спецификация требований (Requirement Specification).
- проверка требований (Requirement validation),
- управление требованиями (Requirement Management).

Инженерия требований к ПО – это дисциплина анализа и документирования требований, которая заключается в преобразовании предложенных заказчиком требований к системе в описание требований к ПО и в их верификации. В основе инженерии лежат модели качества ПО, модели процессов и актеров – действующих лиц, обеспечивающих формирование, планирование и управление требованиями.

Модель процессов – это схема последовательности процессов, которые выполняются от начала проекта и до определения и согласования требований. Процессом является также маркетинг и проверка осуществимости требований.

Управление требованиями к ПО заключается в планировании и контроле выполнения требований в процессе разработки программной системы на этапах ЖЦ.

Модель качества определяет процесс улучшения качества требований к ПО для получения характеристик качества (надежность, реактивность и др.) на этапах ЖЦ.

Выявление требований – это процесс формирования требований из первоисточников, включающий в себя техники выявления (собеседование, сценарии, прототипы, собрания и др.) и мероприятия по идентификации интересов заказчика и разработчика в виде требований к разработке ПО.

Анализ требований – процесс изучения потребностей и целей пользователей, классификация их по требованиям к системе, аппаратуре и ПО, разрешение конфликтов между требованиями, определение границ системы и принципов взаимодействия ПО со средой окружения. Требования классифицируются по функциональному и нефункциональному принципу. *Функциональные требования* отражают функции, которые будет выполнять система или ее ПО, а также поведение ПО при преобразовании входных данных в результаты. *Нефункциональные требования* – это требования, которые определяют условия и среду выполнения функций (защита, доступ к БД, секретность, и др.). Разработка требований и их локализация завершается на этапе проектирования архитектуры и отражается в специальном документе, по которому проводится согласование зафиксированных требований между заказчиком и разработчиком.

Спецификация требований – процесс формализованного описания функциональных и нефункциональных требований, требований к характеристикам качества в соответствии со стандартами качества (ISO 9126–98, ISO/IEC 12119–94), которые должны будут получены на этапах ЖЦ процесса разработки ПО. В спецификации отражаются требования к функциям, качеству и к документации, а также задается в общих чертах архитектура системы и ПО, алгоритмы, логика управления и структура данных. Специфицируются также системные требования, нефункциональные требования и требования к взаимодействию с другими компонентами (БД, СУБД, передача данных, сетевое взаимодействие и др.).

Валидация (аттестация) требований – это проверка требований, изложенных в спецификации, и реализованных в системе и ПО. Заказчик и разработчик ПО проводят экспертизу варианта требований с тем, чтобы продолжить разработку ПО. *Верификация требований* – это процесс проверки правильности требований на их соответствие, непротиворечивость, полноту и выполнимость, а также на соответствие стандартам. В результате проверки требований делается согласованный выходной документ, устанавливающий полноту и корректность требований к ПО, и

обеспечивается продолжение проектирования ПО. Одним из методов аттестации является прототипирование, т.е. быстрая отработка отдельных требований на конкретном инструменте.

Управление требованиями – это руководство процессами формирования требований на всех этапах ЖЦ, которое включает в себя управление изменениями требований и проведение мониторинга – восстановление источника требований. *Трассирование требований* состоит в отслеживании отдельных требований к системе на этапах ЖЦ, и наоборот от продукта к требованиям.

1. 2. ПРОЕКТИРОВАНИЕ ПО

Проектирование ПО – процесс определения архитектуры, компонентов, интерфейсов, других характеристик системы и конечного результата.

Область знаний «Проектирование ПО (Software Design)» состоит из следующих разделов:

– базовые концепции проектирования ПО (Software Design Basic Concepts),

– ключевые вопросы проектирования ПО (Key Issue in Software Design),

– структура и архитектура ПО (Software Structure and Architecture),

– анализ качества проектирования ПО (Software Design Quality Analysis and Evaluation),

– нотации проектирования ПО (Software Design Notations),

– стратегия и методы проектирования ПО (Software Design Strategies and Methods).

К базовым концепциям проектирования ПО относятся методы проектирования архитектуры с использованием принципов (структурного, объектного, компонентного и др.) и техник абстракции, декомпозиции, инкапсуляции и др. Проектируемая система декомпозируется на отдельные компоненты, осуществляется выбор готовых артефактов (нотации, методы и др.) и компонентов и на их основе создается архитектура ПО.

Ключевые вопросы проектирования ПО – это декомпозиция на функциональные компоненты для независимого и параллельного их выполнения; принципы распределения компонентов и способов их взаимодействия между собой в операционной среде, а также механизмы обеспечения качества, живучести системы и др.

Проектирование структуры и архитектуры ПО выполняется архитектурным стилем путем определения основных элементов архитектуры – подсистемы, паттерны, компоненты и связи между ними.

Архитектура проекта – высокоуровневое представление структуры, задаваемое с помощью паттернов, компонентов и их идентификации. В описание архитектуры входит описание логики отдельных компонентов системы и связей между ними. Существуют и другие виды структур ПО, основанные на проектировании образцов, семейств программ и их каркасов.

Паттерн – это элемент структуры, в котором задается взаимодействие совокупности элементов проектируемой системы с определением ролей и ответственности актеров средствами языка UML. *Структурный паттерн включает* в себя типовые композиции объектов и классов, задаваемых с помощью диаграмм классов, объектов, связей и др. *Поведенческий* – схемы взаимодействия классов объектов и их поведение, задаваемые диаграммами активностей, взаимодействия, потоков управления и др.

Анализ и оценка качества проектирования ПО – это системные мероприятия по анализу атрибутов качества, сформулированных в требованиях, оценка различных характеристик ПО (размер, число функций и др.) с применением функционально-ориентированных, структурных и объектно-ориентированных метрик, а также методы статического анализа, моделирования и прототипирования архитектуры ПО.

Нотации проектирования – средства представления артефактов ПО, его структуры и поведения. Существует два типа нотаций: структурные и поведенческие, а также множество различных их представлений.

Структурные нотации – графические способы представления аспектов проектирования, компонентов и их взаимосвязей, элементов архитектуры и их интерфейсов. К нотациям относятся языки спецификаций и проектирования: ADL (Architecture Description Language), UML (Unified Modeling Language), ERD (Entity–Relation Diagrams), IDL (Interface Description Language), классы и объекты, компоненты и классы (CRC Cards), Use Case Driven и др.

Поведенческие нотации отражают динамический аспект поведения систем и их компонентов и задаются с помощью диаграмм: Data Flow, Decision Tables, Activity, Collaboration, Pre-Post Conditions, Sequence и др.

К стратегиям и методам проектирования ПО относятся: методы снизу-вверх, сверху-вниз, абстракции, паттерны и др. Методы проектирования включают в себя структурный анализ, структурные карты, Dataflow-диаграммы, абстрактные структуры данных (диаграммы Джексона) и др.

1.3. КОНСТРУИРОВАНИЕ ПО

Конструирование ПО – создание работающего ПО с привлечением методов кодирования, верификации и тестирования компонентов. К средствам конструирования ПО отнесены языки программирования (ЯП), а также методы и инструментальные системы (компиляторы, СУБД, генераторы отчетов, системы управления версиями, конфигурацией, тестированием и др.). К формальным средствам описания ПО, взаимосвязей между человеком, компьютером и средой окружения отнесены языки конструирования ПО.

Область знаний «Конструирование ПО (Software Construction)» включает в себя следующие разделы:

- снижение сложности (Reduction in Complexity),
- предупреждение отклонений от стиля (Anticipation of Diversity),
- структуризация для проверок (Structuring for Validation),
- использование внешних стандартов (Use of External Standards)

Снижение сложности конструирования ПО и предупреждение отклонений от стиля (лингвистического, формального, визуального и др.) обеспечивается с помощью подходящих стилей конструирования, структуризации ПО и внешних стандартов.

Лингвистический стиль основан на использовании словесных инструкций и выражений для представлений отдельных элементов (конструкций) программ. Он применяется при конструировании несложных конструкций и приводится к виду традиционных функций и процедур, логическому и функциональному их программированию и др.

Формальный стиль используется для точного, однозначного и формального определения компонентов системы. В результате его применения обеспечивается конструирование сложных систем с минимальным количеством ошибок, которые могут возникнуть в связи с неоднозначностью определений или обобщений при неформальном конструировании ПО.

Визуальный стиль является наиболее наглядным стилем конструирования ПО. Например, графический интерфейс освобождает разработчика от подбора необходимых координат и свойств объектов интерфейса. Визуальный язык проектирования UML предоставляет набор удобных диаграмм для задания статической и динамической структуры ПО, текстовое и диаграммное описание структуры ПО с выводом на экран дисплея. В процессе конструирования должны использоваться внешние стандарты ЯП (Ада 95, C++, Паскаль и др.), языков описания данных (XML, SQL и др.), средств коммуникации (COM, CORBA и др.), интерфейсов компонентов (POSIX, IDL, APL), сценариев UML и др.

1.4. ТЕСТИРОВАНИЕ ПО

Тестирование ПО – это процесс проверки работы программы в ОС, основанный на выполнении набора тестовых данных и сравнения полученных результатов с ожидаемыми результатами.

Область знаний «Тестирование ПО (Software Testing)» состоит из следующих разделов:

– основные концепции и определение тестирования (Testing Basic Concepts and Definitions),
– уровни тестирования (Test Levels),
– техники тестирования (Test Techniques),
– метрики тестирования (Test Related Measures),
– управление процессом тестирования (Managing the Test Process).

К основным концепциям относятся терминология, теории и инструменты и принципы организации подготовки и проведения процесса тестирования ПО, а также подходы к аналитической оценки данных об ошибках, собранных в процессе тестирования.

Уровни тестирования:

– *тестирование отдельных элементов,* которое заключается в проверке отдельных, изолированных и независимых частей ПО;

– *интеграционное тестирование* ориентировано на проверку связей и способов взаимодействия (интерфейсов) отдельных компонентов, в том числе расположенных по разным компьютерам распределенной среды;

– *тестирование системы* состоит в проверке функционирования системы, обнаружении отказов и дефектов в ней и их устранение. При этом проводится контроль выполнения нефункциональных требований (безопасность, надежность и др.) к системе, правильность задания и выполнения внешних интерфейсов с операционной средой.

Видами тестирования являются:

– *функциональное тестирование*, которое заключается в проверке соответствия реализованных функций и заданных в требованиях;

– *регрессионное тестирование* – тестирование системы или ее компонентов после внесения в них изменений;

– *тестирование эффективности* – проверка производительности, пропускной способности, максимального объема данных и системных ограничений в соответствии с требованиями;

– *стресс тестирование* – проверка поведения системы при максимально допустимой нагрузке или при превышении;

– *альфа и бета-тестирование* – внутреннее и внешнее тестирование системы. Альфа – без плана, бета – с планом тестирования;

– *тестирование конфигурации* – проверка структуры системы с различными наборами конфигурационных данных.

К техникам тестирования относится тестирование:

– типа *«белого ящика»,* основанное на задании информации о структуре системы;

– типа *«черного ящика»,* основанное на задании тестовых наборов данных для проверки правильности работы компонентов и системы в целом, и не требующее знания их структуры;

– на основе спецификаций, таблиц решений, потоков данных, статистики отказов и др.

Метрики тестирования предназначены для измерения процесса планирования, тестирования и оценки результатов тестирования на основе статистики об отказах и дефектах, покрытия границ тестирования, потоков данных и др.

Управление тестированием заключается в планировании процесса тестирования и измерения показателей качества ПО; в проведении тестирования reuse-компонентов и паттернов; в генерации необходимых тестовых сценариев и среды выполнения ПО; в верификации и валидации реализованных функций и требований к ПО; в сборе данных об отказах, ошибках и др. непредвиденных ситуациях; в подготовке отчетов по результатам тестирования.

1.5. СОПРОВОЖДЕНИЕ ПО

Сопровождение ПО – обеспечение жизнедеятельности ПО после поставки его заказчику, внесение в него изменений, связанных с устранением обнаруженных ошибок при эксплуатации и адаптации ПО к но-

вой среде функционирования, а также соображений о необходимости повышения производительности или улучшения отдельных характеристик ПО,.

Область знаний «Сопровождение ПО (Software maintenance)» состоит из описаний следующих разделов:

– основные концепции (Basic Concepts),

– процесс сопровождения (Process Maintenance),

– ключевые вопросы сопровождения ПО (key Issue in Software Maintenance) ,

– техники сопровождения (Techniques for Maintenance).

Сопровождение рассматривается с точки зрения удовлетворения требований в ПО, корректности его выполнения, обучения и оперативного учета процесса сопровождения.

Основные концепции включают в себя базовые определения и терминологию, подходы к эволюции и сопровождению ПО, а также к оценке стоимости сопровождения и др.

К основным определениям относится ЖЦ ПО и документация. Сопровождение – это процесс выполнения ПО, анализа необходимости модификации и оценки стоимости работ по внесению изменений. Рассматриваются проблемы, связанные с увеличением сложности продукта при большом количестве изменений в ПО.

Процесс сопровождения включает в себя: модели процесса сопровождения, планирование деятельности людей, которые проводят запуск ПО, проверку правильности его выполнения и внесения в него изменений. Процесс сопровождения включает в себя:

– корректировку ПО, т.е. изменение продукта в связи с обнаруженными ошибками и нереализованными задачами;

– адаптацию, т.е. настройку продукта к новым условиям эксплуатации;

– улучшение, т.е. изменение продукта в целях наращивания функций или повышение производительности системы;

– системную проверку продукта для поиска и исправления скрытых ошибок.

К техникам сопровождения относятся реинженерия и реверсная инженерия.

Реинженерия – это повторная реализация наследуемой системы в целях повышения удобства ее эксплуатации и сопровождения путем реорганизации и реструктуризации, перепрограммирования или настройки на другую платформу или среду.

Реверсная инженерия состоит в восстановлении спецификации (графов вызовов, потоков данных и др.) по полученному коду системы (особенно, когда в нее внесено много изменений).

Рефакторинг – это процесс изменения объектов и их интерфейсов для улучшения структурных и качественных показателей объектных программ. Изменения вносятся в отдельные операции над текстами, интерфейсы, среду проектирования и в инструментальные средства поддержки ПО.

1. 6. УПРАВЛЕНИЕ КОНФИГУРАЦИЕЙ ПО

Управление конфигурацией – способ идентификации компонентов системы для обеспечения системного контроля при внесении изменений с сохранением целостности, а также трассирование конфигурации. Это управление обеспечивается руководством по выполнению технических и административных решений проекта, документированию функциональных и физических характеристик конфигурации, контролю изменений этих характеристик; отчетности о внесении изменений и верификации ПО после этих изменений.

Область знаний «Управление конфигурацией ПО» (Software Configuration Management–SCM) состоит из описаний следующих разделов:

– управление процессом конфигурацией (Management of SMC Process),

– идентификация конфигурации ПО (Software Configuration Identification),

– контроль конфигурации ПО (Software Configuration Control),

– учет статуса конфигурации ПО (Software Configuration Status Accounting),

– аудит конфигурации ПО (Software Configuration Auditing),

– управление версиями ПО и доставкой (Software Release Management and Delivery).

Конфигурация системы – состав функций, физических характеристик системного и аппаратного обеспечения, а также готового ПО или комбинаций.

Конфигурация ПО включает в себя набор функций и характеристик ПО, заданных в технической документации и достигнутых в готовом продукте. Элемент конфигурации – график разработки, проектная документация, исходный и исполняемый код, библиотека, инструкции по установке системы и др.

Управление процессом конфигурации включает в себя:

– систематическое отслеживание вносимых изменений в отдельные составные части конфигурации, проведение аудита изменений и автоматизированного контроля за внесением изменений в конфигурацию;

– поддержку целостности конфигурации, ее аудит и обеспечение внесения изменений в один объект конфигурации, а также в связанный с ним другой объект;

– ревизию конфигурации для проверки разработки необходимых программных или аппаратных элементов и согласованности версии конфигурации с заданными требованиями;

– трассировку изменений конфигурации на этапах сопровождения и эксплуатации ПО.

Идентификация конфигурации ПО проводится путем выбора требуемого элемента конфигурации ПО и документирования его функциональных и физических характеристик, а также оформления технической документация на элементы конфигурации.

Контроль конфигурации ПО состоит в проведении работ по оценке, координации и утверждению реализованных изменений в элементы конфигурации с последующей их идентификацией.

Учет статуса конфигурации ПО – это комплекс мероприятий для определения состава системы и проверки правильности внесения изменений в конфигурацию ПО.

Управление версиями ПО заключается в отслеживании версии конфигурации; создании новой версии системы на базе существующей с внесением изменений в исходную конфигурацию; согласование версии с требованиями и изменениями на этапах ЖЦ; обеспечение оперативного доступа к объекту конфигурации.

1. 7. УПРАВЛЕНИЕ ИНЖЕНЕРИЕЙ ПО

Управление инженерией ПО (менеджмент) – руководство работами команды разработчиков в процессе выполнения проекта, определение критериев оценки и измерения процессов и продуктов проекта с использованием общих методов управления, планирования и контроля работ.

Менеджмент проекта – планирование, координация, измерение, мониторинг, контроль и отчет. Управление гарантирует системную, дисциплинированную и измеряемую разработку ПО. Ответственность за координацию человеческих, финансовых и технических ресурсов при реализации целей проекта несет менеджер проекта. В его задачу входит также выполнение целей проекта с соблюдением бюджетных и временных ограничений, стандартов и требований.

Более подробная информация по проблемам управления проектом содержится в ядре знаний – PMBOK (Project Management Body of Knowledge [2, 3], а также в стандарте ISO/IEC 12207 – Software life cycle processes.

Область знаний «Управление инженерией ПО (Software Engineering Management)» состоит из следующих разделов:

– организационное управление (Organizational Management),
– управление процессами и проектом (Process/Project Management),
– инженерия измерения ПО (Software Engineering Measurement).

Организационное управление включает в себя процессы управления, планирование и составление графика работ, оценивание стоимости работ, подбор кадров и контроль выполнения работ. Главные задачи организационного управления – это управление персоналом (тренировка и мотивация, обучение разработчиков), коммуникациями (каналы, медио, встречи, презентации) и рисками (техника управления, селекция проекта, минимизация риска и др.). Специалисты организационной структуры коллектива распределяются по ответственным участкам и подчиняются менеджеру проекта.

Управление процессами проекта включает в себя: исследование, составление и уточнение плана проекта, построение графика работ (сете-

вых и временных диаграмм) с учетом ресурсов, распределение персонала по этапам работ и длительности их выполнения; анализ финансовой, технической, операционной и социальной политики; контроль процессов управления.

Управление продуктом включает в себя уточнение требований и проверку (валидацию) соответствия создаваемого продукта заданным требованиям и верификацию правильности реализованных функций в проекте.

Управление рисками представляет собой процесс определения рисков и разработки мероприятий по уменьшению их влияния на ход выполнения проекта. *Риск* – вероятность проявления неблагоприятных условий, которые могут негативно повлиять на реализацию качества проекта (например, увольнение сотрудника, отсутствие его замены и др.). Предотвращение риска – это действия, которые снимают риск (например, увеличение времени разработки и др.) или уменьшают вероятность появления нового риска.

Измерение процессов инженерии ПО – это определение категорий рисков и отслеживание факторов для регулярного пересчета вероятностей их возникновения; совершенствование процессов управления проектом; оценки временных затрат и стоимости ПО в целях их регулирования и др.; выбор метрик для процессов и продукта и их оценка.

1. 8. ПРОЦЕСС ИНЖЕНЕРИИ ПО

Процесс инженерии ПО – это концепции, инфраструктура, методы определения и измерения процессов ЖЦ, поиск изменений и их реализация, а также анализ и оценка качества продукта.

Область знаний «Процессы инженерии ПО (Software Engineering Process)» состоит из описаний следующих разделов:

– концепции процесса инженерии ПО (Software Engineering Process Concepts),

– инфраструктура процесса (Process Infrastructure),

– определение процесса (Process Definition),

– процесс измерения (Process Measurement),

– количественный анализ проекта (Qualitative Process Analysis),

– выполнение процесса и изменение.(Process Implementation and Change).

Инфраструктура процесса – это виды ресурсов (группы разработчиков, технические средства, программные продуктов и т.п.), а также процесс инженерии ПО (групповой или по типу экспериментальной фабрики – Experience Factory), базирующейся на моделях проекта и продукта, моделях качества и риска, методах управления коллективом и организации разработки проекта.

Определение процесса основывается на типах процессов и моделей ЖЦ (водопадная, спиральная, итерационная и др.), стандартах ЖЦ ПО ISO/IEC 12207 и 15504, IEEE std 1074–91 и 1219–92, а также на методах спецификации процессов и средствах их поддержки.

Процесс измерения предполагает стадию наблюдения за выполнением процесса для сбора информации, моделирования, классификации ошибок и дефектов, а также для статического контроля и измерения отдельных показателей процесса.

Качественный анализ процесса заключается в идентификации и поиске слабых мест в ПО до начала его выполнения. Рассматривается две техники анализа: обзор данных и сравнение данного процесса со стандартным ISO/IEC−12207; сбор данных об атрибутах качества продукта и процессов; анализ причин отказов в функционировании ПО, отката назад от точки возникновения отклонения до точки нормальной работы ПО и выяснения причин.

Выполнение процесса и изменение. Эта процесс развертывания ПО, инспекции проекта, принятие решений о необходимости изменения процесса, организационной структуры проекта и некоторых инструментов управления проектированием.

1. 9. МЕТОДЫ И СРЕДСТВА ИНЖЕНЕРИИ ПО

Методы и средства входят в среду разработки, используемую на процессах ЖЦ. Средства обеспечивают спецификацию требований, конструирование и сопровождение ПО. Методы обеспечивают проектирование, реализацию и выполнение ПО на процессах, а также оценку качества процессов и продуктов.

Область знаний «Методы и средства инженерии ПО (Software Engineering Tools and Methods)» состоит из разделов:

− инструменты (Software Tools),

− методы (Software Methods).

Инструменты инженерии ПО подразделяются на следующие типы инструментов: сбора требований, проектирования ПО (редакторы схем и диаграмм), конструирования ПО (редакторы текстов, компиляторы, отладчики), тестирования (генераторы тестов, среды исполнения тестов), автоматизации процесса инженерии ПО, контроля качества, управления конфигурацией ПО (управление версиями, учетом дефектов и др.), управления инженерией ПО (планирование проекта, управление рисками и др.).

Методы инженерии ПО включают в себя эвристические (неформальные), структурные, объектно-ориентированные методы, ориентированные на данные и на прикладную область, а также формальные методы проектирования и прототипирования.

1.10. КАЧЕСТВО ПО

Качество ПО − набор характеристик продукта или сервиса, которые определяют его способность удовлетворять установленным или предполагаемым потребностям заказчика (пользователя) ПО.

Область знаний «Качество ПО (Software Quality)» состоит из описания следующих разделов:

– концепция качества ПО (Software Quality Concepts),

– определение и планирование качества (Definition & Planning for Quality),

– деятельности и техники гарантии качества и V&V(Activities and Techniques for Software Quality Assurance, Validation–V & Verification – V),

– измерения в анализе качества ПО (Measurement in Software Quality Analysis).

Описание данной области знаний SWEBOK содержит обширный материал по проблеме качества ПО и путей его достижения в процессе проектной деятельности групп разработчиков.

Концепция качества ПО – это современные методы и стандарты определения внешних и внутренних характеристик качества и их метрик, а также модели качества, заданные на множестве внешних характеристик и представленные в стандартах качества. Установлено шесть базовых характеристик качества и для каждой из них по 4–5 атрибутов. К ним относятся: функциональность, надежность, удобства использования, эффективность, сопровождаемость, переносимость. Стандартная модель качества определяет любой тип программного продукта, а конкретная модель качества включает в себя только те характеристики, которые указаны в требовании к конкретному ПО.

Определение и планирование качества ПО основывается на стандартах, планах графиков работ и процедурах проверки и др. План обеспечения качества – это набор действий для проверки процессов обеспечения качества (верификация, валидация и др.) и формирование документа по гарантии качества.

Деятельности и техники гарантии качества включают в себя: инспекцию, верификацию и валидацию ПО.

Инспекция ПО – анализ и проверка различных представлений системы (спецификаций, архитектурных схем, диаграмм и др.) и выполняется на всех этапах ЖЦ разработки ПО.

Верификация ПО – процесс проверки и анализа правильности выполнения функций системы в соответствии со спецификаций. Верификация дает ответ на вопрос, правильно ли создана система. *Валидация* – процесс проверки соответствия ПО функциональным и нефункциональным требованиям и ожидаемым потребностям заказчика. По результатам проверки делается оценка полноты реализации функций и требований.

Измерения в анализе качества ПО – это измерение характеристик процессов, ресурсов и продуктов; оценки модели процесса, документации и др. Для измерения фактических характеристик качества продукта проводится *тестирование ПО* – проверка продукта на тестовых данных и анализ выходных результатов. Тесты разрабатываются так, что бы имитировать работу системы с реальными входными данными и фиксировать возникающие отказы, дефекты и ошибки. Тестирование заканчивается измерением полученных характеристик качества, атрибуты которых заданы в требованиях.

ПРИЛОЖЕНИЕ 2

ХАРАКТЕРИСТИКА ПРОЦЕССОВ ЖЦ СТАНДАРТА 12207

Каждая ПС на протяжении своего существования проходит определенную последовательность этапов от замысла до его воплощения в программы, эксплуатацию и изъятие. Такая последовательность этапов имеет название *жизненного цикла (ЖЦ) разработки*. На каждом этапе ЖЦ происходит определенная совокупность действий и выполнения задач для выпуска продукта, используя ресурсы – стоимость, время, люди.

Все продукты процессов программной инженерии представляют собой определенные описания – тексты требований к разработке, согласование договоренностей, документация, тексты программ, инструкции по эксплуатации и т.п.

Разновидности действий, составляющих процессы жизненного цикла ПС, зафиксированы в международном стандарте ISO/IEC 12207 [1, 2].

Согласно этому стандарту все процессы разделены на три категории (таблица):

– главные процессы;
– вспомогательные процессы;
– организационные процессы.

К главным процессам относятся такие:

– приобретение (acquisition),
– поставка (supply),
– разработка (development),
– эксплуатация (operation),
– сопровождение (maintenance).

Процесс приобретения инициирует ЖЦ ПО и определяет действия организации-покупателя (или заказчика), которая приобретает автоматизированную систему, программный продукт или сервис.

Процесс поставки определяет действия предприятия-поставщика, которое снабжает покупателя системой, программным продуктом или сервисом.

Процесс разработки определяет действия предприятия-разработчика, изготавливающего программный продукт на таких процессах ЖЦ: разработка требований, проектирование, кодирование, тестирование, интеграция, тестирование, эксплуатация, сопровождение.

Процесс эксплуатации определяет действия оператора, который обеспечивает обслуживание системы в процессе ее эксплуатации пользователями (консультации пользователей, изучение их потребностей и т.д.).

Процесс сопровождения определяет действия организации, выполняющей сопровождение программного продукта (управление модификациями, поддержку текущего состояния и удаление программного продукта).

К вспомогательным процессам стандарта отнесены процессы:

- документирования (documentation),
- управления конфигурацией (configuration management),
- обеспечения качества (quality assurance),
- верификации (verification),
- валидации (validation),
- совместного анализа (оценки) (joint review),
- аудита (audit),
- решения проблем (problem resolution).

Вспомогательные процессы поддерживают реализацию основных процессов и способствуют получению требуемого качества ПО. Они инициируются другими процессами.

К организационным процессам стандарта относятся процессы:

- управления (management),
- создания инфраструктуры (infrastructure),
- усовершенствования (improvement),
- обучения (training).

За каждый процесс стандарта отвечает определенный участник разработки или руководитель. Каждый процесс стандарта определяет виды деятельности и задачи, которые в него входят, совокупность результатов видов деятельности и задач, а также некоторые специфические требования.

Стандарт ISO/IEC 12207 предоставляет собой структуру процессов ЖЦ, но не обязывает использовать все процессы или определенную модель ЖЦ ПО, или конкретную методологию разработки ПО. Являясь стандартом высокого уровня, он не задает детали того, как надо выполнять действия или задачи, составляющие процессы.

Стандарт не рекомендует требований к формату и содержанию документов, выпускаемых на разных процессах.

Процессы, действия и задачи приведены в стандарте в наиболее общей естественной последовательности. Это не означает, что в такой же последовательности они должны быть применены в конкретной модели ЖЦ. В зависимости от проекта процессы, действия и задачи стандарта выбираются, упорядочиваются и включаются в модель ЖЦ. При применении они могут перекрывать, прерывать друг друга, выполняться итерационно или рекурсивно.

Поэтому организации, которая намерена применить этот стандарт в своей работе непосредственно, понадобятся дополнительные стандарты или процедуры, определяющие разные детали по применению выбранных элементов ЖЦ. Отметим, что комитет ISO выпускает руководства и процедуры, дополняющие стандарт 12207.

Процессы ЖЦ в стандарте ISO/IEC 12207 Таблица 1

№ п/п	Наименование процессов (подпроцессов)
1. Категория “Основные процессы”	
1.1	Заказ (договор)
1.1.1	Подготовка заказа, выбор поставщика
1.1.2	Мониторинг деятельности поставщика, прием потребителем
1.2	Поставка (приобретение)
1.3	Разработка
1.3.1	Выявление требований
1.3.2	Анализ требований к системе
1.3.3	Проектирование архитектуры системы
1.3.4	Анализ требований к ПО системы
1.3.5	Проектирование ПО
1.3.6	Конструирование (кодирование) ПО
1.3.7	Интеграция ПО
1.3.8	Тестирование ПО
1.3.9	Системная интеграция
1.3.10	Системное тестирование
1.3.11	Инсталляция ПО
1.4	Эксплуатация
1.4.1	Функциональное использование
1.4.2	Поддержка потребителя
1.5	Сопровождение
2. Категория “Процессы поддержки”	
2.1	Документирование
2.2	Управление конфигурацией
2.3	Обеспечение гарантии качества
2.4	Верификация
2.5	Валидация
2.6	Общий просмотр
2.7	Аудит
2.8	Решение проблем
2.9	Обеспечение применимости продукта
2.10	Оценивание продукта
3. Категория “Организационные процессы”	
3.1	Категория
3.1.1	Управление на уровне организации
3.1.2	Управление проектом
3.1.3	Управление качеством
3.1.4	Управление риском
3.1.5	Организационное обеспечение
3.1.6	Измерение
3.1.7	Управления знаниями
3.2	Усовершенствование
3.2.1	Внедрение процессов
3.2.2	Оценивание процессов
3.2.3	Усовершенствование процессов

Кроме этого, стандарт ISO/IEC 12207 имеет первостепенное значение, так как предоставляет основу для принятия ряда других связанных с ним стандартов, таких как стандарты по управлению ПО, обеспечению качества, верификации и валидации, управлению конфигурацией, метриками ПО и т.д.

Из данного стандарта можно выбрать только те процессы, которые более всего подходят для реализации конкретной ПС. Обязательными являются основные процессы, которые присутствуют во всех известных моделях ЖЦ. В зависимости от целей и задач предметной области они могут быть пополнены дополнительными (документирование, обеспечение качества, верификация и валидация и т.п.) и организационными (планирование, управление и др.) процессами этого стандарта. Разработчик сам принимает решение о включении в новую создаваемую модель ЖЦ процесса обеспечения качества компонентов и системы управления им или определения набора проверочных (верификационных) процедур для обеспечения правильности продукта и соответствия (валидация) его заданным требованиям.

Процессы, включенные в модель ЖЦ, предназначены для реализации стандартных задач процессов ЖЦ и могут привлекать другие процессы для выполнения специализированных возможностей системы (например, защита данных). Интерфейсы между двумя любыми процессами ЖЦ должны быть минимальными (вход и выход) и каждый из них должен удовлетворять следующим правилам:

– если процесс А вызывается процессом В и только процессом В, то А принадлежит В;

– если функция (задача) вызывается более чем одним процессом, то она становится отдельным процессом;

– проверки любой функции в ЖЦ является обязательной.

Иными словами, если задача требуется более чем одному процессу, то она может стать процессом, используемым однократно или многократно на протяжении жизни конкретной системы.

Процессы модели ЖЦ ориентированы на разработчика системы, который может выполнять один или несколько процессов. Процесс может быть выполнен одним или несколькими разработчиками, при этом один из них является ответственным за один процесс или за все процессы модели.

Создаваемая модель ЖЦ устанавливает связь с конкретными методиками и соответствующими стандартами в области программной инженерии, либо разрабатывается самостоятельно с учетом модели возможностей и особенностей проекта. Иными словами, каждый процесс ЖЦ подкрепляется выбранными для реализации задач ПС средствами, методами программирования и методикой их применения и выполнения.

СПИСОК ЛИТЕРАТУРЫ

К главе 1

1. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях. – Киев: Диалектика, 1993. – 238 с.

2. *Coad P.,Yourdan E.* Object-oriented analysis.– Second Edition.- Prentice Hall, 1991.– 296p.

3. *Yourdan E.* Modern Srtucrured Analysis. – New York: Yourdan Press/Prentice Hall, 1988.–297p.

4. *DeMarko D.A., McGovan R.L.* SADT: Structured Analysis and Design Technique. New York: Mcgray Hill, 1988. – 378 с.

5. *Yourdan E., Constantine L.* Structured Design. Yourden Press. Engwood Cliffs.N.J.– 1983.

6. *Martin J., Odell J.J.* Object-oriented analysis and design. – Prentice Hall, 1992.– 367p.

7. *Barker R.* CASE-method. Entity Relatiionship Modeling.– Copyrigth ORACLE Corporation UK Limited New York: Publ., 1990. – 312 p.

8. *Schardt J.A.* Assentials of Distributed Object Design M.S.E. Advanced Concepts Center.– 1994.– p.225–234

9. *Rumbaugh J., Blaha V., Premerlani W.* Object–Oriented Modelling and Design, Englewood Cliffs, NJ: Prentice Hall, 1991. – 451p.

10. *Буч. Г.* Объектно-ориентированный анализ.– М.: Бином, 1998. –560 с.

11. *Jacobson I.* Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing.– New York: Addison-Wesley Publ.Co, 1994.– 529 p.

12. *Siegel I.* CORBA Fundamentals and Programming, Wiley Co.Publ.Group, Sohn Wiley & Sons.Inc.– USA.– 1996.– 694p.

13. *Орфали Р., Харки Д.* Эдвардс Дж. Основы CORBA. –М.: Из.–во “Малип”, 1999.–317с.

14 *Рамбо Дж, Джекобсон А., Буч Г.* UML: специальный справочник.– СПб.: Питер, 2002.– 656с.

15. *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.– Издательский дом «Питер».– Москва– Санкт-Петербург... Харьков, Минск, 2005. – 730с.

16 *Лавищева Е.М., Склым Д.Ю.* Подход к проверке правильности интерфейсов и взаимодействий объектов в объектно-ориентированных распределенных приложениях // Проблемы программирования, –№1, 1999.–с.72–79.

17. *Рамбо Дж., Джекобсон А., Буч Г.* UML: специальный справочник.– СПб.: Питер, 2002.– 656с.

18. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.

19. ISO/IEC 12207: 2002.– Information technology – Software life cycle processes) Информационные технологии – Процессы жизненного цикла программного обеспечения.

20. *Software* Engineering Body of Knowledge (SWEBOK). // ISO/IEC JTC1/SC7 N2517. Software & System Engineering Secretariat, Canada, 2001. – 220 p.

21. www.swebok.org, malto:sorlic@borland.ru

22. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии. Учебник:– Киев: Знання, 2001. –269 с.

23. *Jackson M.* Software requirement & specifications.– Wokingham, England: Addison–Wesley, ACM Press Books, 1995. –228 p.

24. *Краснощеков П.С., Петров А.А.* Принципы построения моделей.-М.:Изд-во МГУ, 1983. -264с.

25. *Вигерс К* Формулировка требований к программному обеспечению.– Москва, 2004.– 575с.

26. *The Unified* Modeling Language (UML) Specification. – V. 1.3. UML Specification, revised by the OMG. – July 1999. – 620 p.

27. *Кендалл С.* Унифицированный процесс. Основные концепции.– Москва–С–Петербург–Киев, 2002.– 157с.

28. *Трофимов С.А.* CASE– технологии: практическая работа в Rational Rose.-2-изд.– Бином–Пресс, 2002. – 288с.

К главе 2

1. *Ершов А.П.* Введение в теоретическое программирование.–М.: Главная редакция физико-математической литературы.–1977.–286с.

2. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд. – М.: “Изд-во Бином”, 1998. – 560 с.

3. *Martin J., Odell J.J.* Object-oriented analysis and design.– Prentice Hall, 1992.– 367p.

4. *Rumbaugh J., Blaha V., Premerlani W.* Object-Oriented Modeling and Design, Englewood Cliffs, NJ: Prentice Hall, 1991. – 451p.

5. *Firesmith D.* Object-oriented Methods, Standards and Procedures, Englewood Cliffs: Prentice Hall, 1994. – 367 p.

6. *Jacobson I.* Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing.– New York: Addison–Wesley Publ.Co, 1994. – 529 p.

7. *Coad P., Yourden E.* Object-oriented analysis.– Second Edition.– Prentice Hall, 1991.– 296 p.

8. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях. – Киев: Диалектика, 1993.– 238 с.

9 *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.

10. *Римский Г.В.* Структура и функционирование системы автоматизации модульного программирования // Программирование.– 1980.– №2.– С.31–38.

11. *Лаврищева Е.М., Грищенко В.Н.* Связь разноязыковых модулей в ОС ЕС.– М: Финансы и статистика, 1982.– 127с.

12. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. – Киев.– Наукова Думка, 1991.–213с.

13. *Липаев В.В., Позин Б.А., Штрик А.А.* Технология сборочного программирования. –М.: – Радио и связь, 1992. – 275с.

14. *Марка Д.А., МакГруэн К.* Методология структурного анализа и проектирования.–М.: МетаТехнология, 1997.– 346с.

15. *Yourdon E.* Modern Structured Analysis. –New York: Yourdon Press /Prentice Hall, 1988.– 297 p.

16. *Skidmore S., Mills G., Farmer R.* SSADM: Models and Methods. – Prentice–Hall, Englewood Cliffs, 1994. – 569c

17. *Demark D.A., McGowan R.L.* SADT: Structured Analysis and Design Technique. New York: McCray Hill, 1988. – 378 c.

18. *Barker R.* CASE–method. Entity Relationship Modeling.– Copyright Oracle Corporation UK L limited New York: Publ., 1990. – 312 p.

19. *Боггс У., Боггс М.* UML и Rational Rose. Издат. «Лори», 1999.– 580с.

20. *The Unified* Modeling Language (UML) Specification. – V. 1.3. UML Specification, revised by the OMG. – July 1999. – 620 p.

21. *Кендалл С.* Унифицированный процесс. Основные концепции.– Москва–С–Петербург–Киев, 2002.– 157с.

22. *Рамбо Дж., Джекобсон А, Буч Г.* UML: специальный справочник.– СПб.: Питер, 2002.– 656с.

23. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования//Кибернетика и системный анализ, 2003.– №1.– с.39–55.

24. *Грищенко В.Н., Лаврищева Е.М.* Компонентно-ориентированное программирование. Состояние, направления и перспективы развития//Проблемы программирования. – 2002 г. – № 1–2.– С. 80–90.

25. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии // Проблемы программирования. – 2000. – №1–2.– С.351–360.

26. *Crnkovik I, Larsson S., Stafford J.* Component–Based Software Engineering: building systems from Components at 9th Conference and Workshops on Engineering of Computer –Based Systems.– Software Engineering Notes.– 2002.– vol.27.–N 3.–c.47–50.

27. *Gamma E., Helm R., Johnson R. and Vlissides J.* Design Patterns, Elements of Reusable Object–oriented Software. – N.– Y.: Addison–Wesley, 1995. – 345p.

28. *Component* Object Model. – www.microsoft.com/tech/COM. asp

29. *Meyer B.* On to Components. Computer. – 32, N 1, January 1999. – P.139–140.

30. *Lowy J.* COM and NET Component Services. − O'Reilly, 2001. − 384 p.

31. *Batory D., O'Malley S.* The Design and Implementation of Hierarchical Software Systems with Reusable Components / ACM Transactions on Software Engineering and Methodology. − N 4. − 1, October 1992. − P.355−398.

32. *Weide B., Ogden W., Sweden S.* Reusable Software Components/ Advances in Computers, vol. 33. − Academic Press, 1991. − p.1−65.

33. *Jacobson I., Griss M., Johnson P.* Software Reuse: Architecture, Process and organization for Business Success − Addison Wesley, Reading, MA, May 1997. −501p.

34. *Гост* 30664 (ИСО/МЭК 11404:1996) Информационные технологии. Языки программирования, их среда и системный интерфейс. Зависимые от языков типы данных / Межгосударственный стандарт.− Межгосударственный совет по стандартизации. метрологии и сертификации, 2000. − 112 с.

35. *Lopes C., Kiczales G., Murphy G. Lee A.* Proceedings of the ECOOP on Aspect−Oriented Programming, Kyoto, Japan, April 20, 1998.

36. *Kiselev. I.* Aspect−Oriented Programming with AspectJ. Indianapolis, IN, USA: SAMS Publishing, 2002.− 164p.

37. *Homepage* of the Aspect−Oriented Programming, Xerox Palo Alto Research Center (Xerox Parc) Palo Alto, CA, www.parc.xerox.com/aop

38. *The Aspect−Oriented* Programming. Proceedings ECOOP'97− Object−oriented Programming, 11th Europe Conference, Finland, June 1997. − Springer Verlag, Berlin.

39. *Lieberherr K.* Demeter and Aspect−Oriented Programming. Proceeding of the STJA'97 Conference, Erfurt, Germany, Sept.10−11, 1997. − P.40−43.

40. *Павлов В.* Аспектно-ориентированное программирование // Технология клиент-сервер, № 3−4.− С.3−45.

41. *Berger L., Dery F., Fornarino V.* Interaction between object: and aspect of object−oriented languages / Conference of AOP −1998. P.13−18.

42. *К.Чернецки, У.Айзенекер.* Порождающее программирование. Методы, инструменты, применение.− Издательский дом «Питер».− Москва− СП... Харьков, Минск, 2005.−730с.

43. *Golub G., C.van Loan.* Matrix Computation. Third editor −The John HopkinsUniversity Press, Baltimore and London, 1996.

44. *Reenskaug with Wold P. Lehhe O.A.* Working with Objects: The Ooram Software Engineering Method. Manning Publications Co., Greenwich, CT,− 1996.−311p.

45. *Ndumu D.T., Nwana.* Research and Development challenges for Agent−based system..− IEEProccedings Software Engineering− 1997.− 144.− № 01.−p.26−37.

46. *Wooldridge M.* Agent−based software engineering.− Там же.− p.2−10.

47. *Shoham. Y.* Agent−oriented programming.−Artif.Intell. 1993, 60, №1.− p.51−92.

48. *Плескач В.Л., Рогушина Ю.В.* Агентні технології, Київ.− КНТЕУ.−2005.−337с.

49. *Трахтенгерц Э.А.* Взаимодействие агентов в многоагентных средах //Автоматика и телемеханика.– М.: Наука .–1998.–№8.– с.3–52.

50. *Nwana H.S., Lee J., Jennings N.R.* Coordination Software agent systems / British Telecommunication Technology Journal, 1996.–14(4).

51. *Riechem D.* Intelligent agents // CACM, 1994. – V.37, №7. – p. 20– 31.

52. *Дрейган Р.* Будущее программных агентов.– PC Magazine March 25,– 1997.–190с.

53. *Wegner P.* Interaction Foundation of Object–oriented Programming ECOOP–97 th Europian Conference on OOP Finland, June 9–12, 1997.– 123–139.

54. *Летичевский А.А, Маринченко В.Г.* Объекты в системе алгебраического программирования //Кибернетика и системный анализ. –1997.– N2. – С.160–180.

55. *Летичевский А.А., Капитонова Ю.В., Волков В.А., Вышемирский В.В., Летичевский А.А.* (мол.). Инсерционное программирование // Там же.– 2003.– №1.– 12–32.

56. *Letichevsky A.A., Gilbert D.R.* A General Theory of Action Language //Там же . – 1998. – № 1.– С.16 – 36.

57. *Летичевский А.А., Капитонова Ю.В.* Доказательство теорем в математической информационной среде //Там же. – 1998. – №.4. – С. 3 – 12.

58. *Letichevsky A.A., Gilbert D.R.* A model for interaction of agents and environments //Recent trends in algebra's development technique Language.– 2000. – P.311 – 329.

59. *Редько В.Н.* Экспликативное программирование: ретроспективы и перспективы // Проблемы программирования.– 1998. – №2.–С. 22 – 41.

60. *Никитченко Н.С.* Композиционно–номинативный подход к уточнению понятия программы // Там же.– 1999.– №1.– С. 16–31.

61. *Редько В.Н.* Композиционная структура программологии //Кибернетика и системный анализ. – 1998. – № 4. – С. 47–66.

62. *Редько В.Н.* Основания программологии // Там же. – 2000.– №1.– С. 35–57.

63. *Редько В.Н., Брона Ю.Й., Буй Д.Б.* Реляційні бази даних: табличні алгебри та SQL–подібні мови //Видав. дім “Академперіодика”, Київ, 2001.–195с.

64. *Эпплман Д.* Переход на VB .NET: стратегии, концепции, код. – СПб.: Питер, 2002. – 464 с.

65. *Троелсен Э.* C# и платформа .NET. Библиотека программиста. – СПб.: Питер, 2002. – 800 с.

66. *Просиз Дж.* Программирование для Microsoft .NET. – М.: Издательско-торговый дом “Русская Редакция”, 2003. – 704 с.

67. *Рихтер Дж.* Программирование на платформе Microsoft .NET FRAMEWORK. – М.: Издательско-торговый дом “Русская Редакция”, 2002. – 512 с.

68. *Орлов С.А.* OLE/COM, CORBA и распределенные вычисления.– Открытые системы.–1995, № 3.–с. 37– 43.

69. *Сигел Дж.* CORBA 3. – Москва: Малип, 2002. – 412 с.

70. *Роджерсон Д.* Основы COM. – Руск.пер.– Microsoft Press.– 363с.

71. Программы следующего десятилетия.– Гостинная ОС.– Открытые системы, декабрь 2001.– с.60–72.

72. *Монсон–Хейфел Р.* Enterprise JavaBeans. – СПб:Символ–Плюс, 2002. – 672 с.

73. *Федоров А.Г.* Технологии для Web–сервисов // Компьютер Пресс, 2002. – №6. – С. 32–36.

74. *Холл М.* Сервлеты и JavaServer Pages. Библиотека программиста. – СПб.: Питер, 2001. – 496 с.

75. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.

76. *Яковсон А.* Мечты о будущем программирования. Открытые системы.– М.: 2005.– №12.– с.59–63.

77. *Поспелов Г.С.* Искусственный интеллект – основа новой информационной технологии – М.: Наука, 1988.–279с.

78. *Антимиров В.М., Коваль В.Н.* Доказательное проектирование дискретных устройств на основе алгебраических спецификаций // Логическое управление с применением ЭВМ. – М.: Научн. Совет АН СССР по проблеме «Кибернетика», 1990. – С. 56–63.

79. *Коваль В.Н.* Концепторные языки. Доказательное проектирование.– Киев.: Наукова думка, 2001.– 182с.

80. *Коваль В.Н., Кук Ю.В.* Оптимальные системы обнаружения и классификации движущихся объектов //Кибернетика и системный анализ. – 1993. – №5. – С. 137 – 145.

81. *Хоар Ч., Лауэр П.Е.* Непротиворечивые взаимодополняющие теории семантики языков программирования // М.: Мир, 1980.–с.186–221.

82. *Abrial I.R., Meyer B.* Specification Language Z. –Boston: Massachusetts Computer Associates Inc. 1979. –378 p.

83. *Biorner D.Jones C.B.* The Vienna Development Methods (VDM): The Meta – Language. – Vol. 61 of Lecture Notes in Computer Science.– Springer Verlag, Heiderberg, Germany, 1978.–215p.

84. *The RAISE* Language Group. The RAISE Specification Language. BCS Practitioner Series. – Prentice Hall, 1982.–397 p.

85. *The RAISE* Methods Group. The RAISE Development Methods. BCS Practitioner Series. – Prentice Hall, 1985.–493p.

86. *Цейтлин Г.Е.* Введение в алгоритмику.– Изд.–во Фара, 1999.–310с.

87. *Глушков В.М., Цейтлин Г.Е., Ющенко Е.Л.* Алгебра. Языки. Программирование. – Наукова думка.– 1974.–317с. (перераб. и переизд. 1979г., 1989г.).

88. *Ющенко Е.Л , Сужко С,В,, Цейтлин Г.Е., Шевченко А.И.* Алгоритмические алгебры.– Учебник (укр.).– ИЗММ, 1997.–480с.

89. *Дорошенко А.Е., Финин Г.С., Цейтлин Г.Е.* Алгеброалгоритмические основы программирования.– К.: Наукова думка, 2004.–457с.

90. *Цейтлин Г.Е.* Введение в информатику.–“Изд-во Сфера”.–310с.

91. *Software* Engineering Body of Knowledge (SWEBOK). // ISO/IEC JTC1/SC7 N2517. Software & System Engineering Secretariat, Canada, 2001. – 220 p.

92. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии.– Материалы второй междун.науч.-практ.конф. УкПрог-2000, .Киев, с.351-360.

К главе 3

1. *Corbin J.* The art of Distributed Application. Programming Techn. For Remote Procedure Calls.– Berlin: Springer Verlag, 1992.– 305p.

2. *Open* Software Foundation. Introduce to Open Software Foundation. Distributed Computed Environments.– Englewood Cliffs: Prentice Hall, 1993.– 437p.

3. *Роджерсон Д.* Основы COM. Рус. ред.–Microsoft Press, 1996.–361 с.

4. *Орфали Р., Харки Д., Эрварс Д.* Основы CORBA .– М: НАЛИП, 1999. – 317с.

5.. *CORBA.* The Common Object Request Broker: Architecture and Specification. Revision 2.0. Copyright 1991, 1992, 1995 by Sun Microsystems, Inc.–1995.–621 p.

6. *Мак-Лахлин Бр.* Java и XML. – СПб: Символ-Плюс, 2002. – 544 с.

7. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.

8. *Лаврищева Е.М., Склым Д.Ю.* Подход к проверке правильности интерфейсов и взаимодействий объектов в объектно-ориентированных распределенных приложениях, Ж. Проблемы программирования. – №1, 1999. –с.72–79.

9. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии// Программирование, 2000.– №1–2.– с.351–360.

10. *Лаврищева Е.М., Грищенко В.М.* Сборочное программирование.– Киев, Наукова думка.– 1991. –213с.

11. *Лаврищева Е.М., Грищенко В.Н.* Методы и средства компонентного программирования.– Кибернетика и системный анализ, №1, 2003.– с.39–55

12. *ISO/МЭК* 11404:1996. Информационные технологии. Языки программирования, их среда и системный интерфейс. Независимые от языков типы данных /Межгосударственный стандарт.– Межгосударственный совет по стандартизации, метрологии и сертификации, 2000.–112 с.

13.. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях. – Киев: Диалектика, 1993. – 238 с.

14. *Coad P.,Yourdan E.* Object–oriented analysis.–Second Edition.– Prentice Hall, 1991.– 296p.

15. *Yourdan E.* Modern Srtucrured Analysis.–New York: Yourdan Press /Prentice Hall, 1988.–297p.

16. *DeMarko D.A., McGovan R.L.* SADT: Structured Analysis and Design Technique. New York: Mcgray Hill, 1988. – 378 с.

17. *Yourdan E., Constantine L.* Structured Design. Yourden Press. Engwood Cliffs.N.J.–1983.

18. *Martin J., Odell J.J.* Object−oriented analysis and design.−Prentice Hall.−1992.−367p.

19. *Barker R.* CASE−method. Entity Relatiionship Modeling.−Copyrigth ORACLE Corporation UK Limited New York: Publ., 1990. − 312 p.

20.. *Schardt J.A.* Assentials of Distributed Object Design M.S.E. Advanced Concepts Center.−1994.−p.225−234

21. *Rumbaugh J., Blaha V., Premerlani W.* Object− Oriented Modelling and Design, Englewood Cliffs, NJ: Prentice Hall, 1991. − 451p.

22. *Буч Г.* Объектно-ориентированное проектирование с примерами применения. − Киев: Диалектика, 1992. − 519 с.

23. *Jacobson I.* Object−Oriented Software Engineering. A use Case Driven Approach, Revised Printing. − New York: Addison−Wesley Publ. Co, 1994. − 529 p.

24 *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение.− Издательский дом «Питер».− Москва− Санкт−Петербург... Харьков, Минск, 2005.− 730с.

25 *Рамбо Дж., Джекобсон А., Буч Г.* UML: специальный справочник.− СПб.: Питер, 2002.− 656с.

26. *Кендалл Скотт.* Унифицированный процесс. Основные концепции.− Москва−С−Петербург−Киев, 2002.− 157с.

27. *Song X.* Systematic integration of Design Methods.− IEEE Software. − 1997.− March/April.− p.107−el, D. (1995b). Formal specification languages in knowledge and software engineering. −The KnowledgeEngineering Review, 10(4), p.361 − 404.

28. *Лаврищева Е.М.* Основы технологической подготовки разработки прикладных программ систем обработки данных // Препринт / АН УССР, Ин−т кибернетики им. В.М. Глушкова; 87−5. − Киев.− 1987. − 30 с.

29. *ISO/IEC* 12207: 2002. Information technology − Software life cycle processes) Информационные технологии − Процессы жизненного цикла программного обеспечения.

30. *Андон Ф.И., Коваль Г.И., Коротун Т.М., Суслов В.Ю.*. Основы инженерии качества программных систем / Под ред. И.В. Сергиенко. − К.: Академпериодика, 2002. − 504 с.

31. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии. Учебник (укр. язык). − Киев: Знання, 2001. −269 с.

32. *Соммервил И.* Инженерия программного обеспечения. 6 − издание.−Москва−Санкт−Петербург−Киев, 2002.−623 с.

33. *Wegner P.* Interaction Foundation of Object−oriented Programming ECOOP −97 th Europian Conference on OOP Finland, June 9−12, 1997.− p.123−139.

34. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных приложений.− Киев, Наук. Думка.−1997.−227с.

35. *Типы* данных в языках программирования и данных,− Новосибирск.− Наука, 1987.−52с.

36. *Northrop L.M.* SEI`s Software Product Line Tenets // IEEE Software. −2002. − v.19. − №4. − p.32−39.

К главе 4

1. *Corbin J.* The art of distributed fpplications. Progrmming Techn. For Remote Procedure Calls. − Berlin: Springer Verlag. − 1992.− 305p.

2. *Open* Software Foundation. Inroduce to Open Software Foundation. Disributed Computed Environments. − Englewood Cliffs: Prentice Hall, 1993.− 437p.

3. *Орфали Р., Харки Д., Эрварс Д.* Основы CORBA.− М: НАЛИП, 1999. − 317с.

4. *Слама Д., Гарбис Дж., Рассел П.* Корпоративные системы на основе CORBA. − М.: Издательский дом "Вильямс", 2000. − 368 с.

5. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. − М.: Мир, 2002. − 510с.

6. *Роджерсон Д.* Основы COM. Рус. пер.− Microsoft Press.− 361с.

7. *Орлов С.* OIE/COM, CORBA и распределенные вычисления. Открытые системы.− 1995.−№3.−с.37−43.

8. *ИСО/МЭК* 11404:1996. Информационные технологии. Языки программирования, их среда и системный интерфейс. Независимые от языков типы данных/ Межгосударственный стандарт.−Межгосударственный совет по стандартизации, метрологии и сертификации, 2000. − 112 с.

9. *Джордан Д.* Обработка объектных бах данных в C++. Программирование по стандарту ODMG: Пер. с англ. − М.: Издательский дом "Вильямс", 2001. − 384 с.

10. *Дунаев С. Б.* Доступ к базам данных и техника работы в сети. − М.: Диалог−Мифи, 1999. − 416 с.

11. *Монсон−Хейфел Р.* Enterprise JavaBeans. − СПб: Символ−Плюс, 2002. − 672 с.

12. *Барлет Н., Лесли А., Симкин С.* Программирование на JAVA. Путеводитель.− Киев, 1996.− 736с.

13. *Холл М.* Сервлеты и JavaServer Pages. Библиотека программиста. − СПб.: Питер, 2001. − 496 с.

14. *Иванников В.П., Дышлевый К.В., Мажелей С.Г., Содовская Д.Б., Шебуняев А.Б.* Распределенные объектно-ориентированные среды.− Москва // РАН.ИСП. Труды ИСП.− 2000.− с.84−100.

15. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных систем. − Киев: Наукова думка, 1997г.−228с.

16. *Найк Д.* Стандарты и протоколы Интернета.−Microsoft Press.− Русская редакция, 1999.−256с.

17. *Лаврищева Е.М., Грищенко В.М.* Сборочное программирование.− Киев: Наукова думка, 1991. −213с.

18. *Лаврищева Е. М.* Сборочное программирование. Некоторые итоги и перспективы// Проблемы программирования.− Киев, 1999, №2.− с.20−31.

19. *Грищенко В.М.* Парадигма преобразования данных, передаваемых по сети // Проблемы программирования. − 2001. − № 1−2.− С. 84−94.

20. *Extensible* Markup Language (XML) 1.0 (Second Edition). – W3C Recommendation, 6 October, 2000.

21. *Инг Бей.* Взаимодействие разноязыковых программ. – Москва – С-Петербург – Киев, 2005. – 868 с.

К главе 5

1. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии «Знание».–2001. –269с.

2.. *Соммервилл И.* Инженерия программного обеспечения.– Изд. Дом «Вильямс», Москва *Санкт–Петербург *Киев.– 2002.–623с.

3. *Фаулер М.* Рефакторинг: улучшение соответствующего кода. – СПб.: Символ–Плюс, 2003. – 432 с.

4 *Макконнелл С.* Совершенный код. Практическое руководство по разработке программного обеспечения. Русская редакция.– Питер, 2005.–867с.

5. Д*ейв Т.* Agile – эволюция: направление совершенствования унаследованных программ // Открытые системы.–2006, №8.– с.31–35.

6. *Пантлеймонов А.А.* Аспекты реинженерии приложений с графическим интерфейсом пользователя // Проблемы программирования.– 2001.–№1–2.– С.53–62.

7. *Игнатенко П.П., Неумоин В.Н., Быстров В.М.* Об обеспечении эффективного реинжиниринга прикладных программных систем // Пробл. программирования. – 2001. – №1–2. – с. 42–52.

8. *Гласс Г., Нуазо Р.* Сопровождение программного обеспечения. Пер.с анг. // Под ред. Ю.А. Чернышова.– М.: Мир, 1983.–256с.

9. *Эммерих В.* Конструирование распределенных объектов. Методы и средства программирования интероперабельных объектов в архитектурах OMG/CORBA, Microsoft COM и Java RMI. – М.: Мир, 2002. – 510с.

10. *Хабибуллин И.Ш.* Разработка Web–служб средствами Java. – СПб.: БХВ–Петербург, 213. – 400 с.

11. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.

12. *Ларман К.* Применение UML и шаблонов проектирования: Пер.с англ. – М.: Издательский дом “Вильямс”, 2001. – 496 с.

13. *Уэнди Боггс, Майкл Боггс.* UML и Rational Rose. – М.: “Лори”, 2000г. – 582с.

К главе 6

1. *Марков А.А.* Теория алгоритмов //Москва, АН СССР.–1954.–231с.

2. *Ляпунов А.А.* О логических схемах программ. Проблемы кибернетики вып.1. –Москва, 1958.

3. *Янов Ю.И.* О логических схемах алгоритмов. – Проблемы кибернетики. вып.1.– Москва, 1958.

4. *Hoare C.A.R.* Prof of correctness of data representation // Acta Informatica, 1(4)/-271-287/−1972.−P.214—224.

5. *Андерсон Р.* Доказательство правильности программ.− М.: Мир, 1982.−165с.

6. *Abrial I.R., Meyer B.* Specification Language Z. −Boston: Massachusetts Computer Associates Inc., 1979.−378 p.

7. *Biorner D., Jones C.B.* The Vienna Development Methods (VDM): The Meta − Language.−Vol. 61 of Lecture Notes in Computer Science .− Springer Verlag, Heiderberg, Germany, 1978.−215p.

8. *The RAISE* Language Group. The RAISE Specification Language. BCS Practitioner Series.− Prentice Hall, 1982.−397p.

9. *The RAISE* Methods Group. The RAISE Development Methods. BCS Practitioner Series.− Prentice Hall, 1985.−493p.

10. *Агафонов В.Н.* Спецификации программ: понятийные средства и их организация. − Новосибирск: Наука, 1987.−240с.

11. *Непомнящий В.А., Сулимов А.А.* Об одном подходе к спецификации и верификации трансляторов. − Программирование.−М.: 1983, №4.−с.51−58.

12. *Burstall R.M.* Program proving as hand simulation with a little induction. − Proc. IFIP Congress 74, North−Holland, 1974. −P.80 − 89.

13. *Dijkstra T.W.* Finding the Correctness proof of a concurrent program. − Proc.Konf. Nederland Acad.Wetenach, 1978. − 81. − N2. − p.207− 215.

14. *Pfleeger S.L.* Software Engineering. Theory and Practice. − Prentice Hall, 1998. − 576p.

15. *Непомнящий В.А., Шилов Н.В., Бодин Е.В.* Спецификация и верификация распределенных систем средствами языка Elementary− real."Наука", Программирование", Москва, № 4, 1999,− с.54−67 .

16. *Clarke E.M., Grumberg O., Long D.E.* Model checking and abstraction //ACM Trans. Progr. Languages & Systems. 1994. V. 16. N 5. P. 1512−1542.

17. *De Rover W., Boer P , Hannemann U., Hooman J.* Model Checking. The MIT Press, 1999.

18. *Henzinger A., Manna Z., Pnueli A.* Temporal proof metodologies for real−time systems. Proc. of Symp. on POPL, 1991. −P. 353−366.

19. *Jacobson I.* Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing.− New York: Addison-Wesley Publ.Co, 1994.− 529 p.

20. *Мансуров Н.Н.* Формальные методы для ускоренной разработки телекоммуникационного программного обеспечения// Труды Института системного программирования.− РАН.− Москва: Биоинформсервис, 2000.− с.48−65.

21. *Бабенко Л.П., Лавришева Е.М.* Основы программной инженерии.− Киев: Знання.− 269с.

22. *Dolores R. Wallase M. Ippolito, Cuthill B.*. Reference Information for the Software Verification and Validation Process // NIST Special Publication . − 1996 . − 500−234. − 80p.

23. *Herhart S.L.* Program Verification in the 90's.// Proc.Conf. on Computing in the 1980's, 1978.– P.80–89.

24. *Fei Xie and James C. Browne.* Verified Systems by Composition from Verified Components {feixie, browne}@cs.utexas.edu

25. *Lakhnech, Poel M., and Zwiers J.* Concurrency Verification: Introduction to Compositional and Non–compositional Proof Methods. Cambridge Univ. Press, 2001.

26. *ANSI / IEEE Std. 10122–1986.* Standard for Software Verification and Validation Plans // IEEE . – New York. – 1986. – 61p.

27. *ISO/IEC 12207:2002* – Information technology – Software life cycle processes) Информационные технологии – Процессы жизненного цикла программного обеспечения.

28. *CASE–93.* Proceeding Sixth Intern.//Workshoopen Computer Aided Software Engineering.– Singapure, 1993. – July 19–23. – 418p.

29. *Вудкок Д.* Первые шаги к решению проблемы верификации программ. Открытые системы, 2006.–№8.– с. 36–43.

30 *Hoare T., Misra J.* Verified software: Theories, Tools, Experiments. Vision of Grant Challenge project..– Microsoft Research Ltd and the University of Texas at Austin.–July 2005.– 1–43c.

К главе 7

1. *Майерс Г.* Искусство тестирования программ. Пер. с англ. –М: Финансы и статистика, 1982. – 176 с.

2. *Липаев В.В.* Отладка сложных программ. –М.: Энергоатомиздат, 1993.–296с.

3. *Липаев В.В.*. Тестирование программ.–М: Радио и связь,–1986.–295с.

4. *Канер С., Фолк Д., Нгуен Е.К.* Тестирование программного обеспечения: Пер с англ. – К.: DiaSoft. – 2000. – 544 с.

5. *Weyuker E.J., Ostrand T.J.* Theories of program testing and the application of revealing subdomains // IEEE Trans.Soft.Eng. – 1980, –V.6, –№3, – P. 236–246.

6. *Software unit test* coverage and adequacy / Zhu H., Hall P. A. // ACM Computing Surveys, 29, – № 4, Dec. 1997. – P. 336–427.

7. *Коул Дж., Горем Т. и др.* Принципы тестирования ПО // Открытые системы. – 1998.– №2. www.osp.ru/os/1998/02/60.htm

8. *Бабенко Л.П., Лаврищева К.М.* Основы программной инженерии. – К.: Знання, 2001. – 269 с.

9. *Соммервил И.* Инженерия программного обеспечения.– Изд-во "Вильямс".– Москва, Санкт–Петербург – Киев, 2002.– 623с.

10. *Software Engineering Body* of Knowledge (SWEBOK). // ISO/IEC JTC1/SC7 N2517. Software & System Engineering Secretariat, Canada, 2001. – 220 p.

11. *ISO/IEC 12207:2002* – Information technology – Software life cycle processes) Информационные технологии – Процессы жизненного цикла программного обеспечения.

12. *Основы инженерии качества* программных систем/ Ф.И.Андон, Г.И.Коваль, Т.М. Коротун, В.Ю. Суслов / Под ред. И.В. Сергиенко. − К.: Академпериодика, 2002. − 504 с.

13. *Pfleeger S.L.* Software Engineering. Theory and Practice. − Prentice Hall, 1998. − 576p.

14. *Coulouris G.F. and Dollimore J.* Distributed systems. Concepts and Design, Inter. Comp. Sci. Series.− London: Addison−Wesly Publ. Co, 1993.− 366p.

15. *Perry D.E. and Kaiser C.E.* Adequate testing and object−oriented programming .− Journal of Object−Oriented Programming, January / February.− 1990.− p.13−19.

16. *Wang Y., King J., Kourt J., Ross M.*, S.Staples. On testable object− oriented programming. Software Engineering Notes, volume 22, N4. −1997.− p.84−90

17. *Лаврищева Е.М., Коротун Т.М.* Построение процесса тестирования программных систем // Проблемы программирования.− 2002.−№1− 2.−С.272−281.

18. *Коротун Т.М.* Модели и методы инженерии тестирования программных систем в условиях ограниченных ресурсов,−Автореф. диссеер. Киев.− Институт кибернетики им. Академика Глушкова.−2005.− 21с.

19. *Capability Maturity* Model for Software, Version 1.1 / M.Paulk, B.Curtis at all // CMU− SEI− 93−TR−024, Soft. Engin. Institute, Pittsburg PA 15213, Feb. − Pittsburg. − 1993. − 82 p.

20. *Burnstein, I., T. Suwannasart, and C. R. Carlson,* Developing a Testing Maturity Model: Part I. / Crosstalk, STSC, Hill Air Force Base, Utah, August 1996, − P. 21−24.

21. *Drabick R.* Growth of maturity in the testing process.
http://www.softtest.org.articles/rdrabick3.htm

22. *Towards a* Test Improvement Model / Ericson T., Subotic A., Ursing S. // Proceedings of the Fourth European Conference on Software Testing, Analysis & Review, Amsterdam, December 2−6. − 1996. http://www.brunel.ac.uk/csstrmh/papers/wess99.ps

23. *Koomen, T., and Pol M.*. Improvement of the test process using TPI, 1998. http://www.iquip.nl

24. *Homyen A.* An Assessment Model to Determine Test Process Maturity / Diss., Illinois Institute of Technology, July 1998 Test Improvement Model − TPIM. http:// www.swebok.org/stoneman/version_0.9

25. *Drabick R.* Growth of maturity in the testing process. International Software Testing Institute 1999. http://www.softtest.org/articles/rdrabick3.htm.

К главе 8

1. *Барлоу Р., Прошан Ф.* Математическая теория надежности: Пер.с анг.− М.: Наука, 1969.−483с.

2. *Липаев В.В.* Надежность программного обеспечения. −М.: СИНТЕГ, 1998.−231с.

3. *Липаев В.В.* Методы обеспечения качества крупномасштабных программных систем. − М.: СИНТЕГ, 2003.−510 с.

4. *Майерс Г.* Надежность программного обеспечения,− М.: Мир, 1980.−360с.

5. *Мороз Г.Б., Лаврищева Е.М.* Модели роста надежности программного обеспечения.− Киев: Препринт 92−38, 1992.− 23с.

6. *Коваль Г.И.* Подход к прогнозированию надежности ПО при управлении проектом // Проблемы программирования. −2002. − № 1 − 2. − С. 282 − 290.

7. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии,− Знание, 2001.− 269с.

8. *Fenton N.E., Neil M.* A critique of software defect prediction models // IEEE Trans. On Soft. Eng. − 1999. − V. 25. − N.5. − P. 675 − 689.

9. *Chulani S.* Constructive quality modeling for defect density prediction: COQUALMO // Internat. Symposium on Software Reliability Engineering (ISSRE'99), Boca Raton, N. 1−4. − 1999.

10. *Кузнецов Ю.Н., Кузубов В.И., Волощенко А.Б.* Математическое программирование, М.: Высш. школа, 1976.− 352 с.

11. *Клебанова Т.С., Иванов В.В., Дубровина Н.А.* Методы прогнозирования. − Харьков: ХГЕУ, 2002. − 372 с.

12. *Мороз Г.Б.* Пуассоновские модели роста надежности программного обеспечения и их приложение. Аналитический обзор // УСиМ. − 1996. −№ 1−2. − С. 69 − 85.

13. *Гнеденко Б. В., Беляев Ю.К., Соловьев А.Д.* Математические методы в теории надежности. − М.: Наука − 1965. − 324 с.

14. *Кулаков А.Ф.* Оценка качества программ ЭВМ.− Киев: Техніка, 1984.−167с.

15. *Goel A.L.* Software reliability models& Assumptions, Limitations and Applicability// IEEE Trans.− N2. − p.1411−1423.

16. *Sukert A.N., Goel A.L.* A guidebook for software reliability assessment /Proc. Annual Reliability and Maintainability Symp. − Tokio (Japan). − 1980. −P.186 − 190.

17. *Shick G.J., Wolverton R.W.* An analysis of computing software reliability models /IEEE Tras. Software Eng. − V. SE−4. − № 2. − 1978. − P. 104−120.

18. *Shanthikumar J.G.* Software reliability models: A Review //Microelectron. Reliab. − 1983. −V. 23. −№ 5 − P. 903−943.

19. *Ohba M.* Software Reliability Analysis Models // IBM J.Res.Develop. −1984. − 28. − № 4. − P.428−243.

20. *Musa John D., Anthony Iannino, and Kazuhira Okumoto.* Software Reliability: Measurement, Prediction, Application. Whippany, NJ: McGraw− Hill, 1987.

21. *Schneidewind N.F.* Software Reliability Model with Optimal Selection of Failure Data //IEEE Trans. on Software Eng. − 1993. − № 11. −P. 1095−1104.

22. *Goel Amrit L.,* "Software reliability models: Assumptions, limitations, and applicability. //IEEE Transactions on Software Engineering, Vol. SE−11, № 12. − 1985. −P. 1411−1423.

23. *Musa J.D. Okumoto K. A.* Logarithmic Poisson Time Model for Software Reliability Measurement //Proc. Sevent International Conference on Software Engineering. – Orlando, Florida. – 1984. –P. 230–238.

24. *Yamada S., Ohba M., Osaki S.* S–shaped software reliability grows modeling for software error detection // IEEE Trans. Reliability. – 1983. – R–32. – № 5. – P. 475–478.

25. *Jelinski Z. and Moranda P.* Software reliability research /Statistical computer performance evaluation W.Freiberger, Ed. Academic Press. –1972. – P. 465–484.

26. *Malaiya Y.K., Denton J.* What do the Software Reliability Growth Model Parameters Represent // Proc. IEEE–CS Int. Symp. on Software Reliability Engineering ISSRE. – Nov. 1997. – P. 124 – 135.

27. *Musa J.D.* Operational Profiles in Software Reliability Engineering // IEEE Software – V.10.– № 2, 1993.– P. 14–32.

28. *Musa J.D., Iannino A., Okumoto K.* Software Reliability Measurement, Prediction, Application // McGraw–Hill. – 1987. – 354 p.

29. *Kanoun K., Laprie J.* Software reliability trend analysis from theoretical to practical considerations // IEEE Trans. on Software Eng. – 20. – № 9. –P. 740–747.

30. *ANSI/AIAA* American National Standard Recommended Practice for Software Reliability, R–013–1993, Feb. 23, 1993.

31. *Farr W.* Software Reliability Modeling Survey // Handbook of Software Reliability Engineering, Ed. M. R. Lyu.– McGraw–Hill, 1996. – P. 71–117.

32. *Моррис У.* Наука об управлении. Байесовский подход. – М.: Мир, 1971. –304 с.

33. *Коваль Г.И.* Модели и методы инженерии качества программных систем на ранних стадиях жизненного цикла.– Автореф. канд. физмат. наук.– К.: ИКНАНУ, 2005. – 21с.

34. *ISO/IEC* 12207: 2002. – Information technology – Software life cycle processes.

К главе 9

1. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии (укр.).– Киев: Знання, 2001.–269с.

2. *Соммервил И.* Инженерия программного обеспечения. 6 – издание.– Москва–Санкт–Петербург–Киев, 2002.–623 с.

3. *Чернецки К., Айзенекер У.*. Порождающее программирование. Методы, инструменты, применение.– Издательский дом «Питер».– Москва– Санкт–Петербург... Харьков, Минск, 2005.– 730с.

4. *ISO/IEC* 12207: 1995.– Information technology – Software life cycle processes) Информационные технологии – Процессы жизненного цикла ПО.

5. *Глушков В.М.* Кибернетика, вычислительная техника, информатика.– К.: Наукова думка, Избранные труды в трех томах, том 3.– 1990. – 223с.

6. *Weide B., Ogden W., Sweden S.* Reusable Software Components/ Advances in Computers, vol. 33. − Academic Press, 1991. − p.1−65.

7. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования // Кибернетика и системный анализ, 2003.− №1.− с.39−55.

8. *Northrop L.M.* SEI`s Software Product Line Tenets // IEEE Software. −2002. − v.19. − №4. − 32−39.

9. *ISO/IEC* 9126-1:2001 Software Engineering − product quality. Part 1 Quality model.

10. *ANCI/IEEE* 730−1. IEEE Standard for Software Guality Analisys Plans. −1989.

11. *ISO/IEC* 9126−2. Information Technology. − Software Quality Characteristics and metrics.− 1997.

12. *ДСТУ* 2844−1994. Программные средства ЭВМ. Обеспечение качества. Термины и определения.

13. *ДСТУ* 2850−1994. Программные средства ЭВМ. Обеспечение качества. Показатели и методы оценки качества программного обеспечения.

14. *ДСТУ* 3230−1995. Управление качества и обеспечение качества. Термины и определения.

15. *Липаев В.В.* Методы обеспечения качества крупномасштабных программных средств. − М.: СИНТЕГ, 2003.−520 с.

16. *Липаев В.В.* Качество программных средств. Методические рекомендации / Под ред. А.А. Полякова.− М: Янус−К.− 2002.−400 с.

17. *Ф.И.Андон, Г.И.Коваль , Т.М.Коротун, В.Ю. Суслов.* Основы инженерии качества программных систем. − К: Академпериодика, 2002. − 502 с.

18. *Коваль Г.И.* Модели и методы инженерии качества программных систем на ранних стадиях жизненного цикла, Автореферат канд. диссерт. − Киев, Ин-т кибернетики НАНУ, 2005.− 21с.

19. *Лаврищева Е.М., Коваль Г.И., Коротун Т.М.* Подход к управлению качеством программных систем обработки данных // Кибернетика и системный анализ.− 2006.−№ 6. − С.174−185.

20. *Лаврищева Е.М., Рожнов А.М.* Концепция аналитической оценки характеристик качества программных компонентов // Проблемы программирования.− Киев, 2004, № 1−2. − С.180−187.

21. *Кулаков А.Ф.* Оценка качества программ ЭВМ.−Киев: Техніка, 1984. − 167 с.

22. *NASA−STD−2201/* Software Assurance Standart, 1993.

23. *ISO/IEC* TR 15504, Information Technology−Software Process Assessment (Part 1−9).

24. *Гост 3918-99.* Информационные технологии. Процессы жизненного цикла ПО, 2000. − 49с.

25. *Лаврищева Е.М. Грищенко В.Н.* Области знаний программной инженерии − SWEBOK и подход к обучению этой дисциплины // УСиМ, №1, 2005.−с.38−54 // http:/www.swebok.org

26. *Мороз Г.Б., Лаврищева Е.М.* Модели роста надежности программного обеспечения. − Киев. − 1992. −23с.− (Препр. 92−38).

27. *Коваль Г.И.* Подход к прогнозированию надежности ПО при управлении проектом // Сб. материалов конф. УкрПРОГ'2002, 21–22 мая 2002 г., Киев.–2002. – С. 282–290.

28. *Мороз Г.Б.* Пуассоновские модели роста надежности программного обеспечения и их приложение. Аналитический обзор // УСиМ. – 1996. № 1–2. – С. 69 – 85.

29. *Meyer B.* The role of Object–Oriented Metrics, // Computer, 1998. – №11.–P. 23–125.

30. *IEEE* Software. Measurement. – March/April, 1997.

31. *Haag S., Raja H.K., Sekade L.L.* Quality Function Deployment. Usage in Software Development// Comm. of ACM. –1998. –9, N1.

32. *Zahedi F., Ashrafi N.* Software Reliability Allocation Based on Structure, Utility, Price, and Cost / IEEE Trans. On Softw. Eng. – 1991. – V. 17. – N.4. – P. 345-356.

33. *Musa J.D., Iannino A., Okumoto K.* Software Reliability Measurement, Prediction, Application // McGraw-Hill. – 1987. – 354 p.

34. *Липаев В.В.* Надежность программных систем. – М.: СИНТЕГ, 1998. – 321 с.

35.*Fenton N.E., Neil M.* A critique of software defect prediction models // IEEE Trans. On Soft. Eng. – 1999. – V. 25. – N.5. – P. 675–689.

36. *Hugin Lite* 6.5. Продукт Hugin Expert //www.hugin.com/ roducts_Services/ roducts/Demo/Download/

37. *Коротун Т.М., Лаврищева Е.М.* Построение процесса тестирования программных систем // Проблемы программирования.– 2002. – №1–2. – С.272–281.

38. *Capability Maturity* Model for Software, Version 1.1 / M.Paulk, B.Curtis at all // CMU– SEI– 93–TR–024, Soft. Engin. Institute, Pittsburg PA 15213, Feb. – Pittsburg. – 1993. – 82 p.

К главе 10

1. *Reiter D.J.* Software management, IEEE Computer Society Press, Los Alomos.– 1993.

2. *B. Duncan.* A Guide to the Project Management Body of Knowledge // PMBOK GUIDE.–2000.– Edition / www.pmi.org/publication/download/ 2000welcome.html.

3. *Thayer R.H.*, ed., Software Engineering Project Management, 2nd.ed., IEEE CS Press, Los Alamitos, Calif. 1997.–391p.

4. *Черников А.* Теория и практика управления проектами // Компьютерное обозрение. – 2003.– №10 – С. 24–39.

5. *Гультяев А.К.* Microsoft Project 2002. Управление проектами. – СПб.: Корона принт, 2003. – 589 с.

6. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных приложений. – К.: Наукова думка, 1997. – 228с.

7. *Ройс У.* Управление проектами по созданию программного обеспечения. – М.: Лори, 2002. – 424с.

8. *Боэм Б.*У. Инженерное проектирование программного обеспечения. − М: Радио и связь.−1985.− 511с.

9. *ISO/IEC* TR 16326:1999. Guide for the application of ISO/IEC 12207 to project management.

10. *IEEE Std.* 1058-1998. IEEE Standard for Software Project Management Plans.

11. *Задорожная Н.Т.* Управляемое проектирование документооборота в управляющих информационных системах.− Автореф. канд. диссерт.− Киев, Ин-т кибернетики им. Глушкова, 2004.−21с.

12. *ISO/IEC TR* 15504, Information Technology−Software Process Assessment (Part 1 − 9).

13. *Бабенко Л.П., Лаврищева Е.М.* Основы програмной инженерии. Учебник (укр.язык). − Киев: Знание, 2001. −269 с.

14. *Андон Ф.И.,Суслов В.Ю., Коротун Т.М., Коваль Г.І., Слабосницкая О.О.* Определение затрат на создание ПО автоматизированных систем (укр.) // Проблемы программирования.− 1998.− Вып. 3. − с.23−34.

15. *ГОСТ* 34.602−86 Автоматизированные системы управления. Состав и содержание работ по стадиям создания. − М.: Изд-во стандартов, 1986. − 11 с.

16 *ГОСТ 34.*601−86. Автоматизированные системы. Стадии создания. − М.: Изд-во стандартов, 1986. − 5 с.

17. *ANCI/IEEE* 730−1. IEEE Standard for Software Guality Analisys Plans. −1989.

18. *Pfleeger S.L.* Software Engineering. Theory and Practice. − Prentice Hall, 1998. − 576р.

19. *ISO/IEC* 12207: 1995.− Information technology - Software life cycle processes) Информационные технологии - Процессы жизненного цикла ПО.

20. *Соммервил И.* Инженерия программного обеспечения. 6 изд.− Москва−Санкт−Петербург−Киев, 2002.− 623с.

21. *Брукс Ф.* Мифический человеко-месяц или как создать программные системы. − Спб.: Символ-плюс, 2005. − 304с.

22. *Глушков В.М.* Кибернетика, вычислительная техника, информатика.− К.: Наукова думка, Избранные труды в трех томах, том 3.− 1990. − 223с.

23. *Романов Д.А., Ильина Т.Н., Логинова А.Ю.* Правда об электронном документообороте. − М.: БизнесПРО, 2002. − 219 с

24. *Клепцов М.Я.* ИС ОГУ. − М.: Изд. РАГС, 1996. − 160 с.

25. *Клепцов М.Я, Кононенко А.В., Коновалов С.М., Кушлин В.Н.* Технология разработки и внедрения ИС для органов государственной власти и управления. − М.: Изд. РАГС, 1996. − 362 с.

27. *Лешек А. Мацящек.* Анализ требований и проектирование систем. Разработка информационных систем с использованием UML. − М.: Вильямс, 2002. − 428 с.

28 *Леффингуэлл Дин, Уидриг Дон.* Принципы работы с требованиями к программному обеспечению. Унифицированный подход.− М.: Вильямс, 2002. − 446 с.

29. *Клейнрок Л.* Теория массового обслуживания.– М.: Машиностроение, 1979. –432с.

30 *Кокс Д.Р., Смит У. Л.* Теория очередей.– М.: Мир, 1979.–218с.

31. *Задорожная Н.Т., Лаврищева Е.М.* Управляемое проектирование документооборота в управляющих информационных системах // Программирование.– №4.–2006.–с.37–48.

32. *Bradford C., Devnani-Chulan S., Boehm B. W.* Calibrating the COCOMO II Post-Architecture Model. ICSE 1998. – p.477–480.

33. *CSE* – Center for Software Engineering, " COCOMO II Reference Manual," Computer Science Department, USC Center for Software Engineering, 1999.

К главе 11

1. *Pfleeger S.L.* Software Engineering. Theory and Practice. – Prentice Hall, 1998. – 576p.

2. *Липаев В.В.* Тестирование программ.– М.: Радио и связь, 1986.–296с.

3. *Липаев В.В.* Методы обеспечения качества крупномасштабных программных средств.– М.: СИНТЕГ, 2003.–510с.

4. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии. Учебник (укр. язык). – Киев: Знание, 2001. –269 с.

5. *Основы инженерии* качества программных систем / Ф.И. Андон, Г.И. Коваль, Т.М. Коротун, В.Ю. Суслов / Под ред. И.В. Сергиенко. – К.: Академпериодика, 2002. – 504 с.

6. *Соммервил И.* Инженерия программного обеспечения. 6 изд.– Москва–Санкт–Петербург – Киев, 2002.– 623с.

7. *Ф. Брукс.* Мифический человеко-месяц или как создать программные системы. – СПб.: Символ-плюс, 2005. – 304с.

8. *Лаврищева Е.М. Грищенко В.Н.* Области знаний программной инженерии – SWEBOK и подход к обучению этой дисциплины // УсиМ, №1, 2005. – с.38–54.

К приложению 1

1. www.swebok.org, malto:sorlic@borland.ru

2. *Бабенко Л.П., Лаврищева Е.М.* Основы программной инженерии. Учебник (укр. язык). – Киев: Знання, 2001. –269 с.

3. *Лаврищева К.М.* Основные направления исследований в программной инженерии и пути их развития // Проблемы программирования. – 2003. – №3–4.– С.44–58.

К приложению 2

1. ISO/IEC 12207: 2002.– Information technology – Software life cycle processes) Информационные технологии – Процессы жизненного цикла программного обеспечения.

СОДЕРЖАНИЕ

ЛАВРИЩЕВА ЕКАТЕРИНА МИХАЙЛОВНА

заведующая отделом Института программных систем НАН Украины, доктор физико-математических наук, профессор кафедры теоретической кибернетики и методов оптимального управления Московского физико-технического института при Институте кибернетики имени академика В.М. Глушкова, лауреат премии Совета Министров СССР (1985) и Государственной премии Украины (1991, 2003), руководитель десяти кандидатов наук и автор 100 научных публикаций по проблематике программирования, в том числе, монографии в соавторстве «Связь разноязыковых модулей в ОС ЕС» и «Сборочное программирование» (Москва: Финансы и статистики, 1982; 1991); «Методы инженерии распределенных компьютерных приложений» (Киев: Наукова думка, 1997) и учебника «Основы программной инженерии» (Киев: Знання, 2001).

Подписана книга к печати 13.10.2006.

Наукове видання

Національна академія наук України
Інститут програмних систем

Лавріщева Катерина Михайлівна

МЕТОДИ ПРОГРАМУВАННЯ

Теорія, інженерія, практика

Київ, Науково-виробниче підприємство
«Видавництво Наукова думка» НАН України, 2006

Редактор	М. К. Пуніна
Художній редактор	С.В. Вероцький
Комп'ютерна верстка	О.О. Єрмоленко

Підп. до друку 13.10.06. Формат 60х90/16.
Гарн. «Таймс». Папір офс. №1. Офс. друк. Обл.-вид. арк. 26,78.
Ум.-друк. арк. 28,25. Замовлення № 41-159. Тираж 300 прим.

НВП «Видавництво «Наукова думка» НАН України»
Р.с. № 05417561 від 16.03.95.
01601, Київ, вул. Терещенківська, 3

453