

ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»

А.В. КУПРИЯНОВ

ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ КОМПЛЕКСОВ

*Утверждено Редакционно-издательским советом университета
в качестве учебного пособия*

САМАРА
Издательство СГАУ
2006

УДК 681.3.06, 681.3.07

ББК 32.973

K924



**Инновационная образовательная программа
"Развитие центра компетенции и подготовка
специалистов мирового уровня в области
аэрокосмических и геоинформационных технологий"**

Рецензенты: д-р техн. наук, проф. С. А. Прохоров
канд. физ.-мат. наук, доц. М. Н. Осипов

K924

Куприянов А.В.

Технологии проектирования программных комплексов: учеб. пособие / А.В. Куприянов. - Самара: Изд-во Самар. гос. аэрокосм. ун-та; 2006. - 72 с.: ил.

ISBN 5-7883-0470-9

В данном пособии рассмотрены современные методы проектирования программного обеспечения. Изложение основ инженерного проектирования программных комплексов ориентировано на студентов старших курсов и аспирантов, при этом основное внимание уделяется методологии и техническим аспектам проектирования.

Представлен широкий спектр процессов, ведущих к созданию программных комплексов. Подробно излагаются основы разработки программных средств на основе методологии быстрой разработки приложений RAD.

Также приведены требования, предъявляемые к разработке функциональной спецификации программных комплексов и средств. Приводятся способы спецификации качества программного обеспечения.

В разделах, посвященных разработке архитектуры программных средств, подробно излагаются методология функционального моделирования в рамках структурного подхода и методология объектно-ориентированного моделирования в терминах универсального языка моделирования UML.

Пособие предназначено для студентов специальности 01.05.01 и направления 01.05.00 "Прикладная математика и информатика".

УДК 681.3.06, 681.3.07

ББК 32.973

ISBN 5-7883-0470-9

© Куприянов А.В., 2006
© Самарский государственный
аэрокосмический университет, 2006

ОГЛАВЛЕНИЕ

Введение.....	4
1. Разработка программного обеспечения	5
1.1. Программы и программное обеспечение	5
1.2. Технология программирования	6
1.3. Особенности разработки программных средств	8
2. Основы разработки программных средств.	9
2.1. Жизненный цикл разработки программных средств	9
2.2. Процесс разработки	11
2.3. Быстрая разработка приложений RAD	15
3. Внешнее описание программного средства	22
3.1. Спецификация требований.....	22
3.2. Функциональная спецификация	23
3.3. Определение требований.....	25
3.4. Спецификация качества	26
3.5. Документирование функциональной спецификации	30
4. Архитектура программного средства	32
4.1. Классы архитектур ПС	32
4.2. Архитектурные функции.....	37
5. Структурный подход к проектированию архитектуры.....	38
5.1. Сущность структурного подхода.....	38
5.2. Методология функционального моделирования SADT	40
5.3. Состав функциональной модели	41
5.4. Иерархия диаграмм.....	43
5.5. Связи между функциями	46
6. Объектно-ориентированный подход к разработке архитектуры.....	51
6.1. Классы и объекты	51
6.2. Модели архитектуры	53
6.3. Диаграммы классов в UML	56
Заключение	70
Литература	71

ВВЕДЕНИЕ

Изучение языков программирования обеспечивает разработчика программных систем лишь средствами кодирования алгоритмов, представляемых в виде логических схем. Однако для разработки больших программных систем этого явно недостаточно. Необходимы и другие знания, обеспечиваемые дисциплиной технологии проектирования программных комплексов. Но этот термин не вполне точен, т.к. охватывает только часть возникающих проблем. Решением этих проблем занимаются:

- **методология программирования** или совокупность механизмов программирования, объединенных общим философским подходом (структурное, объектно-ориентированное программирование);

- **технология программирования** – это совокупность производственных процессов создания программных средств и их описания;

- **инженерия программирования** – совокупность методов и средств программирования.

1. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Программы и программное обеспечение

Целью программирования является описание процессов обработки информации или данных.

Данные – это представление фактов и идей в формализованном виде, пригодном для передачи и переработки в некоем процессе, а **информация** – это смысл, который придается данным при их представлении.

Обработка данных – это выполнение систематической последовательности действий с данными. Данные представляются и хранятся на т.н. носителях данных. Совокупность носителей данных, используемых при какой-либо обработке данных, будем называть **информационной средой**.

Набор данных, содержащихся в какой-либо момент в информационной среде, будем называть **состоянием** этой информационной среды.

Процесс можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды.

Описать процесс – значит определить последовательность состояний заданной информационной среды. Если мы хотим, чтобы по заданному описанию требуемый процесс порождался автоматически на каком-либо компьютере, необходимо, чтобы это описание было формализованным. Такое описание называется **программой**.

Программа — это данные, предназначенные для управления конкретными компонентами системы обработки информации (СОИ) в целях реализации определенного алгоритма.

Программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании часто приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на удобном для человека формализованном языке программирования, с которого она автоматически переводится на

язык соответствующего компьютера с помощью другой программы, называемой транслятором. Человеку (программисту), прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы, прояснению общей организации разрабатываемой программы и многое другое. Использование этой информации может существенно упростить задачу понимания программы человеком, поэтому весьма полезно ее как-то фиксировать в виде отдельных документов (часто не формализованных, рассчитанных только для восприятия человеком).

Обычно программы разрабатываются в расчете на то, чтобы ими могли пользоваться люди, не участвующие в их разработке (их называют пользователями). Для освоения программы пользователем помимо ее текста требуется определенная дополнительная документация.

Программа или логически связанная совокупность программ на носителях данных, снабженная программной документацией, называется **программным средством (ПС)**.

Программа позволяет осуществлять некоторую автоматическую обработку данных на компьютере. Программная документация позволяет понять, какие функции выполняет та или иная программа ПС, как подготовить исходные данные и запустить требуемую программу в процесс ее выполнения, а также: что означают получаемые результаты (или каков эффект выполнения этой программы). Кроме того, программная документация помогает разобраться в самой программе, что необходимо, например, при ее модификации. Существенно, что ПС предназначены для многократного использования и применения разными пользователями.

1.2. Технология программирования

Технология программирования – совокупность производственных процессов, приводящая к созданию требуемого ПС, а также описание этой совокупности процессов. Другими словами, техноло-

гию программирования следует понимать в широком смысле как технологию разработки программных средств, включая в нее все процессы, начиная с момента зарождения идеи этого средства, и, в частности, связанные с созданием необходимой программной документации. Каждый процесс этой совокупности базируется на использовании каких-либо методов и средств, например, компьютер (в этом случае будем говорить о компьютерной технологии программирования).

В литературе имеются и другие, несколько отличающиеся, определения технологии программирования. Используется в литературе и близкое к технологии программирования понятие программной инженерии (Software Engineering), определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала.

В технологии программирования акцент делается на изучении процессов разработки ПС (технологических процессов) и порядке их прохождения - методы и инструментальные средства разработки ПС используются в этих процессах (их применение и образует технологические процессы). Тогда как в программной инженерии изучаются, прежде всего, методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей - они могут использоваться в разных технологических процессах (и в разных технологиях программирования); как эти методы и средства образуют технологические процессы - здесь вопрос второстепенный.

Не следует также путать технологию программирования с методологией программирования. Хотя в обоих случаях изучаются методы, но в технологии программирования методы рассматриваются с точки зрения организации технологических процессов, а в методологии программирования методы рассматриваются с точки зрения основ их построения.

1.3. Особенности разработки программных средств

Разработке программных средств присущ ряд специфических особенностей. Прежде всего, следует отметить некоторое противостояние: неформальный характер требований к ПС (постановки задачи) и понятия ошибки в нем, но формализованный основной объект разработки – программы ПС. Тем самым разработка ПС содержит определенные этапы формализации, а переход от неформального к формальному существенно неформален.

Разработка ПС носит творческий характер (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий. Тем самым эта разработка ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.

Следует отметить также особенность продукта разработки. Он представляет собой некоторую совокупность текстов (т.е. статических объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является динамическим). Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.

Продукт разработки имеет и другую специфическую особенность: ПС при своем использовании (эксплуатации) не расходуются и не расходует используемых ресурсов.

2. ОСНОВЫ РАЗРАБОТКИ ПРОГРАММНЫХ СРЕДСТВ

2.1. Жизненный цикл разработки программных средств

Под жизненным циклом ПС понимают весь период его разработки и эксплуатации (использования), начиная от момента возникновения замысла ПС и кончая прекращением всех видов его использования. Жизненный цикл включает все процессы создания и использования ПС.

Различают следующие стадии жизненного цикла ПС (рис. 2.1): разработку ПС, производство программных изделий (ПИ) и эксплуатацию ПС.

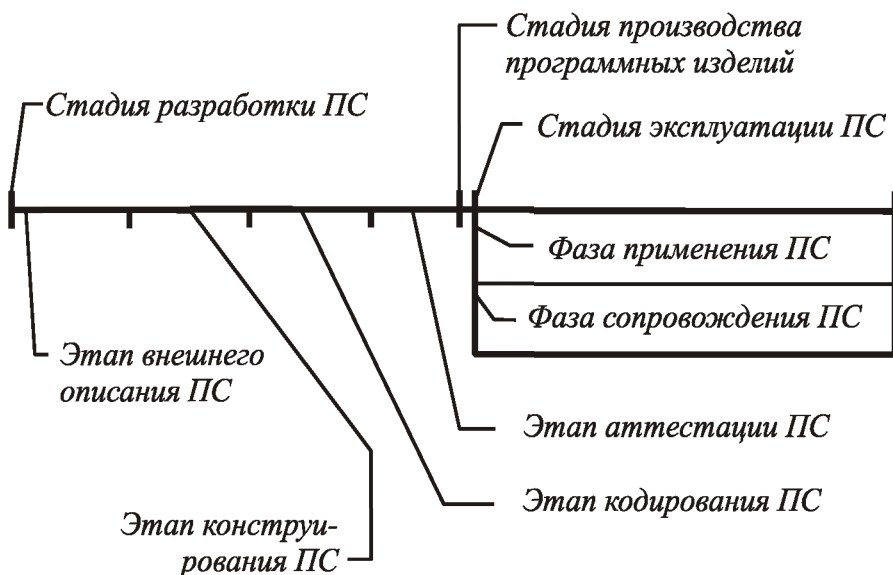


Рис. 2.1. Стадии и фазы жизненного цикла ПС

Стадия разработки (development) ПС состоит из этапа его внешнего описания, этапа конструирования ПС, этапа кодирования (программирование в узком смысле) ПС и этапа аттестации ПС. Всем этим этапам сопутствуют процессы документирования и управление (management) разработкой ПС. Этапы конструирования и кодирования часто перекрываются, иногда довольно сильно. Это означает, что

кодирование некоторых частей программного средства может быть начато до завершения этапа конструирования.

Внешнее описание (Requirements document) ПС является описанием его поведения с точки зрения внешнего по отношению к нему наблюдателя с фиксацией требований относительно его качества. Внешнее описание ПС начинается с определения требований к ПС со стороны пользователей (заказчика).

Конструирование (design) ПС охватывает процессы: разработку архитектуры ПС, разработку структур программ ПС и их детальную спецификацию.

Кодирование (coding) - создание текстов программ на языках программирования, их отладка с тестированием ПС.

На этапе **аттестации** ПС производится оценка качества ПС, после успешного завершения которого разработка ПС считается законченной.

Программное изделие (ПИ) – экземпляр или копия, снятая с разработанного ПС. **Изготовление ПИ** – это процесс генерации и/или воспроизведения (снятия копии) программ и программных документов ПС с целью их поставки пользователю для применения по назначению. Производство ПИ - это совокупность работ по обеспечению изготовления требуемого количества ПИ в установленные сроки [1]. Стадия производства ПС в жизненном цикле ПС является, по существу, вырожденной (не существенной), так как представляет рутинную работу, которая может быть выполнена автоматически и без ошибок. Этим она принципиально отличается от стадии производства различной техники. В связи с этим в литературе эту стадию, как правило, не включают в жизненный цикл ПС.

Стадия эксплуатации ПС охватывает процессы хранения, внедрения и сопровождения ПС, а также транспортировки и применения (operation) ПИ по своему назначению. Она состоит из двух параллельно проходящих фаз: фазы применения ПС и фазы сопровождения ПС.

Применение (operation) ПС - это использование ПС для решения практических задач на компьютере путем выполнения ее программ.

Сопровождение (maintenance) ПС - это процесс сбора информации о его качестве в эксплуатации, устранения обнаруженных в нем ошибок, его доработки и модификации, а также извещения пользователей о внесенных в него изменениях.

2.2. Процесс разработки

Процесс разработки ПС предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, и т. д.

Процесс разработки включает следующие действия:

- 1) подготовительную работу;
- 2) анализ требований к системе;
- 3) проектирование архитектуры системы;
- 4) анализ требований к ПО;
- 5) проектирование архитектуры ПО;
- 6) детальное проектирование ПО;
- 7) кодирование и тестирование ПО;
- 8) интеграцию ПО;
- 9) квалификационное тестирование ПО;
- 10) интеграцию системы;
- 11) квалификационное тестирование системы;
- 12) установку ПО;
- 13) приемку ПО.

Подготовительная работа начинается с выбора модели ЖЦ ПО, соответствующей масштабу, значимости и сложности проекта. Действия

и задачи процесса разработки должны соответствовать выбранной модели. Разработчик должен выбрать, адаптировать к условиям проекта и использовать согласованные с заказчиком стандарты, методы и средства разработки, а также составить план выполнения работ.

Анализ требований к системе подразумевает определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т. д. Требования к системе оцениваются исходя из критериев реализуемости и возможности проверки при тестировании.

Проектирование архитектуры системы на высоком уровне заключается в определении компонентов ее оборудования, ПО и операций, выполняемых эксплуатирующим систему персоналом. Архитектура системы должна соответствовать требованиям, предъявляемым к системе, а также принятым проектным стандартам и методам.

Анализ требований к ПО предполагает определение следующих характеристик для каждого компонента ПО:

- функциональных возможностей, включая характеристики производительности и среды функционирования компонента;
- внешних интерфейсов;
- спецификаций надежности и безопасности;
- эргономических требований;
- требований к используемым данным;
- требований к установке и приемке;
- требований к пользовательской документации;
- требований к эксплуатации и сопровождению.

Требования к ПО оцениваются исходя из критериев соответствия требованиям к системе, реализуемости и возможности проверки при тестировании.

Проектирование архитектуры ПО включает следующие задачи (для каждого компонента ПО):

- трансформацию требований к ПО в архитектуру, определяющую на высоком уровне структуру ПО и состав его компонентов;
- разработку и документирование программных интерфейсов ПО и баз данных;
- разработку предварительной версии пользовательской документации;
- разработку и документирование предварительных требований к тестам и плана интеграции ПО.

Архитектура компонентов ПО должна соответствовать требованиям, предъявляемым к ним и принятым проектным стандартам.

Детальное проектирование ПО включает следующие задачи:

- описание компонентов ПО и интерфейсов между ними на более низком уровне, достаточном для их последующего самостоятельного кодирования и тестирования;
- разработку и документирование детального проекта базы данных; обновление (при необходимости) пользовательской документации; разработку и документирование требований к тестам и плана тестирования компонентов ПО; обновление плана интеграции ПО.

Кодирование и тестирование ПО охватывают следующие задачи:

- разработку (кодирование) и документирование каждого компонента ПО и базы данных, а также совокупности тестовых процедур и данных для их тестирования;
- тестирование каждого компонента ПО и базы данных на соответствие предъявляемым к ним требованиям. Результаты тестирования компонентов должны быть документированы;

- обновление (при необходимости) пользовательской документации;
- обновление плана интеграции ПО.

Интеграция ПО предусматривает сборку разработанных компонентов ПО в соответствии с планом интеграции и тестирование агрегированных компонентов. Для каждого из агрегированных компонентов разрабатываются наборы тестов и тестовые процедуры, предназначенные для проверки каждого из квалификационных требований при последующем квалификационном тестировании. Квалификационное требование - это набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт как соответствующий своим спецификациям и готовый к использованию в условиях эксплуатации.

Квалификационное тестирование ПО проводится разработчиком в присутствии заказчика (по возможности) для демонстрации того, что ПО удовлетворяет своим спецификациям и готово к использованию в условиях эксплуатации. Квалификационное тестирование выполняется для каждого компонента ПО по всем разделам требований при широком варьировании тестов. При этом также проверяются полнота технической и пользовательской документации и ее адекватность самим компонентам ПО.

Интеграция системы заключается в сборке всех ее компонентов, включая ПО и оборудование. После интеграции система, в свою очередь, подвергается квалификационному тестированию на соответствие совокупности требований к ней. При этом также производится оформление и проверка полного комплекта документации на систему.

Установка ПО осуществляется разработчиком в соответствии с планом в той среде и на том оборудовании, которые предусмотрены договором. В процессе установки проверяется работоспособность ПО и баз данных. Если устанавливаемое ПО заменяет существующую систему, разработчик должен обеспечить их параллельное функционирование в соответствии с договором.

Приемка ПО предусматривает оценку результатов квалификационного тестирования ПО и системы и документирование результатов оценки, которые проводятся заказчиком с помощью разработчика. Разработчик выполняет окончательную передачу ПО заказчику в соответствии с договором, обеспечивая при этом необходимое обучение и поддержку.

2.3. Быстрая разработка приложений RAD

Одним из возможных подходов к разработке прикладного ПО является получивший широкое распространение способ так называемой быстрой разработки приложений, или RAD (Rapid Application Development). Подход RAD предусматривает наличие трех составляющих:

- небольших групп разработчиков (от 3 до 7 человек), выполняющих работы по проектированию отдельных подсистем ПО. Это обусловлено требованием максимальной управляемости коллектива;
- короткого, но тщательно проработанного производственного графика (до 3 месяцев);
- повторяющегося цикла, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные в результате взаимодействия с заказчиком.

Команда разработчиков должна представлять собой группу профессионалов, имеющих опыт в проектировании, программировании и тестировании ПО, способных хорошо взаимодействовать с конечными пользователями и трансформировать их предложения в рабочие прототипы.

Жизненный цикл ПО в соответствии с подходом RAD включает четыре стадии:

- анализ и планирование требований;

- проектирование;
- реализация;
- внедрение.

На стадии анализа и планирования требований пользователи осуществляют следующие действия:

- определяют функции, которые должна выполнять система;
- выделяют наиболее приоритетные функции, требующие проработки в первую очередь;
- описывают информационные потребности. Формулирование требований к системе осуществляется в основном силами пользователей под руководством специалистов-разработчиков.

Кроме того, на данной стадии решаются следующие задачи:

- ограничивается масштаб проекта;
- устанавливаются временные рамки для каждой из последующих стадий;
- определяется сама возможность реализации проекта в заданных размерах финансирования, на имеющихся аппаратных средствах и т.п.

Результатом стадии должны быть:

- список расставленных по приоритету функций будущего ПО;
- предварительные модели ПО.

На стадии проектирования часть пользователей принимает участие в техническом проектировании системы под руководством специалистов-разработчиков. Для быстрого получения работающих прототипов приложений используются соответствующие инструментальные средства (CASE-средства). Пользователи, непосредственно взаимодействуя с разработчиками, уточняют и дополняют требования

к системе, которые не были выявлены на предыдущей стадии. На данной стадии выполняются следующие действия:

- более детально рассматриваются процессы системы;
- при необходимости для каждого элементарного процесса создается частичный прототип: экранная форма, диалог, отчет, устраняющий неясности или неоднозначности;
- устанавливаются требования разграничения доступа к данным;
- определяется состав необходимой документации.

После детального определения состава процессов **оценивается количество так называемых функциональных точек (function point)** разрабатываемой системы и принимается решение о разделении на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время.

Под функциональной точкой понимается любой из следующих элементов разрабатываемой системы:

- входной элемент приложения (входной документ или экранная форма);
- выходной элемент приложения (отчет, документ, экранная форма);
- запрос (пара "вопрос/ответ");
- логический файл (совокупность записей данных, используемых внутри приложения);
- интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).

Далее проект **распределяется между различными командами разработчиков**. Опыт разработки крупных систем показывает, что для повышения эффективности работ необходимо разбить проект на отдельные слабо связанные по данным и функциям подсистемы. Реализация подсистем должна выполняться отдельными группами

специалистов. При этом необходимо обеспечить координацию ведения общего проекта и исключить дублирование результатов работ каждой проектной группы, которое может возникнуть в силу наличия общих данных и функций. Результатом данной стадии должны быть:

- общая информационная модель системы;
- функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;
- точно определенные интерфейсы между автономно разрабатываемыми подсистемами;
- построенные прототипы экранных форм, отчетов, диалогов.

Все модели и прототипы должны быть получены с применением тех CASE-средств, которые будут использоваться в дальнейшем при построении системы. Данное требование обусловлено необходимостью избежать неконтролируемого искажения данных при передаче информации о проекте со стадии на стадию.

В отличие от имевшего место ранее подхода, при котором использовались специфические средства прототипирования, не предназначенные для построения реальных приложений, а прототипы выбрасывались после того, как выполняли задачу устранения неясностей в проекте, в подходе RAD каждый прототип развивается в часть будущей системы. Таким образом, на следующую стадию передается более полная и полезная информация.

На стадии реализации выполняется непосредственно сама быстрая разработка приложения:

- разработчики производят итеративное построение реальной системы на основе полученных на предыдущей стадии моделей, а также требований нефункционального характера (требований к надежности, производительности и т.п.);
- пользователи оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает

удовлетворять определенным ранее требованиям. Тестирование системы осуществляется в процессе разработки.

После окончания работ **каждой отдельной команды разработчиков производится постепенная интеграция данной части системы с остальными**, формируется полный программный код, выполняется тестирование совместной работы данной части приложения, а затем тестирование системы в целом. Реализация системы завершается выполнением следующих работ:

- осуществляется анализ использования данных и определяется необходимость их распределения;
- производится физическое проектирование базы данных;
- формулируются требования к аппаратным ресурсам;
- устанавливаются способы увеличения производительности;
- завершается разработка документации проекта.

Результатом стадии является готовая система, удовлетворяющая всем согласованным требованиям.

На стадии внедрения производятся обучение пользователей, организационные изменения и параллельно с внедрением новой системы продолжается эксплуатация существующей системы (до полного внедрения новой). Так как стадия реализации достаточно непродолжительна, планирование и подготовка к внедрению должны начинаться заранее, как правило на стадии проектирования системы.

Приведенная схема разработки программных комплексов не является абсолютной. Возможны различные варианты, зависящие, например, от начальных условий, в которых ведется разработка:

- разрабатывается совершенно новая система;
- существует модель ее деятельности в организации;
- в организации уже существует некоторая компьютерная система, которая может быть использована в качестве

начального прототипа или должна быть интегрирована с разрабатываемой системой.

Следует, однако, отметить, что подход RAD, как и любой другой, не может претендовать на универсальность. Он хорош в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика. Если же разрабатывается крупномасштабная система (например, масштаба отрасли), которая не является законченным продуктом, а представляет собой комплекс программных компонентов, адаптируемых к программно-аппаратным платформам, системам управления базами данных (СУБД), средствам телекоммуникации, организационно-экономическим особенностям объектов внедрения и интегрируемых с существующими разработками, то на первый план выступают такие показатели проекта, как управляемость и качество, которые могут войти в противоречие с простотой и скоростью разработки. Для таких проектов необходимы высокий уровень планирования и жесткая дисциплина проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

Подход RAD не применим для построения сложных расчетных программ, операционных систем или программ управления сложными объектами в реальном масштабе времени, т.е. программ, содержащих большой объем (сотни тысяч строк) уникального кода.

Не годится подход RAD и для приложений, в которых отсутствует ярко выраженная интерфейсная часть, наглядно определяющая логику работы системы (например, приложений реального времени), и приложений, от которых зависит безопасность людей (например, управление самолетом или атомной электростанцией), так как итеративный подход предполагает, что первые несколько версий наверняка не будут полностью работоспособны, что в данном случае исключается.

Оценка размера приложений производится на основе совокупности функциональных точек. Подобная метрика не зависит от языка программирования, на котором ведется разработка. Ориентировочный

состав команды разработчиков приложения, которое может быть выполнено на основе подхода RAD, для хорошо отлаженной среды разработки программных комплексов с максимальным повторным использованием программных компонентов определяется следующим образом:

- менее 1 тыс. функциональных точек — один человек;
- от 1 до 4 тыс. функциональных точек — одна команда разработчиков;
- более 4 тыс. функциональных точек — одна команда разработчиков на 4 тыс. функциональных точек.

Итак, перечислим основные принципы подхода RAD:

- разработка приложений итерациями;
- необязательность полного завершения работ на каждой стадии ЖЦПО;
- обязательность вовлечения пользователей в процесс разработки программных комплексов;
- целесообразность применения CASE-средств, обеспечивающих целостность проекта и генерацию кода приложений;
- целесообразность применения средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности пользователей;
- тестирование и развитие проекта, осуществляемые одновременно с разработкой;
- ведение разработки немногочисленной хорошо управляемой командой профессионалов;
- грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

3. ВНЕШНЕЕ ОПИСАНИЕ ПРОГРАММНОГО СРЕДСТВА

3.1. Спецификация требований

Разработчикам больших программных средств приходится решать весьма специфические и трудные проблемы, особенно, если это ПС должно представлять собой программную систему нового типа, в плохо компьютеризированной предметной области. Разработка ПС начинается с этапа формулирования требований к ПС, на котором, исходя из довольно смутных пожеланий заказчика, должен быть получен документ, достаточно точно определяющий задачи разработчиков ПС. Этот документ мы называем **внешним описанием ПС** (в литературе его часто называют **спецификацией требований**).

Очень часто требования к ПС путают с требованиями к процессам его разработки (к технологическим процессам). Последние включать во внешнее описание не следует, если только они не связаны с оценкой качества ПС. В случае необходимости требования к технологическим процессам можно оформить в виде самостоятельного документа, который будет использоваться при управлении (руководстве) разработкой ПС.

Внешнее описание ПС играет роль точной постановки задачи, решение которой должно обеспечить разрабатываемое ПС. Более того, оно должно содержать всю информацию, которую необходимо знать пользователю для применения ПС. Оно является исходным документом для трех параллельно протекающих процессов: разработки текстов и кодирования программ, входящих в ПС, разработки документации по применению ПС и разработки существенной части комплекта тестов для тестирования ПС. Ошибки и неточности во внешнем описании, в конечном счете, трансформируются в ошибки самой ПС и обходятся особенно дорого, во-первых, потому, что они делаются на самом раннем этапе разработки ПС, и, во-вторых, потому, что они распространяются на три параллельных процесса. Это требует особенно серьезных мер по их предупреждению.

Исходным документом для разработки внешнего описания ПС являются определение требований к ПС. Но так как через этот документ передается от заказчика (пользователя) к разработчику основная информация относительно требуемого ПС, то формирование этого документа представляет собой довольно длительный и трудный итерационный процесс взаимодействия между заказчиком и разработчиком, с которого и начинается этап разработки требований к ПС. Трудности, возникающие в этом процессе, связаны с тем, что пользователи часто плохо представляют, что им на самом деле нужно: использование компьютера в "узких" местах деятельности пользователей может на самом деле потребовать принципиального изменения всей технологии этой деятельности (о чем пользователи, как правило, и не догадываются). Кроме того, проблемы, которые необходимо отразить в определении требований, могут не иметь определенной формулировки [2,3], что приводит к постепенному изменению понимания разработчиками этих проблем. В связи с этим определению требований часто предшествует процесс системного анализа, в котором выясняется, насколько целесообразно и реализуемо "заказываемое" ПС, как повлияет такое ПС на деятельность пользователей и какими особенностями оно должно обладать. Иногда для прояснения действительных потребностей пользователей приходится разрабатывать упрощенную версию требуемого ПС, называемую прототипом ПС. Анализ "пробного" применения прототипа позволяет иногда существенно уточнить требования к ПС.

3.2. Функциональная спецификация

В определении внешнего описания легко бросаются в глаза две самостоятельные его части. Описание поведения ПС определяет функции, которые должно выполнять ПС, и потому его называют функциональной спецификацией ПС. Функциональная спецификация определяет допустимые фрагменты программ, реализующих декларированные функции. Требования к качеству ПС должны быть сформулированы так, чтобы разработчику были ясны цели [4], которые он

должен стремиться достигнуть при разработке этого ПС. Эту часть внешнего описания будем называть спецификацией качества ПС (в литературе ее часто называют нефункциональной спецификацией, но она, как правило, включает и требования к технологическим процессам). Она, в отличие от функциональной спецификации, реализуется неформализованно и играет роль тех ориентиров, которые в значительной степени определяют выбор подходящих альтернатив при реализации функций ПС, а также определяет стиль всех документов и программ разрабатываемого ПС. Тем самым спецификация качества играет решающую роль в обеспечении требуемого качества ПС.

Обычно разработка спецификации качества предшествует разработке функциональной спецификации ПС, так как некоторые требования к качеству ПС могут предопределять включение в функциональную спецификацию специальных функций, например, функции защиты от несанкционированного доступа к некоторым объектам информационной среды. Таким образом, структуру внешнего описания ПС можно выразить формулой:

$$\text{Внешнее описание ПС} = \text{спецификация качества ПС} + \\ + \text{функциональная спецификация ПС}$$

Таким образом, внешнее описание определяет, что должно делать ПС и какими внешними свойствами оно должно обладать. Оно не отвечает на вопрос, как должно быть устроено это ПС и как обеспечить требуемые его внешние свойства. Оно должно достаточно точно и полно определять задачи, которые должны решить разработчики требуемого ПС. В то же время оно должно быть понято представителем пользователем - на его основании заказчиком достаточно часто принимается окончательное решение на заключение договора на разработку ПС. Внешнее описание играет большую роль в обеспечении требуемого качества ПС, так как спецификация качества ставит для разработчиков ПС конкретные ориентиры, управляющие выбором приемлемых решений при реализации специфицированных функций.

3.3. Определение требований

Определение требований к ПС является исходным документом разработки ПС – заданием, выражающем в абстрактной форме потребности пользователя. Они в общих чертах определяют замысел ПС, характеризуют условия его использования. Неправильное понимание потребностей пользователя трансформируется в ошибки внешнего описания. Поэтому разработка ПС начинается с создания документа, достаточно полно характеризующего потребности пользователя и позволяющего разработчику адекватно воспринимать эти потребности.

Определение требований представляет собой смесь фрагментов на естественном языке, различных таблиц и диаграмм. Такая смесь должна быть понятной пользователю, не знающему специальных программистских обозначений. Обычно в определении требований не содержится формализованных фрагментов, но в случае, если пользователи достаточно подготовлены (например, математически) - формализация этих требований составляет содержание дальнейшей работы коллектива разработчиков.

Неправильное понимание требований заказчиком, пользователями и разработчиками связано обычно с различными взглядами на роль требуемого ПС в среде его использования. Поэтому важной задачей при создании определения требований является установление контекста использования ПС, включающего связи между этим ПС, аппаратурой и людьми. Лучше всего этот контекст в определении требований представить в графической форме (в виде диаграмм) с добавлением описаний сущностей используемых объектов (блоков ПС, компонент аппаратуры, персонала и т.п.) и характеристики связей между ними.

Известны три способа определения требований к ПС:

- управляемый пользователем,
- контролируемый пользователем,
- не зависимый от пользователя.

В управляемой пользователем разработке определения требований к ПС определяются заказчиком, представляющим организацию

пользователей. Это происходит обычно в тех случаях, когда организация пользователей (заказчик) заключает договор на разработку требуемого ПС с коллективом разработчиков и требования к ПС являются частью этого договора. Роль разработчика ПС в создании требований заключается, в основном, в выяснении того, насколько понятны ему эти требования, и соответствующей критике рассматриваемого документа. Это может приводить к созданию нескольких редакций этого документа в процессе заключения указанного договора.

В контролируемой пользователем разработке требования к ПС формулируются разработчиком при участии представителя пользователей. Роль пользователя в этом случае сводится к информированию разработчика о своих потребностях в ПС и контролю за тем, чтобы формулируемые требования действительно выражали его потребности в ПС. В конечном счете разработанные требования, как правило, утверждаются представителем пользователя.

В независимой от пользователя разработке требования к ПС определяются без какого-либо участия пользователя (на полную ответственность разработчика). Это происходит обычно тогда, когда разработчик решает создать ПС широкого применения в расчете на то, что разработанное им ПС найдет спрос на рынке программных средств.

С точки зрения обеспечения надежности ПС наиболее предпочтительной является контролируемая пользователем разработка.

3.4. Спецификация качества

Разработка спецификации качества сводится, по-существу, к построению своеобразной модели качества разрабатываемой ПС. В этой модели должен быть перечень всех тех достаточно элементарных свойств, которые требуется обеспечить в разрабатываемом ПС и которые в совокупности образуют приемлемое для пользователя качество ПС. При этом каждое из этих свойств должно быть в достаточной степени конкретизировано с учетом определения требований к разрабатываемому ПС и возможности оценки его наличия у разработанного ПС или необходимой степени обладания им этим ПС.

Для конкретизации качества ПС по каждому из критериев используется стандартизованный набор достаточно простых свойств ПС, однозначно интерпретируемых разработчиками. Такие свойства мы будем называть примитивами качества ПС. Некоторые из примитивов могут использоваться по нескольким критериям. Ниже приводится зависимость критериев качества от примитивов качества ПС.

- *Функциональность*: завершенность.
- *Надежность*: завершенность, точность, автономность, устойчивость, защищенность.
- *Легкость применения*: П-документированность, информативность (только применительно к документации по применению), коммуникабельность, устойчивость, защищенность.
- *Эффективность*: временная эффективность, эффективность по памяти, эффективность по устройствам.
- *Сопровождаемость*: С данным критерием связано много различных примитивов качества. Однако их можно разделить по двум группам, выделив два подкритерия качества: изучаемость и модифицируемость. Изучаемость - это характеристики ПС, которые позволяют минимизировать усилия по изучению и пониманию программ и документации ПС. Модифицируемость - это характеристики ПС, которые упрощают внесение в него необходимых изменений и доработок.
- *Изучаемость*: С-документированность, информативность (здесь применительно и к документации по сопровождению), понятность, структурированность, удобочитаемость.
- *Модифицируемость*: расширяемость, структурированность, модульность.
- *Мобильность*: независимость от устройств, автономность, структурированность, модульность.

Ниже даются определения используемых примитивов качества ПС.

- Завершенность - свойство, характеризующее степень обладания ПС всеми необходимыми частями и чертами, требующимися для выполнения своих явных и неявных функций.
- Точность - мера, характеризующая приемлемость величины погрешности в выдаваемых программами ПС результатах с точки зрения предполагаемого их использования.
- Автономность - свойство, характеризующее способность ПС выполнять предписанные функции без помощи или поддержки других компонент программного обеспечения.
- Устойчивость - свойство, характеризующее способность ПС продолжать корректное функционирование, несмотря на задание неправильных (ошибочных) входных данных.
- Защищенность - свойство, характеризующее способность ПС противостоять преднамеренным или нечаянным деструктивным (разрушающим) действиям пользователя.
- П-документированность - свойство, характеризующее наличие, полноту, понятность, доступность и наглядность учебной, инструктивной и справочной документации, необходимой для применения ПС.
- Информативность - свойство, характеризующее наличие в составе ПС информации, необходимой и достаточной для понимания назначения ПС, принятых предположений, существующих ограничений, входных данных и результатов работы отдельных компонент, а также текущего состояния программ в процессе их функционирования.
- Коммуникабельность - свойство, характеризующее степень, в которой ПС облегчает задание или описание входных данных, а также обеспечивает выдачу полезных сведений в форме и с содержанием, простыми для понимания.
- Временная эффективность - мера, характеризующая способность ПС выполнять возложенные на него функции за определенный отрезок времени.

- Эффективность по памяти - мера, характеризующая способность ПС выполнять возложенные на него функции при определенных ограничениях на используемую память.
- Эффективность по устройствам - мера, характеризующая экономичность использования устройств машины для решения поставленной задачи.
- С-документированность - свойство, характеризующее с точки зрения наличия документации, отражающей требования к ПС и результаты различных этапов разработки данной ПС, включающие возможности, ограничения и другие черты ПС, а также их обоснование.
- Понятность - свойство, характеризующее степень, в которой ПС позволяет изучающему его лицу понять его назначение, сделанные допущения и ограничения, входные данные и результаты работы его программ, тексты этих программ и состояние их реализации. Этот примитив качества синтезирован нами из таких примитивов ИСО [6], как согласованность, самодокументированность, четкость и, собственно, понятность.
- Структурированность - свойство, характеризующее программу ПС с точки зрения организации взаимосвязанных их частей в единое целое определенным образом (например, в соответствии с принципами структурного программирования).
- Удобочитаемость - свойство, характеризующее легкость восприятия текста программ ПС (отступы, фрагментация, формативность).
- Расширяемость - свойство, характеризующее способность ПС к использованию большего объема памяти для хранения данных или расширению функциональных возможностей отдельных компонент.

- Модульность - свойство, характеризующее ПС с точки зрения организации его программ из таких дискретных компонент, что изменение одной из них оказывает минимальное воздействие на другие компоненты.
- Независимость от устройств - свойство, характеризующее способность ПС работать на разнообразном аппаратном обеспечении (различных типах, марках, моделях ЭВМ).

3.5. Документирование функциональной спецификации

С учетом назначения функциональной спецификации ПС и тяжелых последствий неточностей и ошибок в этом документе функциональная спецификация должна быть математически точной. Это не означает, что она должна быть формализована настолько, что по ней можно было бы автоматически генерировать программы, решающие поставленную задачу. А означает лишь, что она должна базироваться на понятиях, построенных как математические объекты, и утверждениях, однозначно понимаемых разработчиками ПС. Достаточно часто функциональная спецификация формулируется на естественном языке. Тем не менее, использование математических методов и формализованных языков при разработке функциональной спецификации весьма желательно.

Функциональная спецификация состоит из трех частей:

- описание внешней информационной среды, к которой должны применяться программы разрабатываемой ПС;
- определение функций ПС, определенных на множестве состояний этой информационной среды (такие функции будем называть внешними функциями ПС);
- описание нежелательных (исключительных) ситуаций, которые могут возникнуть при выполнении программ ПС, и реакций на эти ситуации, которые должны обеспечить соответствующие программы.

В первой части должны быть определены на концептуальном уровне все используемые каналы ввода и вывода и все информационные объекты, к которым будет применяться разрабатываемое ПС, а также существенные связи между этими информационными объектами. Примером описания информационной среды может быть концептуальная схема базы данных или описание сети датчиков и приборов, которой должна управлять разрабатываемая ПС.

Во второй части вводятся обозначения всех определяемых функций, специфицируются все входные данные и результаты выполнения каждой определяемой функции, включая указание их типов и заданий всех соотношений (или ограничений), которым должны удовлетворять эти данные и результаты. И наконец, определяется семантика каждой из этих функций, что является наиболее трудной задачей функциональной спецификации ПС. Обычно эта семантика описывается неформально на естественном языке - примерно так, как это делается при описании семантики многих языков программирования. Эта задача может быть в ряде случаев существенно облегчена при достаточно четком описании внешней информационной среды, если внешние функции задают какие-либо манипуляции с ее объектами.

В третьей части должны быть перечислены все существенные с точки зрения внешнего наблюдателя (пользователя) случаи, когда ПС не сможет нормально выполнить ту или иную свою функцию (например, при обнаружении ошибки во время взаимодействия с пользователем, или при попытке применить какую-либо функцию к данным, не удовлетворяющим соотношениям, указанным в ее спецификации, или при получении результата, нарушающего заданное ограничение). Для каждого такого случая должна быть определена (описана) реакция ПС.

4. АРХИТЕКТУРА ПРОГРАММНОГО СРЕДСТВА

4.1. Классы архитектур ПС

Архитектура ПС - это его строение как оно видно (или должно быть видно) извне, т.е. представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем. В качестве таких подсистем выступают обычно отдельные программы. Разработка архитектуры является первым этапом борьбы со сложностью ПС, на котором реализуется принцип выделения относительно независимых компонент.

Основные задачи разработки архитектуры ПС:

- выделение программных подсистем и отображение на них внешних функций (заданных во внешнем описании) ПС;
- определение способов взаимодействия между выделенными программными подсистемами.

С учетом принимаемых на этом этапе решений производится дальнейшая конкретизация и функциональных спецификаций.

Различают следующие основные классы архитектур программных средств:

- цельная программа;
- комплекс автономно выполняемых программ;
- слоистая программная система;
- коллектив параллельно выполняемых программ.

Цельная программа представляет вырожденный случай архитектуры ПС: в состав ПС входит только одна программа. Такую архитектуру выбирают обычно в том случае, когда ПС должно выполнять одну какую-либо ярко выраженную функцию и ее реализация не представляется слишком сложной. Естественно, что такая архитектура

не требует какого-либо описания (кроме фиксации класса архитектуры), так как отображение внешних функций на эту программу тривиально, а определять способ взаимодействия не требуется (в силу отсутствия какого-либо внешнего взаимодействия программы, кроме как взаимодействия ее с пользователем, а последнее описывается в документации по применению ПС).

Комплекс автономно выполняемых программ состоит из набора программ, такого, что:

- любая из этих программ может быть активизирована (запущена) пользователем;
- при выполнении активизированной программы другие программы этого набора не могут быть активизированы до тех пор, пока не закончит выполнение активизированная программа;
- все программы этого набора применяются к одной и той же информационной среде.

Таким образом, программы этого набора по управлению никак не взаимодействуют - взаимодействие между ними осуществляется только через общую информационную среду.

Слоистая программная система состоит из некоторой упорядоченной совокупности программных подсистем, называемых слоями, такой, что:

- на каждом слое ничего не известно о свойствах (и даже существовании) последующих (более высоких) слоев;
- каждый слой может взаимодействовать по управлению (обращаться к компонентам) с непосредственно предшествующим (более низким) слоем через заранее определенный интерфейс, ничего не зная о внутреннем строении всех предшествующих слоев;

- каждый слой располагает определенными ресурсами, которые он либо скрывает от других слоев, либо предоставляет непосредственно последующему слою (через указанный интерфейс) некоторые их абстракции.

Таким образом, в слоистой программной системе каждый слой может реализовать некоторую абстракцию данных. Связи между слоями ограничены передачей значений параметров обращения каждого слоя к смежному снизу слою и выдачей результатов этого обращения от нижнего слоя верхнему. Недопустимо использование глобальных данных несколькими слоями.

Коллектив параллельно действующих программ представляет собой набор программ, способных взаимодействовать между собой, находясь одновременно в стадии выполнения. Это означает, что такие программы, во-первых, вызваны в оперативную память, активизированы и могут попеременно разделять по времени один или несколько центральных процессоров, а во-вторых, осуществлять между собой динамические (в процессе выполнения) взаимодействия, на базе которых производится их синхронизация. Обычно взаимодействие между такими процессами производится путем передачи друг другу некоторых сообщений.

Простейшей разновидностью такой архитектуры является конвейер. Конвейер представляет собой последовательность программ, в которой стандартный вывод каждой программы, кроме самой последней, связан со стандартным вводом следующей программы этой последовательности (рис. 4.1).



Рис. 4.1. Конвейер одновременно действующих программ

Конвейер обрабатывает некоторый поток сообщений. Каждое сообщение этого потока поступает на ввод первой программе, которая,

обработав его, передает переработанное сообщение следующей программе, а сама начинает обработку очередного сообщения потока. Таким же образом действует каждая программа конвейера: получив сообщение от предшествующей программы и обработав его, она передает переработанное сообщение следующей программе, а последняя программа конвейера выводит результат работы всего конвейера (результатирующее сообщение). Таким образом, в конвейере, состоящем из n программ, может одновременно находиться в обработке до n сообщений. Конечно, в силу того, что разные программы конвейера могут затратить на обработку очередных сообщений разные отрезки времени, необходимо обеспечить каким-либо образом синхронизацию этих процессов (некоторые процессы могут находиться в стадии ожидания либо возможности передать переработанное сообщение, либо возможности получить очередное сообщение).

В более общем случае коллектив параллельно действующих программ может быть организован в систему с портами сообщений.

Порт сообщений представляет собой программную подсистему, обслуживающую некоторую очередь сообщений: она может принимать на хранение от программы какое-либо сообщение, ставя его в очередь, и может выдавать очередное сообщение другой программе по ее требованию. Сообщение, переданное какой-либо программой некоторому порту, уже не будет принадлежать этой программе (и использовать ее ресурсы), но оно не будет принадлежать и никакой другой программе, пока в порядке очереди не будет передано какой-либо программе по ее запросу. Таким образом, программа, передающая сообщение, не будет находиться в стадии ожидания пока программа, принимающая это сообщение, не будет готова его обрабатывать (если только не будет переполнен принимающий порт).

Пример программной системы с портами сообщений приведен на рис. 4.2. Порт U может рассматриваться как порт вводных сообщений для представленного на этом рисунке коллектива параллельно

действующих программ, а порт W - как порт выводных сообщений для этого коллектива программ.

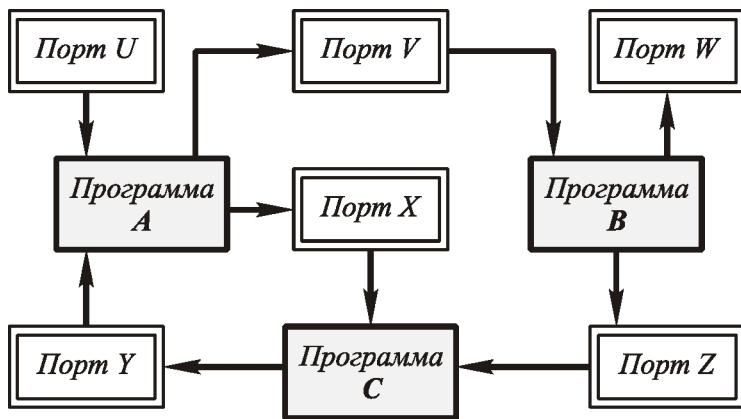


Рис. 4.2. Пример программной системы с портами сообщений

Программные системы с портами сообщений могут быть как жесткой, так и гибкой конфигурации. В системах с портами жесткой конфигурации с каждой программой могут быть жестко связаны один или несколько входных портов. Для передачи сообщения такая программа должна явно указать адрес передачи: имя программы и имя ее входного порта. В этом случае при изменении конфигурации системы придется корректировать используемые программы: изменять адреса передач сообщений. В системах с портами гибкой конфигурации с каждой программой связаны как входные, так и выходные виртуальные порты. Перед запуском такой системы должна производиться ее предварительная настройка с помощью специальной программной компоненты, осуществляющая совмещение каждого выходного виртуального порта с каким-либо входным виртуальным портом на основании информации, задаваемой пользователем. Тем самым при изменении конфигурации системы в этом случае не требуется какой-либо корректировки используемых программ - необходимые изменения должны быть отражены в информации для настройки. Однако в этом случае требуется иметь специальную программную компоненту, осуществляющую настройку системы.

4.2. Архитектурные функции

Для обеспечения взаимодействия между подсистемами в ряде случаев не требуется создавать какие-либо дополнительные программные компоненты (помимо реализации внешних функций) - для этого может быть достаточно заранее фиксированных соглашений и стандартных возможностей базового программного обеспечения (операционной системы). Так в комплексе автономно выполняемых программ для обеспечения взаимодействия достаточно описания (спецификации) общей внешней информационной среды и возможностей операционной системы для запуска программ. В слоистой программной системе может оказаться достаточным спецификации выделенных программных слоев и обычный аппарат обращения к процедурам. В программном конвейере взаимодействие между программами также может обеспечивать операционная система (как это имеет место в операционной системе UNIX).

Однако в ряде случаев для обеспечения взаимодействия между программными подсистемами может потребоваться создание дополнительных программных компонент. Так для управления работой комплекса автономно выполняемых программ часто создают специализированный командный интерпретатор, более удобный в данной предметной области для подготовки требуемой внешней информационной среды и запуска требуемой программы, чем базовый командный интерпретатор используемой операционной системы. В коллективе параллельно действующих программ для управления портами сообщений требуется специальная программная подсистема. Такие программные компоненты никаких внешних функций не выполняют - они реализуют функции, возникшие в результате разработки архитектуры программного средства. В связи с этим такие функции мы будем называть архитектурными.

5. СТРУКТУРНЫЙ ПОДХОД К ПРОЕКТИРОВАНИЮ АРХИТЕКТУРЫ

5.1. Сущность структурного подхода

Сущность структурного подхода к разработке ПС заключается в его декомпозиции (разбиении) на автоматизируемые функции: система разбивается на функциональные подсистемы, которые, в свою очередь, делятся на подфункции, подразделяемые на задачи, и так далее. Процесс разбиения продолжается вплоть до конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны. При разработке системы "снизу-вверх" от отдельных задач ко всей системе целостность теряется, возникают проблемы при информационной стыковке отдельных компонентов.

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов. В качестве двух базовых принципов используются следующие:

- принцип "разделяй и властвуй" - принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения;
- принцип иерархического упорядочивания - принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта). Основными из этих принципов являются следующие:

- принцип абстрагирования - заключается в выделении существенных аспектов системы и отвлечения от несущественных;
- принцип формализации - заключается в необходимости строгого методического подхода к решению проблемы;
- принцип непротиворечивости - заключается в обоснованности и согласованности элементов;
- принцип структурирования данных - заключается в том, что данные должны быть структурированы и иерархически организованы.

В структурном анализе используются в основном две группы средств, иллюстрирующих функции, выполняемые системой и отношения между данными. Каждой группе средств соответствуют определенные виды моделей (диаграмм), наиболее распространенными среди которых являются следующие:

- SADT (Structured Analysis and Design Technique) модели и соответствующие функциональные диаграммы;
- DFD (Data Flow Diagrams) диаграммы потоков данных;
- ERD (Entity-Relationship Diagrams) диаграммы "сущность-связь".

На стадии проектирования ИС модели расширяются, уточняются и дополняются диаграммами, отражающими структуру программного обеспечения: архитектуру ПО, структурные схемы программ и диаграммы экранных форм.

Перечисленные модели в совокупности дают полное описание ИС независимо от того, является ли она существующей или вновь разрабатываемой. Состав диаграмм в каждом конкретном случае зависит от необходимой полноты описания системы.

5.2. Методология функционального моделирования SADT

Методология SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели объекта какой-либо предметной области. Функциональная модель SADT отображает функциональную структуру объекта, т.е. производимые им действия и связи между этими действиями. Основные элементы этой методологии основываются на графическом представлении блочного моделирования и строгом выполнении правил SADT.

Графика блоков и дуг SADT-диаграммы отображает функцию в виде блока, а интерфейсы входа/выхода представляются дугами, соответственно входящими в блок и выходящими из него. Взаимодействие блоков друг с другом описывается посредством интерфейсных дуг, выражающих "ограничения", которые, в свою очередь, определяют, когда и каким образом функции выполняются и управляются. Выполнение правил SADT требует достаточной строгости и точности, не накладывая в то же время чрезмерных ограничений на действия аналитика.

Правила SADT включают:

- ограничение количества блоков на каждом уровне декомпозиции (правило 3-6 блоков);
- связность диаграмм (номера блоков);
- уникальность меток и наименований (отсутствие повторяющихся имен);
- синтаксические правила для графики (блоков и дуг);
- разделение входов и управлений (правило определения роли данных).
- отделение организации от функции, т.е. исключение влияния организационной структуры на функциональную модель.

Методология SADT может использоваться для моделирования широкого круга систем и определения требований и функций, а затем для разработки системы, которая удовлетворяет этим требованиям и реализует эти функции. Для уже существующих систем SADT может быть использована для анализа функций, выполняемых системой, а также для указания механизмов, посредством которых они осуществляются.

5.3. Состав функциональной модели

Результатом применения методологии SADT является модель, которая состоит из диаграмм, фрагментов текстов и глоссария, имеющих ссылки друг на друга. Диаграммы - главные компоненты модели, все функции ИС и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком определяет тип интерфейса. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, показана с левой стороны блока, а результаты выхода показаны с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется дугой, входящей в блок снизу (рис. 5.1).

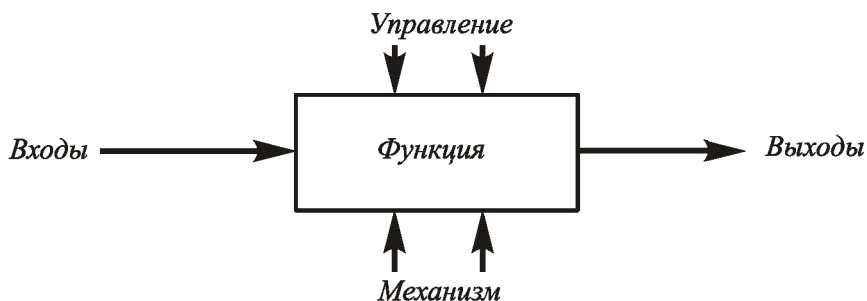


Рис. 5.1. Функциональный блок и интерфейсные дуги

Одной из наиболее важных особенностей методологии SADT является постепенное введение все больших уровней детализации по мере создания диаграмм, отображающих модель.

На рис. 5.2, где приведены четыре диаграммы и их взаимосвязи, показана структура SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждая диаграмма иллюстрирует "внутреннее строение" блока на родительской диаграмме.

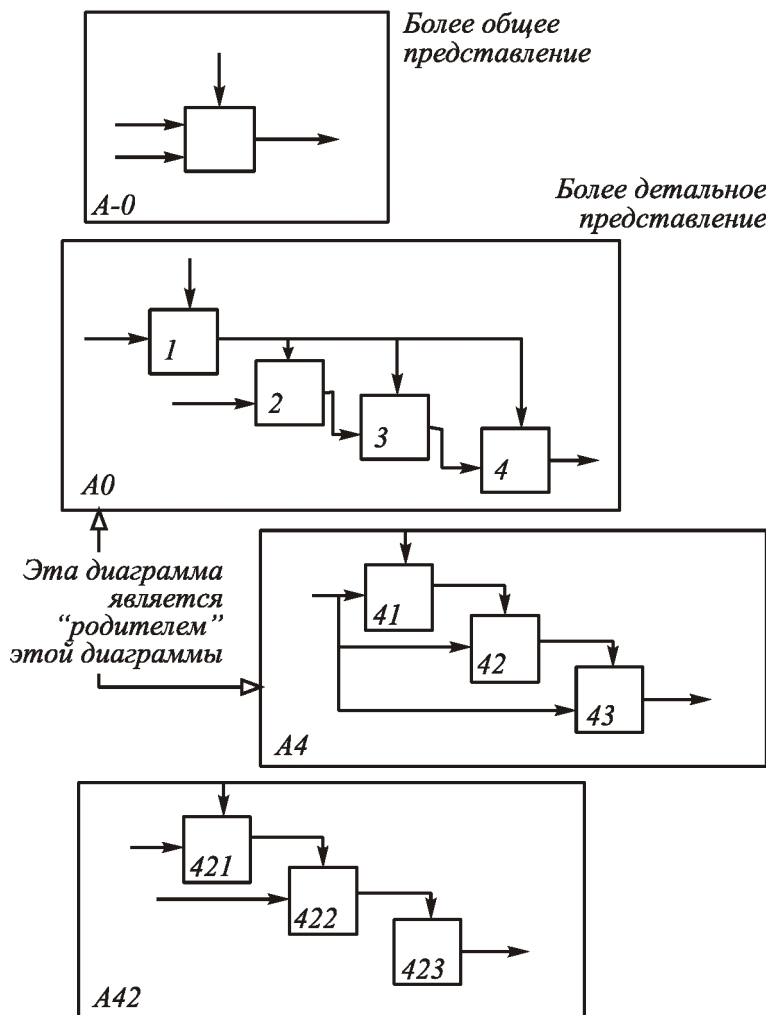


Рис.5.2. Структура SADT-модели. Декомпозиция диаграмм

Построение SADT-модели начинается с представления всей системы в виде простейшей компоненты - одного блока и дуг, изображающих интерфейсы с функциями вне системы. Поскольку единственный блок представляет всю систему как единое целое, имя, указанное в блоке, является общим. Это верно и для интерфейсных дуг - они также представляют полный набор внешних интерфейсов системы в целом.

Затем блок, который представляет систему в качестве единого модуля, детализируется на другой диаграмме с помощью нескольких блоков, соединенных интерфейсными дугами. Эти блоки представляют основные подфункции исходной функции. Данная декомпозиция выявляет полный набор подфункций, каждая из которых представлена как блок, границы которого определены интерфейсными дугами. Каждая из этих подфункций может быть декомпозирована подобным образом для более детального представления.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию. Кроме того, модель не может опустить какие-либо элементы, т.е., как уже отмечалось, родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

5.4. Иерархия диаграмм

Модель SADT представляет собой серию диаграмм с сопроводительной документацией, разбивающих сложный объект на составные части, которые представлены в виде блоков. Детали каждого из основных блоков показаны в виде блоков на других диаграммах. Каждая детальная диаграмма является декомпозицией блока из более общей диаграммы. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Дуги, входящие в блок и выходящие из него на диаграмме верхнего уровня, являются точно теми же самыми, что и дуги, входящие

в диаграмму нижнего уровня и выходящие из нее, потому что блок и диаграмма представляют одну и ту же часть системы.

На рис. 5.3 - 5.4 представлены варианты выполнения функций и соединения дуг с блоками.

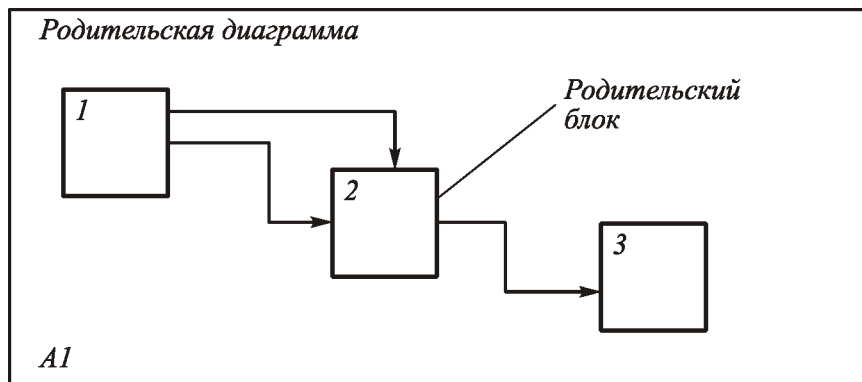


Рис. 5.3. Родительская диаграмма

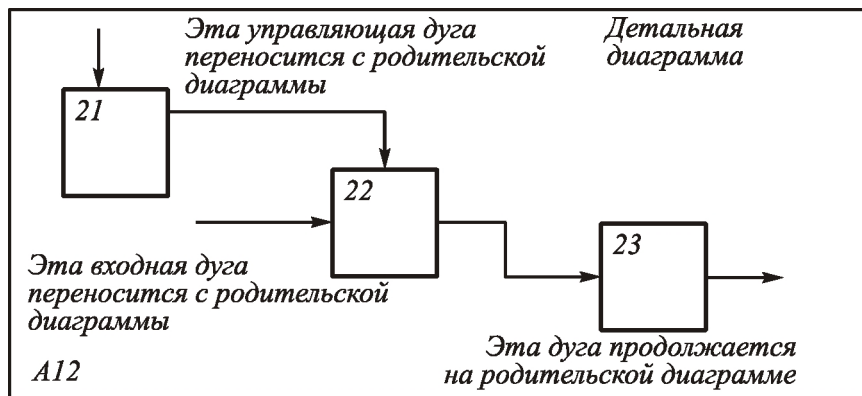


Рис. 5.4. Дочерняя диаграмма

Некоторые дуги присоединены к блокам диаграммы обоими концами, у других же один конец остается неприсоединенным. Неприсоединенные дуги соответствуют входам, управлениям и выходам родительского блока. Источник или получатель этих пограничных дуг может быть обнаружен только на родительской диаграмме. Неприсоединенные концы должны соответствовать дугам на исходной диаграмме. Все граничные дуги должны продолжаться на родительской диаграмме, чтобы она была полной и непротиворечивой.

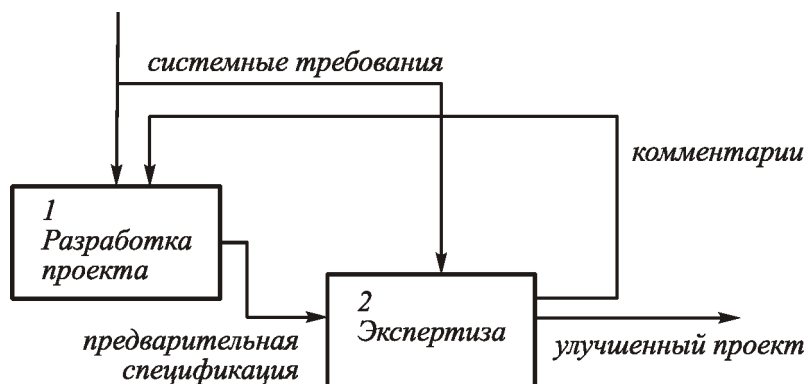


Рис. 5.5. Пример обратной связи

На SADT-диаграммах не указаны явно ни последовательность, ни время. Обратные связи, итерации, продолжающиеся процессы и перекрывающиеся (по времени) функции могут быть изображены с помощью дуг. Обратные связи могут выступать в виде комментариев, замечаний, исправлений и т.д. (рис. 5.5).

Как было отмечено, механизмы (дуги с нижней стороны) показывают средства, с помощью которых осуществляется выполнение функций. Механизм может быть человеком, компьютером или любым другим устройством, которое помогает выполнять данную функцию.

Каждый блок на диаграмме имеет свой номер. Блок любой диаграммы может быть далее описан диаграммой нижнего уровня, которая, в свою очередь, может быть далее детализирована с помощью необходимого числа диаграмм. Таким образом формируется иерархия диаграмм.

Для того, чтобы указать положение любой диаграммы или блока в иерархии, используются номера диаграмм. Например, A21 является диаграммой, которая детализирует блок 1 на диаграмме A2. Аналогично, A2 детализирует блок 2 на диаграмме A0, которая является самой верхней диаграммой модели.

5.5. Связи между функциями

Одним из важных моментов при проектировании ИС с помощью методологии SADT является точная согласованность типов связей между функциями. Различают, по крайней мере, семь типов связывания:

- Случайная
- Логическая
- Временная
- Процедурная
- Коммуникационная
- Последовательная
- Функциональная

Ниже каждый тип связи кратко определен и проиллюстрирован с помощью типичного примера из SADT.

(0) Тип случайной связности: наименее желательный.

Случайная связность возникает, когда конкретная связь между функциями мала или полностью отсутствует. Это относится к ситуации, когда имена данных на SADT-дугах в одной диаграмме имеют малую связь друг с другом. Крайний вариант этого случая показан на рис. 5.6.

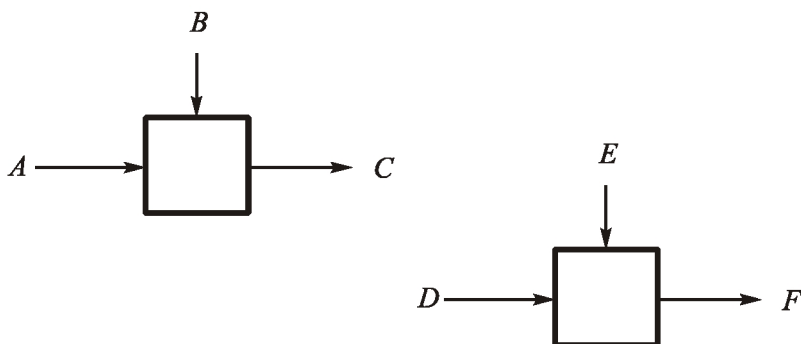


Рис. 5.6. Случайная связность

(1) Тип логической связности. Логическое связывание происходит тогда, когда данные и функции собираются вместе вследствие того, что они попадают в общий класс или набор элементов, но необходимых функциональных отношений между ними не обнаруживается.

(2) Тип временной связности. Связанные по времени элементы возникают вследствие того, что они представляют функции, связанные во времени, когда данные используются одновременно или функции включаются параллельно, а не последовательно.

(3) Тип процедурной связности. Процедурно-связанные элементы появляются сгруппированными вместе вследствие того, что они выполняются в течение одной и той же части цикла или процесса. Пример процедурно-связанной диаграммы приведен на рис. 5.7.

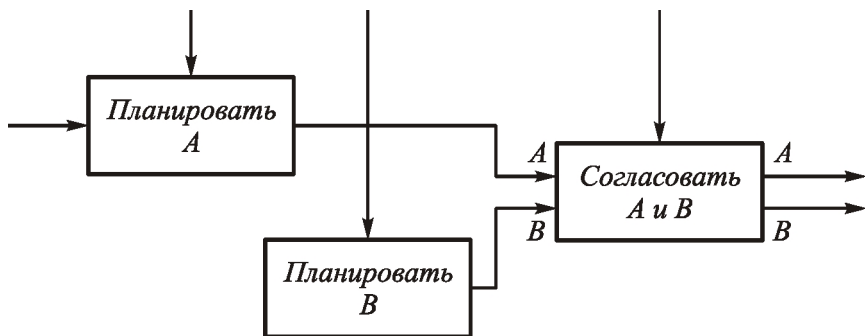


Рис. 5.7. Процедурная связность

(4) Тип коммуникационной связности. Диаграммы демонстрируют коммуникационные связи, когда блоки группируются вследствие того, что они используют одни и те же входные данные и/или производят одни и те же выходные данные (рис. 5.8).

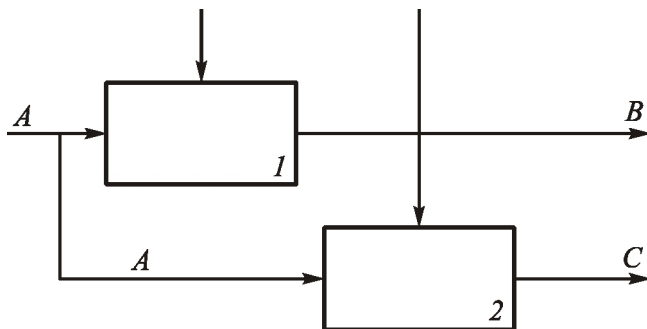


Рис. 5.8. Коммуникационная связность

(5) Тип последовательной связности. На диаграммах, имеющих последовательные связи, выход одной функции служит входными данными для следующей функции. Связь между элементами на диаграмме является более тесной, чем на рассмотренных выше уровнях связей, поскольку моделируются причинно-следственные зависимости (рис. 5.9).

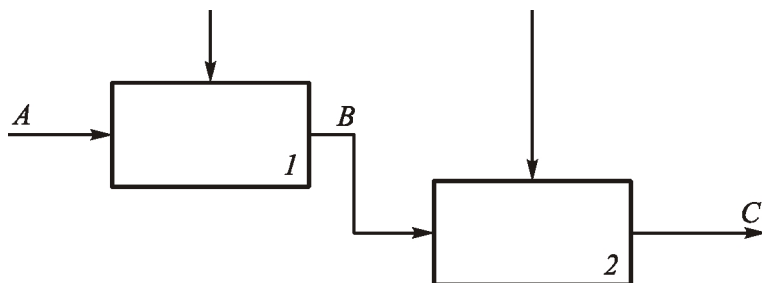


Рис. 5.9. Последовательная связность

(6) Тип функциональной связности. Диаграмма отражает полную функциональную связность при наличии полной зависимости одной функции от другой. Диаграмма, которая является чисто функциональной, не содержит чужеродных элементов, относящихся к последовательному или более слабому типу связности. Одним из способов определения функционально-связанных диаграмм является рассмотрение двух блоков, связанных через управляющие дуги, как показано на рис. 5.10.

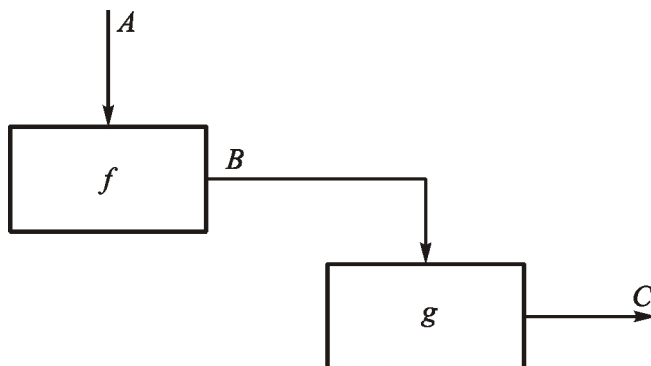


Рис. 5.10. Функциональная связность

Далее в таблице представлены все типы связей, рассмотренные выше. Важно отметить, что уровни 4-6 устанавливают типы связностей, которые разработчики считают важнейшими для получения диаграмм хорошего качества.

Значимость	Тип связности	Для функций	Для данных
0	Случайная	Случайная	Случайная
1	Логическая	Функции одного и того же множества или типа (например, "редактировать все входы")	Данные одного и того же множества или типа
2	Временная	Функции одного и того же периода времени (например, "операции инициализации")	Данные, используемые в каком-либо временном интервале
3	Процедурная	Функции, работающие в одной и той же фазе или итерации (например, "первый проход компилятора")	Данные, используемые во время одной и той же фазы или итерации
4	Коммуникационная	Функции, использующие одни и те же данные	Данные, на которые воздействует одна и та же деятельность
5	Последовательная	Функции, выполняющие последовательные преобразования одних и тех же данных	Данные, преобразуемые последовательными функциями
6	Функциональная	Функции, объединяемые для выполнения одной функции	Данные, связанные с одной функцией

6. ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД К РАЗРАБОТКЕ АРХИТЕКТУРЫ

6.1. Классы и объекты

Объектно-ориентированная архитектурная модель структурирует систему в виде совокупности слабо связанных объектов с четко определенными интерфейсами. Объекты вызывают сервисы, предоставляемые другими объектами.

На этапе объектно-ориентированной декомпозиции определяются классы объектов, их свойства и операции. При реализации системы из этих классов создаются объекты; для координации операций объектов используется какая-либо модель управления.

Преимущества объектно-ориентированного подхода хорошо известны. Поскольку объекты слабо связаны между собой, можно изменять реализацию того или иного объекта, не воздействуя на остальные. Структуру системы легко понять, так как объекты часто являются объектами реального мира. Для непосредственной реализации системных компонентов можно использовать объектно-ориентированные языки программирования.

Вместе с тем, объектно-ориентированный подход имеет и недостатки. При использовании сервисов объекты должны явно ссылаться на имена других объектов и знать их интерфейс. Если при изменении системы требуется изменить интерфейс, необходимо оценить эффект от такого изменения с учетом всех пользователей изменяемого объекта. Многие объекты реального мира сложно представить в виде системных объектов.

В общем случае следует попытаться разложить систему на части так, чтобы архитектура была как можно проще. Согласно хорошему практическому правилу модель архитектуры должна состоять не более чем из семи основных объектов.

Перед выполнением данного этапа проектирования уже должны быть сформированы представления относительно основных объектов проектируемой системы.

На следующем этапе проектирования определяются классы объектов. Структура системы описывается в терминах этих классов. Классы объектов, определенные ранее, неизбежно получают более детальное описание, поэтому иногда приходится возвращаться на данный этап проектирования для переопределения классов.

Существует множество **подходов к определению классов объектов**.

1) Использование грамматического анализа естественного языкового описания системы. Объекты и атрибуты — это существительные, операции и сервисы — глаголы. Такой подход реализован в иерархическом методе объектно-ориентированного проектирования, который широко используется в аэрокосмической промышленности Европы.

2) Использование в качестве объектов ПО событий, объектов и ситуаций реального мира из области приложения, например самолетов, ролевых ситуаций менеджера, взаимодействий, подобных интерактивному общению на научных конференциях, и т.д. Для реализации таких объектов могут потребоваться специальные структуры хранения данных (абстрактные структуры данных).

3) Применение подхода, при котором разработчик сначала полностью определяет поведение системы. Затем определяются компоненты системы, отвечающие за различные поведенческие акты (режимы работы системы), при этом основное внимание уделяется тому, кто иницирует и осуществляет данные режимы. Компоненты системы, отвечающие за основные режимы работы, считаются объектами.

4) Применение подхода, основанного на сценарии, в котором по очереди определяются и анализируются различные сценарии использования системы. Поскольку анализируется каждый сценарий, группа, отвечающая за анализ, должна идентифицировать необходимые объекты, атрибуты и операции. Метод анализа, при котором аналитики и разработчики присваивают роли объектам, показывает эффективность подхода, основанного на сценариях.

Каждый из описанных подходов помогает начать процесс определения объектов. Но для описания объектов и классов объектов необходимо использовать информацию, полученную из разных источников. Объекты и операции, первоначально определенные на основе неформального описания системы, вполне могут послужить отправной точкой при проектировании. Затем для усовершенствования и расширения описания первоначальных объектов можно использовать дополнительную информацию, полученную из области применения ПО или анализа сценариев. Дополнительную информацию также можно получить в ходе обсуждения с пользователями разрабатываемой системы или анализа имеющихся систем.

6.2. Модели архитектуры

Когда взаимодействия между проектируемой системой ПО и ее окружением определены, эти данные можно использовать как основу для разработки архитектуры системы. Конечно, при этом необходимо применять знания об общих принципах проектирования системных архитектур и данные о конкретной предметной области.

Модели системной архитектуры показывают объекты и классы объектов, составляющие систему, и при необходимости типы взаимоотношений между этими объектами. Такие модели служат мостом между требованиями к системе и ее реализацией. А это значит, что к данным моделям предъявлены противоречивые требования. Они должны быть абстрактными настолько, чтобы лишние данные не скрывали отношения между моделью архитектуры и требованиями к системе. Однако чтобы программист мог принимать решения по реализации, модель должна содержать достаточное количество информации.

Эти противоречия можно обойти, разработав несколько моделей разного уровня детализации. Там, где существуют тесные рабочие связи между разработчиками требований, проектировщиками и программистами, можно обойтись одной обобщенной моделью. В этом случае конкретные решения по архитектуре системы можно

принимать в процессе реализации системы. Когда связи между разработчиками требований, проектировщиками и программистами не такие тесные (например, если система проектируется в одном подразделении организации, а реализуется в другом), требуется более детализированная модель.

Следовательно, в процессе проектирования важно решить, какие требуются модели и какой должна быть степень их детализации. Это решение зависит также от типа разрабатываемой системы. Систему последовательной обработки данных можно спроектировать на основе встроенной системы реального времени разными способами с использованием различных моделей архитектуры. Существует множество систем, которым требуются все виды моделей. Но уменьшение числа созданных моделей сокращает расходы и время проектирования.

Существует два типа объектно-ориентированных моделей системной архитектуры.

1. **Статические модели**, которые описывают статическую структуру системы в терминах классов объектов и взаимоотношений между ними. Основными взаимоотношениями, которые документируются на данном этапе, являются отношения обобщения, отношения "используют-используются" и структурные отношения.

2. **Динамические модели**, которые описывают динамическую структуру системы и показывают взаимодействия между объектами системы (но не классами объектов). Документируемые взаимодействия содержат последовательность составленных объектами запросов к сервисам и описывают реакцию системы на взаимодействия между объектами.

В языке моделирования UML поддерживается огромное количество возможных статических и динамических моделей. Здесь рассматриваются три типа моделей.

1. Модели **подсистем**, которые показывают логически сгруппированные объекты. Они представлены с помощью диаграммы классов,

в которой каждая подсистема обозначается как пакет. Модели подсистем являются статическими.

2. Модели **последовательностей**, которые показывают последовательность взаимодействий между объектами. Они представляются в UML с помощью диаграмм последовательности или кооперативных диаграмм. Это динамические модели.

3. Модели **конечного автомата**, которые показывают изменение состояния отдельных объектов в ответ на определенные события. В UML они представлены в виде диаграмм состояния. Модели конечного автомата являются динамическими.

Другие типы моделей: модели **вариантов использования** показывают взаимодействия с системой, модели **объектов** дают описание классов объектов, модели **обобщения и наследования** показывают, какие классы являются обобщениями других классов, модель **агрегирования** выявляет взаимосвязи между коллекциями объектов.

Модель подсистем является одной из наиболее важных и полезных статических моделей, поскольку показывает, как можно организовать систему в виде логически связанных групп объектов. В UML пакеты являются структурами инкапсуляции и не отображаются непосредственно в объектах разрабатываемой системы. Однако они могут отображаться, например, в виде библиотек Java.

Совместная модель пакетов и классов объектов позволяет показать логически сгруппированные системные элементы.

Модель последовательностей — одна из наиболее полезных и наглядных динамических моделей, которая в каждом узле взаимодействия документирует последовательность происходящих между объектами взаимодействий. Опишем основные свойства модели последовательности.

Объекты, участвующие во взаимодействии, располагаются горизонтально сверху диаграммы. От каждого объекта исходит пунктирная вертикальная линия — *линия жизни объекта*.

Время направлено сверху вниз по пунктирным вертикальным линиям. Поэтому в данной модели легко увидеть последовательность операций.

Взаимодействия между объектами представлены маркированными стрелками, связывающими вертикальные линии. Это не поток данных, а представление сообщений или событий, основных в данном взаимодействии.

Тонкий прямоугольник на линии жизни объекта обозначает интервал времени, в течение которого данный объект был управляющим объектом системы. Объект берет на себя управление в верхней части прямоугольника и передает управление другому объекту внизу прямоугольника. Если в системе имеется иерархия вызовов, то управление не передается до тех пор, пока не завершится последний возврат в вызове первоначального метода.

При документировании проекта для каждого значительного взаимодействия необходимо создавать диаграмму последовательностей. Если разрабатывается модель вариантов использования, то диаграмму последовательности нужно создавать для каждого заданного варианта.

Диаграммы последовательностей обычно применяются при моделировании комбинированного поведения групп объектов, однако при желании можно также показать поведение одного объекта в ответ на обрабатываемые им сообщения.

6.3. Диаграммы классов в UML

Диаграмма классов UML – это граф, узлами которого являются элементы статической структуры проекта (классы, интерфейсы, шаблоны классов), а дугами - отношения между узлами (ассоциации,

наследование, зависимости). На диаграмме классов изображаются следующие элементы (рис. 6.1):

- *Пакет (package)* - набор элементов модели, логически связанных между собой.
- *Класс (class)* - описание общих свойств группы сходных объектов.
- *Интерфейс (interface)* - абстрактный класс, задающий набор операций, которые объект произвольного класса, связанного с данным интерфейсом, предоставляет другим объектам.
- *Утилита (utility)* - класс, объединяющий группу общедоступных (глобальных) переменных и процедур.
- *Метакласс (metaclass)* - средство для классификации классов: экземплярами метакласса являются классы. Таким образом, метакласс является элементом абстракции более высокого уровня, чем класс.
- *Объект (object)* - экземпляр класса.
- *Ассоциация (association)* - отношение между классами, являющееся способом описания взаимодействия объектов этих классов.
- *Наследование (inheritance)* - отношение между классами, с помощью которого можно в отдельный класс вынести общие свойства нескольких классов. Это отношение аналогично наследованию в объектно-ориентированных языках программирования.
- *Зависимость (dependency)* между элементами модели. Например, между классом и интерфейсом может быть установлена зависимость "uses", означающая, что класс использует интерфейс.



Рис. 6.1. Элементы объектно-ориентированных диаграмм UML

Рассмотрим теперь подробнее основные элементы диаграммы классов, используемые для описания и разработки архитектуры ПС.

Пакеты

В контексте диаграмм классов пакет - этоместилище для некоторого набора классов и других пакетов. Пакет является самостоятельным пространством имен.

В UML нет каких-либо ограничений на правила, по которым разработчики могут или должны группировать классы в пакеты. Но есть некоторые стандартные случаи, когда такая группировка уместна, например тесно взаимодействующие классы, или более общий случай - разбиение системы на подсистемы.

Отметим, что пакет физически содержит сущности, определенные в нем (говорят, что "сущности принадлежат пакету"). Это означает, что если будет уничтожен пакет, то будут уничтожены и все его содержимое.

При описании классов пакета нередко бывает полезно сослаться на класс, определенный в другом пакете. Это можно сделать, импортировав нужный пакет. Это означает, что в импортирующем пакете станут доступными все классы импортируемого пакета. При этом пространства имен не объединятся: для использования класса надо будет указать его имя с полным путем пакета, в котором он лежит.

Класс

Класс - это сущность, описывающая множество объектов со сходной структурой, поведением и связями с другими объектами.

На диаграммах класс изображается в виде прямоугольника со сплошной границей, разделенного горизонтальными линиями на 3 секции (рис. 6.2.)

Верхняя секция (секция имени) содержит имя класса и другие общие свойства (в частности, стереотип). В средней секции содержится список атрибутов (членов-данных), а в нижней - список операций (функций-членов). Атрибуты хранят инкапсулированные данные класса, а операции описывают поведение объектов класса. Другой

взгляд на поведение и данные класса - это его отношения с другими классами (ассоциации, наследование и др.).

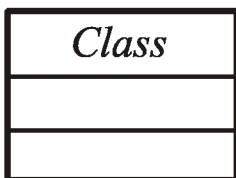


Рис. 6.2. Изображение класса на диаграмме

Любая из секций атрибутов и операций может не изображаться (а также обе сразу). Для отсутствующей секции не нужно рисовать разделительную линию и как-либо указывать на наличие или отсутствие элементов в ней.

На усмотрение конкретной реализации могут быть введены дополнительные секции.

Область видимости класса - это пакет, в котором он описан. Это означает, что имена классов должны быть уникальны среди имен классов того же пакета.

По умолчанию считается, что указываемый класс определен в текущем пакете. Если надо сослаться на класс из другого пакета, это указывается явно:

Имя_пакета::Имя_класса

Так как иерархия пакетов может иметь глубину вложенности большую, чем 1, то путь к классу может содержать более чем один пакет, при этом путь начинается от корня иерархии пакетов:

Имя_пакета1::Имя_пакета2::...::Имя_пакетаN::Имя_класса

Средняя и нижняя секции прямоугольника класса содержат списки его атрибутов и операций. Элементы этих списков можно группировать по некоторым признакам. В этом случае перед группой элементов ставится заключенная в кавычки строка, определяющая свойство,

причем это свойство распространяется на все нижестоящие элементы до нового свойства.

В одном *представлении списка* элемент списка должен присутствовать не более одного раза, хотя в разных представлениях можно задавать различную группировку.

Отметим, что место элемента в списке задается при добавлении его в список и никак не зависит от группировки элементов в представлениях. Кроме того, в некоторое представление списка какие-то его элементы могут не попасть (например, в случае установки фильтра или при ограничении количества видимых элементов). В этом случае вместо этих элементов в списке ставится три точки.

У каждой секции прямоугольника класса может быть имя. При этом, так как секция "Имя класса" обязательна, ее имя не указывается.

Атрибут

Атрибут (attribute) - это инкапсулируемый элемент данных класса, т.е. элемент данных, который содержится в объекте, принадлежащем описываемому классу.

У атрибута должен быть тип (type expression), который может представлять собой простой тип или быть сложным.

В любом случае детали, касающиеся типов атрибутов, не специфицированы UML. Единственное, что определено - это указание имени одного из классов системы. Более подробное описание типа зависит от того, какой язык программирования используется разработчиками.

Атрибут изображается в виде текстовой строки, отражающей различные его свойства:

<видимость> <имя>:<тип> = <начальное_значение> {<свойства>}

<видимость> имеет C++ семантику видимости членов класса:

- *открытый атрибут (public)*, обозначается символом «+»;
- *защищенный атрибут (protected)*, обозначается символом «#»;

- *закрытый атрибут (private)*, обозначается символом «-».

Видимость типа "открытый атрибут" означает, что любая сущность, имеющая доступ к объекту определяемого класса, имеет доступ и к этому атрибуту.

Видимость типа "защищенный атрибут" означает, что к атрибуту имеют доступ только методы данного класса и его потомков.

Видимость типа "закрытый атрибут" означает, что атрибут доступен только методам класса, который его инкапсулирует.

Символ области видимости может быть опущен. Это означает, что область видимости не показывается (а не то, что она не определена или "public" по умолчанию). Также область видимости может изображаться ключевым словом "public", "private" или "protected".

Нет символа, показывающего возможность изменить атрибут. По умолчанию атрибут изменяемый. Указав в его свойствах пометку "{frozen}", можно объявить атрибут неизменяемым.

Для атрибута можно указывать его множественность. Если она не обозначена, то предполагается, что атрибут может хранить ровно одно значение. Множественность может быть определена в квадратных скобках сразу после имени атрибута.

Ассоциация

Ассоциация определяет некоторую связь между классами. В UML одна ассоциация может специфицировать связь между несколькими (возможно, более чем двумя) классами. Такие ассоциации называются N-арными. Мы рассмотрим самый распространенный случай – бинарную ассоциацию.

Бинарная ассоциация

Бинарная ассоциация (binary association) - это ассоциация между ровно двумя классами. Возможна ассоциация класса с самим собой, которая называется рефлексивной ассоциацией.

Изображается ассоциация в виде сплошной ломаной линии, соединяющей два символа класса.

Конец ассоциации (т.е. место ее присоединения к классу) называется ролью (association role). Большая часть информации, касающейся ассоциации, присоединена к ее ролям.

На линии (рядом с линией), изображающей ассоциацию, могут быть следующие пометки:

- <имя ассоциации> - определяет необязательное имя ассоциации;
- <символ класса ассоциации> - класс, уточняющий свойства данной ассоциации (соединяется с линией ассоциации пунктирной линией).

Роль

Роль (association role) - это неотделимая часть ассоциации, описывающая некоторые свойства её "соединения" с классом (роль класса в данной ассоциации).

У роли могут быть следующие свойства:

- *имя роли* - строка, стоящая рядом с концом ассоциации. Поле не обязательное, но если имя задано, оно должно отображаться на диаграмме;
 - *множественность (multiplicity)* - количество представителей (конкретных объектов), которые могут быть связаны с одним партнером ассоциации;
 - *упорядочение* - если множественность больше одного, то множество связанных объектов может быть упорядочено.
- Варианты:

- *unordered* - неупорядочено (по умолчанию);
- *ordered* - упорядочено (задан линейный порядок) задается пометкой {ordered};
- квалификатор (qualifier) - список атрибутов класса с противоположного конца ассоциации, по значениям которых

можно однозначно разбить множество его объектов на подмножества. Используется для связи объекта класса-партнера ассоциации с группой объектов другого класса-партнера ассоциации;

- *навигация (navigability)* - если в направлении, соответствующем роли, поддерживается навигация, на конце линии может быть изображена стрелка. Возможность навигации в направлении роли означает, что партнеры ассоциации могут просматривать объекты, соответствующие этой роли;
- *агрегирование (aggregation)* - показывает, что ассоциация является отношением типа целое/часть;
- *спецификатор интерфейса* - имя типа или интерфейса в следующей форме:

':' <имя>

Это свойство указывает поведение, ожидаемое от объектов ассоциированного класса. Другими словами, спецификатор интерфейса задает поведение, необходимое для удовлетворения требований ассоциации. Реальные классы, как правило, обеспечивают большую функциональность, чем требуется партнерам ассоциации. Указав интерфейс или тип партнера, можно ограничить доступ к его свойствам рамками указанного интерфейса или типа. Если спецификатор интерфейса не указан, то партнеры имеют доступ ко всей функциональности класса;

- *изменяемость* - если связь изменяема, то есть может быть добавлена, удалена и перемещена, то никаких дополнительных пометок не нужно. В противном случае в строке свойств могут присутствовать:
 - *{frozen}* - связи не могут добавляться, удаляться и перемещаться;

- *ordered* - упорядочено (задан линейный порядок) задается пометкой {ordered};

Агрегирование

Если у роли ассоциации установлен атрибут "агрегирование", то вся ассоциация является *отношением агрегирования*. Такой атрибут может быть установлен только у одной из ролей.

Агрегирование (aggregation) - это отношение между классами типа целое/часть. Агрегируемый класс в той или иной форме является частью агрегата. На практике это может быть реализовано по-разному. Например, объект класса-агрегата может хранить объект агрегируемого класса или ссылку на него.

В UML допускается, что один класс агрегируется многими. При этом возможны два варианта. Первый - на диаграмме классов такая множественность означает лишь возможность для соответствующих объектов - т.е. агрегируемый объект может агрегироваться любым объектом, который принадлежит классам, агрегирующим класс данного объекта. Но объект не агрегируется многими объектами. Второй вариант - объект все-таки может агрегироваться несколькими объектами - так называемое каталожное агрегирование. Примером такого агрегирования может быть книга в библиотеке, которая принадлежит сразу многим каталогам.

По умолчанию агрегирование в UML может быть каталожным. Но имеется специальный вид агрегирования, называемый композицией, который этого не допускает.

Композиция - это специальный вид агрегирования (так называемое сильное агрегирование) (рис.6.3). В частности, агрегируемый объект может быть создан только тогда, когда создан агрегат, а с уничтожением агрегата уничтожаются и все агрегируемые объекты. Агрегируемый объект не разделяем, т.е. владеть им может только один агрегат.

Агрегирование изображается на диаграмме полым ромбом на конце линии ассоциации со стороны агрегирующего класса (агрегата).

Композиция показывается также как и агрегирование, но ромбик рисуется не пустым, а заполненным.

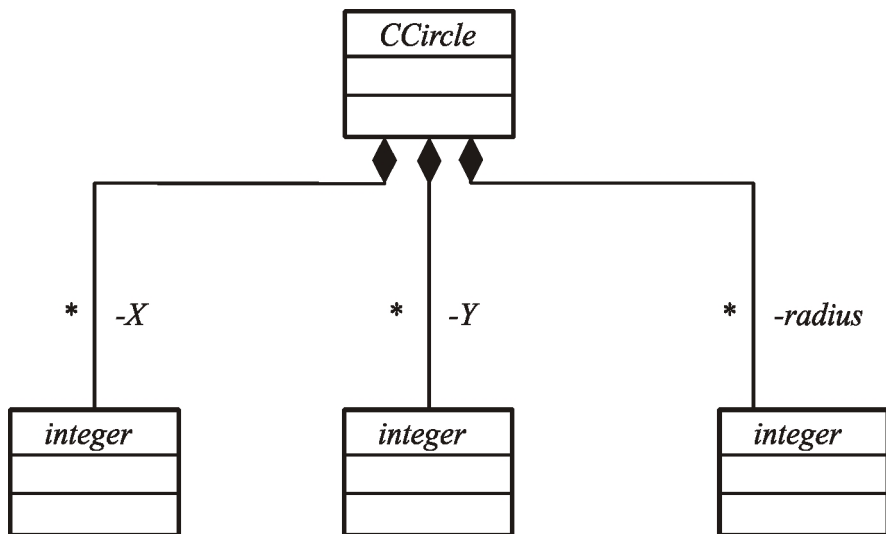


Рис. 6.3 Композиция (сильное агрегирование).

Кроме того, UML допускает «вложенную» нотацию, в которой агрегируемый класс изображается внутри класса-агрегата. При этом перед его именем указывается имя роли и символ « : ».

Наследование

Наследование - это отношение типа общее-частное между элементами модели.

Наследование пакетов означает, что в пакете-наследнике все сущности пакета-предка будут видны под своими собственными именами (т.е. пространства имен объединяются). На практике наследование пакетов применяется достаточно редко (в Rational Rose (классическая реализация UML) такая возможность не поддерживается).

Наследование показывается сплошной линией, идущей от конкретного элемента к более общему (в терминологии ООП - от потомка

к предку, от сына к отцу, или от подкласса к суперклассу). Со стороны более общего элемента рисуется большой полый треугольник.

Один из атрибутов отношения наследования - дискриминатор (discriminator) - строка, задающая имя группы потомков. Его использование полезно, если у данного класса много потомков, и мы хотим разбить их на несколько групп. Отсутствие дискриминатора означает, что дискриминатор - пустая строка (дискриминатор по умолчанию). Изображается дискриминатор текстовой строкой, стоящей возле линии наследования.

Способ изображения наследования показан на рис. 6.4:

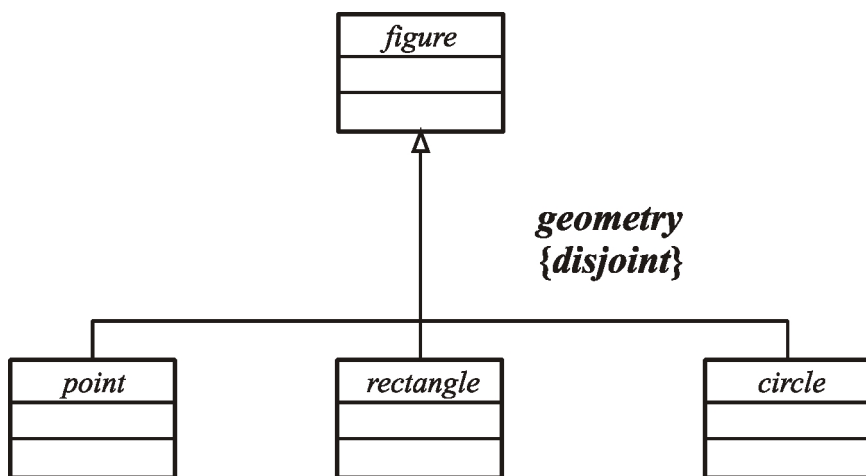


Рис. 6.4. Наследование классов

Зависимость

В некоторых случаях два и более элемента модели могут быть семантически связаны. Например, класс А использует методы класса В. Тогда при изменении класса В необходимо произвести соответствующие изменения в классе А. Поэтому в нотации UML предусмотрено такое отношение, как зависимость. Для рассмотренного примера на диаграмме классов необходимо указать, что класс А зависит от класса

В. Отношение зависимости является универсальным, т.е. с помощью него можно связывать различные типы сущностей UML, например шаблон и метакласс.

Зависимость изображается пунктирной линией, проведенной между двумя элементами диаграммы, и считается, что элемент, привязанный к концу стрелки, зависит от элемента, привязанного к началу этой стрелки. Зависимость может быть снабжена именем и спецификатором.

Существуют следующие виды зависимостей:

- *trace* - показывает историческую связь между двумя элементами, которые представляли одно и то же понятие на разных этапах;
- *refine* - историческая связь между элементами, как правило, показывает, что один элемент как бы произошел от другого;
- *uses* - использование - ситуация когда один элемент модели использует другой;
- *bind* - привязка - устанавливается между шаблоном и экземпляром шаблона;
- *friend* - аналог C++ friend.

Особое место объектно-ориентированной модели определяется тем, что основная цель UML – проведение объектно-ориентированного анализа и проектирования ПС и поддержание перехода к объектно-ориентированной реализации. А объектно-ориентированное ПС строится из классов. Таким образом, основной задачей стадий разработки, предшествующих реализации, является построение модели классов. Естественно, в ходе реализации появятся новые классы, но основной каркас системы не должен меняться, в противном случае анализ и проектирование выполнены некачественно.

Отметим, что предложенная модель классов может использоваться, начиная с анализа и кончая реализацией. При анализе с ее помощью удобно моделировать объекты предметной области и связи между ними. При проектировании на ней изображаются основные элементы

будущей системы. При реализации на модели классов определяются все классы системы и связи между ними. Предполагается, что вместе со средствами реинжиниринга модель классов должна служить эффективным средством визуальной разработки ПО. Отмечается, что использование на разных этапах разработки системы одной нотации облегчает преемственность между этими этапами.

ЗАКЛЮЧЕНИЕ

В современных условиях развития компьютерных технологий разработка программных комплексов превратилась в напряжённую работу коллективов специалистов различной квалификации при больших трудовых затратах. Программы превратились в изделия, массовое производство которых потребовало создания соответствующих технологий.

Общий вектор развития современных технологий программирования от интуитивных, которые были характерны ранним этапам развития вычислительной техники, к более формализованным, поддающимся автоматизации и позволяющим создавать ПС с заданными показателями качества при минимальных затратах.

Компьютерную технологию разработки ПС определяют как технологию программирования, в которой используются программные инструменты для разработки формализованных спецификаций программ и других документов (включая и графические спецификации) с последующей автоматической генерацией программ и документов (или хотя бы значительной их части) по этим спецификациям с программной поддержкой разработки графических требований и графических спецификаций ПС.

Компьютерная технология обеспечивает семантическое понимание документов средствами программной поддержки компьютерной технологии. В результате семантического понимания документов программная поддержка автоматически генерирует программы. Поэтому как для спецификации программ, так и для спецификации других документов широко используются формальные языки уже на ранних этапах разработки ПС. Например, широко используются формальные графические языки спецификаций.

СПИСОК ЛИТЕРАТУРЫ

1. Иан Соммервилл. Инженерия программного обеспечения. 6-е изд. - М.: Вильямс, 2002.
2. Вендров А.М. Проектирование программного обеспечения экономических информационных систем. – М.: Финансы и статистика, 2003.
3. Жоголев Е.А. Введение в технологию программирования. - М.: ДИАЛОГ-МГУ, 1994.
4. Ван Тассэл Д. Стил, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1980.
5. Зелковец М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. - М.: Мир, 1982.
6. Фуксман А.Л. Технологические аспекты создания программных систем. - М.: Статистика, 1979.
7. Майерс Г. Надёжность программного обеспечения. - М.: Мир, 1980.
8. Зиглер К. Методы проектирования программных систем. - М.: Мир, 1985.

Учебное издание

Куприянов Александр Викторович

**ТЕХНОЛОГИИ ПРОЕКТИРОВАНИЯ
ПРОГРАММНЫХ КОМПЛЕКСОВ**

Учебное пособие

Технический редактор *В.В.Сергеев*
Редакторская обработка *Н.С.Куприянова*
Корректорская обработка *В.С. Телпова*
Доверстка *Н.С. Куприянова*

Подписано в печать 28.12.06. Формат 60х84 1/16.

Бумага офсетная. Печать офсетная.

Усл. печ. л. 4,2. Усл. кр.-отг. 4,3. Печ.л. 4,5.

Тираж 50 экз. Заказ . ИП-50/2006

Самарский государственный аэрокосмический университет.
443086 Самара, Московское шоссе, 34.

Изд-во Самарского государственного аэрокосмического университета.
443086 Самара, Московское шоссе, 34.

