Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Екатерина Лаврищева, Владимир Петрухин
Лекция 6. Прикладные и теоретические методы программирования

В последние годы наибольшее развитие и использование получили прикладные методы: объектно-ориентированный, компонентный, сервисно-ориентированный и др. Они составляют основу методологии разработки ПС и практически применяются при создании разных видов программ.

Находят применение формальные и *теоретические методы* программирования (алгебраический, алгебро-алгоритмический, композиционный и др.), которые основываются на логико-алгоритмических и математических подходах.

В данной лекции представлено описание базовых понятий и особенностей методов систематического (прикладного), отдельных методов *теоретического программирования* с целью ознакомления студентов с современной теорией и практикой разработки программ.

## 5.1. Методы систематического программирования

К методам систематического программирования отнесены следующие методы:

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

Каждый метод имеет свое множество понятий и операций для проведения процесса разработки компонентов или ПС. Метод генерирующего программирования использует возможности объектно-ориентированного, компонентного, аспектно-ориентированного методов и др.

### 5.1.1. Структурный метод

Сущность структурного подхода к разработке ПС заключается в декомпозиции (разбиении) системы на автоматизируемые функции, которые в свою очередь делятся на подфункции, на задачи и так далее. Процесс декомпозиции продолжается вплоть до определения конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимосвязаны.

В основе структурного метода лежат такие общие принципы:

- разбивка системы на множество независимых задач, доступных для понимания и решения;
- иерархическое упорядочивание, т.е. организация составных частей проблемы в древовидные структуры с добавлением новых деталей на каждом уровне.

К основным принципам относятся:

- абстрагирование, т.е. выделение существенных аспектов системы и отвлечение от несущественных;
- формализация, т.е. общее методологическое решение проблемы;
- непротиворечивость, состоящая в обосновании и согласовании элементов системы;
- иерархическая структуризация данных.

При структурном анализе применяются три наиболее распространенные *модели проектирования* ПС:

- *SADT* (*Structured Analysis* and *Design Technique*) – модель и соответствующие функциональные диаграммы [5.1];
- *SSADM* (Structured *Systems Analysis* and *Design Method*) – метод структурного анализа и проектирования [5.2];
- *IDEF0* (Integrated Definition Functions) – метод создания функциональной модели, IDEF1 – информационной модели, *IDEF2* – динамической модели и др. [5.3].

На стадии проектирования эти модели расширяются, уточняются и дополняются диаграммами, отражающими структуру или архитектуру системы, структурные схемы программ и диаграммы экранных форм.

**Метод функционального моделирования SADT.** На основе метода *SADT*, предложенного Д.Россом, разработана методология *IDEF0* (*Icam* DEFinition), которая является основной частью программы *ICAM* (Интеграция компьютерных и промышленных технологий), проводимой по инициативе ВВС США.

Методология *SADT* представляет собой совокупность методов, правил и процедур, предназначенных для построения *функциональной модели предметной области*, которая отображает функциональную структуру, производимые функции и действия, а также связи между ними.

Основные элементы этого метода базируются на следующих концепциях:

- графическое представление структуры системы диаграммами, *отображающими функции* в виде блоков, а интерфейсы (вход/выход) – дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описывается в виде ограничений, которые определяют условия управления и выполнения функции;
- наличие ограничений на количество блоков (от 3 до 6) на каждом уровне декомпозиции и связей диаграмм через номера этих блоков;
- уникальность меток и наименований;
- разделение входов и операций управления;
- отделение управления от функций, т.е. исключение влияния организационной структуры на функциональную модель.

Метод *SADT* применяется при моделировании широкого круга систем, для которых определяются требования и функции, а затем проводится их реализация. Средства *SADT* могут применяться при анализе функций в действующей ПС, а также при определении способов их реализации.

Результат применения метода *SADT* – модель, которая состоит из диаграмм, фрагментов текстов и глоссария со ссылками друг на друга. Все функции и интерфейсы представляются диаграммами в виде блоков и дуг. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, указывается с левой стороны блока, а результаты выхода – с правой стороны. Механизм, осуществляющий операцию (человек или автоматизированная система), задается дугой, входящей в блок снизу (рис. 5.1).

Одна из наиболее важных особенностей метода *SADT* – постепенная детализация модели системы по мере добавления диаграмм, уточняющих эту модель.

**Метод SSADM** базируется на таких структурных диаграммах, как последовательность, выбор и итерация. Моделируемый объект задается последовательностью групп, операторами выбора из группы и циклическим выполнением отдельных элементов.


**Рис. 5.1.** Структура модели

Базовая диаграмма – иерархическая и включает в себя: список компонентов описываемого объекта; идентифицированные группы выбранных и повторяемых компонентов, а также последовательно используемых компонентов.

Данный метод представлен моделью ЖЦ со следующими этапами разработки программного проекта (рис. 5.2):

- стратегическое проектирование и изучение возможности выполнения проекта;
- детальное обследование предметной области, включающее в себя анализ и спецификацию требований;
- *логическое проектирование* и спецификация системы;
- физическое проектирование структур данных в соответствии с выбранной структурой БД (иерархической, сетевой и др.);
- конструирование и тестирование системы.

Детальное обследование предметной области проводится для того, чтобы изучить ее особенности, рассмотреть потребности и предложения заказчика, провести анализ требований из разных документов, специфицировать их и согласовать с заказчиком.

Цель стратегического проектирования – определение области действия проекта, анализ информационных потоков, формирование общего представления об архитектуре системы, затратах на разработку и подтверждение возможности дальнейшей реализации проекта. Результат – спецификация требований, которая применяется при разработке логической структуры системы.

*Логическое проектирование* – это определение функций, диалога, метода построения и обновления БД. В логической модели отображаются входные и выходные данные, прохождение запросов и установка связей между сущностями и событиями.


увеличить изображение

**Рис. 5.2.** Жизненный цикл SSADM

Физическое проектирование – это определение типа СУБД и представления данных в ней с учетом спецификации *логической модели данных*, ограничений на память и времени обработки, а также определение механизмов доступа, размера логической БД, связей между элементами системы. Результат – создание документа, включающего в себя:

- спецификацию функций и способов их реализации, описание процедурных, непроцедурных компонентов и интерфейсов системы;
- определение логических и физических групп данных с учетом структуры БД, ограничений на оборудование и положений стандартов на разработку;
- определение событий, которые обрабатываются как единое целое и выдача сообщений о завершении обработки и др.

Конструирование – это программирование элементов системы и их тестирование на наборах данных, которые подбираются на ранних этапах ЖЦ разработки системы.

Проектирование системы является управляемым и контролируемым. Создается сетевой график, учитывающий работы по разработке системы, затраты и сроки. Слежение и контроль выполнения плана проводит организационный отдел. Проект системы задается структурной моделью, в которой содержатся работы и взаимосвязи между ними и их исполнителями, а потоки проектных документов между этапами отображаются в

сетевом графике. Результаты каждого из этапов ЖЦ контролируются и передаются на следующий этап в виде, удобном для дальнейшей реализации другими исполнителями.

### 5.1.2. Объектно-ориентированный метод

*Объектно-ориентированный подход* (ООП) – стратегия разработки, в рамках которой разработчики системы вместо операций и функций мыслят *объектами* [5.4, 5.5] . Объект – это предмет внешнего мира, некоторая сущность, пребывающая в различных состояниях и имеющая множество операций. Операции задают сервисы, предоставляемые объектам для выполнения определенных вычислений, а состояние – это набор атрибутов объекта, поведение которых изменяет это состояние.

Объекты группируются в класс, который служит шаблоном для включения описания всех атрибутов и операций, связанных с объектами данного класса.

Программная система содержит взаимодействующие объекты, имеющие собственное локальное состояние и набор операций для определения состояний других объектов. Объекты скрывают информацию о представлении состояний и ограничивают к ним доступ.

Под *процессом* в ООП понимается проектирование классов объектов и взаимоотношений между ними (рис. 5.3).

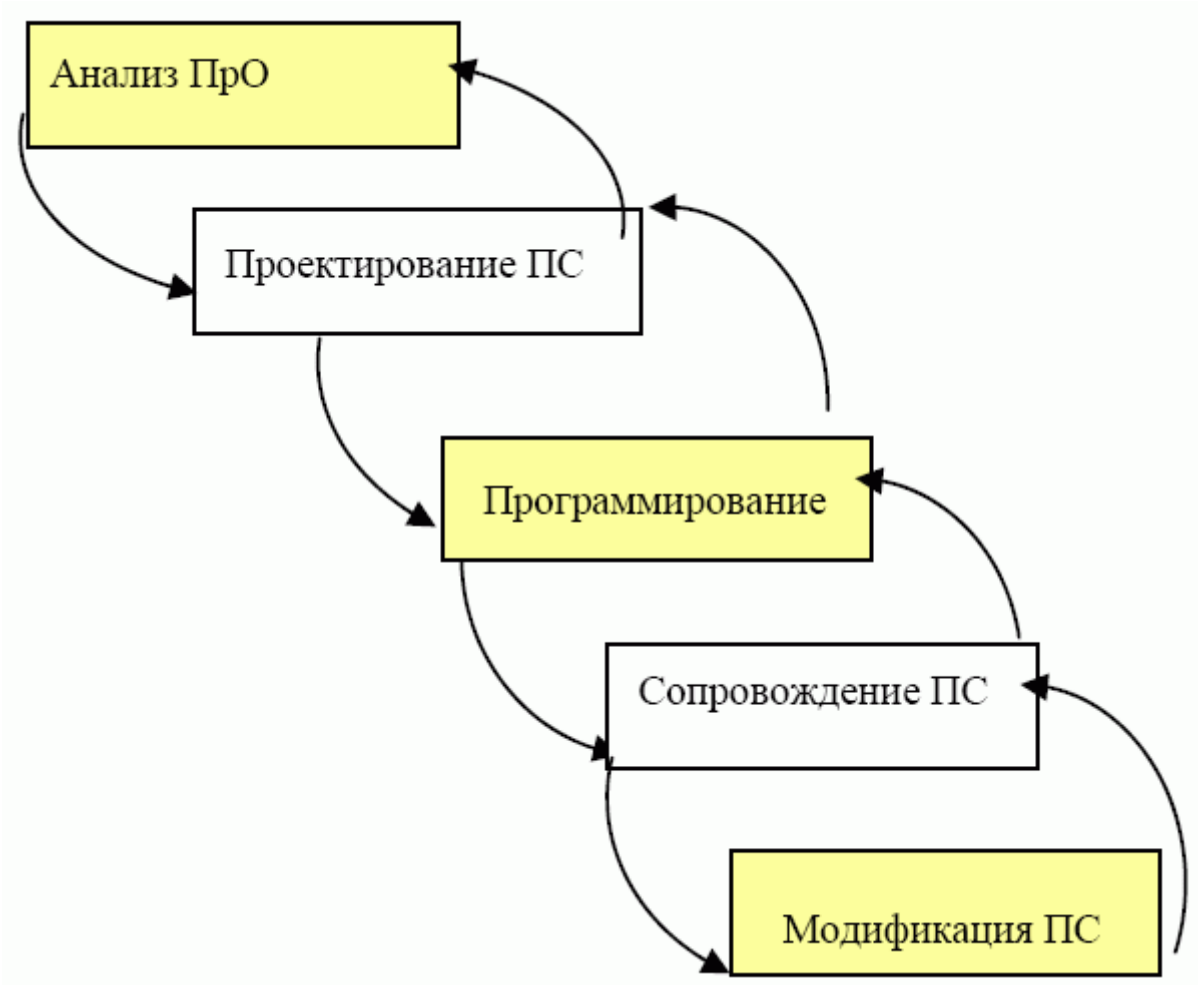



**Рис. 5.3.** ЖЦ разработки ПС в среде ООП

Процесс разработки включает в себя следующие этапы:

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

*проектирование* – уточнение ОМ с учетом описания требований для реализации – реализация ОМ средствами языков программирования C++, Java и др.

*сопровождение* – конкретных задач системы;

*программирование* использование и развитие системы, внесение изменений, как в состав объектов, так и в методы их реализации;

*модификация ПС* – изменение системы в процессе ее сопро-вождения путем добавления новых функциональных возможностей, интерфейсов и операций.

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

Переход к следующему этапу приводит к усовершенствованию результатов предыдущего этапа путем более детальной реализации ранее определенных классов объектов и добавления новых классов.

Результат процесса анализа ЖЦ – модель ПрО и набор других моделей (модель архитектуры, модель окружения и использования), полученных на этапах процесса ЖЦ. Модели отображают связи между объектами, их состояния и набор операций для динамического изменения состояния других объектов, а также взаимоотношения со средой. Объекты инкапсулируют информацию об их состоянии и ограничивают к ним доступ. Модели окружения и использования системы – это две взаимно дополняющие друг друга модели связи системы со средой.

Модель окружения системы – *статическая модель*, которая описывает другие подсистемы из пространства разрабатываемой ПС, а модель использования системы – динамическая модель, которая определяет взаимодействие системы со своей средой. Это взаимодействие определяется последовательностью запросов к сервисам объектов и ответных реакцией системы после выполнения запроса. После определения взаимодействий между объектами проектируемой системы и ее окружением, полученные данные используются для разработки архитектуры системы из объектов, созданных в предыдущих подсистемах и проектах.

Существует два типа моделей системной архитектуры:

*статическая модель* описывает статическую структуру системы в терминах классов объектов и взаимоотношений между ними (обобщение, расширение, использование, структурные отношения);

*динамическая модель* описывает динамическую структуру системы и взаимодействие между объектами во время выполнения системы.

Результат проектирования – это ПС, в которой определены все необходимые объекты статически или динамически с помощью классов и соответствующих методов реализации объектов. Полученная объектно-ориентированная система проверяется на показатели качества на основе результатов тестирования и сбора данных об ошибках и отказах системы. Такую систему можно рассматривать как совокупность автономных и независимых объектов. Изменение метода реализации объекта или добавление новых функций не влияет на другие объекты системы. Объекты могут быть повторно используемыми.

### 5.1.3. UML-метод моделирования

UML (United Modeling Language) – унифицированный язык моделирования является результатом совместной разработки специалистов программной инженерии и инженерии требований [5.6, 5.7]. Он широко используется ведущими разработчиками ПО как метод моделирования на этапах ЖЦ разработки ПС.

В основу метода положена парадигма объектного подхода, при котором концептуальное моделирование проблемы происходит в терминах взаимодействия объектов и включает:

онтологию домена, которая определяет состав классов объектов домена, их атрибутов и взаимоотношений, а также услуг (операций), которые могут выполнять объекты классов;

модель поведения задает возможные состояния объектов, инцидентов, инициирующих переходы с одного состояния к другому, а также сообщения, которыми обмениваются объекты;

модель процессов определяет действия, которые выполняются при проектировании объектов, как компонентов.

Модель требований в UML – это совокупность диаграмм, которые визуализируют основные элементы структуры системы.

Язык моделирования UML поддерживает статические и динамические модели, в том числе модель последовательностей – одну из наиболее полезных и наглядных моделей, в каждом узле которой – взаимодействующие объекты. Все модели представляются диаграммами, краткая характеристика которых дается ниже.

**Диаграмма классов** (*Class diagram*) отображает онтологию домена, эквивалентна структуре информационной модели метода С.Шлеера и С.Меллора, определяет состав классов объектов и их взаимоотношений. Диаграмма задается иконами, как визуальное изображение понятий, и связей между ними. Верхняя часть иконы – обязательная, она определяет имя класса. Вторая и третья части иконы определяют соответственно список атрибутов класса и *список операций* класса.

Атрибутами могут быть типы значений в UML:

`public` (общий) обозначает операцию класса, вызываемую из любой части программы любым объектом системы;

`protected` (защищенный) обозначает операцию, вызванную объектом того класса, в котором она определена или наследована,

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

Пользователь может определять специфические для него атрибуты. Под операцией понимается сервис, который экземпляр класса может выполнять, если к нему будет произведен соответствующий вызов. Операция имеет название и список аргументов.

Классы могут находиться в следующих отношениях или связях.

*Ассоциация* – взаимная зависимость между объектами разных классов, каждый из которых является равноправным ее членом. Она может обозначать 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-клиент может использовать определенную операцию другого класса; классы могут быть связаны отношением трассирования, если один класс трансформируется в другой в результате выполнения определенного процесса ЖЦ.

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

**Моделирование поведения системы.** *Поведение* системы определяется множеством обменивающихся сообщениями объектов и задается *диаграммами: последовательность*, сотрудничество, *деятельности и состояния*.

*Диаграмма последовательности* применяется для задания взаимодействия объектов, с помощью сценариев, отображающих события, связанные с их созданием и уничтожением. Взаимодействие объектов контролируется событиями, которые происходят в сценарии и поддерживаются сообщениями к другим объектам.

*Диаграммы сотрудничества* задает поведение совокупности объектов, функции которых ориентированы на достижение целей системы, а также взаимосвязи тех ролей, которые обеспечивают сотрудничество.

*Диаграмма деятельности* задает поведение системы в виде определенных работ, которые может выполнять система или актер, виды работ могут зависеть от принятия решений в зависимости от заданных условий или ограничений. В качестве примера использования диаграммы деятельности UML приведена структура программы "Оплатить услуги" (рис. 5.4). Данная диаграмма демонстрирует программу расчета и оплаты услуг. В ней выполняется ряд последовательных действий по расчету стоимости за услуги.

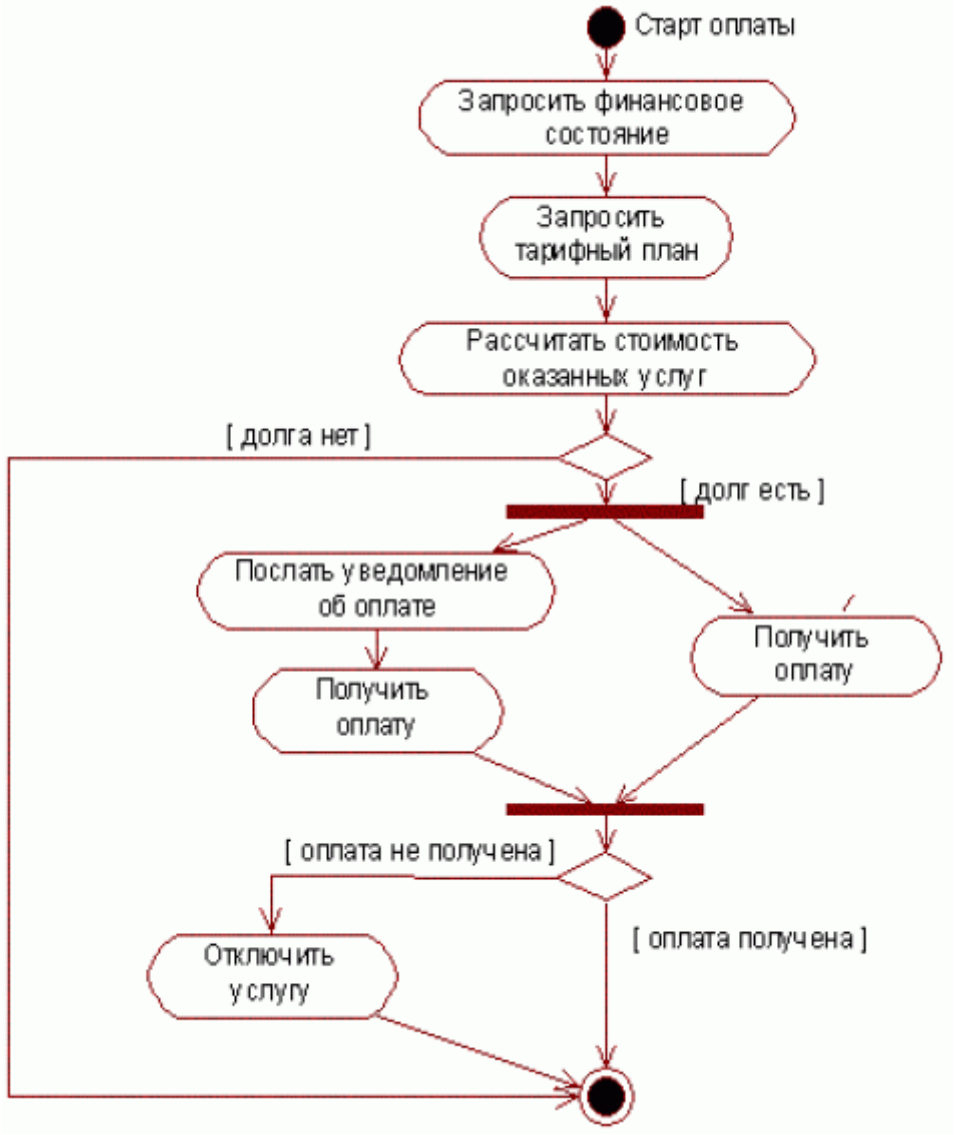


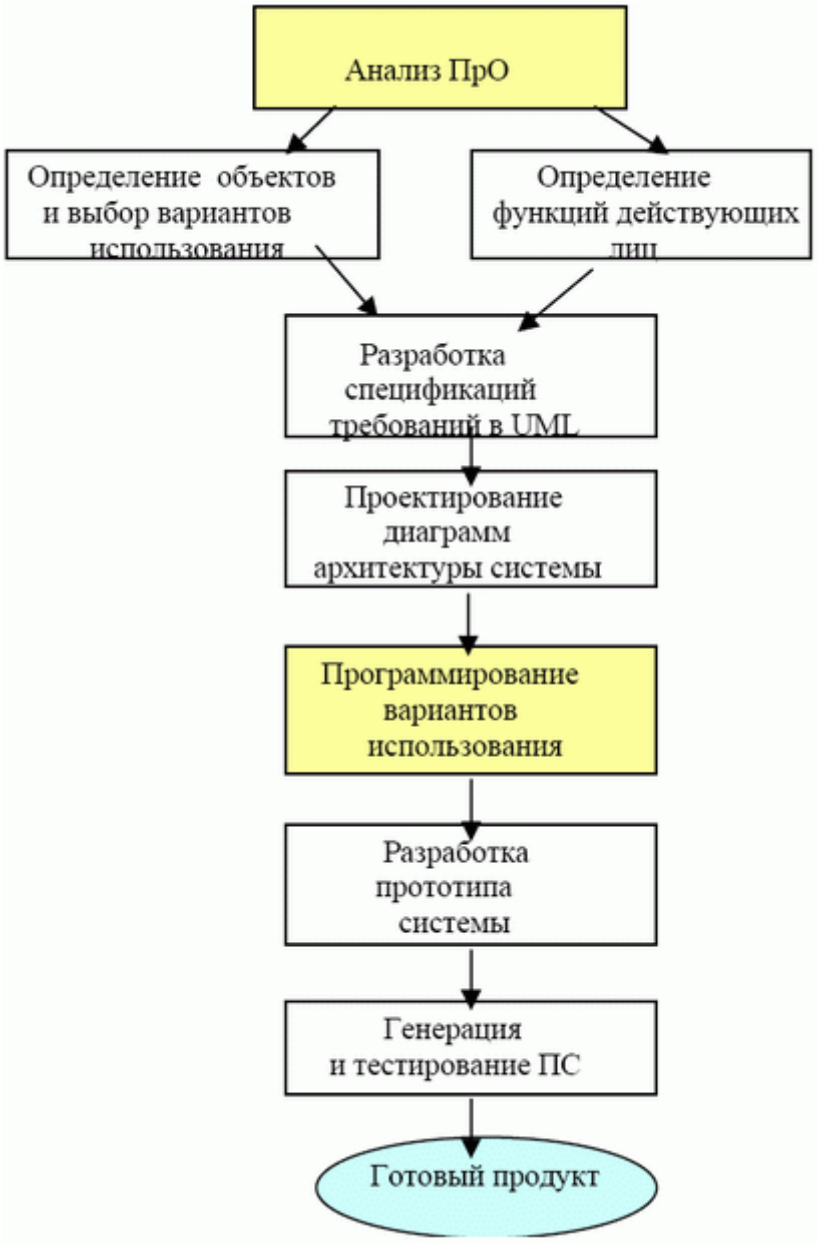
**Рис. 5.4.** Диаграмма программы расчета и оплаты услуг

В зависимости от выполнения условия "долга нет" происходит переход в конечное состояние или на разделение потоков на два параллельных. В левой ветви выполняется действие "послать уведомление об оплате" и "получить оплату", а в правой – "получить оплату". Распараллеливание означает, что пользователь может оплатить услуги, не дожидаясь уведомления. Параллельные потоки сливаются в один, затем снова ветвление алгоритма – условие "оплата не получена", "отключить услугу" и переход в конечное состояние.

*Диаграмма состояний* использует *расширенную модель* конечного автомата и определяет условия переходов, действия при входе и выходе из состояния, а также параллельно действующие состояния. Переход по списку данных инициирует некоторое событие. Состояние зависит от условий перехода, подобно тому, как взаимодействуют две параллельно работающие машины.

**Диаграмма реализации** состоит из диаграммы компонента и размещения.

Построение ПС методом UML состоит в выполнении этапов ЖЦ, приведенных на общей схема реализации ПрО (рис. 5.5).




увеличить изображение

**Рис. 5.5.** Схема моделирования и проектирования ПС в UML

*Диаграмма компонента* отображает структуру системы как композицию компонентов и связей между ними. Диаграмма размещения задает состав *физических ресурсов* системы (узлов системы) и отношений между ними, к которым относятся необходимые аппаратные устройства, на которых располагаются компоненты, взаимодействующие между собой.

Пакет может быть элементом конфигурации построенной системы, на которую можно ссылаться в разных диаграммах.

### 5.1.4. Компонентный подход

По оценкам экспертов, 75 % работ по программированию в информационном мире дублируются (например, программы складского учета, начисления зарплаты, расчета затрат на производство продукции и т.п.). Большинство из этих программ типовые, но каждый раз находятся особенности, которые влияют на их повторную разработку.

Компонентное проектирование сложных программ из готовых компонентов является наиболее производительным [5.8–5.13].

Переход к компонентам происходил эволюционно: от подпрограмм, модулей, функций. При этом усовершенствовались элементы, методы их композиции и накопления для дальнейшего использования (табл. 5.1).

Компонентный подход дополняет и расширяет существующие подходы в программировании, особенно ООП. Объекты рассматриваются на логическом уровне проектирования ПС, а компоненты – это физическая реализация объектов.

Компоненты конструируются как некоторая абстракция, включающая в себя информационный раздел и артефакты (спецификация, код, контейнер и др.). В этом разделе содержатся сведения: назначение, дата изготовления, условия применения (ОС, среда, платформа и т.п.). Артефакт – это реализация (implementation), интерфейс (interface) и схема развертывания (deployment) компонента.

*Реализация* – это код, который будет выполняться при обращении к операциям, определенным в интерфейсах компонента. Компонент может иметь несколько реализаций в зависимости от операционной среды, *модели данных*, *СУБД* и др. Для описания компонентов, как правило, применяются языки объектно-ориентированной ориентации, а также язык JAVA, в котором понятие интерфейса и класса – базовые, используются в инструментах *Javabeans* и Enterprise *Javabeans* и в объектной модели CORBA [5.14].

Таблица 5.1. Схема эволюции элементов компонентов

| Элемент композиции | Описание элемента | Схема взаимодействия | Представление, хранение | Результат композиции |
|---|---|---|---|---|
| Процедура, подпрограмма, функция | Идентификатор | Непосредственное обращение. оператор вызова | Библиотеки подпрограмм и функций | Программа |
| Модуль | Паспорт модуля, связи | Вызов модулей. интеграция модулей | Банк, библиотеки модутей | Программа с модульной структурой |
| Объект | Описание класса | Создание экземпляров классов, вызов методов | Библиотеки классов | Объектно-ориентированная программа |
| Компонент | Описание логики (бизнес), интерфейсов (*APL*, IDL), схемы развертывания | Удаленный вызов в компонентных моделях (CQV1 CORBA,*OSF*,...) | Регозитарий компонентов. серверы и контейнеры компонентов | Распределенное компонентно-ориентированное приложение |
| Сервис | Описание бизнес-логики интерфейсов сервиса (XML, WSDL, ...) | Удаленный вызов (RPC, НИР, SOAP,...) | Индексация и каталогизация сервисов (XML, UDDL..) | Распределенное сервисо-ориентированное приложение |

*Интерфейс* отображает операции обращения к реализации компонента, описывается в языках IDL или *APL*, включает в себя описание типов и операции передачи аргументов и результатов для взаимодействия

компонентов. Компонент, как физическая сущность, может иметь множество интерфейсов.

*Развертывание* – это выполнение физического файла в соответствии с конфигурацией (версией), параметрами настройки для запуска на выполнение компонента.

Компоненты наследуются в виде классов и используются в модели, композиции и в каркасе (Фреймворке) интегрированной среды. Управление компонентами проводится на архитектурном, компонентном или интерфейсном уровнях, между которыми существует взаимная связь.

Компонент описывается в языке программирования, не зависит от операционной среды (например, от среды виртуальной машины JAVA) и от реальной платформы (например, от платформ в системе CORBA), где он будет функционировать.

**Типы компонентных структур.** Расширением понятия компонента является *шаблон* (паттерн) – абстракция, которая содержит описание взаимодействия совокупности объектов в общей кооперативной деятельности, для которой определены роли участников и их ответственности. Шаблон является повторяемой частью программного элемента как схема или взаимосвязь контекста описания для решения проблемы.

*Компонентная модель* – отражает проектные решения по композиции компонентов, определяет типы шаблонов компонентов и допустимые между ними взаимодействия, а также является источником формирования файла развертывания ПС в среде функционирования.

*Каркас* – представляет собой высокоуровневую абстракцию проекта ПС, в которой функции компонентов отделены от задач управления ими. Например, бизнес-логика – это функция компонента, а каркас – управление ими. Каркас объединяет множество взаимодействующих между собою объектов в некоторую интегрированную среду для решения заданной конечной цели. В зависимости от специализации каркас называют "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. Использование такого типа каркаса является характерным для ООП.

Для каркаса типа "черный ящик" в его видимую часть выносятся точки, разрешающие изменять входы и выходы.

**Композиция компонентов** может быть следующих типов:

- *композиция компонент-компонент* обеспечивает непосредственное взаимодействие компонентов через интерфейс на уровне приложения;
- *композиция каркас-компонент* обеспечивает взаимодействие каркаса с компонентами, при котором каркас управляет ресурсами компонентов и их интерфейсами на системном уровне;
- *композиция компонент-каркас* обеспечивает взаимодействие компонента с каркасом по типу "черного ящика", в видимой части которого находится описание файла для развертывания и выполнения определенной функции на сервисном уровне;
- *композиция каркас-каркас* обеспечивает взаимодействие каркасов, каждый из которых может разворачиваться в гетерогенной среде и разрешать компонентам, входящим в каркас, взаимодействовать через их интерфейсы на сетевом уровне.

Компоненты и их композиции, как правило, запоминаются в репозитарии компонентов, а их интерфейсы в репозитарии интерфейсов.

**Повторное использование** в компонентном программировании – это применение готовых порций формализованных знаний, добытых во время реализации ПС, в новых разработках [5.13-5.15].

*Повторно используемые компоненты (ПИК)* – это готовые компоненты, элементы оформленных знаний (проектные решения, функции, шаблоны и др.) в ходе разработки, которые используются не только самими разработчиками, а и другими пользователями путем адаптации их к условиям новой ПС, что упрощает и сокращает сроки ее разработки. В системе Интернет в данный момент имеется много разных библиотек, репозитариев, содержащих ПИК, и их можно использовать в новых проектах.

При создании компонентов, ориентированных на повторное использование, их интерфейсы должны содержать операции, которые обеспечивают разные способы применения компонентов. Как и любые элементы производства, ПИК должны отвечать определенным требованиям, обладать характерными свойствами и структурой, а также иметь механизмы обращения к ним и др.

Главным преимуществом создания ПС из компонентов является уменьшение затрат на разработку за счет выбора готовых компонентов с подобными функциями, пригодными для практического применения и настройки их к новым условиям, на что тратится меньше усилий, чем на аналогичную разработку.

Поиск готовых компонентов основывается на методах классификации и каталогизации. Метод классификации предназначен для представления информации о компонентах с целью быстрого поиска и отбора. Метод каталогизации – для физического их размещения в репозитариях с обеспечением доступа к ним в процессе интеграции.

**Методология компонентной разработки ПС.** Создание компонентной системы начинается с анализа ПрО и построения концептуальной модели, на основе которой создается компонентная модель (рис. 5.7) , включающая проектные решения по композиции компонентов, использованию разных типов шаблонов, связей между ними и операции развертывания ПС в среде функционирования.

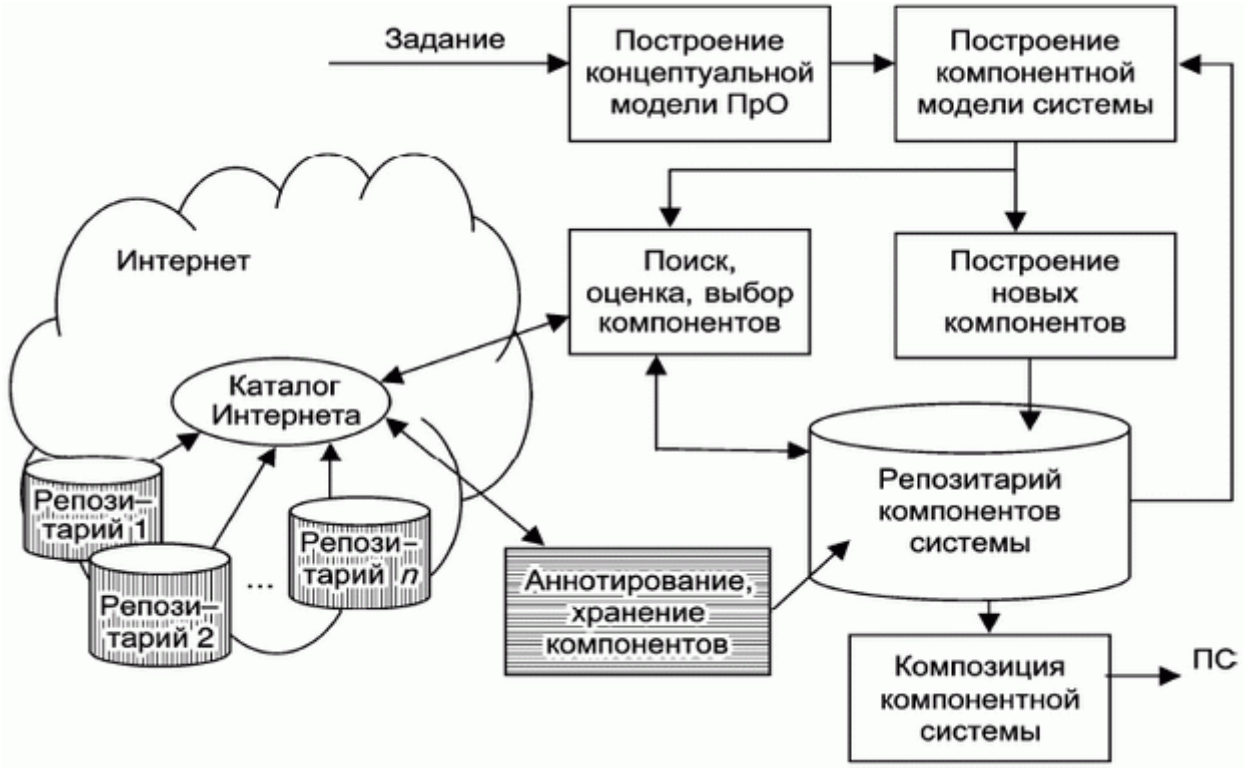



**Рис. 5.7.** Концептуальна схема построения ПС из компонентов в среде Интернет

Готовые компоненты берутся из репозитариев Интернета и используются при создании ПС, технология построения которых описывается следующими этапами ЖЦ.

1. *Поиск, выбор ПИК* и разработка новых компонентов, исходя из *системы классификации* компонентов и их каталогизации, формализованное определение спецификаций интерфейсов, поведения и функциональности компонентов, а также их аннотирования и размещения в репозитарии системы или в Интернет.
2. *Разработка требований* (Requirements) к ПС – это формирование и описание функциональных, нефункциональных и др. свойств ПС.
3. *Анализ поведения* (Behavioral Analysis) ПС заключается в определении функций системы, деталей проектирования и методов их выполнения.
4. *Спецификация интерфейсов и взаимодействий компонентов* (Interface and Interaction Specification) отражает распределение ролей компонентов, интерфейсов, их идентификацию и взаимодействие компонентов через поток действий (workflow).

1. 5. *Интеграция набора компонентов и ПИК* (Application Assembly and Component Reuse) в единую среду основывается на подборе и адаптации ПИК, определении совокупности правил, условий интеграции и построении конфигурации каркаса системы.

1. *Тестирование компонентов и среды* (*Component Testing*) основывается на методах верификации и тестирования для проверки правильности как отдельных компонентов и ПИК, так и интегрированной из компонентов ПС.
2. *Развертывание (System Deployment)* включает оптимизацию плана компонентной конфигурации с учетом среды пользователя, развертку отдельных компонентов и создание целевой компонентной конфигурации для функционирования ПС.
3. *Сопровождения ПС* (System Support and Maintenance) состоит из анализа ошибок и отказов при функционировании ПС, поиска и исправления ошибок, повторного ее тестирования и адаптации новых компонентов к требованиям и условиям интегрированной среды.

### 5.1.5. Аспектно-ориентированное программирование

*Аспектно-ориентированное программирование* (АОП) [5.15–5.17] – это парадигма построения гибких к изменению ПС путем добавления новых аспектов (функций), обеспечивающих безопасность, взаимодействие компонентов с другой средой, также синхронизацию одновременного доступа частей ПС к данным и вызов новых общесистемных средств.

Аспектом может быть ПИК, фрагмент программы, реализующий концепцию взаимодействия компонентов в среде, защиту данных и др. ПС, которая создается из ПИК, объектов, небольших методов и аспектов, дополняется необходимыми фрагментами взаимодействия, синхронизации, защиты и т.п. путем встраивания их в точки компонентов ПС, где они необходимы. В результате встроенные фрагменты дополняют компоненты новым содержательным аспектом и тем самым значительно усложняют процесс вычислений.

Практическая реализация аспектов, размещенных в разных частях элементов ПС, обеспечивается механизмом перекрестных ссылок и точками соединения, через которые осуществляется связь с аспектным фрагментом для получения определенной дополнительной функции.

В основе АОП лежит метод разбиения задач ПрО на ряд функциональных компонентов, определения необходимости применения разного рода дополнительных аспектов и установления точек расположения аспектов в отдельных компонентах, где это требуется. Эти работы выполняются на этапах ЖЦ процесса разработки, способствуют реализации ПС с ориентацией на взаимодействие компонентов или их синхронизацию. Такой подход известен при проведении отладки программ, когда фрагменты отладочных программ встраиваются в отдельные точки исходной программы для выдачи промежуточных результатов. Когда отладка завершается успешно, эти участки удаляются. В случае аспектов – их программные фрагменты остаются в программе.

Создание конечного продукта ПС в АОП выполняется по технологии, соответствующей разработке компонентных систем, с той особенностью, что используемые аспекты определяют особые условия выполнения компонентов в среде взаимодействия. Аспекты можно рассматривать как выполнение разных ролей взаимодействующими лицами, что приближает аспект к роли программного агента, выполняющего дополнительные функции при определении архитектуры системы и повышение качества компонентов.

Для использования аспектов при выработке проектных решений используется механизм фильтрации входных сообщений, с помощью которых проводится изменение параметров и имен текстов аспектов в конкретно заданном компоненте системы. Код компонента становится "нечистым", когда он пересечен аспектами, и при композиции с другими компонентами общие средства (вызов процедур, RPC, RMI, IDL и др.) становятся недостаточными. Это так как аспекты требуют декларативного сцепления описаний и связано с тем, что фрагменты находятся или берутся из различных объектов. Один из механизмов композиции компонентов и аспектов – фильтр композиции, который обновляет аспекты без изменения функциональных возможностей. Фактически фильтрация касается входных и выходных параметров сообщений, которые переопределяют соответствующие имена объектов. Иными словами, фильтры делегируют внутренним частям компонентов параметры, переадресовывая ранее установленные ссылки, проверяют и размещают в буфере сообщений, локализуют ограничения и готовят компонент для выполнения.

В ОО-программах могут быть методы, выполняющие дополнительно некоторые расчеты с обращением на другие методы внешнего уровня. Деметр сформулировал закон [5.17], согласно которому длинные последовательности мелких методов не должны выполняться. В результате создается код алгоритма с именами классов, не задействованных в расчетных операциях, а также дополнительный класс, который расширяет код этими расчетами.

С точки зрения моделирования, аспекты можно рассматривать как каркасы декомпозиции системы, в которых отдельные аспекты пересекают ряд многократно используемых ПИК (рис. 5.8).

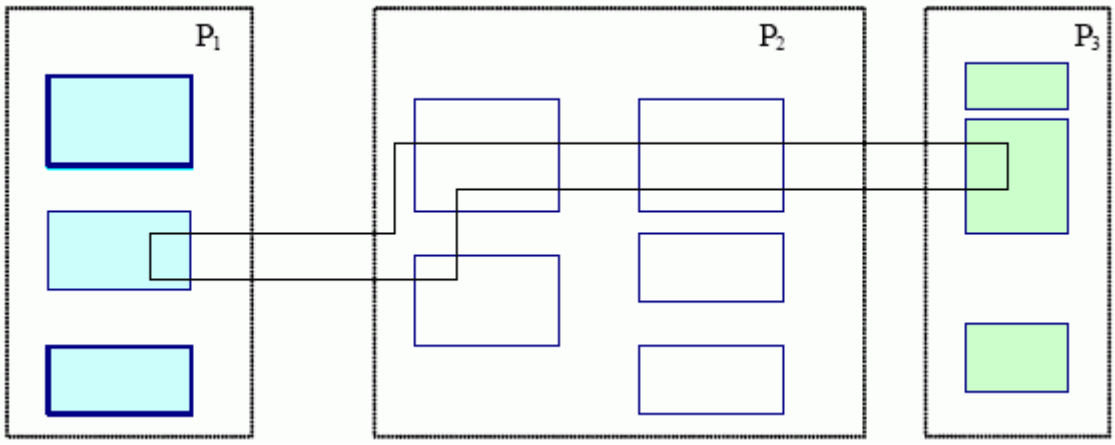



**Рис. 5.8.** Пример расположения аспектов в программах P1, P2 и P3

Разным аспектам проектируемой системы могут отвечать и разные парадигмы программирования: объектно-ориентированные, структурные и др. Они по отношению к проектируемой ПрО образуют мультипарадигмную концепцию обработки, такую как синхронизация, взаимодействие, обработка ошибок и др. со значительными доработками процессов их реализации. Кроме того, этот механизм позволяет устанавливать аспектные связи с другими предметными областями в терминах родственных областей. Языки АОП позволяют описывать

аспекты для разных ПрО. В процессе компиляции пересекаемые аспекты объединяются, оптимизируются, генерируются [5.16] и выполняются в динамике.

Существенной особенностью АОП является построение модели, 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, реализуя шаблоны взаимодействия. Один из недостатков – пересечение отдельных компонентов аспектами может привести к понижению эффективности их выполнения.

Переплетение аспектов с компонентами проявиться на последующих этапах процесса разработки и поэтому требуется минимизация количества сцеплений между аспектами и компонентами через ссылки в вариантах использования, сопоставление с шаблоном или блоком кода, в котором установлены перекрестные ссылки.

В ходе анализа ПрО и построении ее характеристической модели устанавливается связь с дополнительными аспектами, что приводит к статическому или "жесткому" связыванию компонентов и аспектов модели, учету этого случая при компиляции.

Аспекты с точки зрения моделирования можно рассматривать как каркасы декомпозиции системы с многократным использованием. АОП становится мультипарадигмовой концепцией, сущность которой состоит в том, что разным аспектам проектируемой ПС, должны отвечать разные парадигмы программирования. Каждая из парадигм относительно реализации разных аспектов ПС (синхронизации, внедрения, обработки ошибок и др.) требует их усовершенствования и обобщения для каждой новой ПрО.

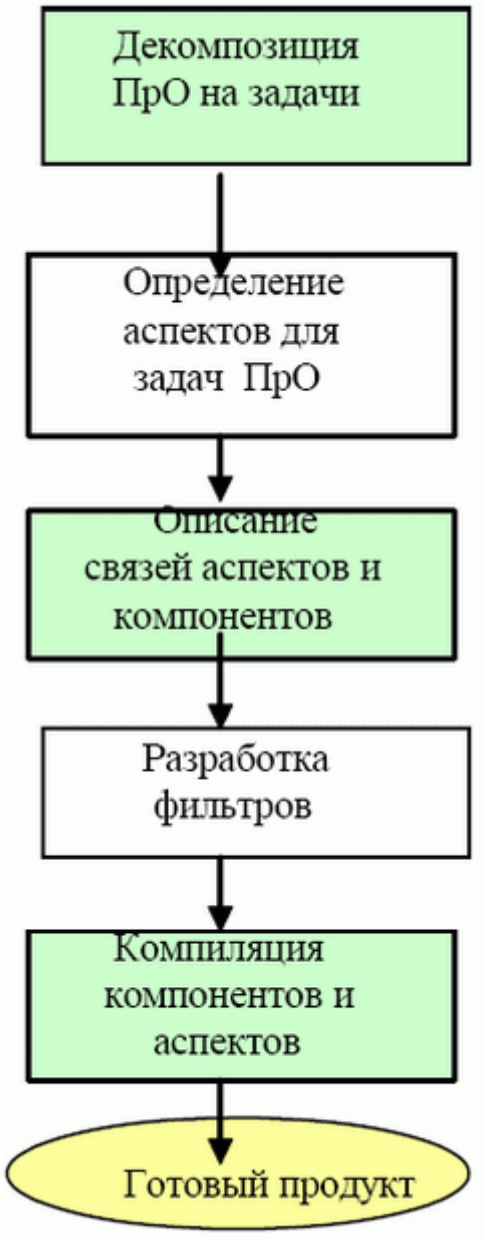



**Рис. 5.9.** Технологическая схема проектирования ПС средствами АОП

В АОП используется модель модульных расширений в рамках метамодельного программирования, которая обеспечивает оперативное использование новых механизмов композиции отдельных частей ПС или их семейств с учетом предметно-ориентированных возможностей языков (например, SQL) и каркасов, которые

поддерживают аспекты. Технология разработки прикладной системы с использованием АОП включает общие этапы (рис. 5.9):

1. Декомпозиция функциональных задач с условием многоразового применения модулей и выделенных аспектов, т.е. свойств их выполнения (параллельно, синхронно, безопасно и т.д.).
2. Анализ языков спецификации аспектов и определение конкретных аспектов для обеспечения взаимодействия, синхронизации и др. задач ПрО.
3. Определение точек встраивания аспектов в компоненты и формирование ссылок и связей с другими элементами.
4. Разработка фильтров и описание связей аспектов с функциональными компонентами, выделенными в ПрО, отображение фильтров в модели EJB на стороне сервера и управление данными с обеспечением безопасности, *защиты доступа* к некоторым данным.
5. Определение механизмов композиции (вызовов процедур, методов, сцеплений) функциональных модулей многоразового применения и аспектов в точках их соединения, как фрагментов свойств управления выполнением этих модулей, или ссылок из этих точек на другие модули.
6. Создание объектной или компонентной модели, дополнение ее входными и выходными фильтрами сообщений, посылающих объектам ссылки, задания на выполнение методов или аспектов.
7. Анализ библиотеки расширений для выбора некоторых функциональных модулей, необходимых для реализации задач ПрО.
8. Компиляция, совместная отладка модулей и аспектов, после чего композиция их в готовый программный продукт.

В процессе создания ПС с применением аспектов используются IP-библиотека расширений, активные библиотеки, Smalltalk и ЯП, расширенные средства описания аспектов [5.17].

*IP-библиотека* содержит функции компиляторов, средства оптимизации, редактирования, отображения и др. Например, библиотека матриц для вычисления выражений с массивами, предоставляющая память и др., получила название библиотеки генерирующего типа.

Иной вид библиотек АОП – *активные библиотеки*, которые содержат не только базовый код реализации понятий ПрО, но и целевой код обеспечения оптимизации, адаптации, визуализации и редактирования. Активные библиотеки пополняются средствами и инструментами *интеллектуализации* агентов, с помощью которых обеспечивается разработка специализированных агентов для реализации конкретных задач ПрО.

### 5.1.6. Генерирующее (порождающее) программирование

Генерирующее программирование (generate programming) – генерация семейств приложений из отдельных элементов компонентов, аспектов, сервисов, ПИК, каркасов и т.п. Базис этого программирования – ООП, дополненный механизмами генерации ПИК, другими многоразовыми элементами, а также свойствами их изменчивости, взаимодействия и др. [5.17].

В нем используются разные методы программирования для поддержки инженерии ПрО как дисциплины инженерного проектирования семейств ПС из разных ранее изготовленных продуктов путем объединения технологии генерации как отдельных ПС, так и их семейств. Эта дисциплина использует методы программирования, соответствующие формализмы и модели для создания более качественных представителей семейства ПС по принципу конвейера.

Главный элемент ПС – это семейство ПС или конкретные его экземпляры, которые генерируются на основе общей генерирующей модели домена (generative *domain model*) и включающей в себя средства определения членов (представителей) семейства, методы сборки членов семейства и базу конфигурации с набором правил развертывания в операционной среде.

Каждый член семейства создается путем интеграции отдельных компонентов, планирования, контроля и оценки результатов *интеграционного тестирования*, а также определения затрат на применение многократно используемых ПИК, в том числе из активной библиотеки [5.17].

Базовый код элементов активной библиотеки содержит целевой код по обеспечению процедур компиляции, отладки, визуализации и др. Фактически компоненты этих библиотек – это интеллектуальные агенты, генерирующие новых агентов в расширяемой среде программирования для решения конкретных задачи ПрО. Эта среда содержит специальные метапрограммы и компоненты библиотек для осуществления отладки, проверки композиции и взаимодействия компонентов. Среда пополняется новыми сгенерированными компонентами для членов семейства, в качестве компонентов многоразового применения.

Реализация целей порождающего программирования по включению ПИК в другие члены семейства проводится по двум сформировавшимся инженерным направлениям [5.11–5.15]:

1. *прикладная инженерия* – процесс производства конкретных ПС из ПИК, ранее созданных в среде независимых систем, или как отдельные элементы процесса инженерии некоторой ПрО;

2. *инженерия* ПрО – построение членов семейства или самого семейства систем путем сбора, классификации и фиксации ПИК в качестве их конструктивных элементов, а также для частей систем для конкретной ПрО. Поиск, адаптация ПИК и внедрение их в новые члены семейства ПС проводится с помощью специальных инструментальных средств типа репозитария.

Составная часть инженерии ПрО – инженерия приложений как способ создания отдельных целевых членов семейства для конструирования из этих членов новые ПИК, многократно используемых проектных решений и генерируемых как системы семейства ПрО.

Основные этапы инженерии ПрО приведены на рис. 5.10:

анализ ПрО и выявление объектов и отношений между ними;

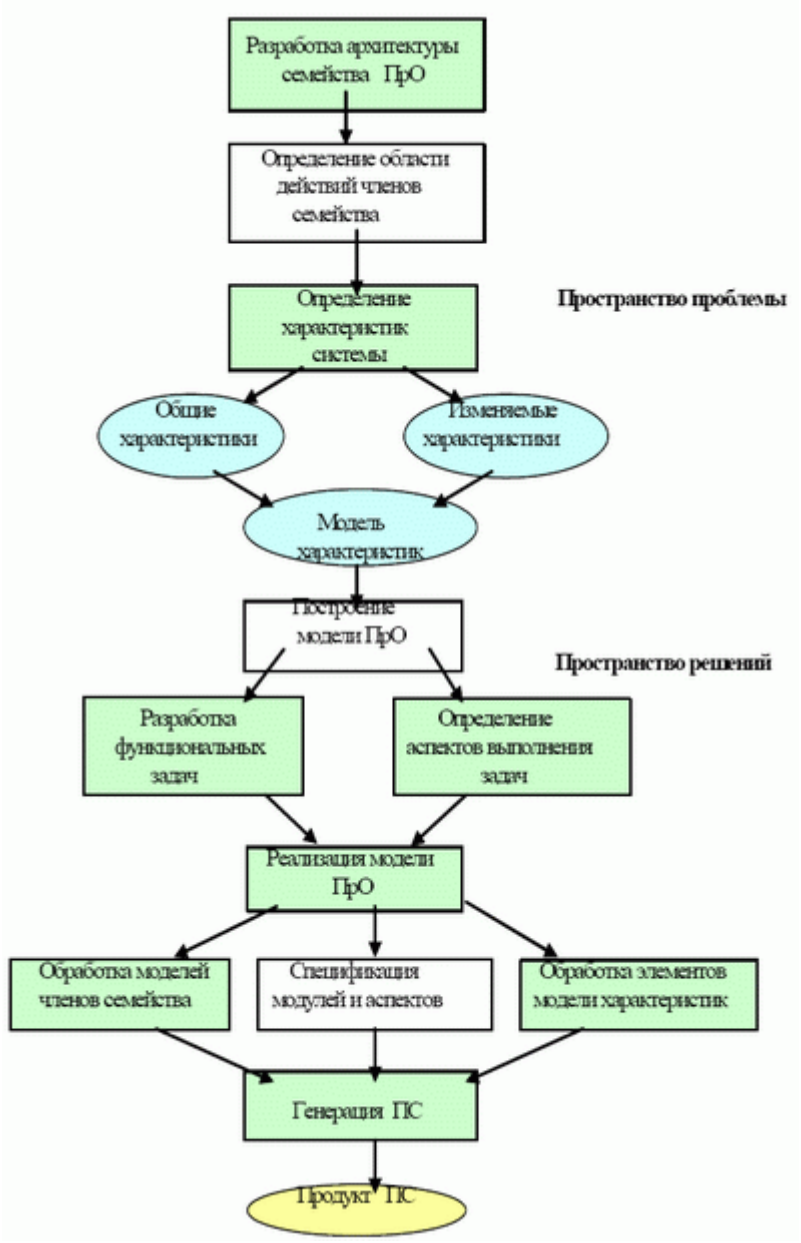
определение области действий объектов ПрО;

определение общих функциональных и изменяемых характеристик, построение модели, устанавливающей зависимость между различными членами семейства;

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

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

генерация отдельного домена, члена семейства и ПС.




увеличить изображение

**Рис. 5.10.** Технологическая схема инженерии ПрО

Генерация доменной модели для семейства ПС основывается на модели характеристик, наборе компонентов реализации задач ПрО, совокупности компонентов и их спецификациях. Результат генерации – готовая

подсистема или отдельный член семейства.

К рассмотренной схеме инженерии ПрО также относятся:

– *корректировка процессов* при включении новых проектных решений или при изменении состава ПИК;

– *моделирование изменчивости и зависимостей* с помощью механизмов изменения моделей (объектных, взаимодействия и др.), добавления новых требований и понятий, а также фиксации их в модели характеристик и в конфигурации системы;

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

ЖЦ разработки с повторным или многократным использованием обеспечивает получение семейства систем, определение области их действия, а также определение общих и изменяемых характеристик представителей семейства, заданных в модели характеристик. При их определении используются пространство проблемы и пространство решений.

*Пространство проблемы* (space problem) – компоненты семейства системы, в которых используется ПИК, объекты, аспекты и др., процесс разработки которых включает в себя системные инструменты, а также созданные в ходе разработки ПрО. Инженерия ПрО объединяет в модели характеристик функциональные характеристики, свойства выполнения компонентов, изменяемые параметры разных частей семейства, а также решения, связанные с особенностями взаимодействия групп членов семейства ПС.

Инженерия ПрО обеспечивает не только разработку моделей членов семейства (подсистем), а и моделирование понятий ПрО, модель характеристик для подсистем и набора компонентов, реализующих задачи ПрО. В рамках инженерии ПрО используются горизонтальные и вертикальные типы компонентов в терминологии системы CORBA [5.14, 5.18].

К горизонтальным типам компонентов отнесены общие системные средства, которые нужны разным членам семейства, а именно: графические пользовательские интерфейсы, СУБД, системные программы, библиотеки расчета матриц, контейнеры, каркасы и т.п.

К вертикальным типам компонентов относятся прикладные системы (медицинские, биологические, научные и т.д.), методы инженерии ПрО, а также компоненты горизонтального типа по обслуживанию архитектуры многократного применения компонентов и их интерфейсов и др.

*Пространство решений* (space solution) – компоненты, каркасы, *шаблоны проектирования* ПрО, а также средства их соединения или встраивания в ПС и оценки избыточности. Элементы пространства реализуют решение задач этой ПрО. Каркас оснащен механизмом изменения параметров модели, которые требуют избыточную фрагментацию "множество мелких методов и классов". *Шаблоны проектирования* обеспечивают создание многократно используемых решений в различных типах ПС. Для задания и реализации таких аспектов, как синхронизация, удаленное взаимодействие, защита данных и т.п. применяются технологии ActiveX и *JavaBeans*, а также новые механизмы композиции и др.

Примером систем поддержки инженерии ПрО и реализации горизонтальных методов является система DEMRAL [5.17, 5.14], предназначенная для разработки библиотек: численного анализа, распознавания речи, графовых вычислений и т.д. Основные виды элементов этой библиотеки – абстрактные типы данных (*abstract data types – ADT*) и алгоритмы. DEMRAL позволяет моделировать характеристики ПрО и представлять их в характеристической модели и *предметно-ориентированных языках* описания конфигурации.

Система конструирования RSEB в среде генерирующего программирования использует методы, относящиеся к вертикальным методам, а также ПИК и Use Case при проектировании больших ПС. Методы вертикального типа вызывают различные горизонтальные методы, относящиеся к разным прикладным подсистемам. При работе над отдельной частью семейства могут применяться аспекты взаимодействия, структуры, потоков данных и др. Важную роль при этом выполняет графический пользовательский интерфейс и метод обеспечения взаимодействия компонентов в распределенных средах (например, в CORBA).

### 5.1.7. Агентное программирование

Понятие интеллектуального и программного агента появилось более 20 лет назад, их роль в программной инженерии все время возрастает [5.19–5.23]. Так, в [5.23] Джекобсон отметил перспективу использования агентов в качестве менеджеров проектов, разработчиков архитектуры с помощью диаграмм use case и др.

Основной теоретический базис данного программирования – темпоральная, модальная и мультимодельная логики, дедуктивные методы доказательства правильности свойств агентов и др.

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 [5.19]. Агент обладает следующими свойствами:

автономность – это способность действовать без внешнего управляющего воздействия;

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

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

способность к взаимодействию с другими агентами (или людьми).

Основными задачами программного агента являются:

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 связаны знания типа убеждение, намерение, обязательства и т.п. 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 через сообщения или запросы и выполняют заданные действия или операции в соответствии с имеющимися знаниями.

Агенты могут быть локальными и распределенными (рис. 5.11). Процессы локальных агентов протекают в клиентских серверах сети, выполняют заданные функции и влияют на общее состояние среды функционирования. Распределенные агенты располагаются в разных узлах сети, выполняют автономно (параллельно, синхронно, асинхронно) предназначенные им функции и могут влиять на общее состояние распределенной среды.

Характер взаимодействия между агентами зависит от совместимости целей, компетентности и т.п. [5.21].

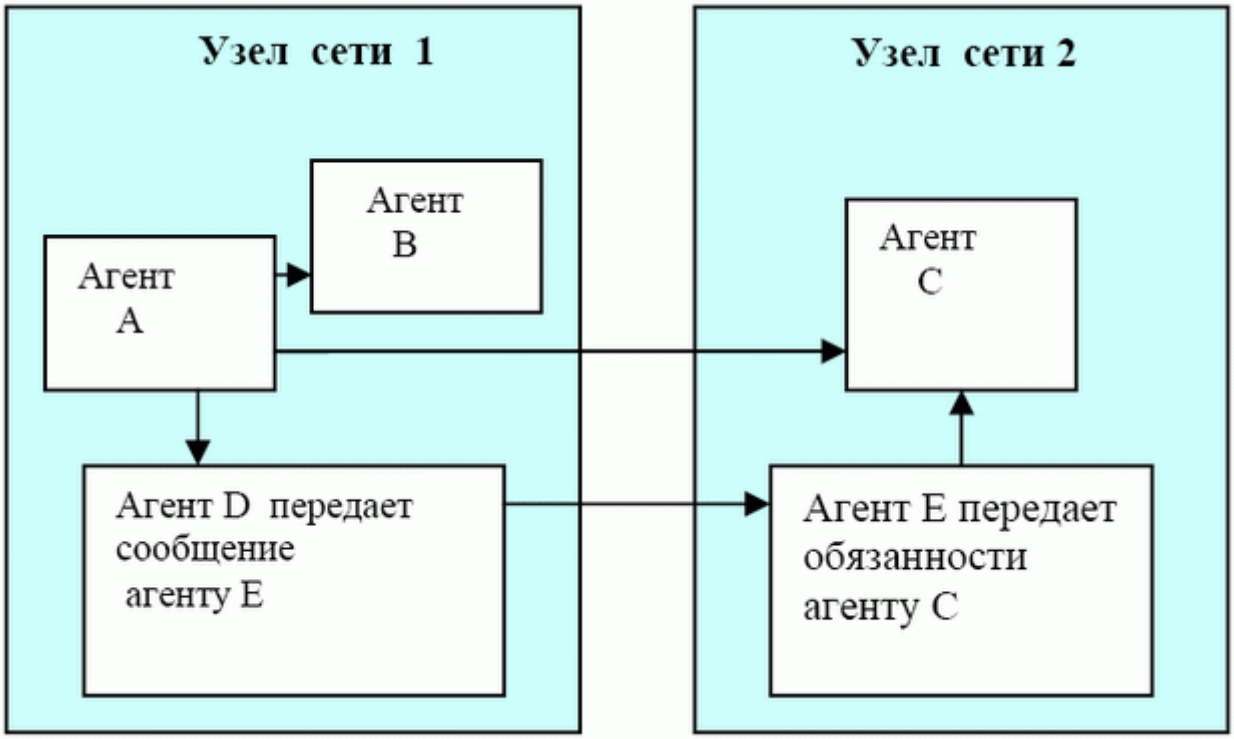



**Рис. 5.11.** Пример взаимодействия агентов в разных средах

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.

Агенты взаимодействуют между собой с помощью разных механизмов, а именно: координация, коммуникация, кооперация или коалиция.

Под *координацией агентов* понимается процесс обеспечения последовательного функционирования при согласованности их поведения и без взаимных конфликтов. Координация агентов определяется:

взаимозависимостью целей других агентов-членов коалиции, а также от 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.

Главное средство коммуникации агентов – транспортный протокол TCP/IP или протокол агентов ACL (Agent Communication Languages). Управление агентами (Agent Management) выполняется с помощью сервисов: передача сообщений между агентами, доступ агента к серверу и т.п. *Коммуникация* агентов базируется на общем протоколе, языке HTML и декларативном или процедурном (Java, Telescript, ACL и т.п.) языке описания этого протокола.

Примером активной и скоординированной деятельности агентов по поиску необходимой информации является среда Интернет. В нем агенты обеспечивают доступ к информационным ресурсам, а также выполняют ее анализ, интеграцию, фильтрацию и передачу результата запроса пользователю.

Каждый агент выполняет определенную функцию, передает друг другу задание на последующее действие по доступу к информационному ресурсу, извлечению необходимой информации и передачи ее для обработки следующим агентам. При этом могут возникать нерегулярные состояния (тупики, отсутствие ресурса и др.).

Одной из систем построения агентов, основанной на обмене сообщениями в ACL, является JATLite, которая с помощью Java-классов создает новых агентов, вычисляющих определенные функции в распределенной среде. Система Agent Builder предназначена для конструирования программных агентов, которые описываются в языке Java и могут взаимодействовать на языке *KQML* (Knowledge Guery and Manipulation Language) [5.19–5.23].

Построенные агенты выполняют функции: менеджера проекта и онтологий, визуализации, отладки и др. Реализацию механизмов взаимодействий агентов обеспечивает система JAFMAS, ряд других мультиагентных систем, что описано в [5.19].

## 5.2. Теоретическое программирование

*Теоретическое программирование* включает в себя формальные методы, основанные на *спецификации программ*, и методы, основанные на математических дисциплинах (логика, *алгебра*, *комбинаторика*) и обеспечивающие математический метод анализа и осмысления задач ПрО, а также разработку программ с математической символикой, правильность которых надо доказывать, чтобы получить на компьютере требуемые результаты. В связи с этим получили развитие отдельные направления в *теории программирования*, которые и предоставляют аппарат конструирования структур программ с применением математических методов.

Так, направление, связанное с использованием математического аппарата для описания действий над объектами как *алгебраические операции* базовых основ алгебры, получило название алгебраического программирования, алгоритмики и др. Украинские ученые разработали теоретические направления программирования, в частности:

- алгебраическое, инсерционное программирование (Летичевский А.А. и др.) [5.24, 5.25];
- экспликативное программирование (Редько В.Н.) и наука о программах (программология), объединяющая логический и математический аппарат для конструирования программ [5.26–5.28];
- алгебро-алгоритмическое программирование (Цейтлин Г.Е.), объединяющее алгебраический аппарат, теорию алгоритмов и операции над множествами [5.29–5.30]. Остановимся на их краткой характеристике.

### 5.2.1. Алгебраическое, инсерционное программирование

Алгебраическое программирование (АП) – это конструирование программ с алгебраическими преобразованиями и функциями интеллектуальных агентов. В основе математического аппарата АП лежит алгебра языка действий AL (Action Language) и понятие транзитивной системы [5.24, 5.25], в качестве механизма определения поведения систем и механизмов ее эквивалентности. В качестве понятий в общем случае могут быть компоненты, программы и их спецификации, объекты, взаимодействующие друг с другом и со средой их существования.

Основу АП составляет математическая модель, которая включает в себя следующие понятия:

- агент как транзитивная система с поведением и его завершением;
- поведение агентов, задаваемое в языке AL с помощью операций $a, u, u + v$, констант $\Delta$, $\perp$, $0$, граничных условий и рекурсий;
- среда, состоящая из агентов и функций погружения, которая обозначается $env$ и имеет в качестве параметров состояние среды и агентное выражение;
- функция развертывания функциональных выражений в простые агентные выражения;
- транзитивная система, представленная в виде композиции среды и системы взаимодействующих агентов, погруженных в эту среду.

Всякая транзитивная система имеет историю функционирования, которая включает последовательность действий и состояний –конечных или бесконечных. История функционирования хранит одно из

соответствующих состояний: успешное завершение вычислений в среде транзитивной системы; тупиковое состояние, когда каждая из параллельно выполняющихся частей системы находятся в состоянии ожидания; неопределенное состояние, возникающее при выполнении алгоритма с бесконечными циклами.

Расширение понятия транзитивных систем – это *множество заключительных состояний* с успешным завершением функционирования системы и без неопределенных состояний. Главный инвариант состояния транзитивной системы – поведение системы, задаваемое выражениями алгебры поведения $F(A)$ на множестве операций алгебры действий – префиксинг $a \cdot u$, недетерминированный выбор $u + v$ одного из двух поведений $u$ и $v$ со свойством ассоциативности и коммутативности. Конечное поведение системы задается константами $\Delta$, $\perp$, $0$, которые обозначают соответственно состояние успешного завершения, неопределенного и тупикового. Алгебра поведения включает отношение $\leq$, элемент $\perp$ как наименьший и операции поведения, являющиеся монотонными. Средствами алгебры поведения $F(A)$ доказана теорема про наименьшую неподвижную точку.

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

**Среда E**, где находится объект, определяется как агент в алгебре действий AL и функции погружения от двух аргументов $Ins(e, u) = e[u]$. Первый аргумент – это поведение среды, второй – поведение агента, который погружается в эту среду с заданным состоянием. Значения функций погружения – это новое состояние одной и той же среды. Базовым понятием является "действие", которое трансформирует состояние агентов, поведение которых, в конце концов, изменяется.

Поведение агентов характеризуется состоянием с точностью до слабой эквивалентности. Каждый агент рассматривается как транзитивная система с действиями, определяющими не детерминированный выбор и *последовательную композицию* (т.е. примитивные и сложные действия). *Последовательная композиция* – ассоциативная, а *параллельная композиция* – ассоциативная и коммутативная. *Параллельная композиция* раскладывается на комбинацию действий компонентов.

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

Язык действий AL имеет синтаксис и семантику. Синтаксис языка задает правила описания действий, семантика – функции, которые определяются средствами и выражениями языка и ставят в соответствие заданным выражениям значения в некоторой семантической области. Разные *семантические функции* могут давать равные абстракции и *свойства программ*. Семантика может быть вычислительной и интерактивной. Каждая алгебра действий – это гомоморфный образ алгебры примитивных действий, когда все слагаемые разные, а их представление однозначно с точностью до ассоциативности и коммутативности при детерминированном выборе.

Агенты рассматриваются как значения транзитивных систем с точностью до бисимиляционной эквивалентности, которая характеризуется непрерывной алгеброй с аппроксимацией и двумя операциями: недетерминированным выбором и префиксингом. Среда вводится как агент, в нее погружается функция, которая имеет поведение (агент и среда). Произвольные непрерывные функции могут быть использованы как функции погружения в эти функции. 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.

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 – группы, кольца, поля. Теория АП обеспечивает создание математической информационной среды с универсальными математическими конструкциями, вычислительными механизмами, учитывающими особенности разработки ПС и функционирования.

Алгебраическое программирование концентрирует внимание на проблемах *интеллектуализации* и аспектах поведения агентов в распределенной среде, куда они погружаются. Оно постепенно перешло в *инсерционное программирование* путем вставки, погружения агентов в разнообразные среды для преобразования поведения

агентов на основе модели поведения, соответствующей размеченной транзитивной системы и бисимуляционной эквивалентности [5.25]. Данное программирование обобщает алгебраическое преобразование множества состояний информационной среды на объекты, обладающие поведением. Схема создания агентных программ представлена на рис. 5.12.

Главное действующее лицо инсерционного программирования – агент, обладающий поведением при его погружении в среду, которую он также меняет. Характерной особенностью является недетерминированное поведение агентов и сред, как это происходит в реальных системах. При этом программа агента требует не интерпретации, а моделирования, поскольку она представляется транзитивной системой в виде композиции среды и системы взаимодействующих агентов, погруженных в эту среду. Язык действий – порождающий, в нем задается синтаксис действий, агентных функциональных выражений и внутренних состояний среды. Кроме того, в этом языке описывается семантика функций развертывания агентных функциональных выражений и погружений агентов в среду.

Программа – набор параметров и начальных агентных выражений. Параметры могут быть фиксированные и переменные, изменяющиеся при переходе от одной среды в другую. Особенно это касается параметров развертывания.

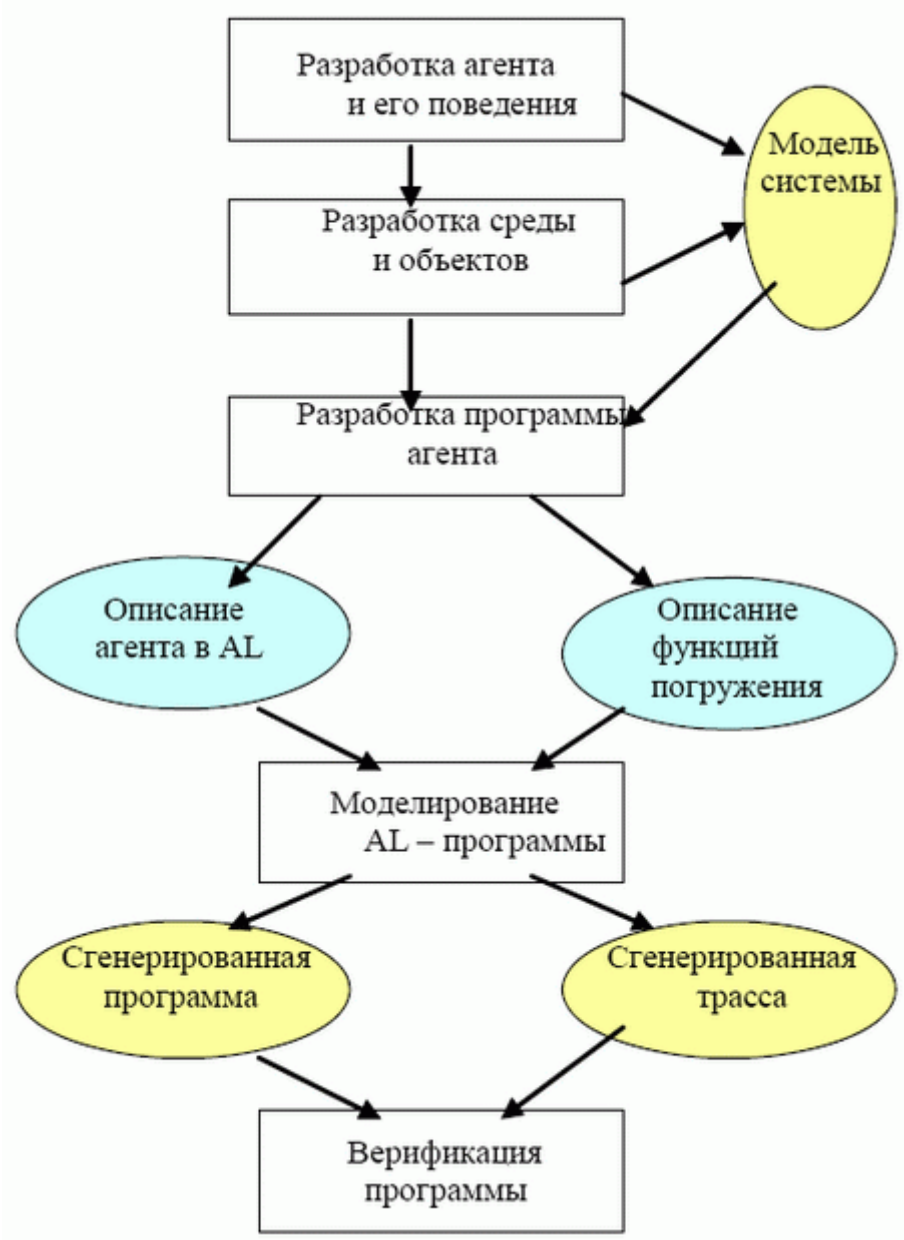



**Рис. 5.12.** Технологическая схема в АП

На следующем уровне описания программы находятся функции, которые преобразуют любое агентное выражение в AL-программу действий. Например, функция $U$ развертывания функциональных выражений может быть представлена в виде простых агентных выражений:

$$U(u+v) = U(u) + U(v), U(u) = u,$$

где $u, v$ – параметры агентных выражений.

Функция развертывания агентных функциональных выражений задает семантику AL-программы в виде программы функционирования агента в любой среде. Если функция развертывания обрабатывается за конечное число шагов, то она содержит конечное выражение. Множество всех переходов, заключительных и тупиковых состояний - перечислимое, даже если функция развертывания имеет бесконечное множество нетривиальных итераций. Состоянием транзитивной системы являются ограниченные выражения, определяемые операцией выбора $+$, соотношением $x + 0 = a$ и отношением перехода с правилом $u \to U(u)$.

Третий уровень программы - это функция развертывания погружений, задаваемых в алгебре действий. Эти функции определяются на объектах среды и агентов, т.е. агентных выражениях. Для корректности функций погружения необходимо, чтобы она зависела только от поведения агента и среды и была непрерывной функцией. Следующим шагом разработки программ является ее реализация в ЯП, например в C++, когда уточняются типы данных и параметры. Затем проводится верификация полученной программы для проверки правильности выполнения ее поведения в заданной модели. Более подробно об инсерционном программировании и средствах его автоматизации в [5.25].

### 5.2.2. Экспликативное, номинативное программирование

Экспликативное программирование (ЭП) ориентировано на разработку теории дескриптивных и декларативных программных формализмов, адекватных моделям структур данных, программ и средств конструирования из них программ [5.26-5.28]. Для этих структур решены проблемы существования, единства и эффективности. Теоретическую основу ЭП составляют логика, конструктивная математика, информатика, композиционное программирование и классическая теории алгоритмов. Для изображения алгоритмов программ используются алгоритмические языки и методы программирования: функциональное, логическое, структурное, денотационное и др.

Принципами ЭП являются:

*принцип развития* 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 реализацию проблемы с учетом *информационности данных* и *апликативности*. Программа рассматривается как функция, вырабатывающая выходные данные на основе входных данных. Функция - это объект, которому сопоставляется *денотат* имени функции с помощью отношения *именования* ( *номинации* ) ;

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

Развитие понятия функции осуществляется с помощью *принципа композиционности*, т.е. составления программ (функций) из более простых программ для создания новых объектов с более сложными именами (дескрипциями) для функций. Они включают номинативные (именные) и языковые выражения, термы и формулы.

Таким образом, процесс развития программы осуществляется в виде цепочки понятий:

данные - функция - имя функции - композиция - дескрипция.

Из них триада - "данные - функция - композиция" задает семантический аспект программы, а триада "данные - имя функции - дескрипция" - синтаксический аспект. Главным в ЭП является семантический аспект, система композиций и номинативности (КНС), ориентированная на систематическое описание номинативных отношений при построении данных, функций, композиций и дескрипций [5.27].

КНС содержит специальные языковые системы для описания разнообразных классов функций, которые называются композиционно-номинативными языками функций. Такие системы тесно связаны с алгебрами функций и данных, построены в семантико- синтаксическом стиле. Они отличается от традиционных систем (*моделей программ*) теоретико-функциональным подходом, классами однозначных $n$ -арных функций номинативными отображениями и структурами данных.

Для построения математически простых и адекватных *моделей программ* параметрического типа используется КНС и методы универсальной алгебры, математической логики и теория 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.

В рамках ЭП разработаны новые 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 [5.29].

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 отображение специального типа. Класс аппликативных композиций предназначен для конструирования широкого класса программ.

Практическая проверка теоретического аппарата формализации дедуктивных и ОО БД прошла в ряде экспериментальных проектов в *классе манипуляционных* данных БД заданных в SQL-подобных языках.

### 5.2.3. Алгоритмика программ

Алгоритмика программ – структурная схемотология построения последовательных и параллельных программ с аппаратом формальных алгебраических преобразований и канонических форм описания логических и операторных выражений. Ее основу составляют системы алгебраических алгебр (САА), расширенные формализмами для представления логических условий параллельных программ и методами символьной мультиобработки [5.30].

Построение и исследование алгебры алгоритмов началось с проектирования логических структур ЭВМ под руководством академика В.М.Глушкова. В результате была построена теория САА, которая затем Г.Е.Цейтлиным была положена в основу создания обобщенной теории структурированных схем алгоритмов и программ, называемой *алгоритмикой* [5.30].

Основными понятиями алгебры алгоритмики являются:

- операции над множествами, булевы операции, предикаты, функции и операторы;
- бинарные и n-*арные отношения*, эквивалентность, частично и полностью упорядоченные множества;
- граф-схемы и операции над графовыми структурами;
- операции сигнатуры САА, аксиомы и правила вывода *свойств программ* на основе сверточной, развертночной и комбинированной стратегий;
- *символьная обработка* и методы синтаксического анализа программ.

Объекты алгоритмики – 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 в ЯП и реализуются соответствующими системами программирования в машинное представление.

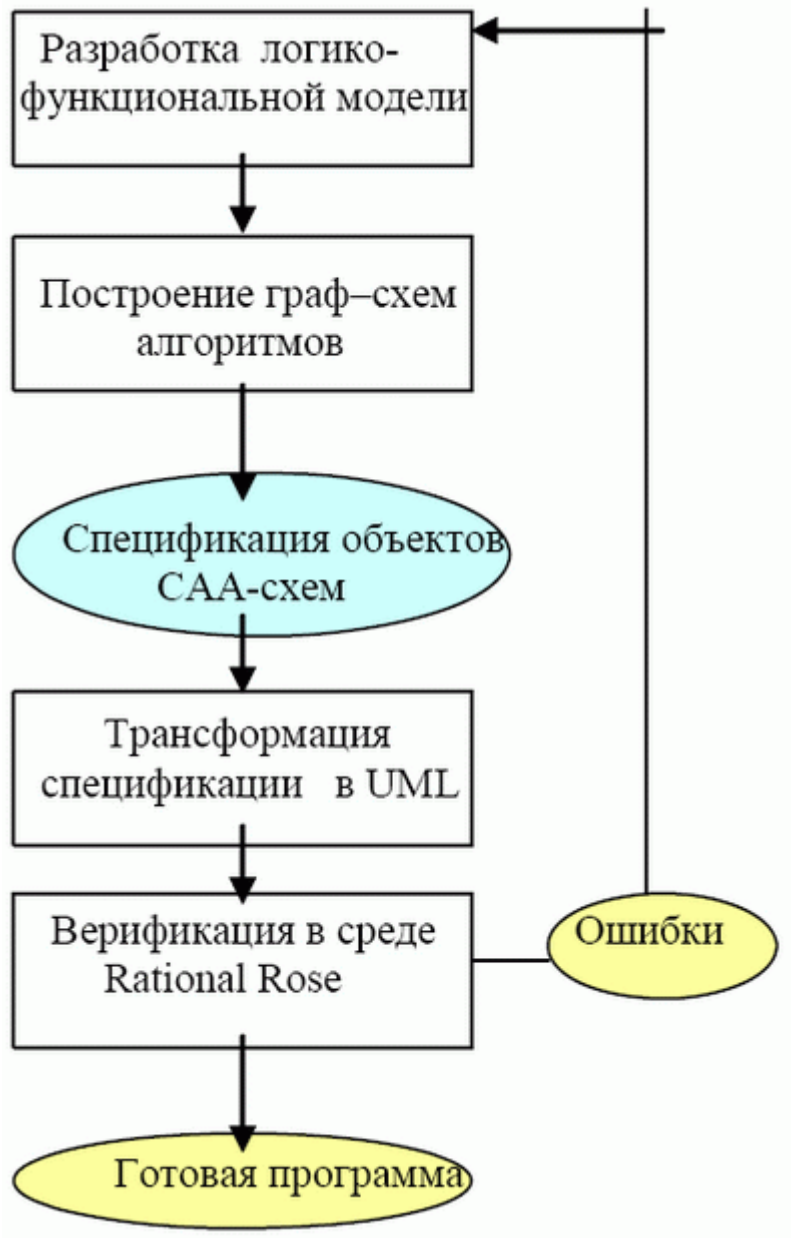
В рамках алгоритмики разработаны специальные инструментальные средства реализации алгоритмов программ, которые используют современные объектно-ориентированные средства и метод моделирования UML. Тем самым обеспечивается полный цикл работ по практическому применению разработанной теории алгоритмики для реализации прикладных задач, начиная с постановки задачи, формирования требований и разработки алгоритмов до получения программ решения этих задач (рис. 5.13).

**Алгебра алгоритмов.** Под алгеброй алгоритмов $\mathrm{AA} = \{A, \Omega\}$ понимается основа $\mathrm{A}$ и сигнатура $\Omega$ операций над элементами основы алгебры. С помощью операции 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* и противоречия. Алгебра, которой удовлетворяют перечисленные операции, называется булевой.

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




увеличить изображение

**Рис. 5.13.** Схема проектирования программ в алгебре алгоритмики

Основные объекты алгебры алгоритимики – 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 $\mathrm{A}$ и $\mathrm{B}$ записывается в виде композиции $\mathrm{A} * \mathrm{B}$ ; альтернативное выполнение операторов $\mathrm{A}$ и $\mathrm{B}$ ( $u(\mathrm{A},\mathrm{B})$ ) означает, если $u$ истинно, то выполняется $\mathrm{A}$,иначе $\mathrm{B}$ ; цикл $(u(\mathrm{A},\mathrm{B}))$ выполняется, пока не станет истинным условие $u$ ( $u$ -логическая переменная).

С помощью этих элементарных конструкций строится более сложная схема $\Pi$ алгоритма:

$$\Pi ::= \{[u_1]\mathrm{A}_1\},$$
$$\mathrm{A}_1 ::= \{[u_2]\mathrm{A}_2 * D\}$$
$$\mathrm{A}_2 ::= \mathrm{A}_3 * C,$$
$$\mathrm{A}_3 ::= \{[u]\mathrm{A}, B\},$$
$$u ::= u_2 \wedge u_1.$$

Проведя суперпозицию путем свертки данной схемы алгоритма $\Pi$, получаем формулу:

$$\Pi ::= \{[u_1]\{[u_2]([u_2 \wedge u_1]\mathrm{A}, \mathrm{B}) * \mathrm{C}\} * D\}.$$

Важный результат в области алгоритмики – проведение сопоставительного анализа аппарата алгебры алгоритмики с известными алгебрами. Дадим краткую характеристику.

**Алгебра Дейкстры АД** = $\{\mathrm{ACC}, L(2), \mathrm{СИГН}\}$ – двухосновная алгебра, элементами которой являются множество $\mathrm{ACC}$ операторов, представленных структурными блок-схемами, множество $L(2)$ булевых функций в сигнатуре $\mathrm{СИГН}$, в которую входят операции дизъюнкции, конъюнкции и отрицания, принимающие значения из $L(2)$. С помощью специально разработанных механизмов преобразования АД в алгебру алгоритмики установлена связь между альтернативой и циклом, т.е. $\{[u]\mathrm{A}\} = ([u]\mathrm{E}, \mathrm{A} * \{[u]\mathrm{A}\})$, произвольные операторы представлены суперпозицией основных операций и констант.

Операция фильтрации $\Phi(u) = \{[u]\mathrm{E}, N\}$ в АД представлена суперпозицией тождественного $\mathrm{E}$ и неопределенного $N$ операторов 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** $\mathrm{АЯ} =< \{AHC, L2)\}; CИГН >$, где $\mathrm{АНС}$ -совокупность неструктурных схем, $L(2)$ – совокупность различных булевских функций, $\mathrm{СИГН}$ – сигнатура из композиции $\mathrm{A} * \mathrm{B}$ и операция неструктурного перехода $\Pi(u, F)$, а также операции дизъюнкции, конъюнкции и отрицания. $\mathrm{АЯ}$ включает операции построения неструктурных логических схем программ. Схема Янова состоит из *предикатных символов* множества $P(p_1, p_2, \ldots,)$, операторных символов множества $A\{\mathrm{a}_1 a_2, \ldots\}$ и графа переходов. Оператор в данной алгебре – это пара $A\{p\}$, состоящая из символов множества $\mathrm{A}$ 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 они знаками $+$ и $-$. Преобразователь имеет один преемник, а распознаватель – два. Каждый распознаватель включает в себя условие выполнения схемы. Преобразователь обрабатывает операторы, включающие логические переменные, принадлежащие множеству $(p_1, p_2, \ldots)$.

Каждая созданная схема $\mathrm{АЯ}$ отличается большой сложностью, требует серьезного преобразования при переходе к представлению программы в виде соответствующей последовательности действий, условий перехода и безусловного перехода. В работе [5.30] разработана теория интерпретации схем Янова и доказательство эквивалентности двух операторных схем исходя из особенностей алгебры алгоритмики.

Для представления схемы Янова аппаратом алгебры алгоритмики сигнатура операций $\mathrm{АЯ}$ вводятся композиции $\mathrm{A} * \mathrm{B}$ и операции условного перехода, который в зависимости от условия $u$ выполняет переход к следующим операторам или к оператору, помеченному меткой (типа $goto$). Условный переход трактуется как бинарная операция $\Pi(u, F)$, 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 $\Pi(u)$ вместо метки и булевы операции конъюнкции и отрицания. Эквивалентность выполненных операций преобразования обеспечивает правильность неструктурного представления.

**Система алгебр Глушкова** $\mathrm{АГ} = \{\mathrm{ОП}, \mathrm{УС}, \mathrm{СИГН}\}$, где $\mathrm{ОП}$ и $\mathrm{УС}$ – множества операторов суперпозиции, входящих в сигнатуру $\mathrm{СИГН}$, и логических условий, определенных на

информационном множестве $\text{ИМ}$, $\text{СИГН} = \{\text{СИГНад} \cup \text{Прогн.}\}$, где $\text{СИГНад}$ – сигнатура операций Дейкстры, $\text{Прогн}$. – операция прогнозирования. Сигнатура САА включает в себя операции алгебры АД, обобщенной трехзначной булевой операции и прогнозирования (левое умножение условия на оператор $u = (\text{A} * u')$ с порождением предиката $u = \text{УС}$ такого, что $u(m) = u'(m'),\ m' = \text{A}(m), \text{A} \in \text{ОП})$. ИМ – множество обрабатываемых данных и определение операций из множеств ОП и УС. Сущность операция прогнозирования состоит в проверке условия $u$ в состоянии $m$ оператора $\text{A}$ и определения условия $u'$, вычисленного в состоянии $m'$ после выполнения оператора $\text{A}$. 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 до алгебры алгоритмики. Этот специальный процесс отличается громоздкостью и рассматриваться не будет, при необходимости можно обратиться к [5.30].

**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 $S = \{\{D_i | i \in I\}; \text{СИГН}_0, \text{СИГН}_n\}$, где $D_i$ – основы или сорта, $\text{СИГН}_0, \text{СИГН}_n$ – 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 объектном программировании. Тем самым устанавливается связь с современными тенденциями развития современного программирования.

Практическим результатом исследований алгебры алгоритмики является построение оригинальных инструментальных систем проектирования алгоритмов и программ на основе современных средств поддержки ООП (Rational Rose). Детальное знакомство с данной темой можно продолжить, изучая приведенные ниже источники.

## Контрольные вопросы и задания

1. Дайте характеристику структурного метода программирования.
2. Приведите основные особенности и возможности объектно-ориентированного программирования.
3. Какие диаграммы имеются в языке UML для визуального проектирования программ?
4. Приведите основные типы компонентов и пути их развития в компонентном программировании.
5. Назовите базовые понятия в компонентном программировании.
6. Приведите базовые структуры в компонентном программировании.
7. Определите основные понятия и этапы жизненного цикла компонентного программирования.
8. Определите основные элементы аспектно-ориентированного программирования.
9. Определите основные элементы агентного программирования.
10. Определите понятие агента и его место в программировании.
11. Дайте характеристику инженерии ПрО.
12. Определите объекты генерирующего программирования и дайте краткую характеристику.
13. Что такое пространство проблем и пространство решений?
14. Представьте *теоретические методы* программирования.
15. Что такое алгоритмика и ее алгебра?
16. Покажите сущность перехода от других алгебр к алгебре алгоритмики.