УДК 681.3

# СОВРЕМЕННЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ: ВОЗМОЖНОСТИ И ИНСТРУМЕНТЫ

*Е.М. Лаврищева*

Институт программных систем НАНУ, пр.Академика Глушкова,40
тел. 526 3470, e-mail: lem@isofts.kiev.ua

Анализируются современные широко используемые методы систематического и теоретического программирования. Рассмотрены их возможности и инструменты для проектирования программных систем. Среди этих методов определяются перспективные – порождающее и агентно-ориентированное.

Modern wide using the methodical and the theoretical methods programming are analyzed. Opportunities and tools of these methods programming for systems designing are considered. Generative and agent-oriented methods are perspective among the others in future.

## Введение

В последние годы в программировании четко прослеживалось становление и развитие объектно-ориентированного проектирования (ООП) программных систем (ПС) [1, 2]. Определилась не только технологическая база разработки разных видов ПС, но и инструментальные поддержки (Rational Rose, Rational Software, RUP, Demral , OOram и др.).

Совершенствование ООП достигло такого уровня, что он стал базисом генерирующего (порождающего) программирования, в котором определены пути его развития и устранения ряда недостатков, связанных не только с особенностью проектирования уникальных ООП ПС, но и с отсутствием механизмов использования готовых компонентов, свойств изменчивости, взаимодействия, синхронизации и др. [3] В данном программировании объединены и другие концепции и методы программирования с целью обеспечения инженерии предметной области (ПрО), ориентированной на проектирование семейств ПС из объектов, компонентов, аспектов, сервисов, повторного использования компонентов (ПИК), систем, характеристик и т.п. Все это расширило рамки всех концепций и положило начало объединенной целостной и стройной технологии генерации как отдельных ПС, так и их семейств. Порождающее программирование заложило базис будущего программирования, основанный на современных методах программирования, новых формализмах и объединяющих моделях, посредством которых новое поколение программистов будут создавать более долговечные и качественные программные изделия семейств ПС с использованием конвейерной автоматизации.

Другие методы систематического программирования ПС (структурный, модульный, компонентный и др.) достигли своего апогея, и без бурных всплесков широко и последовательно используются на практике с применением соответствующих инструментов проектирования ПС [4 – 9].

Важное место среди методов программирования заняло аспектно-ориентированное программирование (АОП) [10, 11], главной задачей которого является построение 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 и др. Аспект в программировании – это нечто, в чем проявляется интерес одного или нескольких заинтересованных лиц [12]. Им может быть функция, ПИК, элемент или часть готовой программы, компонент, концепция взаимодействия, защиты и др. Созданная по модели ПС в рамках АОП может включать набор ПИК, мелкие методы и аспекты, которые пересекают (переплетают) их и тем самым усложняют процесс вычислений. Для преодоления такого рода явлений аспект представляется модулем или функцией как средства связи с другими объектами ПС и снижения количества переплетений кодами аспектов функциональных компонентов системы. Реализация аспектов в различных частях программного кода решается также путем установления перекрестных ссылок и точек соединения, через которые обеспечивается связь аспекта с транзакциями, обработкой ошибок, распределенным доступом и т.п.

Наряду с практическими методами программирования ПС развивались и теоретические методы (алгебраическое, инсерционное, агентное, композиционное, экспликативное и др.), которые основываются на математических, алгебраических и логико-алгоритмических подходах и методах формального построения и доказательства программ. Авторами украинской теоретической школы программирования [13-16] созданы новые теоретические средства для решения ключевых проблем программирования. Теория алгебраического программирования обеспечивает описание математических конструкций, вычислений, алгебраических преобразований, доказательство математических теорем, а также новых механизмов создания интеллектуальных агентов [17–19].

Экспликативное программирование предлагает 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 [20–22].

В данной статье рассматриваются основные возможности и особенности широко распространенных методов систематического и теоретического программирования. Определяются перспективные из них.

## 1. Методы систематического программирования

К ним отнесем следующие:

– модульный, сборочный;
– структурный;
– объектно-ориентированный;
– моделирование в UML;
– компонентный;
– аспектно-ориентированный и др.

Остановимся на краткой характеристике возможностей основных методов, инструментальной поддержке и направлениях развития.

### 1.1. Модульное, сборочное программирование

**Возможности.** Одной из ключевых проблем раннего программирования (70–90е годы прошлого столетия) является формирование понятия модуля и использование его в качестве строительных блоков новых ПС. Программирование с помощью модулей прошло длинный путь своего развития от библиотек стандартных подпрограмм общего назначения, библиотек, банков модулей до современных библиотек классов, методов вычисления матриц и т.п.

*Модуль* – это логически законченная часть программы, выполняющая определенную функцию. Он обладает такими свойствами: завершенность, независимость, самостоятельность, раздельная трансляция, повторное использование и др. Модуль раньше других программных объектов накапливался в библиотеках или Банках программ и модулей как готовых «деталей», из которых собирается ПС и адаптируется к новым условиям среды обработки. Программирование с помощью модулей привело к появлению *сборочного программирования (СБ)* как метода проектирования ПС снизу вверх из более простых элементов – модулей. Идею сборки модулей по принципу конвейера сформулировал академик В.М.Глушков в 1976 г. С этого момента в Институте кибернетики АН УССР проводились разработки методов автоматизации построения сложных программ из модулей. В том числе система автоматизации программ – АПРОП (1976-1990 гг.), основанная на понятиях: модуль с паспортными характеристиками, интерфейс для стыковки модулей и автоматизированная сборка по принципу фабрики программ.

Базовой концепцией СБ из готовых модулей, записанных в языках программирования – ЯП (Фортран, ПЛ/1, Алгол, Ассемблер), является интерфейс, обеспечивающий связь модулей в этих языках. Идея интерфейса значительно позже получила развитие, когда мировое сообщество пришло к необходимости объединения разных видов программ для разных систем и машин. 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 СБ определяется как модуль-посредник, в котором описываются операторы вызова модуля, его параметры и их типы. Другой тип интерфейса – это интерфейс пары ЯП, сущность которого заключается в преобразовании несовпадающих структур и типов данных в этих ЯП, которое возникает при вызове процедур в этих ЯП. Это преобразование осуществляется статически с помощью заранее разработанных функций и макросов релевантного преобразования библиотеки интерфейса, которая для указанных ЯП включала более 65 таких элементов.

Созданная концепция интерфейса модулей для класса ЯП положена в основу метода сборки (интеграции) модулей в ПС [5, 23–25]. Метод сборки основывается на математических формализмах определения связей (по данным и по управлению) между разнородными модулями и функциях преобразования данных в модуле-посреднике для каждой пары разноязыковых модулей, в которых содержатся операторы CALL к вызывающему модулю.

Задача обеспечения интерфейса решается путем построении взаимно-однозначного соответствия между множеством фактических параметров $V= \{v^1, v^2, \ldots, v^к\}$ вызывающего модуля и множеством формальных параметров $F = \{ f^1, f^2, \ldots, f^{кI} \}$ вызываемого модуля и их отображения в релевантные типы с помощью алгебраических систем преобразования данных в классе ЯП.

Алгебраическая система $G_\alpha^t = <X_\alpha^t, \Theta_\alpha^t>$, где $X_\alpha^t$ – множество значений типа данных, а $\Theta_\alpha^t$ – множество операций над этим типом, ставит в соответствие типу $T_\alpha^t$ языка $l_\alpha$ операции преобразования и изоморфное отображение алгебраических систем $G_\alpha^t$ в $G_\beta^d$. Построенный класс алгебраических систем $\Sigma = \{G_\alpha^b, G_\alpha^c, G_\alpha^i, G_\alpha^r, G_\alpha^a, G_\alpha^z\}$ для типов данных ЯП (b-boolean, c-character, i-integer, r -real, a -array, z–record и

др.) учитывает все виды преобразований. Доказан изоморфизм алгебраических систем класса Σ, кроме значений $X_\alpha^t$ и $X_\beta^q$ для типов $T_\alpha^t$ и $T_\beta^q$ из-за отсутствия соответствия.

Инструменты. Система АПРОП [23 – 24] воплотила метод сборки разноязыковых модулей. В дальнейшем в качестве готовых элементов ПС стали использовать любые порции формализованных знаний (объекты, компоненты, программы, каркасы и др.), полученные при реализации отдельных программ и систем. Возникли новые проблемы их использования, например перенос на другую платформу или другую среду функционирования, т.е. проблема интероперабельности. Стандартным механизмом решения этой проблемы на данный момент является обеспечение связи Java- и C/C++- компонентов с помощью интерфейса JNI (Java Native Interface). Обращение Java-классов к функциям и библиотекам на других ЯП включает: 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 [25]. Для платформы .Net интероперабельность реализована средствами языка CLR (Common Language Runtime), в который транслируются объекты в ЯП (C#, Visual Basic, C++, Jscript), используется библиотека стандартных классов независимо от ЯП и стандартные средства генерации в представление .Net-компонента. Особенности ОС и архитектур компьютеров учитывает также среда CORBA. Стандарт ДСТУ [26] обеспечивает независимо от ЯП спецификацию типов данных разноязыковых компонентов средствами специального языка, механизмов их агрегации и преобразования. Язык этого стандарта – альтернатива IDL, API, RPC.

## 1.2. Структурное программирование

**Возможности.** Основу структурного метода программирования составляет декомпозиция создаваемой системы на отдельные функции и задачи, которые в свою очередь разбиваться на более мелкие. Процесс декомпозиции продолжается вплоть до определения конкретных процедур. Для функций разрабатываются программные компоненты и устанавливаются связи между ними [27-29].

Метод структурного программирования базируется на двух общих принципах:

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

– 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 решение проблемы;

– непротиворечивость – обоснование и согласование элементов системы;

– структуризация данных согласно иерархической организации.

Сформировался ряд моделей этого метода программирования, главными из которых являются:

– SADT (Structured Analysis and Design Technique) – структурный анализ и техника проектирования [29];

– SSADM (Structured Systems Analysis and Design Method) – метод структурного анализа и проектирования [28] и др.

На стадии проектирования эти модели дополняются структурными схемами и диаграммами.

*SADT* представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели ПрО. Функциональная модель в SADT отображает структуру функций объекта автоматизации, производимые действия и связи между ними. Концепции метода: графическое представление блочного моделирования функций в виде блоков с интерфейсами (вход/выход) в виде дуг, задающих взаимодействие блоков в соответствии с правилами:

– строгость и точность количества блоков на каждом уровне декомпозиции (от 3 до 6 блоков) и связь диаграмм через номера этих блоков;

– уникальность меток и наименований;

– разделение входов и управлений (определение роли данных).

– исключение влияния организационной структуры на функциональную модель.

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

*SSADM* базируется на таких структурах: последовательность, выбор и итерация (цикл). Моделируемый объект задается сгруппированной последовательностью компонентов, следующих друг за другом, операторами выбора компонентов из группы и цикличеким выполнением отдельных компонентов. Каждый компонент изображается на диаграмме четырехугольником с прямыми или закругленными углами и дугами с входными и выходными данными.

Базовая диаграмма является иерархической и включает: список всех компонентов описываемого объекта; идентифицированные группы выбранных и повторяемых, а также последовательных компонентов. Модель процесса включает структурные диаграммы, которые в SSADM создаются в процессе:

– определения функций;

– моделирования взаимосвязей событий и сущностей;

– логического проектирования данных;

– проектирования диалога;

– логического проектирования БД;

– физического проектирования.

Наиболее распространенным средством моделирования данных являются диаграммы "сущность-связь" (ERD), с помощью которых определяются объекты (сущности) ПрО, их свойства (атрибуты) и отношения (связи) [24]. Сущность (Entity) – реальный либо воображаемый объект, существенный для ПрО. Каждая сущность и ее экземпляр имеют уникальные имена. Сущность обладает 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 – с одним экземпляром сущности-родителя. Таким образом, экземпляр сущности-потомка может существовать только при наличии сущности «родитель».

**Инструменты.** Основным инструментом является комплекс инструментальных, методических и организационных средств системы SSADM. Она принята государственными органами Англии в качестве основного системного средства и используется многими организациями как внутри страны, так и за ее пределами. На основе идеологии структурного программирования создан ряд CASE-средств (SilverRun, Oracle Disigner, Erwin для прямой и обратной связи с Rational Rose и др.).

## 1.3. Объектно-ориентированное проектирование (ООП)

**Возможности.** ООП [1, 2, 30–36] представляет собой стратегию проектирования ПС из объектов. *Объект* – это нечто, способное пребывать в различных состояниях и имеющее множество операций. Состояние определяется как набор атрибутов объекта. Операции, связанные с объектом, предоставляют сервисы другим объектам для выполнения определенных вычислений. Объекты объединяются в классы, каждый из которых имеет описания всех атрибутов и операций.

ПС состоит из взаимодействующих объектов, имеющих локальное состояние и способных выполнять набор операций, определяемый состоянием объекта. Объекты инкапсулируют информацию об их состоянии и ограничивают к ним доступ.

*Процессы ООП* – это проектирование классов объектов и взаимоотношений между ними. Проект ПС реализуется в виде исполняемой программы, в которой все необходимые объекты создаются динамически с помощью классов. ООП включает три этапа проектирования ПС [31]:

– *Анализ ОО*. Создание ОО модели предметной области, в которой объекты отражают реальные ее сущности и операции, выполняемые ими;

– *Проектирование ОО*. Уточнение ОО модели с учетом требований для реализации конкретных задач системы;

– *Программирование ОО*. Реализация ОО модели средствами C++, Java и др.

Данные этапы могут происходить друг за другом, и на каждом этапе может применяться одна и та же система нотаций. Переход к следующему этапу приводит к усовершенствованию результатов предыдущего этапа путем более детального описания определенных ранее классов объектов и определения новых.

Изменение реализации какого-нибудь объекта или добавление ему новых функций не влияет на другие объекты системы. Четкое соответствие между реальными объектами (например, аппаратными средствами) и управляющими объектами ПС облегчает понимание и реализацию проекта в целом.

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

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

**Инструменты.** Одной из инструментальной поддержки метода ООП является RUP [37] – неофициальный стандарт разработки одиночных ПС из элементов Use Case, отвечающих требованиям конечных пользователей. Варианты использования – главные элементы этого процесса включают группы сценариев, описывающих применение ПС и интеграцию на этапах разработки с измерением времени выполнения этапов и измерения компонентов процесса на качество. При измерении времени выполнения этапов используются контрольные точки ПС. Компоненты процессов RUP описываются в категориях действий, рабочих потоков и исполнителей, которые измеряются и оцениваются. Основным недостатком ООП является отсутствие возможности задавать такие аспекты, как синхронизация выполнения объектов, удаленное взаимодействие в распределенной среде, защита данных, устойчивость и т.д. Этот недостаток относится и к компонентным технологиям ActiveX и JavaBeans и получит развитие в генерирующем программировании.

## 1.4. Метод моделирования UML

Метод UML (United Modeling Language – унифицированный язык моделирования) [34–40] широко используется программными организациями как метод моделирования ПС исходя из таких базовых понятий:

– 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.

Язык UML поддерживает задание статических и динамических моделей, в том числе концептуальной модели и модели последовательностей, узлы которой определяют последовательность взаимодействий между объектами. В концептуальной модели отражаются требования к системе в виде совокупности нотаций – диаграмм, которые визуализируют основные элементы структуры проектируемой системы. Эффективное представление взаимодействий между разными участниками разработки сопровождается заданием комментариев: традиционного текста или стереотипа.

Стереотип – это средство метаклассификации элемента, который представлен в диаграмме с описанием назначения (например, <<треугольник>>,<<точка>>). Он может расширяться и адаптироваться к конкретным областям применения.

ПС описывается с помощью рассматриваемых ниже диаграмм.

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

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

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

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

Под операцией понимается сервис, который экземпляр класса может выполнять, если к нему будет произведен соответствующий вызов.

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, участвующих в связи (0 – ни одного, 1 – один, n – 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++).

Диаграммы моделирования поведения системы. Под *поведением* системы понимается множество объектов, обменивающихся сообщениями с помощью следующих диаграмм:

– последовательность, упорядочивающая взаимодействие объектов при обмене сообщениями;

– сотрудничество, определяющее роль объектов во время их взаимодействия;

– активность, показывающая потоки управления при взаимодействии объектов;

– состояние, указывающее на динамику изменения объектов.

*Диаграмма последовательности* применяется для задания взаимодействия объектов. Любому из объектов сценария ставится в соответствие линия жизни, которая отображает ход событий от его создания до разрушения. Взаимодействие объектов контролируется событиями, которые происходят в сценарии и стимулируют посылки сообщений другому объекту.

*Диаграммы сотрудничества* представляют совокупность объектов, поведение которых имеет значение для достижения целей системы, и взаимоотношения ролей, важных в сотрудничестве. Диаграмма позволяет моделировать статическое взаимодействие объектов без учета фактора времени. При параметризации она представляет абстрактную схему сотрудничества – *паттерн*, в котором определяется множество конкретных схем.

*Диаграмма деятельности* задает поведение системы в виде работ, которые может выполнять система или актер, которые зависят от принятия решений в зависимости от сложившихся условий. Эта диаграмма предусматривает параллельное выполнение нескольких видов деятельностей и их синхронизацию.

*Диаграмма состояний* отражает условия переходов и действия при входе в состояние, его активность при выходе из этого состояния.

*Диаграмма реализации* состоит из диаграммы компонента и диаграммы размещения. Диаграмма компонента отображает структуру системы как композицию компонентов и связей между ними в виде графа, узлами которого являются компоненты, а дуги – отношения зависимости. Диаграмма размещения позволяет задать состав физических ресурсов системы (узлов системы) и отношений между ними.

*Пакет* – это совокупность элементов (объектов, классов, подсистем и т.п.), представленная группой подсистем разного уровня детализации. 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.

**Инструменты.** Rational Rose – средство визуального моделирования ОО ПС. Rose является CASE-средством, графические возможности которого основаны на использовании UML и позволяют проектировать разные виды диаграмм с заданием всех свойств, отношений и взаимодействий. Средства системы обеспечивают проектирование и моделирование общей модели процессов (абстрактной) 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 поставляются Rose DataModeler 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.

## 1.5. Компонентное программирование

**Возможности.** По оценкам экспертов, 75 % работ по программированию в мире дублируются. Поэтому переход к повторному использованию компонентов (ПИК) наиболее производителен. Этот процесс происходил эволюционно: от подпрограмм, модулей и объектов до компонентов путем совершенствования этих элементов и методов, их спецификации и композиции. Компонентное программирование обобщает ООП. Объекты рассматриваются на логическом уровне проектирования программной системы, а компоненты – как непосредственная физическая реализация объектов. Один компонент может быть реализацией нескольких объектов или даже некоторой части объектной системы, полученной на уровне проектирования [7, 8, 45, 46]. Компоненты в отличие от объектов могут изменяться и пополняться новыми функциями и интерфейсами. Они конструируются в виде некоторой программной абстракции, состоящей из трех частей: информационной, внешней и внутренней.

*Информационная часть* содержит описание: назначение, дата изготовления, условие применения (ОС, платформа и т.п.), возможность ПИК, среды окружения и др.

*Внешняя часть* определяет взаимодействие компонента с внешней средой и с платформой, на которой он будет выполняться и включает следующие характеристики:

– интероперабельность – способность взаимодействовать с компонентами других сред;

– переносимость – способность компонента выполняться на разных платформах компьютеров;

– интеграционность – объединение компонентов на основе интерфейсов в более сложные ПС;

– нефункциональность – обеспечение безопасности, надежности работы компонентов и т.п.

*Внутренняя часть* – это программный фрагмент кода, кластер, системная или абстрактная структура, представленные в виде проекта компонента, спецификации, выходного кода и др. Данная часть компонента включает:

– интерфейс (interface),

– реализацию (implementation),

– схемы развертки (deployments).

*Интерфейс* отображает способ обращения к другим компонентам с помощью средств языков IDL или APL. В нем указываются типы данных и операции передачи параметров для взаимодействия компонентов друг с другом. Каждый компонент может реализовывать определенную совокупность интерфейсов.

*Реализация* – это код, который будет выполняться на компьютере с использованием заданных интерфейсов. Компонент может иметь несколько реализаций в зависимости от ОС, модели данных и соответствующей системы управления БД.

*Развертка* – это создание физического файла, готового для выполнения, содержит все необходимые данные для настройки и запуска компонента.

Компонент описывается в ЯП, не зависит от ОС и реальной платформы, где он будет функционировать. Компоненты наследуются в виде классов и используются в модели или композиции, а также в каркасе интегрированной среды. Управление компонентами осуществляется на трех уровнях: архитектурном, компонентном и интерфейсном, между которыми есть взаимная связь.

Компонент может быть представлен разными структурами.

К*онтейнер* – структура, в которой заданы функции, порядок их выполнения, вызываемые события, сервис, интерфейс, необходимый для обращения к провайдеру сервиса.

*Паттерн* – абстракция, которая содержит описание взаимодействий совокупности объектов, ролей участников и их ответственности. Может быть и ПИК.

*Каркас* – представляет собой высокоуровневую абстракцию проекта ПС, в которой отделены функции компонентов от задач управления ними. В бизнес-функциях компонентов каркас задает надежное управление, объединяет множество взаимодействующих между собою объектов в некоторую интегрированную среду для решения заданной конечной цели и может быть “белым или черным ящиком”.

Каркас типа “белый ящик” включает абстрактные классы для представления цели объекта и его интерфейса. При реализации эти классы наследуются в конкретные классы с указанием соответствующих методов реализации, что соответствует ООП. Каркас типа «черный ящик» в видимой части имеет точки входа и выхода.

*Компонентная модель* включает многочисленные 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 следующие типы:

– компонент-компонент обеспечивает непосредственное их взаимодействие через интерфейс на уровне приложения;

– композиция каркас-компонент способствует взаимодействию каркаса с компонентами, управлению ресурсами компонентов и их интерфейсами на системном уровне;

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

– каркас-каркас создает возможность взаимодействовать с каркасом, каждый из которых может разворачиваться в гетерогенной среде и взаимодействовать через интерфейсы на сетевом уровне.

ПИК – это готовые компоненты, которые используются в других ПС [44–46], они упрощают и сокращают сроки разработки новых ПС благодаря:

– отображению в них базовых функций и понятий ПС;

– скрытию представления данных, операций обновления и получения доступа к этии данным;

– обработке возникающих исключительных ситуаций и др.

Как и любые элементы промышленного производства, ПИК должны отвечать определенным требованиям, иметь характерные свойства, структуру, интерфейс и др. Для практического применения ПИК проводится их классификация и каталогизация, а также после выбора настройка на новые условия среды функционирования. Классификация ориентирована на унификацию представления информации о компонентах для поиска и отбора в среде хранения. Каталогизация направлена на физическое размещение ПИК в разных хранилищах (репозитариях, банках компонентов и др.) для извлечения их из них с целью использования.

**Инструменты.** Примером инструментария по разработке ПС из ПИК является *система RSEB* [47]. В ней реализован метод, основанный на нотации UML, объектно-ориентированном методе и использовании ПИК многократного применения. Построение разных ПС проводится с использованием элементов Use Case. *Компонент* в RSEB — это тип, класс или любое другое программное средство (например, Use Case анализа, проектирования или реализации), разработка которого осуществляется с учетом его многократного применения. Компонентная система в RSEB рассматривается как ПС, содержащая ряд ПИК и повторно используемых нефункциональных характеристик. В качестве примера компонентной системы можно привести повторно используемые каркасы графических пользовательских интерфейсов и математические библиотеки [48]. Практически компонентная система позволяет производить другие прикладные системы, если в ней выделены родовые подсистемы с многократно используемыми решениями. Инженерия предметной области в системе RSEB состоит из двух процессов конструирования:

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

2) компонентных систем из семейства. Процесс разработки различных частей ПС направлен на конструирование и реализацию надежных, расширяемых и гибких компонентов повторного использования

*Система OOram* также обеспечивает поддержку инженерии разработки ПС с помощью ПИК на основе следующих процессов [49]: создание модели, например ролевой, ее синтез и разработку спецификации объектов; процесс разработки системы в соответствии с ЖЦ, начиная с формулирования требований и заканчивая введением в действие ПС; процесс производства ПС из набора компонентов многократного применения в соответствии с этапами: анализ рынка, разработка и упаковка продукта, маркетинг ПИК и др.

## 1.6. Аспектно-ориентированное программирование (АОП)

**Возможности.** АОП базируется на методах ООП разбиения задач ПрО на ряд функциональных компонентов, а также на аспектах (синхронизации, взаимодействия, защиты и др.), которые встраиваются в отдельные компоненты как некоторые их реализации и влияют на композицию компонентов [10, 11, 50– 54]. В качестве аспекта может быть некоторая идея, которая интересует нескольких заинтересованных лиц и представляется в виде варианта использования, функции, как обязательный элемент компонента или программы. Некоторые аспекты могут действовать на протяжении ЖЦ процесса разработки, они кодируются, тестируются и упрощают результат разработки. Как правило, создание продукта связано с выполнением разных ролей в процессе разработки, которые могут выполнять агенты с такими функциями: определение архитектуры, управление проектом, повышение качества и продуктивности разработки ПС.

Так как современные языки программирования не позволяют инкапсулировать аспекты в проектные решения системы, то в некоторые инструментальные компонентные системы введен механизм фильтрации входных сообщений, с помощью которых выполняется изменение параметров и имен текстов аспектов в конкретном компоненте. «Нечистый» код компонента (код с пересекаемыми его аспектами) требует разработки новых подходов к композиции компонентов, ориентированных на ПрО и на выполнение ее функций [51, 52].

Общие средства композиции объектов или компонентов (вызов процедур, RPC, RMI, IDL и др.) в АОП являются недостаточными, так как аспекты требуют декларативного сцепления между частичными описани-

ями, а также связывания отдельных обрывков из различных объектов. Одним из механизмов композиции является фильтр композиции, суть которого состоит в обновлении 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 установленные ссылки, проверяют и размещают в буфере сообщения, локализуют ограничения на синхронизацию и готовят компонент для выполнения.

Поскольку в ОО-программах может содержаться много мелких методов, которые самостоятельно не выполняют расчетов и обращаются к другим методам, расположенным в областях внешнего уровня, Деметер сформулировал закон [53, 54], согласно которому не разрешаются длинные последовательности методов, связанные с передачами параметров через внутренние объекты. 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, взаимодействия и др. пересекают ряд многократно используемых ПИК. Разным аспектам проектируемой системы могут отвечать и разные парадигмы программирования: объектно-ориентированные, структурные и др. Они по отношению к 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 и генерируются [51] в динамике выполнения.

Существенной чертой любых аспектов является модель, которая пересекает структуру другой модели, для которой первая 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 образцы 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. Одним из путей оптимальной реализации аспектов является минимизация сцепления между аспектами и компонентами, которая реализуется ссылками на языковые конструкции, варианты использования, сопоставление с образцом. Реализация аспектов в различных блоках кода позволяет устанавливать перекрестные ссылки между ними, тем самым декларируется 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 через косвенный код или таблицы виртуальных таблиц для повторного связывания или статическое («жесткое») связывание в период компиляции.

Для целей АОП хорошо подходит модель модульных расширений, создаваемая в рамках метамодельного программирования. Она предлагает оперативное распространение новых механизмов композиции в отдельные части ПС или их семейств с учетом предметно-ориентированных возможностей языков (например, SQL) и каркасов, которые поддерживают разного рода аспекты [51].

**Инструменты.** Для эффективной реализации аспектов разработана IP-библиотека расширений. В ней размещены некоторые функции компиляторов, методов, средства оптимизации, редактирования, отображения. и др. Например, библиотека матриц, с помощью которой вычисляются выражения с массивами, обеспечивается 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 [48, 52, 53].

## 1.7. Порождающее (генерирующее) программирование

**Возможности.** Порождающее программирование (generate programming) – это парадигма разработки ПС, основанная на моделировании групп или отдельных элементов ПС, таких, что при описании списка конкретных требований до системы из этих элементов, аспектов, элементарных ПИК и каркасов конфигурации автоматически *генерируется* промежуточный или конечный продукт [3].

Данное программирование представляет объединенную целостную и стройную концепцию создания инженерии ПрО путем проектирования семейств ПС на основе объектов, компонентов, аспектов, сервисов, ПИК, систем, характеристик и т.п. По существу это программирование является дальнейшим развитием ООП в направлении использования ПИК, каркасов и разных аспектов в создаваемых ПС. 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 (generative domain model), т.е. модели семейства, включающей средст-

ва определения ее членов, компоненты реализации и ПИК, из которых собирается любой член этого семейства, и базу конфигураций, отображающую спецификации членов семейства.

В созданном программном члене семейства отражается максимум знаний о его производстве, а именно конфигурации, инструментарии измерения и оценки, методах тестирования и планирования, отладки, визуального представления и пр. Эти аспекты отображают специфику ПрО, многократно используемых ПИК, представленных в активных библиотеках [53].

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 разным инструментальным системам (компиляторами, анализаторами кода, отладчиками и т.п.) в виде описания функций. Фактически компоненты активных библиотек выполняют роль интеллектуальных агентов, в процессе взаимодействия которых создаются новые специализированные 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, отладку, визуализацию, взаимодействие и др. Кроме того, есть возможность пополнять ее новыми сгенерированными компонентами в рамках отдельных ПС семейства, которые относятся к компонентам многоразового применения. Вместе с тем ЯП компонентов дополняются новыми аспектами, которые расширяют одновременно и ПрО новыми возможностями.

Целью порождающего программирования является разработка правильных компонентов для целого семейства и после этого автоматически предоставлять их другим семействам. Реализации этой цели соответствует два сформировавшихся направления использования ПИК:

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

2) *инженерия* ПрО – построение семейства ПС путем сбора, классификации и фиксации ПИК, опыта конструирования систем или готовых частей систем для конкретной ПрО. При этом создаются системные инструментальные системы поддержки поиска, адаптации ПИК и внедрения их в новую ПС семейства.

Разбиение инженерии ПрО на конструирование семейства приложений и конструирование компонентных систем обусловлено четким разделением задач по разработке общей архитектуры и многократно используемых решений для отдельных подсистем.

Основными этапами инженерии ПрО являются: анализ ПрО и выявление характеристик; определение области действий ПрО; определение общих и изменяемых характеристик в характеристической модели. Эта модель устанавливает зависимость между различными членами семейства и в пределах самого члена семейства, а также создает базис для производства конкретных программных членов семейства с механизмами изменчивости независимо от средств их реализации. На основе характеристической модели и компонентов реализации генерируется доменная модель для семейства с использованием данной модели, знания о конфигурациях и спецификации высокого уровня компонентов автоматически генерируются в некоторый член семейства. При этом используются такие стандартные системные средства, как JavaBeans с графическим интерфейсом, визуальными компонентами или C++ с компонентами стандартной библиотеки шаблонов.

Функциональные компоненты представляются в виде объектов, процедур, модулей, которым необходимы такие свойства, как безопасность, синхронизацияи др. Эти свойства, как правило, выражаются небольшими фрагментами кода в нескольких функциональных компонентах. Они могут пересекать ряд компонентов и представлять собой отдельные аспекты в терминологии АОП. Смесь компонентов и аспектов образует ряд небольших классов и методов, что в конце концов усложняет создаваемую ПС.

Инженерия ПрО ориентирована на разработку решений, связанных с группами ПС, обеспечивает поддержку мультисистемного проектирования и моделирование изменчивости. Она состоит в следующем: разработка моделей групп систем семейства; моделирование понятий данной ПрО и выявление альтернативных методов реализации этапов; разработка групп систем для последующего их повторного использования; системное моделирование характеристик компонентов в соответствии с характеристической моделью семейства ПрО; реализация процесса сборки каждого члена семейства на основе базы знаний о конфигурации.

Для выполнения инженерии ПрО используются следующие процессы:

*корректировка процессов* для разработки решений на основе ПИК;

*моделирование изменчивости и зависимостей*, которое начинается с разработки словаря описания различных понятий, фиксации их в характеристической модели и в справочной информации сведений об изменчивости моделей (объектных, Use Case, взаимодействия и др.). Фиксация зависимостей между характеристиками 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 новые понятия – пространство задач, а в технологии реализации компонентов на основе каркаса конфигураций – пространство решений.

Пространство задач. Областью разработки является семейство систем, в которых используются компоненты многократного применения. Процесс разработки с повторным использованием организуется таким образом, чтобы в нем применять не только ПИК, но и инструменты, созданные в ходе разработки ПрО. В рамках

инженерии ПрО разрабатывается характеристическая модель, которая обобщает характеристики системы и изменяемые параметры разных частей семейства, а также решения, связанные с группами ПС.

Инженерия ПрО включает разработку моделей групп систем, моделирование понятий ПрО, разработку характеристических моделей и групп систем для последующего их повторного использования.

В данном программировании нашли отражение идеи международного OMG-комитета, касающиеся видов ПрО горизонтального и вертикального типов. К горизонтальным ПрО отнесены общие системные средства: графические пользовательские интерфейсы, СУБД, системные программы, библиотеки расчета матриц, контейнеры, каркасы и т.п., а к вертикальным видам ПрО – прикладные системы (медицинские, биологические, научные и т.д.), методы инженерии ПрО и горизонтальные методы обслуживания архитектуры многократного применения, интерфейсов, библиотек и др.

Пространство решений включает компоненты, каркасы и образцы проектирования. При этом каркас оснащается изменяемыми параметрами модели, что может привести к излишней его фрагментации и появлению «множества мелких методов и классов». Каркас обеспечивает динамическое связывание аспектов и компонентов в процессе реализации изменчивости между разными приложениями. *Образцы* проектирования обеспечивают создание многократно используемых решений в различных ПС. Для задания 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 поддержки инженерии ПрО и реализации горизонтальных методов является система DEMRAL [52, 54], 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 конфигурирования.

Система конструирования RSEB [47] базируется на вертикальных методах, ПИК и ориентирована на использование элементов Use Case 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 или метод взаимодействия компонентов в распределенной открытой среде (например, в CORBA).

## 2. Методы теоретического программирования

К ним отнесем следующие:
– агентный;
– алгебраический, инсорционный;
– экспликативный, композиционный;
– дескрипторный и др.

Остановимся на краткой характеристике этих методов и основных направлений их развития.

### 2.1. Агентное программирование

**Возможности.** Понятие интеллектуального агента появилось более 20 лет назад, их роль в программной инженерии все время возрастает. Так, в [12] сказано, что перспективными ролями агентов будут: разработчик 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 [55–64].

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 решает задачи, достигая заданных целей путем выполнения программы и изменения состояния среды. Он имеет такие свойства:
– автономность – способность действовать без внешнего управляющего воздействия;
– реактивность – возможность реагировать на изменения данных и среды и воспринимать их;
– активность – способность ставить цели и выполнять заданные действия для их достижения;
– социальность – взаимодействие 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 убеждения, намерения, обязательства и т.п. [58, 59, 64]. Эти понятия входят в концептуальную модель и объединяются между собой операционными планами реализации целей каждого агента. Для достижения целей интеллектуальные агенты взаимодействуют друг с другом, устанавливают связь между собой через сообщения или запросы 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 им функции и могут влиять на общее состояние среды. В обоих случаях характер взаимодействия между агентами зависит от таких факторов: совместимость целей, компетентность, 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 посредством координации, коммуникации, кооперации или коалиции.

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

– взаимозависимостью целей других агентов – членов коалиции, а также возможным влиянием 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). Управление агентами (Agent Management) выполняется с помощью таких сервисов: передача сообщений между ними, 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 [55–57].

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

Агенты функционируют в среде и избирают те действия, которые они могут выполнить. Модель этой среды состоит из модели информационных ресурсов, их свойств, правил работы с ними и средств задания сообщений. В результате выполнения функций агенты создают некоторое поведение среды, которое в любой момент времени находится в некотором состоянии, а агент, выполняя заданные действия, изменяет его в целевое состояние и учитывает возможность возникновения нерегулярных состояний (тупиков, отсутствие ресурса и др.).

Среда – это пространство состояний, которое задается графом с вершинами – состояниями и дугами — действиями, обеспечивающими переход из одного состояния в другое.

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, синхронное или асинхронное изменение состояния и т.п.

**Инструменты.** Одной из систем построения агентов, основанной на обмене сообщениями в ACL, является JATLite. Она включает 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 функции менеджера проекта и онтологий, визуализации, отладки и др. На реализацию механизмов взаимодействий агентов ориентирована и система JAFMAS. Ряд мультиагентных систем описано в [54].

## 2.2. Алгебраическое программирование (АП)

Данное программирование предназначено для определения интеллектуальных агентов и сред их функционирования с применением математического аппарата, в качестве которого используется понятие транзитивной системы [13, 14, 17-19]. Данный аппарат позволяет определить поведение систем и их эквивалентность. Примерами транзитивных систем могут быть компоненты, программы и их спецификации, объекты, взаимодействующие друг с другом и средой их существования. Эволюция такой системы описывается с помощью истории функционирования систем, которая может быть конечной или бесконечной, и включает обзорную часть в виде последовательности действий и скрытую часть в виде последовательности состояний. История функ-

ционирования включает 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.

Главным инвариантом состояния транзитивной системы является 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 $u+v$ одной из двух поведений $u$ и $v$, который является ассоциативным и коммутативным. Конечное поведение задается константами $\Delta$, $\perp$, 0, обозначающими соответственно состояние успешного завершения, неопределенного и тупикового состояния. Алгебра поведения частично задается отношением $\leq$, в котором элемент $\perp$ является наименьшим, а операции алгебры поведения – монотонными. Теоретический результат алгебры *F(A)* – доказательство теоремы про наименьшую неподвижную точку.

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

*Среда E* определяется как агент над алгеброй действий *A* и функцией погружения от двух аргументов *Ins(e, u) = e[u]*. Первый аргумент – поведение среды, второй – поведение агента, который погружается в эту среду в заданном состоянии. Алгебра агента – параметр среды. Значения 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", которое трансформирует 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 выражается через параллельную композицию агентов над той же самой областью действий и соответствующей комбинацией действий. Другой – через функцию погружения агента в некоторую среду, результатом трансформации которой является новая среда.

Язык действий *A* 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 есть 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 – 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: недетерминированным выбором и префиксингом. Среда вводится как агент, в нее погружается функция, имеющая поведение типа агента или среды. Произвольные непрерывные функции могут быть использованы как функции погружения, и эти функции определены значениями логики переписывания. Трансформации поведения среды, которые определяются функциями погружения, составляют новый тип – 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 использовать 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 [14].

**Инструменты.** Система алгебраического программирования (АПС) основывается на типах системных объектов, базовых вычислительных механизмах и языковых конструкциях *AL* [17, 18]. 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 (*A*-модуль); интерпретаторами алгебраических модулей и т.п. *A*-модуль – это представление структуры данных, которые определяются в АП-модулях. Он наследует тип, начальное именование и имеет динами-

ческий 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.

## 2.3. Экспликативное программирование (ЭП)

**Возможности.** Данное программирование [15, 16, 20-22] 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 составляют логика, конструктивная математика, информатика, композиционное программирование и классическая теория алгоритмов. Для изображения алгоритмов программ используются разные языки и методы программирования: функциональное, логическое, структурное, денотационное и др.

Принципами ЭП последовательных и многоуровневых программ и их уточнений являются следующие [15].

*Принцип развития* понятия программы состоит 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 *аппликативности*, т.е. рассмотрение программы как функции, вырабатывающей выходные данные на основе входных. Функция является объектом, которому сопоставляется *денотат* имени функции с помощью отношения *именования* (*номинации).*

Развитие понятия функции осуществляется с помощью п*ринципа 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 включающие 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 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*. Главными в ЭП являются семантический аспект, система композиций и номинативности (КНС), ориентированные на систематическое изучение номинативных отношений при построении данных, функций, композиций 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, 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. Используются для построения математически простых и адекватных моделей программ параметрического типа с использованием методов универсальной алгебры, математической логики и теории алгоритмов. Данные в КНС рассматриваются на трех уровнях: абстрактном, булевском и номинативном. Класс номинативных данных обеспечивает построение именных данных, многозначных номинативных данных или мультиименных данных, задаваемых рекурсивно.

Имеются средства 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 аппликативности, 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 функций, 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 на 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 запро-

сов, заданных в SQL-подобных языках запросов [20-22], что подтверждает практическую сторону аппарата формализации дедуктивных и ОО БД.

## Заключение

Анализ рассмотренных методов систематического и теоретического программирования показывает постоянное их развитие, совершенствование, пополнение новыми возможностями и возможностью объединения. Классическим примером объединения является язык UML, в результате ООП обогатился новыми возможностями, которые удовлетворили многих пользователей визуальным и наглядным моделированием ПС на основе разнообразных диаграмм. Алгебраическое и экспликативное программирование впитало в себя не только возможности логического, функционального и процедурного программирования, но и специфические особенности абстрактного представления функций, программ, данных и агентов.

Большой интерес представляет объединенная концепция порождающего программирования, позволяющая объединять разного рода объекты, компоненты и методы их производства. Каждый вид программирования, используемый в этом программировании, может развиваться как самостоятельно, так и в сообществе с другими. Это программирование позволяет создавать семейства ПС, изменять их, итерационно развивать, а также переориентировать на применение более эффективных методов. Выгода от создания ПС в среде этого программирования зависит не только от входящих методов программирования и инвестиций в ПИК, но и от многократного применения изготовленных из них членов семейства. Значительный выигрыш дает построение порождающей модели ПрО для семейства, когда все члены семейства не являются одиночными программами, а удовлетворяют принципам моделирования характеристик, оптимизации и генерации как на уровне процедур и классов, так и на уровне компонентов, подсистем и элементов разных библиотек с помощью автоматизированного генератора.

Как считают многие специалисты в области информатики, перспективными среди рассмотренных методов программирования являются методы порождающего и агентно–ориентированного программирования.


1. *Буч Г.* Объектно-ориентированный анализ.– М.:"Бином", 1998. – 560 с.
2. *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.
3. *Чернецки К., Айзенекер У.* Порождающее программирование. Методы, инструменты, применение. – Изд. дом «Питер».– М., С-Пет.– 2005. – 730 с.
4. *Римский Г.В.* Структура и функционирование системы автоматизации модульного программирования // Программирование. – 1980. – №2. – С.31–38.
5. *Лаврищева Е.М., Грищенко В.Н.* Сборочное программирование. – Киев. – Наукова Думка, 1991. – 213с.
6. *Yourdan E.* Modern Srtucrured Analysis. – New York: Prentice Hall, 1988. – 297 p.
7. *Грищенко В.Н., Лаврищева Е.М.* Методы и средства компонентного программирования // Кибернетика и системный анализ. – 2003. – №1. – С.39-55.
8. *Грищенко В.Н., Лаврищева Е.М.* Компонентно-ориентированное программирование. Состояние, направления и перспективы развития // Проб. программирования. – 2002. – № 1–2. – С. 80–90.
9. *WCOP'99* – Proc. of the Fourth International Workshop on Component-Oriented Programming // Ed. Boush J., Szyperski C. , Weck W. –
10. ISSN 1581. – C.113.
11. *Lopes C., Kiczales G., Murphy G., Lee A.* (organizers). Proc. of the ECOOP on Aspect-Oriented Programming, Kyoto, Japan, April 20, 1998.
12. *Elrad T., Filman R.E.* Aspect-oriented programming // Com. of the ACM. – 2001. – Vol.44, № 10. – P.33–38.
13. *Яковсон Айвар.* Мечты о будующем программирования. Открытые системы. – М.: 2005. – №12. – С.59–63.
14. *Летичевский А.А, Маринченко В.Г.* Объекты в системе алгебраического программирования // Кибернетика и системный анализ. – 1997. – №2. – С.160–180.
15. *Летичевский А.А., Капитонова Ю.В., Волков В.А., Вышемирский В.В., Летичевский А.А.* (мол.). Инсерционное программирование //
16. Там же.– 2003. – №1. – С.12-32.
17. *Редько В.Н.* Экспликативное программирование: ретроспективы и перспективы // Проб. программирования. – 1998. – №2. – С. 22– 41.
18. *Никитченко Н.С.* Композиционно-номинативный подход к уточнению понятия программы // Там же. – 1999. – №1. – С.16–31.
19. *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.
20. *Letichevsky A.A., Gilbert D.R.* A General Theory of Action Language // Кибернетика и системный анализ. – 1998. – № 1. – С.16–36.
21. *Летичевский А.А., Капитонова Ю.В.* Доказательство теорем в математической информационной среде // Там же. – 1998. – №.4. – С. 3 – 12.
22. *Редько В.Н.* Композиционная структура программологии // Там же. – С. 47-66.
23. *Редько В.Н.* Основания программологии // Там же.– 2000. – № 1.– С. 35-57.
24. *Редько В.Н., Брона Ю.Й., Буй Д.Б.* Реляційні бази даних: табличні алгебри та SQL–подібні мови //Видав. дім "Академперіодика",
25. Київ, 2001. – 195 с.
26. *Лаврищева Е.М., Грищенко В.Н.* Связь разноязыковых модулей в ОС ЕС. – М: Финансы и статистика. – 1982. – 127 с.
27. *Лаврищева Е.М.* Сборочное программирование. Некоторые итоги и перспективы // Проб. программирования. – 1998. – №1–2. – С.20 - 31.
28. *Лаврищева Е.М.* Парадигма интеграции в программной инженерии // Там же. – 2000. – №1–2. – С.351-360.
29. *ДСТУ* 3901-99 (ISO/IEC 11404:1996, ГОСТ 30664-99) Інформаційні технології. Мови програмування, їхні середовище і системний інтерфейс. Незалежні від мов типи даних / О.Л.Перевозчикова, О.І.Ткаченко, О.Т.Ткаченко та інші. – Київ: Держстандарт України, 2000. – 112 с.
30. *Марка Д.А., Мак Груэн К.* Методология структурного анализа и проектирования. – М.: МетаТехнология, 1997. – 346 с.
31. *Skidmore.S, Mills.G, Farmer R.* SSADM: Models and Methods. – Prentice-Hall, Englewood Cliffs, 1994. – 693 p.
32. *DeMarko D.A., McGovan R.L.* SADT: Structured Analysis and Design Technique. – New York: Mcgray Hill, 1988. – 378 p.
33. *Barker R.* CASE-method. Entity Relatiionship Modeling.Copyrigth Oracle Corporation UK Limited New York: – 1990. – 312 p.
34. *Шлеер С., Меллор С.* Объектно-ориентированный анализ: моделирование мира в состояниях. – Киев: Диалектика, 1993. – 238с.
35. *Martin J., Odell J.J.* Object-oriented analysis and design. – Prentice Hall. – 1992. – 367 p.

36. *Rumbaugh Jet. al*. Object Oriented Modeling and Design.– Englewood Cliffs: Prentice Hall, 1992.– NJ. – 271 p.
37. *Firesmith D.* Object-oriented Methods, Standarts and Procedures. – Englewood Cliffs: Printice Hall, 1994. – 367 p.
38. *Jacobson I.* Object-Oriented Software Engineering. A use Case Driven Approach, Revised Printing. – New York: Addison-Wesley Publ.Co. – 1994. – 529 p.
39. *Coad P.,Yourden E.* Object-oriented analysys. – Second Ed. – New York: Prentice Hall. – 1991.– 296 p.
40. *Кендалл Скотт*. Унифицированный процесс. Основные концепции. – Москва–С–Петербург–Киев. – 2002. – 157 с.
41. *Боггс У., Боггс М.* UML и Rational Rose. – С-Перербург. – Издат. «Лори».– 1999.– 580 с.
42. *Буч.Г., Рамбо Дж., Джекобсон А*. Язык UML. Руководство пользователя. – М.: ДМК, 2000. – 430 с.
43. *Рамбо Дж., Джекобсон А , Буч Г*. UML: специальный справочник. – СПб.: Питер.– 2002.– 656 с.
44. *Gamma E., Helm R., Johnson R., Vlissides J*. Design Patterns, Elements of Reusable Object-oriented Software, – N.– Y.: Addison-Wesley, 1995. – 345 p.
45. *Component* Object Model. – www.microsoft.com/tech/com.asp
46. *Meyer B.* On to Components // Computer. – 1999.– Vol. 32, № 1, Jan.1999. – P.139–140.
47. *Lowy J.* COM and .NET Component Services. – O'Reilly, 2001. – 384 p.
48. *Batory D., O'Malley S.* The Design and Implementation of Hierarchical Software Systems with Reusable Componets // ACM Trans. on Software Eng. and Methodology. – 1992.– Vol. 1, № 4, Oct. 1992. – P.355–398.
49. *Weide B., Ogden W., Zweben S.* Reusable Software Components/ Advances in Computers. – 1991. – Vol. 33. – P.1-65.
50. *Lacobson I., Griss M., Jonsson P.* Software Reuse: Architecture, Process and organization for Business Success – Addison Wesley, May 1997. – 501 p.
51. *Golub G., C.van Loan* . Matrix Computation. 3 ed. – Baltimore and London. – The John Hopkins University Press, 1996. – 315 p.
52. *Reenskaug with Wold P. and Lehhe O.A.* Working with Objects: The Ooram Software Engineering Method. Manning Publications Co., Greenwich , CT, 1996. – 311 p.
53. *Homepage* of the Aspect-Oriented Programming, Xerox Palo Alto Research Center (Xerox Parc) Palo Alto, CA. – www.parc.xerox.com/aop
54. *The Aspect-Oriented* Programming. Proc. ECOOP'97 – Object–oriented Programming, 11th Europe Conf., Finland, June 1997. – Springer Verlag, Berlin.
55. *Lieberherr K.* Demeter and Aspect-Oriented Programming // Proc. of the STJA'97 Conference, Erfurt, Germany, Sept.10–11, 1997. – P.40–43.
56. www.prakinf.tu-ilmenau.de
57. *Berger L., Dery F., Fornarino V.* Interaction between object:and aspect of object-oriented languages // Conf. of AOP. – 1998. – P.13–18.
58. *Ndumu D.T., Nwana H.S.* Research and Development challenges for Agent–based system // IEE Proc. Software Eng. – 1997. – Vol. 144, № 01. – P.26–37.
59. *Wooldridge M.* Agent–based software engineering. – Там же. – P.2–10.
60. Shoham. Y. Agent-oriented programming.–Artif.Intell. – 1993. – Vol 60, №1. – P.51–92.
61. *Плескач В.Л., Рогушина Ю.В.* Агентні технології, Київ: КНТЕУ, 2005. – 337 с.
62. *Трахтенгерц Э.А*. Взаимодействие агентов в многоагентных средах //Автоматика и телемеханика. – М.: Наука. – 1998. – №8. – с.3–52.
63. *Agent software*. – http://www.agentlink.org
64. Agent software examples. – http://www.agents/tools/index.html
65. *Nwana H.S., Lee J., Jennings N.R.* Coordination Software agent systems // British Telecommunication Technology Jo., 1996. – 14(4). P.27– 39
66. *Riechem D.* Intelligent agents // CACM, 1994. – Vol 37, №7. – P.20–31.
67. *Дрейган Р.* Будущее программных агентов // PC Magazine. - March 25. – 1997. – 190 с.