ВЫСШЕЕ ОБРАЗОВАНИЕ

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Л.Г. Гагарина,
Е.В. Кокорева,
Б.Д. Виснадул

Л. Г. Гагарина, Е. В. Кокорева,
Б. Д. Виснадул

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Под редакцией профессора Л. Г. Гагариной

*Допущено Учебно-методическим объединением вузов
по университетскому политехническому образованию в качестве
учебного пособия для студентов высших учебных заведений,
обучающихся по направлению 230100 «Информатика
и вычислительная техника», специальности 230105
«Программное обеспечение вычислительной техники
и автоматизированных систем»*

Москва
ИД «ФОРУМ» — ИНФРА-М
2008

УДК 004(075.8)
ББК 32.973-018я73
Г12

Рецензенты:
доцент кафедры «Программное обеспечение ЭВМ и информационные технологии» МГТУ им. Н. Э. Баумана, кандидат физико-математических наук *Т. Н. Романова*;
доктор техн. наук, профессор *О. И. Лисов*
(Институт искусств и информационных технологий)

Гагарина Л. Г., Кокорева Е. В., Виснадул Б. Д.

Г12 Технология разработки программного обеспечения: учебное пособие / под ред. Л. Г Гагариной. — М.: ИД «ФОРУМ»: ИНФРА-М, 2008. — 400 с.: ил. — (Высшее образование).

ISBN 978-5-8199-0342-1 (ИД «ФОРУМ»)
ISBN 978-5-16-003193-4 (ИНФРА-М)

Рассматриваются основные понятия и определения, классификация программного обеспечения, этапы создания программного продукта в рамках жизненного цикла, освещается современное состояние технологий разработки программных продуктов. Изложены существующие подходы к оценке качества процессов создания программного обеспечения, произведен анализ требований и определение спецификаций программного обеспечения. Большое внимание уделено вопросам проектирования ПО, его надежности, коллективной разработки с помощью современных систем контроля версий.

Теоретический материал сопровождается практикумом в виде лабораторных работ по изложенной тематике.

Предназначено для студентов и преподавателей высших учебных заведений технического профиля, а также для специалистов любой предметной области, проявивших интерес к современному состоянию методов и средств программирования.

УДК 004(075.8)
ББК 32.973-018я73

ISBN 978-5-8199-0342-1 (ИД «ФОРУМ»)
ISBN 978-5-16-003193-4 (ИНФРА-М)

© Л. Г Гагарина, 2008
© Е. В. Кокорева, 2008
© Б. Д. Виснадул, 2008
© ИД «ФОРУМ», 2008

Введение

В настоящее время в условиях развивающегося информационного общества с учетом всеобщего применения и распространения компьютерных и телекоммуникационных технологий и систем, а также в связи с реализацией объявленной ранее Президентом РФ программы на создание в стране единого образовательного и информационного пространства появление учебного пособия, освещающего теоретические и практические вопросы разработки программного обеспечения, особенно актуально.

Кроме того, поскольку Интернет в нашей стране доступен в основном пользователям, проживающим в крупных городах, а информатизация общества невозможна без хорошо развитой коммуникационной инфраструктуры, наибольший спрос ожидает научно-техническую литературу прикладного характера, к какой и относится представленная книга. К тому же системное изложение материала, богатый иллюстративный материал и масса жизненных примеров, несомненно, придают изданию особую ценность.

Согласно «Концепции информатизации сферы образования Российской Федерации» и положениям Федеральной целевой программы «Электронная Россия» одной из особенностей перспективной системы образования в нашей стране является опережающее образование, в рамках которого изучаются последние достижения в области информатизации, ее средства, методы, а также перспективы дальнейшего развития и практического использования. Анализ содержания представленного пособия позволяет утверждать, что его следует рассматривать не только как важное средство информационной поддержки учебного процесса в качестве эффективного педагогического инструмента, но и как необходимый инструментарий опережающего образования.

В представленном учебном пособии достаточно полно изложены понятия жизненного цикла программного обеспечения,

процесс его производства: методы, технология и инструментальные средства, тестирование, отладка и сопровождение программ.

Основное отличие данного пособия от аналогичных изданий заключается в освещении основных дидактических единиц дисциплины «Технология разработки программного обеспечения» с учетом пропедевтической подготовки потенциального читателя студента высшего учебного заведения. Именно поэтому только на базе основных понятий и определений в области разработки программных средств возможно освещение проблем документирования, проектирования программного обеспечения; рассмотрение вопросов технологического цикла разработки программных систем. Весьма интересными и своевременными для будущих специалистов современного глобального общества являются разделы об организации коллективной работы по созданию программ и организации процесса разработки с применением инструментальных средств поддержки.

Материал пособия прошел апробацию в Московском институте электронной техники и используется в учебном процессе различных факультетов.

Глава 1
ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

1.1. Технология программирования в историческом аспекте

Чтобы разобраться в существующих технологиях программирования и определить основные тенденции их развития, целесообразно рассматривать эти технологии в историческом контексте, выделяя основные этапы развития программирования как науки.

Первый этап — «стихийное» программирование. Этот этап охватывает период от момента появления первых вычислительных машин до середины 60-х годов XX в. В этот период практически отсутствовали сформулированные технологии и программирование фактически было искусством. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных (рис. 1.1). Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

Рис. 1.1. Структура первых программ

Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и

мнемоники кодов операций. В результате программы стали более «читаемыми».

Создание языков программирования высокого уровня, таких как FORTRAN и ALGOL, существенно упростило программирование вычислений, снизив уровень детализации операций. Это, в свою очередь, позволило увеличить сложность программ.

Революционным было появление в языках средств, позволяющих оперировать подпрограммами. (Идея написания подпрограмм появилась гораздо раньше, но отсутствие средств поддержки в первых языковых средствах существенно снижало эффективность их применения.) Подпрограммы можно было сохранять и использовать в других программах. В результате были созданы огромные библиотеки расчетных и служебных подпрограмм, которые по мере надобности вызывались из разрабатываемой программы.

Типичная программа того времени состояла из основной программы, области *глобальных данных* и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части (рис. 1.2).

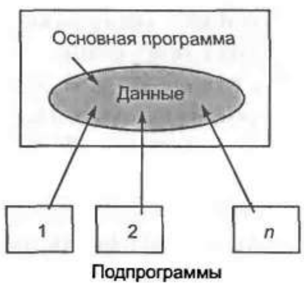


Рис. 1.2. Архитектура программы с глобальной областью данных

Слабым местом такой архитектуры было то, что при увеличении количества подпрограмм возрастала вероятность искажения части глобальных данных какой-либо подпрограммой. Например, подпрограмма поиска корней уравнения на заданном интервале по методу деления отрезка пополам меняет величину интервала. Если при выходе из подпрограммы не предусмотреть восстановления первоначального интервала, то в глобальной области окажется неверное значение интервала. Чтобы сократить количество таких ошибок в подпрограммах, было предложено размещать *локальные данные* (рис. 1.3).

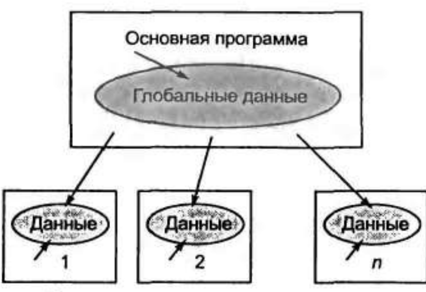


Рис. 1.3. Архитектура программы, использующей подпрограммы с локальными данными

Сложность разрабатываемого программного обеспечения при использовании подпрограмм с локальными данными по-прежнему ограничивалась возможностью программиста отслеживать процессы обработки данных, но уже на новом уровне. Однако появление средств поддержки подпрограмм позволило осуществлять разработку программного обеспечения нескольким программистам параллельно.

В начале 60-х годов XX в. разразился «кризис программирования». Он выражался в том, что фирмы, взявшиеся за разработку сложного программного обеспечения, такого как операционные системы, срывали все сроки завершения проектов [8]. Проект устаревал раньше, чем был готов к внедрению, увеличивалась его стоимость, и в результате многие проекты так никогда и не были завершены.

Объективно все это было вызвано несовершенством технологии программирования. Прежде всего стихийно использовалась разработка «снизу—вверх» — подход, при котором вначале проектировали и реализовывали сравнительно простые подпрограммы, из которых затем пытались построить сложную программу. В отсутствие четких моделей описания подпрограмм и методов их проектирования создание каждой подпрограммы превращалось в непростую задачу, интерфейсы подпрограмм получались сложными, и при сборке программного продукта выявлялось большое количество ошибок согласования. Исправление таких ошибок, как правило, требовало серьезного изменения уже разработанных подпрограмм, что еще более осложняло ситуацию, так как при этом в программу часто вносились новые ошибки, которые также необходимо было исправлять... В конечном итоге

процесс тестирования и отладки программ занимал более 80 % времени разработки, если вообще когда-нибудь заканчивался. На повестке дня самым серьезным образом стоял вопрос разработки технологии создания сложных программных продуктов, снижающей вероятность ошибок проектирования.

Анализ причин возникновения большинства ошибок позволил сформулировать новый подход к программированию, который был назван «структурным» [19, 23].

Второй этап — структурный подход к программированию (60—70-е годы XX в.). *Структурный подход к программированию* представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит *декомпозиция* (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших (до 40—50 операторов) подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название *процедурной* декомпозиции.

В отличие от используемого ранее процедурного подхода к декомпозиции, структурный подход требовал представления задачи в виде иерархии подзадач простейшей структуры. Проектирование, таким образом, осуществлялось «сверху—вниз» и подразумевало реализацию общей идеи, обеспечивая проработку интерфейсов подпрограмм. Одновременно вводились ограничения на конструкции алгоритмов, рекомендовались формальные модели их описания, а также специальный метод проектирования алгоритмов — метод пошаговой детализации.

Поддержка принципов структурного программирования была заложена в основу так называемых *процедурных* языков программирования. Как правило, они включали основные «структурные» операторы передачи управления, поддерживали вложение подпрограмм, локализацию и ограничение области «видимости» данных. Среди наиболее известных языков этой группы стоит назвать PL/1, ALGOL-68, Pascal, C.

Одновременно со структурным программированием появилось огромное количество языков, базирующихся на других концепциях, но большинство из них не выдержало конкуренции. Какие-то языки были просто забыты, идеи других были в дальнейшем использованы в следующих версиях развиваемых языков.

Дальнейший рост сложности и размеров разрабатываемого программного обеспечения потребовал развития *структурирова-*

ния данных. Как следствие этого в языках появляется возможность определения пользовательских типов данных. Одновременно усилилось стремление разграничить доступ к глобальным данным программы, чтобы уменьшить количество ошибок, возникающих при работе с глобальными данными. В результате появилась и начала развиваться технология модульного программирования.

Модульное программирование предполагает выделение групп подпрограмм, использующих одни и те же глобальные данные в отдельно компилируемые *модули* (библиотеки подпрограмм), например модуль графических ресурсов, модуль подпрограмм вывода на принтер (рис. 1.4). Связи между модулями при использовании данной технологии осуществляются через специальный интерфейс, в то время как доступ к реализации модуля (телам подпрограмм и некоторым «внутренним» переменным) запрещен. Эту технологию поддерживают современные версии языков Pascal и С (С++), языки Ада и Modula.

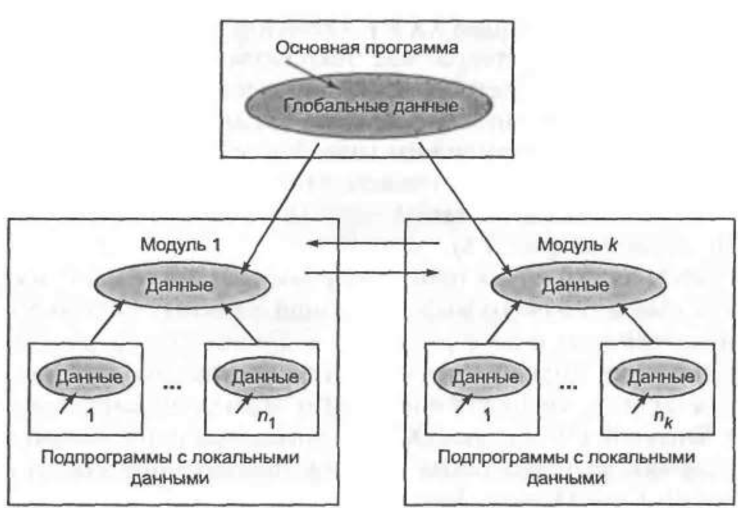


Рис. 1.4. Архитектура программы, состоящей из модулей

Использование модульного программирования существенно упростило разработку программного обеспечения несколькими программистами. Теперь каждый из них мог разрабатывать свои модули независимо, обеспечивая взаимодействие модулей через

специально оговоренные межмодульные интерфейсы. Кроме того, модули в дальнейшем без изменений можно было использовать в других разработках, что повысило производительность труда программистов.

Практика показала, что структурный подход в сочетании с модульным программированием позволяет получать достаточно надежные программы, размер которых *не превышает 100 000 операторов* [10]. Узким местом модульного программирования является то, что ошибка в интерфейсе при вызове подпрограммы выявляется только при выполнении программы (из-за раздельной компиляции модулей обнаружить эти ошибки раньше невозможно). При увеличении размера программы обычно возрастает сложность межмодульных интерфейсов, и с некоторого момента предусмотреть взаимовлияние отдельных частей программы становится практически невозможно. Для разработки программного обеспечения большого объема было предложено использовать *объектный подход.*

Третий этап — объектный подход к программированию (с середины 80-х до конца 90-х годов XX в.). *Объектно-ориентированное программирование* определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности *объектов*, каждый из которых является экземпляром определенного типа (*класса*), а классы образуют иерархию с *наследованием* свойств [10, 24, 29]. Взаимодействие программных объектов в такой системе осуществляется путем передачи *сообщений* (рис. 1.5).

Объектная структура программы впервые была использована в языке имитационного моделирования сложных систем Simula, появившемся еще в 60-х годах XX в. Естественный для языков моделирования способ представления программы получил развитие в другом специализированном языке моделирования языке Smalltalk (70-е годы XX в.), а затем был использован в новых версиях универсальных языков программирования, таких как Pascal, C++, Modula, Java.

Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку. Это приводит к более полной локализации данных и интегрированию их с подпрограммами обработки, что позволяет вести практически независимую разработку отдельных частей (объектов) програм-

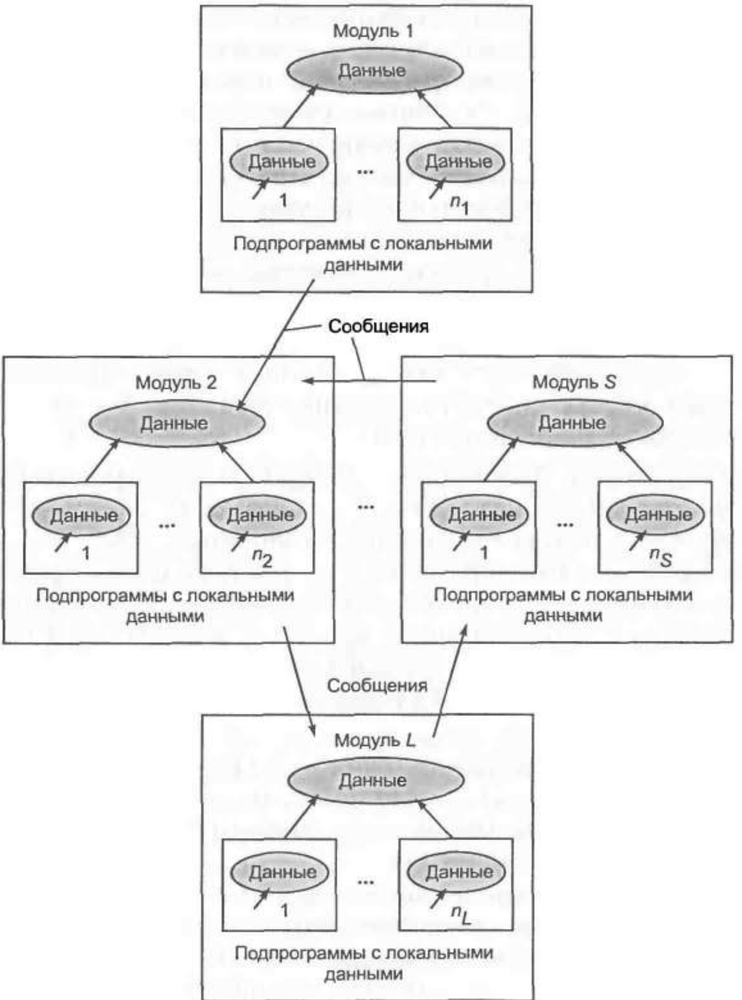


Рис. 1.5. Архитектура программы при объектно-ориентированном подходе

мы. Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения. Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кодов и появляется возможность создания библиотек классов для различных применений.

Бурное развитие технологий программирования, основанных на объектном подходе, позволило решить многие проблемы. Так, были созданы среды, поддерживающие *визуальное программирование*, например Delphi, C++ Builder, Visual C++ и т. д. При использовании визуальной среды у программиста появляется возможность проектировать некоторую часть, например интерфейсы будущего продукта, с применением визуальных средств добавления и настройки специальных библиотечных компонентов. Результатом визуального проектирования является заготовка будущей программы, в которую уже внесены соответствующие коды.

Использование объектного подхода имеет много преимуществ, однако его конкретная реализация в объектно-ориентированных языках программирования, таких как Pascal и C++, имеет существенные недостатки:

1) фактически отсутствуют стандарты компоновки двоичных результатов компиляции объектов в единое целое даже в пределах одного языка программирования. Компоновка объектов, полученных разными компиляторами C++, в лучшем случае проблематична, что приводит к необходимости разработки программного обеспечения с использованием средств и возможностей одного языка программирования высокого уровня и одного компилятора, а значит, требует наличия исходных кодов используемых библиотек классов;

2) изменение реализации одного из программных объектов, как минимум, связано с перекомпиляцией соответствующего модуля и перекомпоновкой всего программного обеспечения, использующего данный объект.

Таким образом, при использовании этих языков программирования сохраняется зависимость модулей программного обеспечения от адресов экспортируемых полей и методов, а также структур и форматов данных. Эта зависимость объективна, так как модули должны взаимодействовать между собой, обращаясь к ресурсам друг друга. Связи модулей нельзя разорвать, но можно попробовать стандартизировать их взаимодействие, на чем и основан компонентный подход к программированию.

Четвертый этап — компонентный подход и CASE-технологии (с середины 90-х годов XX в. до нашего времени). *Компонентный подход* предполагает построение программного обеспечения из отдельных компонентов физически отдельно существующих частей программного обеспечения, которые взаимодействуют между

собой через *стандартизованные двоичные интерфейсы*. В отличие от обычных объектов объекты-компоненты можно собрать в динамически вызываемые библиотеки или исполняемые файлы, распространять в двоичном виде (без исходных текстов) и использовать в любом языке программирования, поддерживающем соответствующую технологию. На сегодня рынок объектов стал реальностью, так, в Интернете существуют узлы, предоставляющие большое количество компонентов, рекламой компонентов забиты журналы. Это позволяет программистам создавать продукты, хотя бы частично состоящие из повторно использованных частей, т. е. использовать технологию, хорошо зарекомендовавшую себя в области проектирования аппаратуры.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model — компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture — общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM фирмы Microsoft является развитием технологии OLE I (Object Linking and Embedding — связывание и внедрение объектов), которая использовалась в ранних версиях Windows для создания составных документов. Технология COM определяет *общую парадигму взаимодействия программ любых типов*: библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции (*службы*), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 1.6). Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM — распределенная COM).

По технологии COM приложение предоставляет свои службы, используя специальные объекты — *объекты* COM, которые являются экземплярами *классов* COM. Объект COM так же, как обычный объект, включает поля и методы, но в отличие от обычных объектов каждый объект COM может реализовывать несколько интерфейсов, обеспечивающих доступ к его полям и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов). При этом интерфейс обычно объединяет

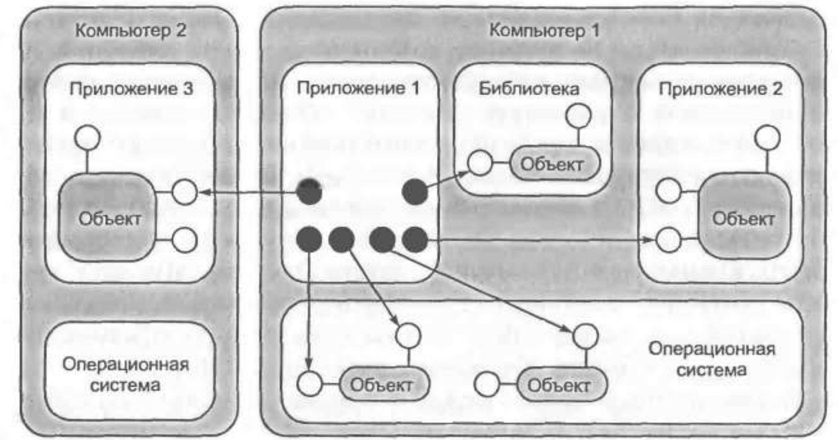


Рис. 1.6. Взаимодействие программных компонентов различных типов

несколько однотипных функций. Кроме того, классы COM поддерживают *наследование интерфейсов*, но не поддерживают *наследования реализации*, т. е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя.

Каждый интерфейс имеет имя, начинающееся с символа «I», и глобальный уникальный идентификатор IID (Interface IDentifier). Любой объект COM обязательно реализует интерфейс IUnknown (на схемах этот интерфейс всегда располагают сверху). Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта.

Объект всегда функционирует в составе *сервера* — динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта. Различают три типа серверов:

1) внутренний сервер; реализуется динамическими библиотеками, которые подключаются к приложению-клиенту и работают в одном с ними адресном пространстве, наиболее эффективный сервер, кроме того, он не требует специальных средств;

2) локальный сервер; создается отдельным процессом (модулем, exe), который работает на одном компьютере с клиентом;

3) удаленный сервер; создается процессом, который работает на другом компьютере.

Например, Microsoft Word является локальным сервером. Он включает множество объектов, которые могут использоваться другими приложениями.

Для обращения к службам клиент должен получить указатель на соответствующий интерфейс. Перед первым обращением к объекту клиент посылает запрос к библиотеке COM, хранящей информацию обо всех зарегистрированных в системе классах COM объектов, и передает ей имя класса, идентификатор интерфейса и тип сервера. Библиотека запускает необходимый сервер, создает требуемые объекты и возвращает указатели на объекты и интерфейсы. Получив указатели, клиент может вызывать необходимые функции объекта.

Взаимодействие клиента и сервера обеспечивается базовыми механизмами COM или DCOM, поэтому клиенту безразлично местонахождение объекта. При использовании локальных и удаленных серверов в адресном пространстве клиента создается *proxy-объект* — заместитель объекта COM, а в адресном пространстве сервера COM — заглушка, соответствующая клиенту. Получив задание от клиента, заместитель упаковывает его параметры и, используя службы операционной системы, передает вызов заглушке. Заглушка распаковывает задание и передает его объекту COM. Результат возвращается клиенту в обратном порядке.

На базе технологии COM и ее распределенной версии DCOM были разработаны компонентные технологии, решающие различные задачи разработки программного обеспечения.

OLE-*automation* или просто Automation (автоматизация) — технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений. Вводит понятие *диспинтерфейса* (dispinterface) — специального интерфейса, облегчающего вызов функций объекта. Эту технологию поддерживает, например, Microsoft Excel, предоставляя другим приложениям свои службы.

ActiveX — технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения как сосредоточенного на одном компьютере, так и распределенного в сети. Предполагает использование визуального программирования для создания компонентов — элементов управления ActiveX. Полученные таким образом элементы управления можно устанавливать на компьютер дистанционно с удаленного сервера, причем устанавливаемый код зависит от используемой операционной системы. Это позволяет применять элементы управления ActiveX в клиентских частях приложений Интернета.

Основными преимуществами технологии ActiveX, обеспечивающими ей широкое распространение, являются:

1) быстрое написание программного кода, поскольку все действия, связанные с организацией взаимодействия сервера и клиента, берет на программное обеспечение COM, программирование сетевых приложений становится похожим на программирование для отдельного компьютера;

2) открытость и мобильность — спецификации технологии недавно были переданы в Open Group как основа открытого стандарта;

3) возможность написания приложений с использованием знакомых средств разработки, например Visual Basic, Visual C++, Borland Delphi, Borland C++ и любых средств разработки на Java;

4) большое количество уже существующих бесплатных программных элементов ActiveX (к тому же практически любой программный компонент OLE совместим с технологиями ActiveX и может применяться без модификаций в сетевых приложениях);

5) стандартность — технология ActiveX основана на широко используемых стандартах Интернет (TCP/IP, HTML, Java), с одной стороны, и стандартах, введенных в свое время Microsoft и необходимых для сохранения совместимости (COM, OLE).

MTS (Microsoft Transaction Server — сервер управления транзакциями) — технология, обеспечивающая безопасность и стабильную работу распределенных приложений при больших объемах передаваемых данных.

MIDAS (Multitier Distributed Application Server — сервер многозвенных распределенных приложений) — технология, организующая доступ к данным разных компьютеров с учетом балансировки нагрузки сети.

Все указанные технологии реализуют компонентный подход, заложенный в COM. Так, с точки зрения COM элемент управления ActiveX — внутренний сервер, поддерживающий технологию OLE-automation. Для программиста же элемент ActiveX — «черный ящик», обладающий свойствами, методами и событиями, который можно использовать как строительный блок при создании приложений.

Технология CORBA, разработанная группой компаний OMG (Object Management Group — группа внедрения объектной технологии программирования), реализует подход, аналогичный COM, на базе объектов и интерфейсов CORBA. Программное ядро CORBA реализовано для всех основных аппаратных и про-

граммных платформ, и потому эту технологию можно использовать для создания распределенного программного обеспечения в гетерогенной (разнородной) вычислительной среде. Организация взаимодействия между объектами клиента и сервера в CORBA осуществляется с помощью специального посредника, названного VisiBroker, и другого специализированного программного обеспечения.

Отличительной особенностью современного этапа развития технологии программирования, кроме изменения подхода, является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые были названы CASE-технологиями (Computer-Aided Software/System Engineering — разработка программного обеспечения программных систем с использованием компьютерной поддержки). Без средств автоматизации разработка достаточно сложного программного обеспечения на настоящий момент становится трудно осуществимой: память человека уже не в состоянии фиксировать все детали, которые необходимо учитывать при разработке программного обеспечения. На сегодня существуют CASE-технологии, поддерживающие как структурный, так и объектный (в том числе и компонентный) подходы к программированию.

Появление нового подхода не означает, что отныне все программное обеспечение будет создаваться из программных компонентов, но анализ существующих проблем разработки сложного программного обеспечения показывает, что он будет применяться достаточно широко.

1.2. Основные понятия и определения

Технология программирования — совокупность методов и средств, применяемых в процессе разработки программного обеспечения.

Программа (program, routine) — упорядоченная последовательность команд (инструкций) компьютера для решения задачи.

Программное обеспечение (software) — совокупность программ обработки данных и необходимых для их эксплуатации документов.

Задача (problem, task) — проблема, подлежащая решению.

Приложение (application) — программная реализация на компьютере решения задачи.

Термин «задача» в программировании означает единицу работы вычислительной системы, требующую выделения вычислительных ресурсов (процессорного времени, памяти).

Процесс создания программ можно представить как последовательность следующих действий:

1) постановка задачи;

2) алгоритмизация решения задачи;

3) программирование.

Постановка задачи (problem definition) — это точная формулировка решения задачи на компьютере с описанием входной и выходной информации.

Алгоритм — система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных (выходной информации) в желаемый результат (выходную информацию) за конечное число шагов.

Программирование (programming) — теоретическая и практическая деятельность, связанная с созданием программ.

По отношению к ПО компьютерные пользователи делятся на следующие группы:

1) системные программисты. Занимаются разработкой, эксплуатацией и сопровождением системного программного обеспечения;

2) прикладные программисты. Осуществляют разработку и отладку программ для решения различных прикладных задач;

3) конечные пользователи. Имеют элементарные навыки работы с компьютером и используемыми ими прикладными программами;

4) администраторы сети. Отвечают за работу вычислительных сетей;

5) администраторы баз данных. Обеспечивают организационную поддержку базы данных.

Сопровождение программы — поддержка работоспособности программы, переход на ее новые версии, внесения изменений, исправление ошибок и т. д.

Основные характеристики программ:

1) алгоритмическая сложность;

2) состав функций обработки информации;

3) объем файлов, используемых программой;

4) требования к операционной системе (ОС) и техническим средствам обработки, в том числе объем дисковой памяти, раз-

мер оперативной памяти для запуска программы, тип процессора, версия ОС, наличие вычислительной сети и т. д.

Показатели качества программы:

1) мобильность (многоплатформенность) — независимость от технического комплекса системы обработки данных, ОС, сетевых возможностей, специфики предметной области задачи и т. д.;

2) надежность — устойчивость, точность выполнения предписанных функций обработки, возможность диагностики возникающих ошибок в работе программы;

3) эффективность как с точки зрения требований пользователя, так и расхода вычислительных ресурсов;

4) учет человеческого фактора — дружественный интерфейс, контекстно-зависимая подсказка, хорошая документация;

5) модифицируемость — способность к внесению изменений, например, расширение функций обработки, переход на другую техническую базу обработки и т. п.

6) коммуникативность — максимально возможная интеграция с другими программами, обеспечение обмена данными между программами.

Все программы по характеру использования и категориям пользователей можно разделить на два класса — утилитарные программы и программные продукты.

Утилитарные программы («программы для себя») предназначены для удовлетворения нужд их разработчиков. Чаще всего такие программы выполняют роль отладочных приложений, являются программами решения задач, не предназначенных для широкого распространения.

Программные продукты (изделия) используются для удовлетворения потребностей пользователей, широкого распространения и продажи.

В настоящее время существуют и другие варианты легального распространения программных продуктов, которые появились с использованием глобальных телекоммуникаций:

- freeware — бесплатные программы, свободно распространяемые, поддерживаются самим пользователем, который правомочен вносить в них необходимые изменения;
- shareware некоммерческие (условно-бесплатные) программы, которые могут использоваться, как правило, бесплатно.

Ряд производителей использует OEM-программы (Original Equipment Manufacturer), т. е. встроенные программы, устанавливаемые на компьютеры или поставляемые вместе с компьютерами.

Программный продукт должен быть соответствующим образом подготовлен к эксплуатации (отлажен), иметь необходимую техническую документацию, предоставлять сервис и гарантию надежной работы программы, иметь товарный знак изготовителя, а также наличие кода государственной регистрации.

1.3. Классификация программного обеспечения

Можно выделить три класса ПО:

1) системное;

2) пакеты прикладных программ (прикладное ПО);

3) инструментарий технологии программирования (инструментальные средства для разработки ПО).

Системное ПО направлено:

- на создание операционной среды функционирования других программ;
- обеспечение надежной и эффективной работы самого компьютера и вычислительной сети;
- проведение диагностики и профилактики аппаратуры компьютера и вычислительных сетей;
- выполнение вспомогательных технологических процессов (копирование, архивация, восстановление файлов программ и БД и т. п.).

Системное ПО (System Software) — совокупность программ и программных комплексов для обеспечения работы компьютера и вычислительных сетей.

Прикладное ПО служит программным инструментарием решения функциональных задач и является самым многочисленным классом ПО. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей. Таким образом, прикладное ПО — комплекс взаимосвязанных программ для решения задач определенного класса предметной области.

Инструментарий технологии программирования обеспечивает процесс разработки программ и включает специализирован-

ное ПО, которое является инструментальным средством разработки. ПО данного класса поддерживает все технологические этапы процесса проектирования, программирования, отладки и тестирования создаваемых программ. Пользователями данного ПО являются системные и прикладные программисты.

1.3.1. Системное программное обеспечение

В составе системного ПО можно выделить две составляющие:

1) базовое ПО;
2) сервисное ПО.

Базовое ПО — минимальный набор программных средств, обеспечивающих работу компьютера.

Сервисное ПО — программы и программные комплексы, которые расширяют возможности базового ПО и организуют более удобную среду работы пользователя.

В базовое ПО входят:

1) операционная система;
2) операционные оболочки (текстовые, графические);
3) сетевая операционная система.

Компьютер выполняет действия в соответствии с предписаниями программы, созданной на одном из языков программирования. При работе пользователя на компьютере часто возникает необходимость выполнять операции с прикладной программой в целом, организовать работу внешних устройств, проверить работу различных блоков, скопировать информацию и т. д. Эти операции используются для работы с любой программой. Поэтому целесообразно из всего многообразия операций, выполняемых компьютером, выделить типовые и реализовать их с помощью специализированных программ, которые следует принять в качестве стандартных средств, поставляемых вместе с аппаратной частью. Программы, организующие работу устройств и не связанные со спецификой решаемой задачи, вошли в состав комплекса программ, названного операционной системой.

Операционная система — совокупность программных средств, обеспечивающих управление аппаратной частью компьютера и прикладными программами, а также их взаимодействием между собой и пользователем.

Операционная система образует автономную среду, не связанную ни с одним из языков программирования. Любая при-

кладная программа связана с ОС и может эксплуатироваться только на компьютерах, где имеется аналогичная системная среда. Программа, созданная в среде одной ОС, не функционирует в среде другой ОС, если в ней не обеспечена возможность конвертации (преобразования) программ.

Для работы с ОС необходимо овладеть языком этой среды — совокупностью команд, структура которых определяется синтаксисом этого языка.

ОС выполняет следующие функции:

- управление работой каждого блока ПО и их взаимодействием;
- управление выполнением программ;
- организацию хранения информации во внешней памяти;
- взаимодействие пользователя с компьютером (поддержку интерфейса пользователя).

Обычно ОС хранится на жестком диске, а при его отсутствии выделяется специальный гибкий диск, который называется системным диском. При включении компьютера ОС автоматически загружается с диска в оперативную память и занимает в ней определенное место. ОС создается не для отдельной модели компьютера, а для серии компьютеров, в структуру которых заложена и развивается во всех последующих моделях определенная концепция.

Операционные системы для ПК делятся:

- на одно- и многозадачные (в зависимости от числа параллельного выполнения прикладных процессов);
- одно- и многопользовательские (в зависимости от числа пользователей, одновременно работающих с ОС);
- непереносимые и переносимые на другие типы компьютеров;
- несетевые и сетевые, обеспечивающие работу в ВС ЭВМ.

Операционные системы, как правило, сравнивают по следующим критериям:

- управлению памятью (максимальный объем адресуемого пространства, технические показатели использования памяти);
- функциональным возможностям вспомогательных программ (утилит) в составе ОС;
- наличию компрессии магнитных дисков;
- возможности архивирования файлов;
- поддержке многозадачного режима работы;

- наличию качественной документации;
- условию и сложности процесса инсталляции.

Наиболее распространенными ОС для персональных компьютеров в настоящее время являются:

- Windows 2000 (Microsoft);
- Windows XP (Microsoft);
- UNIX;
- OS/2 (IBM).

ОС **OS/2** разработана IBM для ПК на основе системной прикладной архитектуры, ранее используемой для больших ЭВМ. Это многозадачная, многопользовательская ОС, обеспечивающая как текстовый, так и графический интерфейс пользователя. **OS/2** обеспечивает:

- поддержку графического интерфейса пользователя;
- одновременную обработку нескольких приложений;
- многопоточную обработку нескольких задач одного приложения;
- 32-разрядную обработку данных;
- сжатие данных при записи на магнитный диск;
- защиту памяти.

Важными особенностями **OS/2** является высокопроизводительная файловая система HPFS (High Performance File System), имеющая преимущества для серверов БД (в отличие от MS DOS поддерживаются длинные имена файлов), а также поддержка мультипроцессорной обработки — до 16 процессоров типа Intel и Power PC. **OS/2 Warp** имеет встроенный доступ в Интернет, систему распознавания речи VoiceType, интегрированную версию Lotus Notes Mail для передачи электронной почты. В **OS/2** могут выполняться прикладные программы Windows 3.x, но не могут выполняться приложения, работающие в среде Windows NT.

ОС **UNIX**, созданная корпорацией Bell Laboratory, реализует принцип открытых систем и широкие возможности по объединению в составе одной вычислительной системы разнородных технических и программных средств. UNIX обладает следующими свойствами:

- переносимость прикладных программ с одного компьютера на другой;
- поддержка распределенной обработки данных в сети ЭВМ;
- сочетаемость с процессорами RISC.

UNIX получила распространение для суперкомпьютеров, работающих станций и профессиональных ПК, имеет большое число версий, разработанных различными производителями ПО.

Сетевая ОС — это комплекс программ, обеспечивающий обработку, передачу и хранение данных в сети. Сетевая ОС предоставляет пользователям различные виды сетевых служб (управление файлами, электронная почта, процессы управления сетью и т. д.), поддерживает работу в абонентских системах. Сетевые ОС используют архитектуру «клиент — сервер» или одноранговую архитектуру. Вначале эти ОС поддерживали только ЛВС, сейчас они распространяются и на объединения ЛВС. Наибольшее распространение имеют Windows 95, LAN Server, NetWare.

Windows NT является многозадачной, предназначенной для архитектуры «клиент — сервер» и использования различных протоколов транспортного уровня сетевой ОС (IPX/SPX, TCP/IP, Net Bios), имеет 32-разрядную архитектуру и обеспечивает следующие функции локальной сети:

- возможность каждой абонентской системы в сети быть сервером или клиентом;
- совместную работу группы пользователей;
- адресацию оперативной и внешней памяти большого размера;
- многозадачность и многопоточность обработки данных;
- поддержку мультипроцессорной обработки и др.

В табл. 1.1 приведены характеристики некоторых ОС.

Таблица 1.1. **Характеристики операционных систем**

ОС	Объем ОЗУ, Мбайт	Память на МД, Мбайт	Многопроцессорная обработка	Средства взаимодействия прикладных программ
OS/2 Warp 4.0	12—24	100—300	Нет	Java, Open DOC
MS Windows 2000	64	1024	Есть	Open JL, PM API, WIN 32
Ms Windows XP	128	1024	Есть	OLE, Open JL, Win 32

Операционные оболочки — специальные программы, предназначенные для облегчения общения пользователя с командами ОС. Операционные оболочки имеют текстовый и графический варианты интерфейса конечного пользователя. Наиболее попу-

лярны такие текстовые оболочки MS DOS, как Norton Commander, DOS Navigator, Volkov Commander и др. Эти программы существенно упрощают задание управляющей информации для выполнения команд ОС, уменьшают сложность работы конечного пользователя.

Среди графических оболочек MS DOS можно выделить Windows 3.1, Windows 3.11 for Work Group, которые расширяют набор основных (диспетчер файлов, графический редактор, текстовый редактор и т. д.) и сервисных функций, обеспечивающих создание пользователю систем обработки информации в составе одноранговых локальных сетей.

Расширением базового ПО компьютера является набор сервисных, дополнительно устанавливаемых программ, которые можно классифицировать по функциональному признаку следующим образом:

- программы диагностики работоспособности компьютера;
- антивирусные программы, обеспечивающие защиту компьютера, обнаружение и восстановление зараженных файлов;
- программы обслуживания дисков, обеспечивающие проверку качества поверхности магнитного диска, контроль сохранности файловой системы на логическом и физическом уровнях, сжатие дисков, резервное копирование данных на внешние носители и др.;
- программы архивирования данных, которые обеспечивают процесс сжатия информации с целью уменьшения объема памяти для ее хранения;
- программы обслуживания сети.

Эти программы часто называют утилитами.

Утилиты — программы, служащие для выполнения вспомогательных операций обработки данных или обслуживания компьютеров.

Наибольшее распространение имеют следующие утилиты:

- Norton Utilities;
- программа резервного копирования HP Colorado BackUp for Windows 95.

Антивирусные программы (AVP Toolkit У Касперского; Norton Antivirus for Windows 95 (автоматическое создание резервных копий данных boot — секторов и критических файлов); Anti-Virus (Central Point Soft Ware Inc.) — обнаружение зараженных файлов без исправления как в DOS, так и в Windows и др.).

1.3.2. Инструментарий технологии программирования

Инструментарий технологии программирования — это программные продукты, предназначенные для поддержки технологии программирования (рис. 1.7).

Рис. 1.7. Инструментарий технологии программирования

Средства для создания приложений — совокупность языков и систем программирования, инструментальные среды пользователя, а также различные программные компоненты для отладки и поддержки создаваемых программ.

Язык программирования — это формализованный язык для описания алгоритма решения задач на компьютере. Языки программирования можно условно разделить на следующие классы:

- машинные языки — это языки, воспринимаемые аппаратной частью компьютера (машинные коды);
- машинно-ориентированные языки, отражающие структуру конкретного типа компьютера (ассемблер);
- процедурно-ориентированные языки — это языки, в которых имеется возможность описания программы как совокупности процедур, или подпрограмм (Си, Паскаль и др.);
- проблемно-ориентированные языки, предназначенные для решения задач определенного класса (ЛИСП, ПРОЛОГ).

Другой классификацией языков является их деление на языки, ориентированные на реализацию основ структурного программирования, основанного на модульной структуре программного продукта и типовых управляющих структурах алгоритмов обработки данных различных программных модулей, и объект-

но-ориентированные языки, поддерживающие понятие объектов, их свойств и методов обработки.

Системы программирования включают:

- компилятор (транслятор);
- интегрированную среду разработки программ (не всегда);
- отладчик;
- средства оптимизации кода программ;
- набор библиотек;
- редактор связей;
- сервисные средства (утилиты) (для работы с библиотеками, текстовыми и двоичными файлами);
- справочные системы;
- систему поддержки и управления продуктами программного комплекса.

Компилятор транслирует всю программу без ее выполнения. Трансляторы (интерпретаторы) выполняют пооперационную обработку и выполнение программы.

Отладчики (debugger) — специальные программы, предназначенные для трассировки и анализа выполнения других программ. Трассировка — это обеспечение выполнения в пооператорном варианте.

Инструментальная среда пользователя — это специальные средства, встроенные в пакеты прикладных программ, такие как:

- библиотека функций, процедур, объектов и методов обработки;
- макрокоманды;
- клавишные макросы;
- языковые макросы;
- конструкторы экранных форм и объектов;
- генераторы приложений;
- языки запросов высокого уровня;
- конструкторы меню и др.

Интегрированные среды разработки программ объединяют набор средств для их комплексного применения на технологических этапах создания программы.

Средства для создания информационных систем (ИС) и технологий поддерживают полный цикл проектирования сложной информационной системы или технологии от исследования объекта автоматизации до оформления проектной и прочей документации на информационную систему или технологию. Они позволяют вести коллективную работу над проектом за счет воз-

можности работы в локальной сети, экспорта-импорта любых фрагментов проекта, организации управления проектом.

Одним из современных средств разработки ИС является **CASE-технология** (CASE — Computer-Aided System Engineering) — программный комплекс, автоматизирующий весь технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем.

Средства CASE-технологий делятся:

- на встроенные в систему реализации — все решения по проектированию и реализации привязки к выбранной СУБД;
- независимые от системы реализации — все решения по проектированию ориентированы на унификацию (определение) начальных этапов жизненного цикла программы и средств их документирования, обеспечивают большую гибкость в выборе средств реализации.

Основное достоинство CASE-технологии — это поддержка коллективной работы над проектом за счет возможности работы в локальной сети разработчиков, экспорта (импорта) любых фрагментов проекта, организованного управления проектами.

В некоторых CASE-системах поддерживается кодогенерация программ — создание каркаса программ и создание полного продукта.

Примеры программных продуктов для создания приложений: Visual C++, Delphi, Visual Basic и т. д.

1.3.3. Пакеты прикладных программ

Классификация пакетов прикладных программ (ППП) приведена на рис. 1.8.

Проблемно-ориентированные ППП. Для некоторых предметных областей возможна типизация функций управления, структуры данных и алгоритмов обработки. Это вызвало разработку значительного количества ППП одинакового функционального назначения:

- ППП автоматизированного бухгалтерского учета;
- ППП финансовой деятельности;
- ППП управления персоналом;
- ППП управления производством;
- банковские информационные системы и т. п.

Рис. 1.8. Классификация пакетов прикладных программ

Основные тенденции развития:

- создание программных комплексов в виде автоматизированных рабочих мест (АРМ) управленческого персонала;
- создание интегрированных систем управления предметной областью на базе вычислительных сетей, объединяющих АРМы;
- организация данных больших информационных систем в виде распределенной БД на сети ЭВМ;
- наличие простых языков средств конечного пользователя и др.

ППП автоматизированного проектирования предназначены для поддержки работы конструкторов и технологов, связанных с разработкой чертежей, схем, графическим моделированием и конструированием. Отличительными особенностями этого класса ППП являются высокие требования к аппаратному обеспечению, наличие библиотек встроенных функций, объектов, интерфейсов с графическими системами и БД (AutoCAD).

К **ППП общего назначения** относят:

1. **Системы управления базами данных** (СУБД), обеспечивающие организацию и хранение локальных БД на автономно работающих компьютерах либо централизованное хранение БД на файл-сервере и сетевой доступ к ним. В современных СУБД (на-

пример, MS Access 2.0) содержатся элементы CASE-технологии процесса проектирования, в частности:

- визуализирована схема БД;
- осуществлена автоматическая поддержка целостности БД при различных видах обработки (включение, удаление, модификация);
- предоставляются так называемые мастера, обеспечивающие поддержки процесса проектирования;
- созданы шаблоны (прототипы) структур БД, отчетов, форм и т. д.

2. **Серверы БД** — это ПО, предназначенное для создания и использования при работе в сети интегрированных БД в архитектуре «клиент — сервер». Многопользовательские СУБД в сетевом варианте обработки информации хранят данные на файл-сервере, специально выделенном компьютере, но сама обработка ведется на рабочих станциях. Серверы БД в отличие от этого большую часть обработки (хранение, поиск, извлечение и передачу данных клиенту) данных выполняют самостоятельно, одновременно обеспечивая данными большое число пользователей сети. Общим для различных видов серверов БД является использование реляционного языка SQL (Structured Query Language) для реализации запросов к данным. Большинство серверов БД поддерживает несколько платформ, широкий спектр протоколов передачи данных. Проблемы: обеспечение целостности данных, тиражирование данных по узлам сети и синхронное обновление.

3. **Генераторы отчетов (серверы отчетов)**, обеспечивающие реализацию запросов и формирование отчетов в печатном или экранном виде в условиях сети с архитектурой «клиент — сервер». Сервер отчетов подключается к серверу БД, использующему драйверы сервиса БД (Crystal Reports, Profit for Windows).

4. **Текстовые процессоры**, предназначенные для работы с текстовыми документами. Развитием данного направления являются издательские системы (Microsoft Word).

5. **Табличные процессоры**, являющиеся удобной средой для вычислений конечным пользователем, содержат средства деловой графики, средства специализированной обработки (Microsoft Excel).

6. **Средства презентационной графики** — специализированные программы, предназначенные для создания изображений и их показа на экране, подготовки слайд-фильмов, мультфильмов и их проектирования (Microsoft PowerPoint).

7. **Интегрированные пакеты** набор нескольких программных продуктов, функционально дополняющих друг друга, поддерживающие единые информационные технологии, реализованные на единой операционной и вычислительной платформе (Microsoft Office). Компоненты интегрированных пакетов могут работать изолированно друг от друга, имеют общий интерфейс, благодаря этому их лучше осваивать.

Методо-ориентированные ППП. Данный класс охватывает программные продукты, обеспечивающие независимо от предметной области и функции информационных систем математические, статистические и другие методы решения задач. Наиболее распространены методы математического программирования, решения дифференциальных уравнений, имитационного моделирования, исследования операций (Storm, SYSTAT, SAS и другие).

Офисные ППП. Данный класс охватывает программы, обеспечивающие ориентационное управление деятельностью офиса:

- **органайзеры (планировщики)** — ПО для планирования рабочего времени, составления протоколов встреч, расписаний, ведение записей и телефонной книжки. В состав входят: калькулятор, записная книжка, часы, календарь и т. п.
- **программы-переводчики, средства проверки орфографии, распознавание текста** (Tiger — система распознавания русского языка, Stylus Lingvo Office, содержащий Fine Reader, Stylus for Windows — переводчик на указанный язык, корректор орфографии Lingvo Corrector и резидентный словарь Lingvo);
- **коммуникационные пакеты**, предназначенные для организации взаимодействия пользователей с удаленными абонентами или информационными ресурсами сети;
- **браузеры, средства создания WWW-страниц**;
- **средства электронной почты** (Pegasys Mail).

Настольные издательские системы. Данный класс ПО включает программы (PageMaker, CorelDraw, PhotoShop for Windows и т. д.), обеспечивающие информационную технологию компьютерной издательской деятельности:

- форматирование и редактирование текстов;
- автоматическую разбивку текста на страницы;
- компьютерную верстку печатной страницы;
- монтирование графики;
- подготовку иллюстраций и т. п.

Программные средства мультимедиа. Основное значение данных программных средств — создание и использование аудио- и видеоинформации для расширения информационного пространства пользователя (различные БД компьютерных произведений искусства, библиотеки звуковых записей и т. д.).

Системы искусственного интеллекта:

- программы-оболочки для создания экспертных систем путем наполнения баз знаний и правил логического вывода;
- готовые экспертные системы для принятия решений в рамках определенных предметных областей;
- системы анализа и распознавания речи, текста и т. п.

Примеры систем искусственного интеллекта: FIDE, MYSIN, Guru и др.

Контрольные вопросы

1. Перечислите основные характеристики программ.
2. Приведите существующую классификацию программного обеспечения.
3. Дайте определение и перечислите основные характеристики системного программного обеспечения.
4. Дайте определение и перечислите основные характеристики прикладного программного обеспечения.
5. Дайте определение и охарактеризуйте инструментарий технологии программирования.

Глава 2
ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И ПОДХОДЫ

2.1. Особенности создания программного продукта

2.1.1. Принципы работы с требованиями к программному обеспечению. Проблематика проектирования

Согласно статистическим исследованиям группы Стендиша (Standish Group), в США ежегодно тратится более 250 млрд долларов на разработку приложений информационных технологий в рамках примерно 175 000 проектов. Причем 31 % проектов будет прекращен до завершения. Затраты на 52,7 % проектов составят 189 % от первоначальной оценки. В таком случае американские компании и правительственные учреждения потратят 81 млрд долларов на программные проекты, которые так и не будут завершены. Эти же организации заплатят дополнительно 59 млрд долларов за программные проекты, которые хотя и завершатся, но значительно превысят первоначально отведенное на них время [11].

Первым шагом на пути решения любой проблемы является осознание основных причин ее возникновения. В отчете группы Стендиша указано три наиболее часто встречающихся ключевых фактора, создающих проблемы при проектировании программного обеспечения:

- недостаток исходной информации от клиента — 13 % всех проектов;
- неполные требования и спецификации — 12 % проектов;

- изменение требований и спецификаций — 12 % всех проектов.

В остальном данные сильно расходятся. Конечно, проект может потерпеть неудачу из-за нереалистично составленного графика или неправильно распределенного времени (4 % проектов), нерационального подбора персонала и выделения ресурсов (6 %), несоответствия технологических навыков (7 %), а также по другим причинам. Тем не менее, если считать, что приведенные цифры представляют реальное положение дел в отрасли, то, по крайней мере, неудачи третьей части проектов объясняются причинами, непосредственно связанными со сбором и документированием требований, а также с управлением ими.

Несмотря на то что большинство проектов действительно превышает отведенное время и бюджет, оказалось, что около 9 % проектов крупных компаний были завершены вовремя и в пределах бюджета; аналогичного успеха удалось достигнуть в 16 % проектов мелких компаний. Возникает очевидный вопрос: «Каковы главные “факторы успеха” в этих проектах?» Согласно проведенному исследованию тремя наиболее важными факторами были следующие:

- подключение к разработке пользователя — 16 % всех успешных проектов;
- поддержка со стороны исполнительного руководства — 14 % всех успешных проектов;
- четкая постановка требований — 12 % всех успешных проектов.

Двумя самыми главными проблемами, упоминавшимися почти в половине ответов, оказались:

- спецификации требований;
- управление требованиями клиента.

2.1.2. Оценка стоимости ошибок

Некоторое время назад ряд компаний провел исследование оценки стоимости ошибок, возникающих на разных этапах создания программ. Каждая фирма действовала независимо, тем не менее результаты получены примерно одинаковые: если стоимость усилий, необходимых для обнаружения и устранения ошибок на стадии написания кода, принять за единицу, то стоимость выявления и устранения ошибки на стадии выработки требований бу-

дет в 5—10 раз меньше, а стоимость обнаружения и устранения ошибки на стадии сопровождения — в 20 раз больше (рис. 2.1).

Рис. 2.1. Оценка стоимости ошибок на разных этапах создания ПО

Откуда берется такая высокая стоимость ошибки? Ко времени обнаружения ошибки в требованиях группа разработчиков уже могла потратить время и усилия на создание проекта по этим ошибочным требованиям. В результате проект, вероятно, придется отбросить или пересмотреть [1].

Истинная природа ошибки может быть замаскирована; при проведении тестирования и проверок на данной стадии все думают, что имеют дело с ошибками проектирования, и значительное время и усилия могут быть потрачены впустую.

В зависимости от того, где и когда при работе над проектом разработки программного приложения был обнаружен дефект, цена его может разниться в 50—100 раз. Причина состоит в том, что для его исправления придется затратить средства на некоторые (или все) нижеперечисленные действия.

1. Повторная спецификация.
2. Повторное проектирование.
3. Повторное кодирование.
4. Повторное тестирование.
5. Замена заказа — сообщить клиентам и операторам о необходимости заменить дефектную версию исправленной.
6. Внесение исправлений — выявить и устранить все неточности, вызванные неправильным функционированием ошибочно специфицированной системы, что может потребовать выплаты определенных сумм возмущенным клиентам, повторного выполнения определенных вычислительных задач на ЭВМ и т. п.

7. Списание той части работы (кода, части проектов и т. п.), которая выполнялась с наилучшими побуждениями, но оказалась ненужной, когда обнаружилось, что все это создавалось на основе неверных требований.

8. Отозвание дефектных версий встроенного программного обеспечения и соответствующих руководств. Если принять во внимание, что программное обеспечение сегодня встраивается в различные изделия — от наручных часов и микроволновых печей до автомобилей, — такая замена может коснуться как этих изделий, так и встроенного в них программного обеспечения.

9. Выплаты по гарантийным обязательствам.

10. Ответственность за изделие, если клиент через суд требует возмещение убытка, причиненного некачественным программным продуктом.

11. Затраты на обслуживание   представитель компании должен посетить клиента, чтобы установить новую версию программного обеспечения.

12. Создание документации.

2.1.3. Управление требованиями

Требования задают возможности, которые должна предоставлять система, так что соответствие или несоответствие некоторому множеству требований часто определяет успех или неудачу проекта. Поэтому имеет смысл узнать, что собой представляют требования, записать их, упорядочить и отслеживать их изменения. Определение управления требованиями выглядит следующим образом [3].

Управление требованиями — это систематический подход к выявлению, организации и документированию требований к системе, а также процесс, в ходе которого вырабатывается и обеспечивается соглашение между заказчиком и выполняющей проект группой по поводу меняющихся требований к системе.

Учитывая, что системе будут предъявлены сотни, если не тысячи требований, то очень важно организовать их.

Поскольку невозможно удерживать в памяти более нескольких десятков фактов, для успешного взаимодействия различных участников процесса необходимо обеспечить документирование требований. Требования следует записать так, чтобы они были

доступны для ознакомления; это может быть документ, модель, база данных или листок на доске объявлений.

Кроме того, очень важными факторами являются размер проекта и его сложность. Управление требованиями наиболее важно в больших проектах, в которых участвует множество людей и число требований к проекту велико. Допустим, таких требований 1000. Тогда придется столкнуться с задачами организации, определения приоритетов, управления доступом, а также обеспечения ресурсов для выполнения всех этих требований.

2.1.4. Последовательность работы с требованиями. Анализ проблемы

У пользователя есть технические или бизнес-задачи, для решения которых им нужны программисты. Задача последних состоит в том, чтобы понять проблемы пользователей в их собственной проблемной плоскости и на их языке и построить системы, удовлетворяющие их требованиям. Для понимания проблемы пользователей существует ряд профессиональных приемов, о которых пойдет речь ниже [3].

Программисты должны понять потребности пользователей и других заинтересованных лиц, на чью жизнь повлияет создание программы.

Следующим шагом осуществляется переход в область решения — непосредственно к программированию. Однако для начала будет полезно сформулировать знания о предметной области. На данном этапе составляется список функций, которые должна реализовывать система.

Для того чтобы провести анализ, полезно определить, что же собственно представляет собой проблема. По определению Гауса и Вайнберга [11], проблема — это разница между желаемым и воспринимаемым.

Иногда самым простым решением является изменение бизнес-процесса, а не создание новой системы. Как всегда, начинать следует с определения цели. Цель анализа состоит в том, чтобы добиться лучшего понимания решаемой проблемы до начала разработки. Для этого необходимо осуществить следующие пять этапов.

1. Достигнуть соглашения об определении проблемы.

2. Выделить основные причины — вопросы, стоящие за проблемой.

3. Выявить заинтересованных лиц и пользователей.

4. Определить границу системы решения.

5. Выявить ограничения, которые необходимо наложить на решение.

Этап 1. Достижение соглашения об определении проблемы.

Первый шаг состоит в достижении соглашения об определении проблемы, которую необходимо решить. Один из простейших способов заключается в том, чтобы просто записать проблему и выяснить, все ли согласны с такой постановкой.

В рамках этого процесса зачастую полезно рассмотреть преимущества предлагаемого решения, причем их следует описывать на языке клиентов/пользователей. Это обеспечивает дополнительную содержательную основу для понимания реальной проблемы. Рассматривая эти преимущества с точки зрения клиента, программисты также достигают лучшего понимания их взгляда на проблему в целом.

Часто бывает полезно записать проблему в стандартной форме (табл. 2.1). Создание подобной таблицы является простым, но действенным средством, чтобы удостовериться в том, что все участники проекта работают вместе над осуществлением общей цели.

Таблица 2.1. **Структурирование проблемы**

Элемент	Описание
Проблема	Описание проблемы
Воздействует на что (кого) и результатом чего является	Указание лиц, на которых оказывает влияние данная проблема. Описание воздействия данной проблемы на заинтересованных лиц и бизнес-деятельность
Выигрыш от решения может состоять в следующем	Указание предлагаемого решения. Список основных предоставляемых решением преимуществ

Этап 2. Выявление основных причин — вопросов, стоящих за проблемой.

На данном этапе важно понять корневые причины, лежащие в основе проблемы, и ее проявления.

Например, электронный магазин решил бороться с проблемой недостаточной прибыльности. Для этого был проведен ана-

лиз причин плохих продаж. Получено, что следующие причины ведут к слишком большим остаткам продукции на складе:

1) устаревшие готовые изделия;
2) неправильные заказы на покупку;
3) повреждения при доставке;
4) производственные дефекты;
5) возвраты клиентами;
6) прочее.

Однако нужно ли устранять все эти причины? Зачастую нет. Некоторые корневые причины просто не стоят того, чтобы их устранять. Нужно определить влияние каждой корневой причины и устранять только те, которые наиболее серьезно влияют на саму проблему. В примере, допустим, наибольшее влияние оказывает корневая причина «Неправильные заказы на покупку».

Этап 3. Выявление заинтересованных лиц и пользователей.
В этом процессе могут помочь ответы на следующие вопросы:

- Кто является пользователем системы?
- Кто является заказчиком (экономическим покупателем) системы?
- На кого еще окажут влияние результаты работы системы?
- Кто будет оценивать и принимать систему, когда она будет представлена и развернута?
- Существуют ли другие внешние или внутренние пользователи системы, чьи потребности следует учесть?
- Кто будет заниматься сопровождением новой системы?
- Не забыли ли мы кого-нибудь?

Этап 4. Определение границ системы.
Мир делится на две части (рис. 2.2):

- создаваемая система;
- то, что взаимодействует с системой, — фактор.

Рис. 2.2. Границы системы

Очень важно правильно определить факторы. Для этого следует ответить на приводимые ниже вопросы.

- Кто будет управлять системой?
- Кто будет осуществлять сопровождение системы?
- Откуда система получает информацию?
- Какие внешние системы будут взаимодействовать с системой?

Этап 5. Выявление ограничений, налагаемых на решение.

Ограничения уменьшают степень свободы, которой располагают разработчики при реализации решения. Каждое ограничение может существенно сузить возможность создания предполагаемого решения. Следовательно, в процессе планирования необходимо тщательно изучить все ограничения (табл. 2.2).

Таблица 2.2. **Возможные источники ограничений системы**

Источник	Образцы вопросов
Экономический	Какие финансовые или бюджетные ограничения следует учесть? Существуют ли соображения, касающиеся себестоимости и ценообразования? Существуют ли вопросы лицензирования?
Политический	Существуют ли внешние или внутренние политические вопросы, влияющие на потенциальное решение? Существуют ли проблемы в отношениях между подразделениями?
Технический	Существуют ли ограничения в выборе технологий? Должны ли мы работать в рамках существующих платформ или технологий? Запрещено ли использование любых новых технологий? Должны ли мы использовать какие-либо закупаемые пакеты программного обеспечения?
Системный	Будет ли решение создаваться для существующих систем? Должны ли разработчики обеспечивать совместимость с существующими решениями? Какие операционные системы и среды должны поддерживаться?
Эксплуатационный	Существуют ли ограничения информационной среды или правовые ограничения? Юридические ограничения? Требования безопасности? Какими другими стандартами ограничены разработчики?
Графики и ресурсы	Ограничена ли команда программистов существующими ресурсами? Могут ли разработчики привлекать сотрудников со стороны? Можно ли увеличить штат? Временно?

2.1.5. Преграды на пути выявления требований

Синдром «да, но...

Одну из самых неприятных проблем, с которыми сталкиваются разработчики, можно назвать синдромом «да, но...» [3]. Это первичная наблюдаемая реакция пользователя на каждый разработанный фрагмент программного обеспечения, которая могла бы быть выражена следующими словами:

- «О, это действительно здорово! Можем реально использовать это, классная работа, молодцы, мальчики!» и т. д.
- «Да, но как насчет?.. Нельзя ли было?.. А что, если?..

Причина синдрома «да, но...» кроется глубоко в природе проектирования программного обеспечения как интеллектуального неосязаемого процесса. Проблема усугубляется тем, что команда разработчиков крайне редко предоставляет что-либо пользователям для обсуждения до окончания разработки (создания программного кода).

Реакция пользователей является следствием человеческой природы. Подобную реакцию можно часто наблюдать и при других повседневных обстоятельствах. Пользователи никогда ранее не видели новую систему или что-либо подобное; они не понимают, что программисты подразумевают, когда описывают ее. И вот теперь она перед ними — впервые после стольких месяцев (или лет) ожидания они имеют возможность взаимодействовать с системой. И оказывается, что это не совсем то, чего они ожидали!

Как это ни грустно, но нужно принять факт существования синдрома «да, но...» в качестве объективной реальности и сделать некоторые выводы, которые помогут членам команды смягчить влияние этого синдрома в будущих проектах:

- синдром «да, но...» является следствием человеческой природы и неотъемлемой частью разработки любого приложения;
- разработчики могут существенно уменьшить воздействие этого синдрома путем применения методов, которые выявят эти «но» как можно раньше. Выявив их на более ранних этапах, можно направить большую часть усилий на разработку программ, которые уже прошли тест на «да, но...

Синдром «пользователь и разработчик»

Синдром «пользователь и разработчик» является следствием расхождения взглядов пользователей и разработчиков. Пользователи и разработчики, как правило, принадлежат к различным мирам, говорят на разных языках и имеют различный опыт, мотивацию и цели. Предлагаются рекомендации по смягчению данной ситуации (табл. 2.3).

Таблица 2.3. **Рекомендации решения проблемы**

Проблема	Решение
Пользователи не знают, чего хотят, а если и знают, то не могут это выразить	Признать пользователя экспертом в предметной области и ценить его в этом качестве; пытаться использовать альтернативные методы общения и выявления требований
Пользователи думают, что они знают, чего хотят, до тех пор, пока разработчики не предоставят им то, что те якобы хотели	Как можно раньше предлагать альтернативные методы выявления: раскадровку, ролевые игры, прототипы и т. п.
Аналитики думают, что они понимают проблемы пользователя лучше его самого	Поставить аналитика на место пользователя. Провести ролевую игру в течение часа или всего дня
Все считают, что другие руководствуются политическими мотивами	Такова человеческая натура, поэтому пусть все остается, как есть

Функции

Использование функций — удобный способ описания возможностей без лишних подробностей.

Такой подход имеет недостаток. Если команда при обсуждении не поймет, какая потребность стоит за функцией, это может привести к неприятным последствиям. Тем не менее это высокий уровень абстракции, удобный для описания возможностей системы.

Рекомендуемое количество функций, которое дает полное представление о разрабатываемой системе, — 25—99, однако желательно, чтобы их число не превышало 50.

После того как все функции перечислены, можно приступить к принятию решения вида «отложить до следующей версии», «реализовать немедленно», «полностью отвергнуть» или «исследовать дополнительно». Это процесс корректировки мас-

штаба лучше проводить на уровне функций, а не на уровне требований, иначе можно увязнуть в деталях.

Чтобы лучше работать с этой информацией, введем понятие атрибутов функций — элементов данных, которые обеспечат дополнительную информацию о каждой функции (табл. 2.4).

Таблица 2.4. **Атрибуты функций**

Атрибут	Описание/примеры значений функции
Статус	• Предлагаемая • Утверждаемая • Включенная
Приоритет/полезность	• Критичная • Важная • Полезная
Трудоемкость	• Низкий уровень • Средний уровень • Высокий уровень
Риск	Вероятность того, что функция вызовет нежелательные последствия, такие как увеличение расходов, отставание от графика или даже закрытие проекта. • Высокий • Средний • Низкий
Стабильность	Вероятность того, что данная функция будет меняться или будет меняться ее понимание командой. • Высокая • Средняя • Низкая
Целевая версия	Указание версии продукта, в которой впервые появится реализация данной функции
Назначение	Информация для разработчиков
Обоснование	Ссылка на источник запрашиваемой функции

Методы выявления требований:
- интервьюирование и анкетирование;
- совещания, посвященные требованиям;
- мозговой штурм и отбор идей;
- раскадровки;
- прецеденты;
- обыгрывание ролей;
- создание прототипов.

Интервьюирование и анкетирование. Интервью помогает понять проблему, не оказывая влияния на ответы пользователя. Ниже приведены примеры контекстно-свободных вопросов:

- почему существует проблема?
- как она решается в настоящее время?
- как заказчик хотел бы ее решать?
- кто такие пользователи?
- каковы их навыки в компьютерной области?

После этого интервьюер перечисляет основные пункты, чтобы проверить, все ли было правильно понято: «Итак, вы сказали мне...» (перечисляются описанные заказчиком проблемы своими словами). «Какие еще проблемы вы испытываете?»

Правила подготовки интервью:

1. Все вопросы должны быть составлены заранее.
2. Перед интервью необходимо познакомиться с информацией о клиенте.
3. Кратко записывайте ответы.

Совещания. Хорошо проведенное совещание по вопросам требований имеет множество преимуществ:

- помогает создать команду, подчиненную одной цели — успеху данного проекта;
- все заинтересованные лица получают возможность высказать свое мнение, никто не остается в стороне;
- формирует соглашение между заинтересованными лицами и командой разработчиков по поводу того, что должно делать приложение;
- может высветить и разрешить политические вопросы, которые влияют на успех проекта;
- результат, предварительное определение системы на уровне функций, немедленно становится известным.

Все материалы к совещанию должны быть предоставлены участникам заранее.

Мозговой штурм. Все основные участники собираются в одной комнате, им раздаются материалы для заметок — не менее 25 листов формата от 7 × 12 до 12 × 17.

Правила проведения мозгового штурма:

1. Не допускается критика или дебаты.
2. Дайте свободу фантазии.
3. Генерируйте как можно больше идей.
4. Переделывайте и комбинируйте идеи.

Идеи каждый записывает на листок; позже они оглашаются. Критиковать их нельзя, дискутировать тоже не стоит, чтобы никого не обидеть. Идею надо попробовать развить или скомбинировать с какой-либо другой; если ничего не получится, то ее просто изымут из рассмотрения.

Идеи обязательно записывают на бумагу по следующим причинам:

1. Сохраняется авторская формулировка.

2. Существует гарантия, что они не будут утрачены.

3. Есть перспектива развития идеи в дальнейшем.

4. Обеспечивается непрерывный творческий процесс — не надо ждать, пока секретарь запишет мысль участника.

Раскадровка. Цель раскадровки в раннем выявлении реакции типа «да, но... Существует три типа раскадровок.

1. Пассивные. Пользователю излагают некую историю с применением рисунков, схем, картинок с экрана.

2. Активные. Пользователю показывают «еще не созданный фильм» с применением анимации или слайдов.

3. Интерактивные. Пользователь приобретает реальный опыт взаимодействия с системой (почти так же, как и на практике).

Применение прецедентов. Рисуются схемы с факторами; в овале пишется вид выполняемого ими действия (рис. 2.3).

Рис. 2.3. Схема применения прецедентов

Обыгрывание ролей. Этот способ позволяет разработчику прочувствовать проблемы пользователя. Разработчик на время становится на место клиента, причем можно заранее написать сценарий для программистов, по которому они попытаются выполнить действия пользователей.

Также очень эффективным способом является составление сценариев на основе CRC-карточек. Карточки описывают объект, его поведение и взаимодействие с другими объектами системы. Участники делят карточки между собой и используют их для ролевой игры, выполняя действия согласно предписаниям. Эффективный способ для выявления проблем проектирования системы.

Прототипы требований. Прототип требований к ПО — это частичная реализация системы, созданная для того, чтобы помочь разработчикам, пользователям и клиентам точнее определить требования к системе. Этот метод также помогает решить проблему «да, но...» [3].

Таким образом, на базе знаний по составлению требований к ПО и при их корректном постоянном применении разработчик лучше поймет проблемы клиента, как следствие, создаст качественный программный продукт, отвечающий потребностям пользователей, и снизит риск краха проекта из-за функционального несоответствия приложения.

2.2. Оценка качества процессов создания программного обеспечения

Переход от штучной разработки программных продуктов к промышленному программированию обусловил повышение требований к качеству создаваемого ПО. В настоящее время существует несколько стандартов, связанных с оценкой качества этих процессов, которое обеспечивает компания-разработчик. К наиболее интересным для рассмотрения относятся:

- международные стандарты серии ISO 9000 (ISO 9000 — ISO 9004);
- CMM — Capability Maturity Model — модель зрелости (совершенствования) процессов создания программного обеспечения, предложенная SEI (Software Engineering Institute —

институт программирования при университете Карнеги — Меллон);
- процесс сертификации программ на базе информации об их использовании.

2.2.1. Серия стандартов ISO 9000

Одной из важнейших проблем обеспечения качества программных средств является формализация характеристик качества и методология их оценки. Для определения адекватности качества функционирования, наличия технических возможностей программных средств к взаимодействию, совершенствованию и развитию необходимо использовать стандарты в области оценки характеристик их качества. Основой регламентирования показателей качества программных средств ранее являлся международный стандарт ISO 9126:1991 (ГОСТ Р ИСО/МЭК 9126—93) «Информационная технология. Оценка программного продукта. Характеристики качества и руководство по их применению». Завершается разработка и формализован последний проект состоящего из четырех частей стандарта ISO 9126-1—ISO 9126-4 для замены небольшой редакции 1991 г. Проект состоит из следующих частей под общим заголовком «Информационная технология — характеристики и метрики качества программного обеспечения»: «Часть 1. Характеристики и субхарактеристики качества; Часть 2. Внешние метрики качества»; «Часть 3. Внутренние метрики качества»; «Часть 4. Метрики качества в использовании» [32].

В России в области обеспечения жизненного цикла и качества сложных комплексов программ в основном применяется группа устаревших ГОСТов, которые отстают от мирового уровня на 5—10 лет.

Первая часть стандарта — ISO 9126-1 — распределяет атрибуты качества программных средств по шести характеристикам, используемым в остальных частях стандарта. Исходя из принципиальных возможностей их измерения все характеристики могут быть объединены в три группы, к которым применимы разные категории метрик:
- категорийным, или описательным (номинальным), метрикам наиболее адекватны функциональные возможности программных средств;

- количественные метрики применимы для измерения надежности и эффективности сложных комплексов программ;
- качественные метрики в наибольшей степени соответствуют практичности, сопровождаемости и мобильности программных средств.

В части стандарта ISO 9126-1 даются определения с уточнениями из остальных его частей для каждой характеристики программного средства, а также для субхарактеристик качества.

За последние несколько лет создано множество стандартов ISO, регламентирующих процессы и продукты жизненного цикла программных средств и баз данных, которые могут служить основой для систем обеспечения качества программных продуктов.

Вторая и третья части стандарта — ISO 9126-2 и ISO 9126-3 — посвящены формализации соответственно внешних и внутренних метрик характеристик качества сложных программных средств. Все таблицы содержат унифицированную рубрикацию, где отражены имя и назначение метрики; метод ее применения; способ измерения, тип шкалы метрики; тип измеряемой величины; исходные данные для измерения и сравнения; а также этапы жизненного цикла программного средства (по ISO 12207), к которым применима метрика.

Четвертая часть стандарта — ISO 9126-4 — предназначена для покупателей, поставщиков, разработчиков, сопровождающих, пользователей и менеджеров качества программных средств. В ней обосновываются и комментируются выделенные показатели сферы (контекста) использования программных средств и группы выбранных метрик для пользователей.

Выбор показателей качества

Исходными данными и высшим приоритетом при выборе показателей качества в большинстве случаев являются назначение, функции и функциональная пригодность соответствующего программного средства. Достаточно полное и корректное описание этих свойств должно служить базой для определения значений большинства остальных характеристик и атрибутов качества. Принципиальные и технические возможности и точность измерения значений атрибутов характеристик качества всегда ограничены в соответствии с их содержанием. Это определяет рациональные диапазоны значений каждого атрибута, которые могут

быть выбраны на основе здравого смысла, а также путем анализа прецедентов в спецификациях требований реальных проектов.

Процессы выбора и установления метрик и шкал для описания характеристик качества программных средств можно разделить на два этапа:

- выбор и обоснование набора исходных данных, отражающих общие особенности и этапы жизненного цикла проекта программного средства и его потребителей, каждый из которых влияет на определенные характеристики качества комплекса программ;
- выбор, установление и утверждение конкретных метрик и шкал измерения характеристик и атрибутов качества проекта для их последующей оценки и сопоставления с требованиями спецификаций в процессе квалификационных испытаний или сертификации на определенных этапах жизненного цикла программного средства.

На первом этапе за основу следует брать всю базовую номенклатуру характеристик, субхарактеристик и атрибутов, стандартизированных в ISO 9126. Их описания желательно предварительно упорядочить по приоритетам с учетом назначения и сферы применения конкретного проекта программного средства. Далее необходимо выделить и ранжировать по приоритетам потребителей, которым необходимы определенные показатели качества проекта программного средства с учетом их специализации и профессиональных интересов. Подготовка исходных данных завершается выделением номенклатуры базовых, приоритетных показателей качества, определяющих функциональную пригодность программного средства для определенных потребителей.

На втором этапе, после фиксирования исходных данных, которое должен выполнить потребитель оценок качества, процессы выбора номенклатуры и метрик начинаются с ранжирования характеристик и субхарактеристик для конкретного проекта и их потребителя. Далее этими специалистами для каждого из отобранных показателей должна быть установлена и согласована метрика и шкала оценок субхарактеристик и их атрибутов для проекта и потребителя результатов анализа. Для показателей, представляемых качественными признаками, желательно определить и зафиксировать в спецификациях описания условий, при которых следует считать, что данная характеристика реализуется в программном средстве. Выбранные значения характеристик качества и их атрибутов должны быть предварительно про-

верены разработчиками на их реализуемость с учетом доступных ресурсов конкретного проекта и при необходимости откорректированы.

Оценка качества

Методологии и стандартизации оценки характеристик качества готовых программных средств и их компонентов (программного продукта) на различных этапах жизненного цикла посвящен международный стандарт ISO 14598, состоящий из шести частей. Рекомендуется следующая общая схема процессов оценки характеристик качества программ:

- установка исходных требований для оценки — определение целей испытаний, идентификация типа метрик программного средства, выделение адекватных показателей и требуемых значений атрибутов качества;
- селекция метрик качества, установление рейтингов и уровней приоритета метрик субхарактеристик и атрибутов, выделение критериев для проведения экспертиз и измерений;
- планирование и проектирование процессов оценки характеристик и атрибутов качества в жизненном цикле программного средства;
- выполнение измерений для оценки, сравнение результатов с критериями и требованиями, обобщение и оценка результатов.

Для каждой характеристики качества рекомендуется формировать меры и шкалу измерений с выделением требуемых, допустимых и неудовлетворительных значений. Реализация процессов оценки должна коррелировать с этапами жизненного цикла конкретного проекта программного средства в соответствии с применяемой, адаптированной версией стандарта ISO 12207.

Функциональная пригодность — наиболее неопределенная и объективно трудно оцениваемая субхарактеристика программного средства. Области применения, номенклатура и функции комплексов программ охватывают столь разнообразные сферы деятельности человека, что невозможно выделить и унифицировать небольшое число атрибутов для оценки и сравнения этой субхарактеристики в различных комплексах программ.

Оценка корректности программных средств состоит в формальном определении степени соответствия комплекса реализованных программ исходным требованиям контракта, техниче-

ского задания и спецификаций на программное средство и его компоненты. Путем верификации должно быть определено соответствие исходным требованиям всей совокупности компонентов комплекса программ, вплоть до модулей и текстов программ и описаний данных.

Оценка способности к взаимодействию состоит в определении качества совместной работы компонентов программных средств и баз данных с другими прикладными системами и компонентами на различных вычислительных платформах, а также взаимодействия с пользователями в стиле, удобном для перехода от одной вычислительной системы к другой с подобными функциями.

Оценка защищенности программных средств включает определение полноты использования доступных методов и средств защиты программного средства от потенциальных угроз и достигнутой при этом безопасности функционирования информационной системы. Наиболее широко и детально методологические и системные задачи оценки комплексной защиты информационных систем изложены в трех частях стандарта ISO 15408:1999-1—ISO 15408:1999-3 «Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий».

Оценка надежности — измерение количественных метрик атрибутов субхарактеристик в использовании: завершенности, устойчивости к дефектам, восстанавливаемости и доступности/готовности.

Потребность в ресурсах памяти и производительности компьютера в процессе решения задач значительно изменяется в зависимости от состава и объема исходных данных. Для корректного определения предельной пропускной способности информационной системы с данным программным средством нужно измерить экстремальные и средние значения длительностей исполнения функциональных групп программ и маршруты, на которых они достигаются. Если предварительно в процессе проектирования производительность компьютера не оценивалась, то, скорее всего, понадобится большая доработка или даже замена компьютера на более быстродействующий.

Оценка практичности программных средств проводится экспертами и включает определение понятности, простоты использования, изучаемости и привлекательности программного средства. В основном это качественная (и субъективная) оценка в баллах, однако некоторые атрибуты можно оценить количественно по трудоемкости и длительности выполнения операций

при использовании программного средства, а также по объему документации, необходимой для их изучения.

Сопровождаемость можно оценивать полнотой и достоверностью документации о состояниях программного средства и его компонентов, всех предполагаемых и выполненных изменениях, позволяющей установить текущее состояние версий программ в любой момент времени и историю их развития. Она должна определять стратегию, стандарты, процедуры, распределение ресурсов и планы создания, изменения и применения документов на программы и данные.

Оценка мобильности — качественное определение экспертами адаптируемости, простоты установки, совместимости и замещаемости программ, выражаемое в баллах. Количественно эту характеристику программного средства и совокупность ее атрибутов можно (и целесообразно) оценить в экономических показателях: стоимости, трудоемкости и длительности реализации процедур переноса на иные платформы определенной совокупности программ и данных.

Система управления качеством

Выбор характеристик и оценка качества программных средств — лишь одна из задач в области обеспечения качества продукции, выпускаемой компаниями разработчиками ПО. Комплексное решение задач обеспечения качества программных средств предполагает разработку и внедрение той или иной системы управления качеством. В мировой практике наибольшее распространение получила система, основанная на международных стандартах серии ISO 9000, включающей десяток с лишним документов, в том числе стандарт, регламентирующий обеспечение качества ПО (ISO 9000/3). Эти стандарты должны служить руководством для ведущих специалистов компаний, разрабатывающих ПО на заказ.

Определения характеристик и субхарактеристик качества (ISO 9126-1):

Функциональные возможности — способность программного средства обеспечивать решение задач, удовлетворяющих сформулированные потребности заказчиков и пользователей при применении комплекса программ в заданных условиях.

Функциональная пригодность — набор и описания субхарактеристики и ее атрибутов, определяющие назначение, номенк-

латуру, основные, необходимые и достаточные функции программного средства, соответствующие техническому заданию и спецификациям требований заказчика или потенциального пользователя.

Правильность (корректность) — способность программного средства обеспечивать правильные или приемлемые для пользователя результаты и внешние эффекты.

Способность к взаимодействию — свойство программных средств и их компонентов взаимодействовать с одной или большим числом компонентов внутренней и внешней среды.

Защищенность — способность компонентов программного средства защищать программы и информацию от любых негативных воздействий.

Надежность — обеспечение комплексом программ достаточно низкой вероятности отказа в процессе функционирования программного средства в реальном времени.

Эффективность — свойства программного средства, обеспечивающие требуемую производительность решения функциональных задач, с учетом количества используемых вычислительных ресурсов в установленных условиях.

Практичность (применимость) — свойства программного средства, обусловливающие сложность его понимания, изучения и использования, а также привлекательность для квалифицированных пользователей при применении в указанных условиях.

Сопровождаемость — приспособленность программного средства к модификации и изменению конфигурации и функций.

Мобильность — подготовленность программного средства к переносу из одной аппаратно-операционной среды в другую.

2.2.2. CMM

В ноябре 1986 г. американский институт Software Engineering Institute (SEI) совместно с Mitre Corporation начал разработку обзора зрелости процессов разработки программного обеспечения, который был предназначен для помощи в улучшении их внутренних процессов [31].

Разработка такого обзора была вызвана запросом американского федерального правительства на предоставление метода оценки субподрядчиков для разработки ПО. Реальная же проблема состояла в неспособности управлять большими проекта-

ми. Во многих компаниях проекты выполнялись со значительным опозданием и с превышением запланированного бюджета. Необходимо было найти решение данной проблемы.

В сентябре 1987 г. SEI выпустил краткий обзор процессов разработки ПО с описанием их уровней зрелости, а также опросник, предназначавшийся для выявления областей в компании, в которых были необходимы улучшения. Однако большинство компаний рассматривало данный опросник в качестве готовой модели, вследствие этого через 4 года вопросник был преобразован в реальную модель, Capability Maturity Model for Software (CMM). Первая версия CMM (Version 1.0), вышедшая в 1991 г., в 1992 г. была пересмотрена участниками рабочей встречи, в которой принимали участие около 200 специалистов в области ПО, и членами общества разработчиков.

В результате был выпущен стандарт CMM, Version 1.1, который до настоящего времени активно используется во всем мире.

Причины такого интереса к CMM понятны. Несмотря на то что и сами разработчики ПО, и их руководство зачастую очень хорошо знают свои постоянные проблемы, они не могут прийти к единому мнению о том, какие изменения необходимы компании в первую очередь. Без выработки единой стратегии проведения улучшений руководство не может найти взаимопонимания со своими сотрудниками относительно наиболее приоритетных задач по улучшению. Для достижения максимального результата от усилий, потраченных на улучшение процессов, необходимо иметь поэтапную стратегию развития, которая позволит улучшать зрелость процессов разработки постепенно, эволюционным путем.

Постоянное улучшение процессов базируется на постепенном взращивании культуры компании, а не на проведении революционных инноваций. В CMM представлена схема такого постепенного улучшения, разделенная по пяти уровням зрелости процессов. Эти пять уровней представляют собой шкалу (рис. 2.4) для оценки уровня зрелости процессов разработки ПО в компании и для измерения их параметров.

Приведем основные характеристики каждого уровня:

1. Начальный уровень. Процесс разработки носит хаотический характер. Определены лишь немногие из процессов, и успех проектов зависит от конкретных исполнителей.

2. Повторяемость. Установлены основные процессы управления проектами: отслеживание затрат, графика работ и функцио-

Рис. 2.4. Принцип последовательного повышения уровня зрелости: возможности развития организации

нальности. Упорядочены некоторые процессы, необходимые для того, чтобы повторить предыдущие достижения на аналогичных проектах (проектах с аналогичными приложениями).

3. Разработка. Процессы разработки ПО и управления проектами описаны и внедрены в единую систему процессов компании. Во всех проектах используется стандартный для организации процесс разработки и поддержки ПО, адаптированный под конкретный проект.

4. Контроль. Собираются детальные количественные данные по функционированию процессов разработки и качеству конечного продукта. Анализируется значение и динамика этих данных.

5. Улучшение качества. Постоянное улучшение процессов основывается на количественных данных по процессам и на пробном внедрении новых идей и технологий.

2.2.3. Процесс сертификации программ на базе информации об их использовании

Это новый подход к сертификации, который, по утверждению его разработчиков, дает надежные гарантии качества для коммерческих программных пакетов [30].

Методов сертификации качества программного обеспечения становится все больше и больше. Популярные подходы, основанные на процессах, такие как ISO 9000 и SEI-CMM, вынуждают создателей программного обеспечения жестко придерживаться выбранных стандартов и процессов разработки. Такие подходы зачастую требуют участия аудиторов, которые проверяют документацию производителя и то, как он выполняет данное им обещание. Но даже если аудитор по сертификации может убедиться в чистоте намерений производителя, одна эта проверка вовсе не гарантирует, что созданное программное обеспечение будет высокого качества.

Принимая во внимание эту проблему, можно предложить методологию сертификации, которая не требует, чтобы производитель давал клятвы, а аудиторы проверяли, насколько эти клятвы выполняются. Можно с уверенностью сказать, что только полностью независимая сертификация продукта и есть тот подход, которому могут доверять потребители.

Пригласив третью сторону для выдачи сертификатов качества на программное обеспечение, разаботчики снимут с себя ответственность, как это делает врач, приглашая для консультации другого доктора или проводя еще один анализ. Конечные пользователи тоже выиграют от того, что беспристрастная оценка продукту будет даваться независимыми организациями. Таким образом, есть разумные причины для появления учреждений, которые будут сертифицировать качество программного обеспечения.

Организации, выполняющие такую сертификацию, называются лабораториями по сертификации программного обеспечения (Software Certification Laboratories — SCL). Их достоинство в том, что они смогут предоставить равные возможности всем производителям, если, конечно, каждый продукт будет тестироваться в равных условиях.

SCL до сих пор не получили широкого распространения именно из-за ответственности, которую берет на себя организация, выдающая сертификат. Если сертифицированное ПО не оправдывает ожиданий при использовании его в реальном производственном процессе, удостоверитель, выдавший сертификат, несет за это определенную ответственность. Суды в США печально известны своей исключительной требовательностью к тем, кто называет себя профессионалами, будь то врачи, юристы и инженеры.

Вот и SCL могут потерять свое реноме и лишиться права заниматься сертификацией в случае ошибки [2]. Чтобы несколько снизить этот риск, необходимо использовать точные методы при принятии решений о сертификации — в идеале автоматизированные. К сожалению, даже если SCL использует лучшие методики статистического анализа и динамического тестирования, не всегда можно понять, какому реальному воздействию подвергнется программа в руках пользователей. Таким образом, SCL сталкивается с проблемой точного определения того, как будет вести себя программная система — самое главное, что требуется от уполномоченных по выдаче сертификатов.

Процесс сертификации, о котором идет речь, использует автоматизированные процедуры, что позволяет отказаться от услуг аудиторов. Этот процесс использует ресурсы тестирования конечными пользователями, полагаясь на доказавшие свою состоятельность методы, подобные тем, благодаря которым Linux стала самой популярной и надежной из всех разновидностей Unix [3, 4]. Предлагаемый процесс, полностью основанный на продуктах, как таковых, оценивает качество работы программного обеспечения, а не качество процессов создания кода.

Модель ограниченной гарантии на ПО

Компонентная разработка программ (CBSE — component-based software engineering) предполагает создание программных систем из фрагментов. Достоинства различных стратегий обсуждаются многие годы, но нет никаких признаков того, что компонентная разработка вот-вот станет стандартным подходом к созданию программных систем.

Уверения в надежности компонентов должны исходить не только от их автора. SCL, которые будут проводить собственное тестирование при сертификации программного обеспечения, столкнутся с теми же проблемами, которые стоят сейчас перед его разработчиками: неадекватный уровень тестирования и неверные предположения о том, как именно будет использоваться продукт. Действительно, уровень тестирования, который SCL в состоянии выполнить и не разориться, может оказаться даже ниже того, который провел сам разработчик. Таким образом, было бы ошибкой думать, что SCL могут выдавать сертификаты, основываясь только на результатах собственного тестирования.

Гарантии на ПО должны опираться на результаты его использования в условиях реального мира, причем в массовом порядке. Подобное тестирование продемонстрирует стабильность продукта в конкретных средах и в конкретных секторах рынка. Без этого гарантии будут или слишком узкими, или не заслуживающими доверия.

Сертификация ПО с участием пользователей

Если SCL и разработчики не могут выполнять адекватное тестирование продукта, то кто же может? Для решения этой задачи можно объединить пользователей, только надо сделать так, что пользователям это будет выгодно. После этого встает вопрос, как лучше всего использовать их возможности сделать реальными гарантии на программное обеспечение.

Сертификация продуктов с помощью самих пользователей — это наилучший способ предоставить гарантии на ПО, поскольку при этом продукт оценивается в ситуациях, когда он применяется в условиях реального мира.

Такой подход предполагает также сбор информации о характере применения продукта. Можно измерить частоту, с которой вызываются определенные функции, и просто собирать файлы ввода, которые используются для создания профилей. Эти наборы информации будут служить в качестве основы для выпуска ограниченных гарантий на ПО.

Производитель программного обеспечения представляет конечный продукт для включения в него резидентного инструментария тестирования, в результате этого создается специальная, «инструментальная» копия. Производитель предоставляет копии SCL, которая передает инструментальную копию предварительно отобранным пользователям, работающим в разных отраслях. Эти тестировщики будут использовать продукт по согласованию с SCL. Предлагаемая Microsoft модель бета-тестирования — прекрасный пример того, как среди всех желающих отобрать только тех, кто действительно сможет предоставить самый большой объем полезной информации.

Заметим, что было бы возможно и потенциально предпочтительно для SCL самой встраивать необходимый инструментальный код, не привлекая для этого производителя. Это можно было бы сделать, если бы специалисты SCL получили доступ к определенным файлам с исходными текстами или к интерфей-

сам. По сути, независимая организация в первую очередь призвана сделать так, чтобы производитель как можно меньше знал о пользователях, принимающих участие в тестировании, и не мог влиять на результаты процесса сертификации.

Периодически SCL собирает информацию от пользователей и суммирует ее. Это позволяет SCL предоставить разработчику статистические данные о том, как использовался продукт и как он вел себя во время испытаний. При этом SCL может собирать статистику от участников тестирования таким образом, чтобы восстановить личность конкретного пользователя было бы невозможно.

SCL должна определить, получит ли данный продукт гарантию и что должно быть указано в этой гарантии: платформа, секторы рынка, операционные среды. Со временем, собрав дополнительные данные, SCL могла бы предложить более широкую гарантию и снизить указанные в ней ограничения.

Особую сложность представляют системы, критичные к безопасности. Нельзя рассчитывать на то, что SCL найдет желающих полетать на самолете, на котором установлены несертифицированные программы управления полетами, или использовать медицинские приборы с несертифицированными программами только потому, что необходимо выяснить, имеют ли эти программы право на сертификат. По мнению авторов, SCL сначала должна сертифицировать программное обеспечение для областей, которые не являются критически важными. После того как продукт получит сертификат для работы в таких условиях, он будет пользоваться большим доверием у производителей приложений для критичных к безопасности систем в том случае, если его утилиты ранее были сертифицированы для некритических сред, которые отражают условия критичных к безопасности приложений.

Если программное обеспечение прекрасно работает в критичных к безопасности применениях и если SCL сумеет собрать достаточно доказательств этого, можно попробовать сертифицировать его для критически важных областей.

Достоинства модели сертификации с участием пользователей

Подобный подход обладает рядом преимуществ. Эти преимущества, однако, не могут быть реализованы до тех пор, пока точно не известно, какие данные необходимы для того, чтобы выдать ограниченную гарантию на программное обеспечение.

В конце концов, объем необходимых данных будет зависеть от того, насколько строгую и полную гарантию хотим получить.

Эти данные должны включать в себя результаты по оценке надежности, предположений о функциональности, мониторинга звонков о возникновении исключительных ситуаций, использования в работе и анализа охвата кода. Хотя возможно проведение и другого анализа, эти параметры, по-видимому, являются наиболее важными индикаторами того, достаточно ли хорошо изучена работа компонента, чтобы его можно было сертифицировать для использования в конкретных средах и секторах рынка. В дальнейшем можно вместе со специалистами по сертификации выбрать технологии резидентного тестирования, которые обеспечивают получение данных, достаточных для того, чтобы предложить страховую премию на программное обеспечение и гарантии на ПО.

Чтобы быстро добиться получения сертификата, недобросовестные разработчики могут попытаться изменить инструментарий таким образом, чтобы добавить фальшивые датчики, которые помешают проведению тестирования. И SCL должна уменьшить этот риск.

2.3. Жизненный цикл программы

2.3.1. Понятие технологии разработки программы

В современном мире всеобщей компьютеризации и информатизации требования, предъявляемые к программному обеспечению (ПО) вообще и к программным продуктам (ПП), программным средствам (ПС) и программам в частности, весьма высоки. В связи с этим обеспечение удовлетворяющих пользователя потребительских качеств программы, таких как надежность, быстродействие, соответствие заявленным возможностям, полнота документации, возможности расширения, развития и т. д., без строгого соблюдения определенной технологии практически невозможно.

Рассмотрим сначала основные термины и определения. Политехнический словарь [26] оперирует словом «технология» (от *греч.* techne — искусство, мастерство, умение и *логия*) в широком смысле как совокупностью «методов обработки, изготовления, изменения состояния, свойств, формы сырья, материала или по-

луфабрикатов, применимых в процессе производства, для получения готовой продукции»; как наукой «о способах воздействия на сырье, материалы и полуфабрикаты соответствующими орудиями производства. Разработка технологии осуществляется по отраслям производства». В Энциклопедическом словаре [25] определение примерно то же, более того, задача науки технологии заключается в выявлении «физических, химических, механических и др. закономерностей с целью определения и использования на практике наиболее эффективных и экономичных производственных процессов». В Толковом словаре [24] технология — это «совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства».

Итак, на основании анализа вышеприведенных определений под технологией программирования в широком смысле следует понимать технологию разработки программного средства, как совокупность абсолютно всех технологических процессов его создания — от момента зарождения идеи о данном ПС до составления необходимой программной документации. Каждый процесс указанной совокупности базируется на использовании неких методов и средств, например компьютерных (в этом случае будем говорить о компьютерной технологии программирования).

В литературе имеются и другие, отличные от приведенного понятия технологии программирования. Используется также близкое обсуждаемому понятие программной инженерии, определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств. Главное различие между технологией программирования и программной инженерией в качестве учебных дисциплин заключается в способах рассмотрения и систематизации материала. В технологии программирования акцент делается на изучении процессов разработки ПС (технологических процессов) и порядке их прохождения — в этих процессах используются определенные методы и инструментальные средства разработки ПС (их применение и образует технологический процесс), 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). Вопросом о том, каким образом эти методы и средства создают технологический процесс, в этом случае никто не задается.

Не следует также путать технологию программирования с методологией программирования, несмотря на то, что в обоих случаях изучаются соответствующие методы. Дело в том, что в технологии программирования методы рассматриваются «сверху», т. е. с точки зрения организации технологических процессов, а в методологии программирования — «снизу», т. е. с точки зрения основ их построения.

Имея в виду, что надежность является неотъемлемым атрибутом ПС, технологию программирования здесь будем рассматривать как технологию разработки надежных ПС. Это значит, во-первых, обсуждение всех процессов разработки ПС (от идеи создания до «утилизации»), а во-вторых, вопросов построения программных конструкций, описания функций и принимаемых решений с точки зрения их человеческого восприятия. И наконец, в качестве продукта технологии появится надежное программное средство. Все вышеперечисленное будет существенно влиять на выбор методов и инструментальных средств при разработке ПС [20].

Кратко резюмируем сказанное. Целью программирования является выполнение систематической последовательности действий для достижения автоматической обработки данных, таким образом, технология разработки программного обеспечения или, проще, технология программирования терминологически обозначает совокупность процессов для создания программного продукта требуемой функциональности.

Результатом таких процессов является программное средство — совокупность логически связанных программ на носителях данных, снабженных программной документацией и предназначенных для людей, не участвовавших в процессе разработки.

Поскольку технологический процесс разработки программного обеспечения вообще аналогичен процессу разработки программного средства (в частности), далее будем рассматривать специфику технологии программирования именно по отношению к ПО.

2.3.2. Основа разработки программного обеспечения

В основе разработки и дальнейшего применения программного обеспечения пользователем лежит понятие жизненного цикла, который, в сущности, является моделью его создания и

использования, отражающей различные состояния, начиная с момента осознания необходимости появления данного ПО и заканчивая моментом его полного выхода из употребления [5].

Существует несколько моделей жизненного цикла (ЖЦ), каждая из которых определяет различную методологию создания систем, тем не менее все без исключения модели ЖЦ включают в себя пять этапов и связей между ними с детальным описанием действий, моделей и результатов каждого этапа. Приведем названия и краткое содержание каждого этапа в соответствии с ГОСТ 19.102—77.

1. Техническое задание:
   - постановка задачи;
   - выбор критериев эффективности;
   - проведение предварительных научно-исследовательских работ (НИР);
   - разработка ТЗ.
2. Эскизный проект:
   - структура входных и выходных данных;
   - уточнение методов решения;
   - общий алгоритм;
   - разработка документации эскизного проекта.
3. Технический проект:
   - уточнение структуры входных и выходных данных;
   - разработка алгоритмов;
   - формы данных;
   - семантика и синтаксис языка;
   - структура программы;
   - конфигурация технических средств;
   - план работ.
4. Рабочий проект:
   - программирование и отладка;
   - разработка документов;
   - подготовка и проведение испытаний;
   - корректировка программы и документов по итогам испытаний.
5. Внедрение:
   - передача программы и документов для сопровождения;
   - оформление акта;
   - передача в Фонд алгоритмов и программ (ФАП).

2.3.3. Модели жизненного цикла

Исторически в ходе развития теории проектирования программного обеспечения и по мере его усложнения утвердились четыре основные модели ЖЦ.

Первой по времени появления и самой распространенной явилась **каскадная модель** (рис. 2.5).

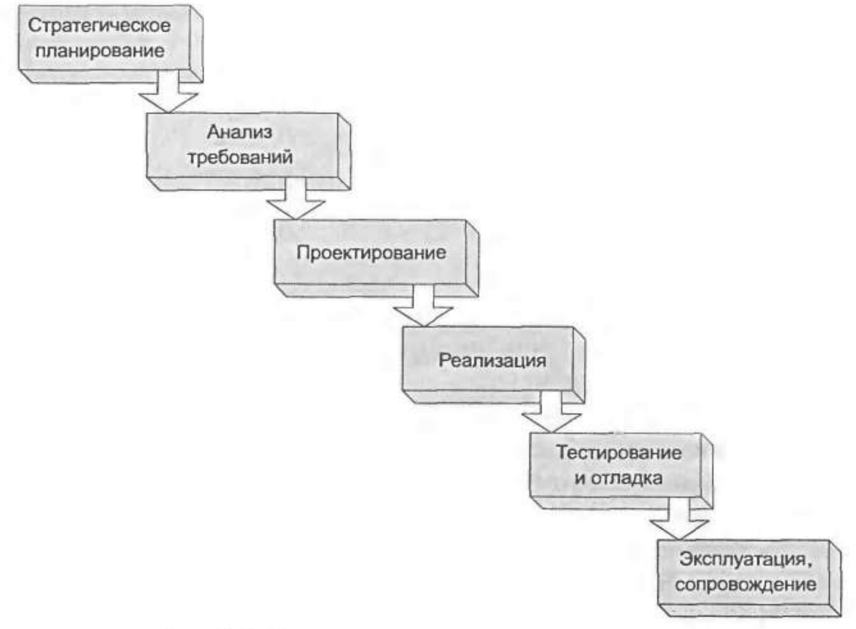


Рис. 2.5. Каскадная модель жизненного цикла ПО

Каскадная модель характеризуется следующими основными особенностями:

- последовательным выполнением входящих в ее состав этапов;
- окончанием каждого предыдущего этапа до начала последующего;
- отсутствием временного перекрытия этапов (последующий этап не начнется, пока не завершится предыдущий);
- отсутствием (или определенным ограничением) возврата к предыдущим этапам;
- наличием результата только в конце разработки.

Выявление и устранение ошибок в каскадной модели производится только на стадии тестирования, которая может растянуться во времени или вообще никогда не завершиться.

Следующей стадией развития теории проектирования ПО стала **итерационная модель ЖЦ**, или так называемая поэтапная модель с промежуточным контролем (рис. 2.6). Основной ее особенностью является наличие обратных связей между этапами, вследствие этого появляется возможность проведения проверок и корректировок проектируемой ИС на каждой стадии разработки. В результате трудоемкость отладки по сравнению с каскадной моделью существенно снижается. Итерационность модели проявляется в обработке ошибок, выявленных промежуточным контролем. Если на каком-либо этапе в ходе промежуточной проверки обнаружена ошибка, допущенная на более ранней стадии разработки, необходимо повторить весь цикл работ этой стадии. При этом анализируются причины ошибки и корректируются в случае необходимости исходные данные этапа или его содержание (последовательность действий).

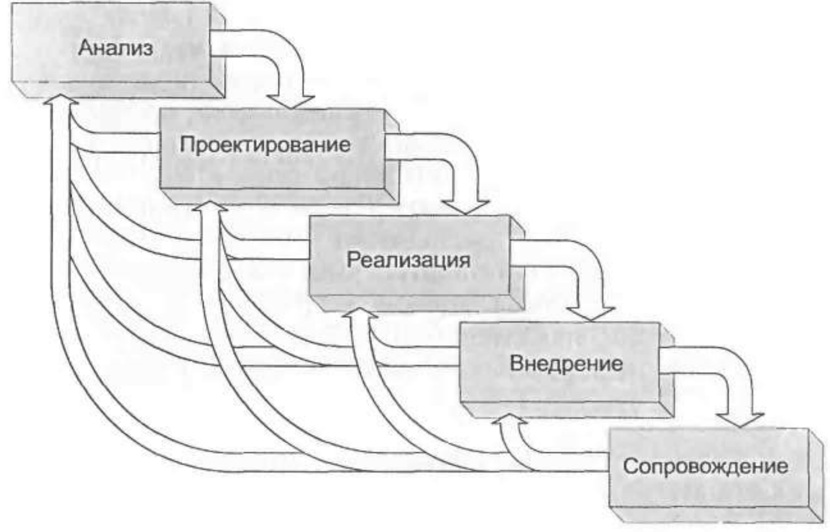


Рис. 2.6. Итерационная модель жизненного цикла ПО

К сожалению, в процессе разработки системы могут измениться начальные требования, и в этом случае итерационная модель может оказаться неэффективной.

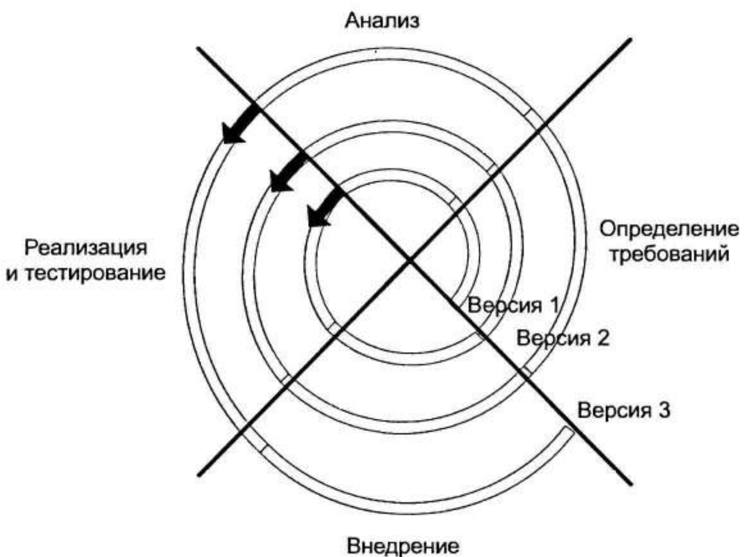


Рис. 2.7. Спиральная модель жизненного цикла ПО

Третья модель ЖЦ ПО — **спиральная** (spiral) **модель** (рис. 2.7) — поддерживает итерации поэтапной модели, но особое внимание уделяется начальным этапам проектирования: анализу требований, проектированию спецификаций, предварительному проектированию и детальному проектированию. Каждый виток спирали соответствует поэтапной модели создания фрагмента или версии ПО, уточняются цели и требования к программному обеспечению, оценивается качество разработанного фрагмента или версии и планируются работы следующей стадии разработки (витка). Таким образом, углубляются и конкретизируются все детали проектируемого ПО, в результате получается продукт, который удовлетворяет всем требованиям заказчика.

2.3.4. Rational Objectory Process — модель жизненного цикла (методология объектно-ориентированного программирования)

Известно, что объектно-ориентированное проектирование программного обеспечения стало результатом появления объектно-ориентированного программирования (ООП), т. е. примене-

ние новой методологии, как всегда[1], началось с этапа кодирования. Ранние стадии описания предметной области и разработки архитектуры системы не поддерживались, первые варианты использования объектно-ориентированной методологии в большей степени являлись чистым повторением принципов ООП. Такие вопросы, как декомпозиция предметной области, спецификация требований, интерфейс пользователя, не рассматривались, однако успехи объектно-ориентированного программирования заставили распространить новую технологию на весь жизненный цикл ПО. В результате все преимущества подхода применяются не только в процессе кодирования, но и на более ранних этапах. Таким образом, были определены основные компоненты методологии:

- модель жизненного цикла;
- действия;
- нотация языка.

2.3.5. Жизненный цикл UML[2] (Rational Objectory Process)

Фирма Rational Software, разработавшая язык UML, предложила также и свою модель ЖЦ, которая называется Rational Objectory Process (ROP). Означенная технология прямого перевода не имеет, так как rational в данном случае употребляется в значении «рациональный» и как название фирмы одновременно, во-вторых, слова objectory в английском языке не существует, его лингвообразование аналогично слову repository (накопитель).

Перечислим основные свойства ROP-технологии.

Rational Objectory Process — итеративный процесс, в течение которого происходит последовательное уточнение результатов.

---

[1] В исторической последовательности развития программных средств первыми появились узкоориентированные приложения (*«программа, предназначенная для вычисления числа π с точностью до 200 знака»*), следом — системы программирования (ранние их версии назывались *системами автоматизации программирования*), затем — операционные системы.

[2] Unified Modeling Language — унифицированный язык моделирования.

Rational Objectory Process направлен именно на создание моделей, а не на разработку каких-либо других элементов проекта (например, текстовых документов).

Действия Rational Objectory Process определяются в первую очередь блоками использования (use case) (рис. 2.8).

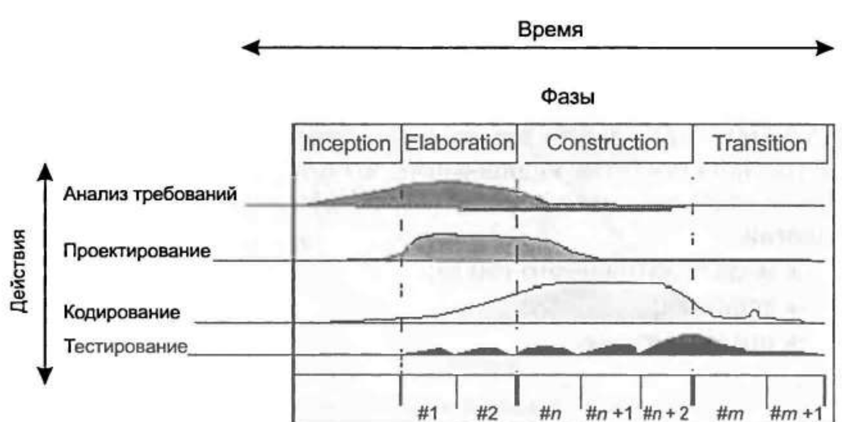


Рис. 2.8. Модель жизненного цикла UML

Rational Objectory Process разбит на циклы, каждый из которых, в свою очередь, состоит из четырех фаз:

- начальная стадия (Inception);
- разработка (Elaboration);
- конструирование (Construction);
- ввод в эксплуатацию (Transition).

Результатом работы каждого такого цикла является своя версия программной системы.

Каждая стадия завершается в четко определенной контрольной точке (milestone). В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

Начальная стадия может принимать множество разных форм. Для крупных проектов — это всестороннее изучение всех возможностей реализации на протяжении нескольких месяцев. Здесь же вырабатывается бизнес-план проекта, определяется его стоимость, примерный доход, а также ограничения ресурсов — иными словами, выполняется некоторый начальный анализ оценки проекта.

Окончанием начального этапа могут служить следующие результаты:

- начальный проектный словарь терминов;
- общее описание системы — основные требования к проекту, его характеристики и ограничения;
- начальная модель вариантов использования;
- начальный бизнес-план;
- план проекта, отражающий стадии и итерации;
- один или несколько прототипов.

На стадии разработки выявляются более детальные требования к системе, выполняется высокоуровневый анализ предметной области и проектирование базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта.

Самым важным результатом стадии разработки является описание базовой архитектуры будущей системы. Эта архитектура включает:

- модель предметной области, которая служит отправным пунктом для формирования основных абстракций предметной области;
- технологическую платформу, определяющую основные элементы технологии реализации системы и их взаимодействие.

Стадия разработки занимает примерно пятую часть времени создания проекта, результатом которой являются:

- оценка времени реализации каждого варианта использования;
- идентификация всех наиболее серьезных рисков и возможности их ликвидации.

Сущность стадии конструирования заключается в определении последовательности итераций конструирования и вариантов использования, реализуемых на каждой итерации, которые являются одновременно инкрементными и повторяющимися.

При этом необходимо отметить следующее:

- итерации являются инкрементными в соответствии с выполняемой функцией. Каждая итерация добавляет очередные конструкции к вариантам использования, реализованным во время предыдущих итераций;
- итерации являются повторяющимися по отношению к разрабатываемому коду. На каждой итерации некоторая часть

существующего кода переписывается с целью сделать его более гибким.

Результатом стадии конструирования является продукт, готовый к передаче пользователям и содержащий, как правило, руководство пользователей и готовый к интеграции на требуемых платформах.

Назначением стадии ввода в эксплуатацию является передача готового продукта в полное распоряжение конечных пользователей. Данная стадия включает:

- бета-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;
- параллельное функционирование с существующей (legacy) системой, которая подлежит постепенной замене;
- оптимизацию производительности;
- обучение пользователей и специалистов службы сопровождения.

2.3.6. Специфицирование и планирование

Процесс разработки начинается с создания концептуального описания будущего продукта, задающего «по-крупному» его образ, видение (этот документ так и называется «vision statement») в контексте требований рынка. Главным действующим лицом на этом этапе является «менеджер по продукту» (product manager) — специалист-маркетолог, знающий ситуацию на рынке и запросы потенциальных пользователей. Его задача — донести до менеджеров по разработке ПО потребительские свойства будущего продукта, т. е. указать, какие цели и требования пользователей необходимо удовлетворить, какие для этого заложить функциональные возможности (product features) и в каком порядке в соответствии с существующими приоритетами следует их ранжировать.

На основании vision statement менеджеры по разработке составляют функциональную спецификацию. Здесь функциональные особенности будущего продукта прописываются все еще с точки зрения будущего пользователя, правда, с большей степенью подробности, глубины и формализованности. Затрагиваются вопросы архитектуры проекта, определяются основные компоненты и взаимосвязи между ними. Принципиально то, что эта начальная спецификация вовсе не должна фиксировать всю

функциональность будущего продукта, как и детали каждой из уже определенных функций. В течение последующих этапов работы эта спецификация подвергнется ревизии по мере того, как разработчики будут больше узнавать о самом продукте, обретающем материальное воплощение и в связи с этим способность «сообщать» о целесообразности наличия (формы) той или иной функции.

На изменение функциональности будут влиять и внешние факторы, в том числе те реальные и потенциальные рыночные продукты, которые так или иначе конкурируют с разрабатываемым ПО. Наконец, функциональность зависит и от таких прозаических факторов, как недостаток ресурсов, отставание от графика или просто изъяны в реализации, которые невозможно или некогда исправить. В этом смысле корпоративная культура компании Microsoft не предполагает каких-либо комплексов: здесь без колебаний «режут по живому», отдавая приоритет своевременности выброса продукта на рынок. Статистические данные по основным продуктам Microsoft показывают, что в среднем около 25 % содержащихся в исходной спецификации (разрекламированных, а потому ожидаемых потребителем) функциональных особенностей исчезают ко времени выпуска продукта; если же считать и то, что привнесено, то конечная функциональность будет отличаться от исходной на 30 % и более.

На основе функциональной спецификации менеджеры по разработке, постоянно консультируясь с проектировщиками, начинают на модульной основе создавать горизонтальную архитектуру продукта. Главное, что на этой стадии все основные функции ПО разбиваются на несколько групп (обычно три-четыре группы). Соответственно, формируется столько же подпроектов, работа над которыми будет вестись последовательно. Разбиение производится на основе уже имеющейся классификации функций по степени важности. Наиболее важные ($^1/_3$ от общего количества, если групп всего 3) попадают в первый подпроект; другие, менее приоритетные, реализуются в рамках второго подпроекта, и наконец, прочие, наименее значимые функции выполняются в последнем подпроекте. Каждый подпроект заканчивается выпуском промежуточной «контрольной» версии продукта (milestone release).

Определенная таким образом архитектура проекта отображается на организационную структуру: для реализации отдельных функций в рамках одного подпроекта назначаются небольшие ко-

манды (small feature teams), которые и работают параллельно и максимально автономно. Для них определяется график работы и выделяются необходимые ресурсы, за рамки которых выходить не рекомендуется. Причем большое значение на рассматриваемом этапе придается сознательному принятию этих рамок самими разработчиками, которые имеют возможность детально проанализировать назначенную им задачу. В результате график, планируемый с точностью до дня, часто оказывается чересчур оптимистичным.

Следует обратить внимание на весьма упрощенный подход к разработке архитектуры программного продукта: по сути, само понятие архитектуры низводится до вспомогательного инструментария, подчиненного интересам организационного планирования и управления. Между тем, с точки зрения современных представлений хорошо структурированная архитектура продукта есть необходимое условие его успешной разработки и — что особенно важно! — дальнейшего развития. Именно поэтому виднейшие авторитеты в области Software Engineering, например Гради Буч [3], рассматривают архитектуру, определяющую логическую и физическую структуры программной системы, как основу надлежащих стратегических и тактических проектных решений в целях построения качественного продукта.

Известно, что многие продукты компании Microsoft, созданные на шаткой основе заведомо неполной и постоянно изменяемой спецификации, во многом ущербны. Именно в этом разгадка удивительных на первый взгляд отличий последовательных версий одного и того же продукта, не говоря уже о продуктах разных, но в то же время преемственных идеологически. Яркий тому пример — реализация механизмов DLE — OLE — ActiveX. Консервативная «несущая конструкция», каковой является архитектура ПО, гнется и ломается под грузом малоуправляемых мутаций и деформаций. В результате — масса проблем при реализации такого желательного принципа разработки, как повторное использование кода: достаточно сказать, что 50 % кода изменяется каждые 18 месяцев!

2.3.7. Процесс разработки

Каждая из параллельно работающих в рамках реализации подпроекта команд обычно состоит из менеджера по разработке (program manager), трех—восьми разработчиков и такого же ко-

личества тестировщиков. Каждая команда выполняет полный цикл разработки, включая проектирование, кодирование и прототипирование (вкупе с тестированием) своей задачи по реализации той функции, за которую она ответственна.

Разработчики выполняют проектирование, кодирование и отладку своего кода. Небезынтересно отметить, что обычно никакой особой проектной документации не ведется — на фирме издавна считается, что ее ведение наложило бы излишние ограничения на динамизм разработки. Функции же документации выполняет сам код; при этом он содержит очень мало комментариев, что всегда и являлось самой, пожалуй, отличительной особенностью хакеров. К примеру, комментарии в коде Excel составили лишь около 1 % его общего объема. С учетом же того, что команде предоставлена возможность, чтобы не сказать — обязанность, не рассматривать исходную спецификацию как догму, а, наоборот, оперативно откликаться на поступающие извне либо генерируемые внутри нее самой предложения по ее изменению, в конечном счете оказывается, что единственным источником для понимания реализации функции оказывается этот самый скупо откомментированный код. А отдуваться в итоге приходится местным «техническим писателям», и стоит ли удивляться количеству плохо документированных (и вовсе не документированных) возможностей, которыми традиционно славится руководство Microsoft. Стоит отметить, что при данном подходе к кодированию сложно применять такую современную технику, как «инспекция кода», позволяющую резко увеличить количество обнаруживаемых дефектов при сокращении затрат и усилий на тестирование [13].

Команды и отдельные разработчики, имея значительную свободу в процессе реализации подпроекта, должны тем не менее следовать нескольким жестким правилам, которые необходимы для синхронизации параллельно протекающей работы, что, собственно, и позволяет им функционировать как единому слаженному коллективу, способному относительно быстро и дешево справиться с масштабным проектом. Коль скоро идет работа над единым проектом, то разрабатываемый продукт всегда существует в виде доступной всем командам централизованной базы данных, содержащей «контрольную» (эталонную) версию файлов с исходным кодом (master version). Действующий в Microsoft механизм работы с единым проектом обычно называют «сборкой» (build), что подразумевает периодическое выполнение

процедуры генерации новой текущей версии продукта из частично или полностью законченных разработчиками компонентов. Эта процедура позволяет, не дожидаясь конца разработки, сразу же увидеть, как реализованные (реализуемые) отдельные функции работают в контексте всего продукта, и оперативно выявлять и корректировать проблемы.

Принятая в Microsoft технология подразумевает ежедневное выполнение процесса сборки (daily build), состоящего из нескольких шагов. Прежде всего, любой разработчик имеет возможность «скачать» (check out) необходимые ему для работы файлы из общей базы. После этого он имеет полную свободу по внесению в этот код изменений, необходимых для реализации и отладки той функции, за которую он ответствен. В любой момент разработчик может выполнить свою сборку (private build) и сгенерировать таким образом персональную версию продукта (private release).

Примерно половину своего рабочего времени разработчик тратит на написание кода; другую половину использует на тестирование, отладку и прямое общение с потребителями продукта. При этом нельзя сказать, что типичный майкрософтовский разработчик использует в своей работе самые современные методы и инструменты. Так, очень многие продолжают (как это ни покажется в стенах Microsoft удивительным) использовать Unix Source Code Control System просто потому, что привыкли к этой системе. Это и отражение того факта, что корпоративный менеджмент полагает, что лучше тратить время непосредственно на разработку, чем на освоение нового инструментария. Зато немаловажно, что почти все команды физически сосредоточены в одном месте (в корпоративном кампусе в Редмонде), используют общие языки программирования (в основном это Си и С++), более того — общий «фирменный» стиль кодирования и стандартизованные средства разработки. Это помогает параллельно работающим командам в обсуждении проектных идей и решений, потому что полностью автономной работы команд над общим проектом достигнуть невозможно.

Каждый разработчик работает в паре со «своим» тестировщиком; задача последнего — выполнять непрерывное тестирование той самой «персональной» промежуточной версии, которую собирает разработчик. Надо сказать, что по сравнению со своими коллегами-разработчиками тестировщики используют более современные методы и средства, включая автоматически генери-

руемые и запускаемые тесты и технику регрессионного тестирования. Конечно, не всякая фирма позволит себе такую роскошь, как придание каждому разработчику его персонального тестировщика, но при майкрософтовских масштабах продаж это экономически оправдано, а по сути и необходимо — это плата за качество проектирования.

По крайней мере, дважды в неделю разработчик должен встраивать (check in) разрабатываемый «персональный» код в общую базу, где находится текущая «эталонная» версия (а можно это делать и каждый день). При этом он производит компиляцию и компоновку с обязательным выполнением регрессионного теста, который позволяет проверить реакцию общей «сборки» на вновь поступивший код. В случае проявления при этом какого-либо дефекта разработчик обязан тут же его «зафиксировать» (М. Кусумано, являющийся автором получивших мировую известность книг о японских промышленных технологиях, подмечает здесь сходство с действующим в сборочном цехе корпорации «Тойота» правилом, требующим немедленной остановки конвейера любым сборщиком, если он обнаружил дефект в собираемом автомобиле).

Операцию по встраиванию своего кода в общую эталонную базу разработчики имеют право выполнять до определенного назначенного часа; затем в дело вступает специально назначенный разработчик (project build master), который ежедневно генерирует полную «сборку» продукта на основе текущей эталонной версии исходного кода всего продукта. Эта процедура следует «сценарию сборки» (build script) в виде автоматически выполняемой последовательности команд и включает шаги по полной компиляции кода и получению в конечном итоге одного или нескольких исполняемых файлов. При этом могут создаваться различные «библиотечные файлы», позволяющие конечным пользователям настроить продукт в соответствии со своей спецификой. Таким образом, ежедневно производится выпуск внутренней версии продукта (internal release), генерируемой для каждой платформы (Windows, Mac.), а также для каждого значимого рынка (американский, европейские и т. п.). Вся эта технология, направленная на периодическую интеграцию функций и «стабилизацию» кода в его текущем состоянии, позволяет реализовать такой базисный принцип, как постоянное наличие во всех необходимых версиях «готового» (пусть еще далеко не полностью) продукта, который можно предъявить потребителю.

2.3.8. Выпуск продукта и механизмы обратной связи

В процессе разработки непрерывно используется несколько метрик. Хотя в соответствии с общей корпоративной философией здесь не придают такого значения количественным индикаторам, как, например, в Motorola или HP, чья корпоративная культура предусматривает значительно более «выстроенные» процессы разработки. Тем не менее менеджеры каждый день отслеживают прогресс «ежедневных сборок» продукта именно на основе метрик, показывающих, сколько новых «багов» выявлено, сколько, наоборот, исправлено, и наконец, сколько всего остается «активных» дефектов.

В конце каждого подпроекта — после истечения срока параллельной работы команд над реализацией назначенных им функций — предусмотрен специальный период «буферное время» (buffer time), в течение которого разрешаются всякие не предусмотренные планом, но неизбежно возникающие проблемы, особенно вызванные оперативно вносимыми изменениями в спецификации и взаимозависимостью функций, над которыми работали разные команды. Этот период используется и как тривиальное продление периода разработки подпроекта, потому что выходы за пределы временного графика наблюдаются почти всегда. Для проектов из разряда приложений буферное время занимает 20—30 % от всей продолжительности подпроекта, а для системных программ — 50 %. Только после этого выпускается очередная «контрольная» версия (milestone release), с которой активно работают пользователи.

Важнейшим механизмом, обеспечивающим эту обратную связь с потенциальными потребителями продукта на протяжении всего процесса разработки, включая даже период работы над первым подпроектом, является институт лабораторий пользователя (Usability Lab). Первая такая лаборатория была открыта в 1989 г. и имела четыре тестовых комплекта, каждый из которых включает две разделенные полупрозрачным зеркалом комнаты: тестовую (test room), где пользователь имеет возможность «поиграть» с продуктом, и наблюдательскую (observation room), где располагается сотрудник, в чьи функции входит отслеживание всех деталей работы пользователя. Через три года была введена в действие еще одна лаборатория с пятью тестовыми комплектами; в 1995 г. добавили лабораторию, получившую название Microsoft Home, и не случайно: чтобы заставить пользователей чувствовать себя в бук-

вальном смысле «как дома», в ней сымитирована домашняя обстановка, включая кухню, столовую и детскую. Наконец, в 1996 г. появилась лаборатория с пятью тестовыми комплектами, включающая специальное эргономическое оборудование.

В этих лабораториях работают более ста сотрудников (usability engineers), имеющих не только компьютерное образование, но и обладающих знаниями в специальных областях психологии и эргономики. Кроме того, непосредственно разработчикам, особенно отвечающим за интерфейсную часть, вменено в обязанность периодически присутствовать на тестовых экспериментах. Что касается контингента испытателей, то их стремятся подбирать так, чтобы они представляли все категории потенциальных пользователей — для этого накоплена и постоянно пополняется обширная база данных.

Главное, что измеряется и выражается в специальных метриках, — это легкость освоения продукта и удобство работы с ним. Упомянем две метрики: первая показывает процент пользователей, которым удалось без обращения к руководству выполнить некоторое осмысленное действие; вторая метрика выражает процент корректных шагов на пути к выполнению задачи, сделанных с первой же попытки. Опытным путем установлено, что для большинства продуктов на ранней стадии разработки вторая метрика (correct first-try rate) получается в районе 60 %; цель же, которой в конечном счете стараются добиться, — это 90 %.

Пользователи обычно уходят довольными: в качестве награды за несколько часов, проведенных в лаборатории, они получают в дар коробку с майкрософтовским продуктом. Довольны и принимающие их сотрудники: в их распоряжении оказывается обширный материал для осмысления, который немедленно принимается во внимание. Но надо ли радоваться тем будущим потребителям, которые хотят получить качественный продукт, — не столь ясно. Ведь вносимые на ходу изменения в проект, удовлетворяющие в первую очередь критерию легкости использования, зачастую оборачиваются «заплатами» на уже написанных модулях, еще более расшатывающими и без того не слишком крепкую архитектуру. Наиболее очевидные последствия — разбухание кода, приводящее к повышенным требованиям на ресурсы при менее эффективном исполнении.

Эксперименты проводятся не только в корпоративных лабораториях, но и «на выезде» в офисах, школах и университетах, по месту жительства возможных потребителей. Кроме того, в по-

следней фазе разработки — фазе стабилизации — прошедшие всестороннее внутреннее тестирование «beta» версии отправляются для опытной эксплуатации к партнерам корпорации, принадлежащим к категориям OEM и ISV; здесь задействованы и многочисленные добровольцы-индивидуалы. После этого компания приступает к подготовке выпуска «финальной» версии продукта («golden master» discs), а также необходимой документации. Но даже после выпуска «финальной» версии работа над продуктом не прекращается. Уже установилась традиция в среднем через 12 месяцев выпускать исправленную и дополненную версию, а через 24 месяца — радикально переработанную (с большим количеством новых функций и измененной архитектурой). Небезынтересно отметить, что работа отвечающих на телефонные звонки и другие обращения о помощи инженеров службы поддержки финансируется за счет бюджета команд разработчиков данного продукта; поэтому последние заинтересованы в постоянной минимизации дефектов в каждой последующей версии сравнительно с предыдущей.

Контрольные вопросы

1. Расскажите об особенностях создания программного продукта.
2. Что такое жизненный цикл программного обеспечения?
3. Каковы основные свойства каскадной (итерационной) модели жизненного цикла?
4. Из каких этапов состоит модель жизненного цикла UML?
5. Какова стоимость исправления ошибок в ПО на различных стадиях его разработки?
6. Что такое «управление требованиями»?
7. В чем заключается анализ проблемы?
8. Какие виды ограничений на создаваемое ПО необходимо выявить в процессе работы над требованиями?
9. Каковы существующие методы выявления требований к ПО?

Глава 3
АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Процесс проектирования программных продуктов начинается с определения требований к разрабатываемому программному обеспечению и его исходных данных. В результате анализа требований получают спецификации программного обеспечения в виде текстовых описаний, структурных схем и диаграмм. В процессе определения спецификаций строят общую модель предметной области и конкретизируют основные функции программного продукта и его поведение при взаимодействии с окружающей средой [1].

3.1. Определение требований к программным продуктам

Один из наиболее ответственных этапов создания программного продукта — этап постановки задачи. На этом этапе принимаются важные решения относительно функций создаваемого ПО, эксплуатационных ограничений, накладываемых на него. Производится выбор архитектуры, среды разработки ПО, интерфейса пользователя и др. От этого выбора будет зависеть качество и стоимость конечного программного продукта.

3.1.1. Функциональные требования

Функциональные требования описывают сервисы, предоставляемые программной системой, ее поведение в определенных ситуациях, реакцию на те или иные входные данные и действия,

которые система позволит выполнять пользователям. Иногда сюда добавляются сведения о том, чего система делать не должна [37, 61].

Каждый программный продукт предназначен для выполнения определенных функций. Для того чтобы определить, подходит та или иная программа для решения задач, необходимо иметь четкий набор критериев, на основании которого можно сделать правильный выбор.

При написании функциональных требований необходимо учитывать, что чем они будут подробнее, тем более точная оценка работ по срокам и стоимости будет произведена перед разработкой технического задания на создание программного обеспечения. Если на дальнейших этапах разработки ПО не возникнет дополнений к изначально сформулированным функциональным требованиям, то эта оценка будет достаточно точной. В то же время при описании требований не надо углубляться в какие-то мелкие детали. Необходимо описывать именно функции программы, а не то, какую кнопочку надо нажать в верхнем левом углу окна программы, чтобы получить результат. Такие детали должны быть подробно проработаны уже в процессе разработки технического задания.

Функциональные требования документируются в *спецификации требований* к программному обеспечению, где описывается как можно более полно ожидаемое поведение системы.

Необходимо, чтобы функциональная спецификация программного средства была математически точной. Желательно даже, чтобы при ее разработке применялись математические методы и формализованные языки. Она должна базироваться на четких понятиях и утверждениях, однозначно понимаемых разработчиками и заказчиками программного продукта.

Функциональная спецификация состоит из трех частей:

1. Описание внешней информационной среды, с которой будет взаимодействовать разрабатываемое программное обеспечение. Должны быть определены все используемые каналы ввода и вывода и все информационные объекты, к которым будет применяться разрабатываемое ПС, а также существенные связи между этими информационными объектами.

2. Определение функций программного обеспечения, определенных на множестве состояний этой информационной среды. Вводятся обозначения всех определяемых функций, специфицируются их входные данные и результаты выполнения, с указани-

ем типов данных и заданий всех ограничений, которым должны удовлетворять эти данные и результаты. Определяется содержание каждой из этих функций.

3. Описание исключительных ситуаций, если таковые могут возникнуть при выполнении программ, и реакций на эти ситуации, которые должны обеспечить соответствующие программы. Должны быть перечислены все существенные случаи, когда программное обеспечение не сможет нормально выполнить ту или иную свою функцию. Для каждого такого случая должна быть определена реакция программы.

3.1.2. Эксплуатационные требования

Эксплуатационные требования определяют характеристики разрабатываемого программного обеспечения, проявляемые в процессе его использования. К таким характеристикам относят [1]:

- правильность — функционирование в соответствии с техническим заданием. Это требование является обязательным для всякого программного продукта, но поскольку никакое тестирование не дает гарантии 100%-ной правильности, речь может идти об определенной *вероятности* наличия ошибок. Вероятность сбоя системы управления космическими полетами должна быть близка к нулю;
- универсальность — обеспечение правильной работы при любых допустимых данных и защиты от неправильных данных. Так же как в предыдущем случае, доказать универсальность программы невозможно, поэтому имеет смысл говорить о *степени* ее универсальности;
- надежность (помехозащищенность) — обеспечение полной повторяемости результатов, т. е. обеспечение их правильности при наличии различного рода сбоев. Источниками помех могут являться технические и программные средства, а также люди, работающие с этими средствами. В настоящее время существует достаточное количество способов избежать потерь информации при сбоях. Например, прием «создания контрольных точек», при котором сохраняются промежуточные результаты, что позволяет после сбоя программы продолжить работу с данными, записанными в последней контрольной точке. Возможно также

уменьшить количество ошибок, используя дублирование систем или ввод избыточной информации;

- проверяемость — возможность проверки получаемых результатов. Для этого необходимо документально фиксировать исходные данные, установленные режимы и другую информацию, которая влияет на получаемые результаты. Особенно это сказывается, когда сигналы поступают непосредственно от датчиков;
- точность результатов — обеспечение погрешности результатов не выше заданной. Величина погрешности зависит от точности исходных данных, степени адекватности используемой модели, точности выбранного метода и погрешности выполнения операций в компьютере. Жесткие требования к точности предъявляют системы навигации (например, система стыковки космических аппаратов) и системы управления технологическими процессами;
- защищенность — обеспечение конфиденциальности информации. Наиболее жесткие требования предъявляются к системам, в которых хранится информация, связанная с государственной и коммерческой тайной. Для обеспечения защиты информации используют программные, криптографические, правовые и другие методы;
- программная совместимость — возможность совместного функционирования с другим программным обеспечением. Чаще всего в данном случае речь идет о функционировании программы под управлением заданной операционной системы. Однако может потребоваться обмен данными с некоторой другой программой. В этом случае точно оговаривается формат передаваемых данных;
- аппаратная совместимость — возможность совместного функционирования с некоторым оборудованием. Это требование формулируют в виде минимально возможной конфигурации оборудования, на котором будет работать данное программное обеспечение. Если предполагается использование нестандартного оборудования, то для него должны быть описаны интерфейсы;
- эффективность — использование минимально возможного количества ресурсов технических средств (например, времени микропроцессора, объема оперативной памяти, объема внешней памяти, количества внешних устройств и др.). Эффективность оценивается по каждому ресурсу отдельно,

поэтому требования эффективности часто противоречат друг другу. Например, чтобы уменьшить время выполнения программы, необходимо увеличить объем оперативной памяти;

- адаптируемость — возможность быстрой модификации с целью приспособления к изменяющимся условиям функционирования. Оценить эту характеристику количественно практически невозможно. Можно только констатировать, что при разработке данного ПО использовались приемы, облегчающие его модернизацию;
- повторная входимость — возможность повторного выполнения без перезагрузки с диска. Данное требование обычно предъявляется к программному обеспечению, резидентно загруженному в оперативную память (например, драйверы);
- реентерабельность — возможность «параллельного» использования несколькими процессами. Чтобы удовлетворить этому требованию, необходимо создавать копию данных, изменяемых программой, для каждого процесса.

Четко сформулировать спецификации требований к разрабатываемому ПО, чтобы затем занести их в техническое задание, — достаточно сложная и ответственная задача, которая требует проведения предпроектных исследований.

3.2. Выбор архитектуры программного обеспечения

В технологии программирования нет четкого определения архитектуры ПО. Приведем некоторые из встречающихся в литературе.

Архитектурой программного обеспечения называют совокупность базовых концепций (принципов) его построения [1].

Архитектура ПС — это его строение, как оно видно (или должно быть видно) извне его, т. е. представление ПС как системы, состоящей из некоторой совокупности взаимодействующих подсистем [37].

Архитектура программы или компьютерной системы — это структура или структуры системы, которые включают элементы

программы, видимые извне свойства этих элементов и связи между ними [50].

Архитектура — это структура организации и связанное с ней поведение системы. Архитектуру можно рекурсивно разобрать на части, взаимодействующие посредством интерфейсов, связи, которые соединяют части, и условия сборки частей. Части, которые взаимодействуют через интерфейсы, включают классы, компоненты и подсистемы [51].

Архитектура программного обеспечения системы или набора систем состоит из всех важных проектных решений по поводу структур программы и взаимодействий между этими структурами, которые составляют системы. Проектные решения обеспечивают желаемый набор свойств, которые должна поддерживать система, чтобы быть успешной. Проектные решения предоставляют концептуальную основу для разработки системы, ее поддержки и обслуживания [52].

Как мы видим, выбор архитектуры разрабатываемого ПО определяется задачами, поставленными перед разработчиками, функциональными и эксплуатационными требованиями.

С точки зрения количества пользователей, работающих с одной копией ПО, *различают*:

- однопользовательскую архитектуру;
- многопользовательскую (сетевую) архитектуру.

Кроме того, в рамках однопользовательской архитектуры различают [1]:

- программы. *Программа* (program, routine) — упорядоченная последовательность формализованных инструкций для решения задачи с помощью компьютера. Это самый простой вид архитектуры, который обычно используется при решении небольших задач;
- пакеты программ. *Пакеты программ* представляют собой несколько отдельных программ, решающих задачи определенной прикладной области. Например, пакет графических программ, пакет математических программ. Пакет программ реализуется как набор отдельных программ, каждая из которых сама вводит необходимые данные и выводит результаты, т. е. программы пакета связаны между собой только принадлежностью к некоторой прикладной области;
- программные комплексы. *Программные комплексы* представляют собой совокупность программ, совместно обеспе-

чивающих решение небольшого класса сложных задач одной прикладной области. При этом для выполнения некоторой задачи программой-диспетчером последовательно вызываются несколько программ из программного комплекса. Поскольку несколько программ для решения одной задачи работают с одними и теми же исходными данными и промежуточными результатами, желательно хранить эти данные и результаты вызовов в оперативной памяти или в файлах в пределах одного пользовательского проекта. Программы комплекса могут компилироваться как самостоятельные единицы или совместно. Программа-диспетчер может иметь примитивный интерфейс и простую справочную систему [1];

- программные системы. *Программные системы* представляют собой организованную совокупность программ (подсистем), позволяющую решать широкий класс задач из некоторой прикладной области. Программы, входящие в программную систему, взаимодействуют через общие данные. Программные системы имеют достаточно развитый интерфейс, что требует их тщательного проектирования и разработки.

Многопользовательскую архитектуру реализуют системы, построенные по принципу «клиент — сервер».

3.3. Структура и формат данных. Статические, полустатические и динамические структуры

На этапе определения спецификаций для разработки качественного программного обеспечения необходимо определить структуру и формат используемых в программах данных [41].

Структура данных — это множество элементов данных и связей между ними.

Независимо от содержания и сложности любые данные в памяти компьютера представляются в виде последовательности двоичных разрядов (битов), а их значениями являются соответствующие двоичные числа. Битовые последовательности слабо структурированы и неудобны для практического применения. На практике обычно применяют более сложно организованные структуры данных.

3.3.1. Классификация структур данных

С понятием структуры данных тесно связано понятие *типа* данных.

Различают *физическую* и *логическую* структуры данных. Физическая структура в отличие от логической отражает способ представления данных в памяти компьютера и называется еще внутренней.

По составу различаются *простые* структуры (типы) данных и *интегрированные* (сложные). Простые структуры не могут быть расчленены на составные части, большие, чем биты. С точки зрения физической структуры для простого типа четко определен его размер и способ размещения в памяти компьютера. С точки зрения логической структуры простые структуры являются неделимыми единицами. Интегрированные структуры данных включают в себя другие структуры данных — простые или интегрированные.

Между отдельными элементами структур могут наличествовать или отсутствовать явно заданные связи. В зависимости от этого следует различать: *несвязные* структуры (векторы, массивы, строки, стеки, очереди) и *связные* структуры (связные списки).

По признаку *изменчивости* различают структуры *статические, полустатические, динамические*. Под изменчивостью понимают изменение числа элементов структуры или связей между этими элементами. Классификация структур данных по признаку изменчивости приведена на рис. 3.1.

По признаку упорядоченности элементов структуры можно делить на *линейные* и *нелинейные*. Пример нелинейных структур — многосвязные списки, деревья, графы.

Линейные структуры, в свою очередь, делятся на структуры с *последовательным* распределением (векторы, строки, массивы, стеки, очереди) и структуры с *произвольным связным* распределением (односвязные, двусвязные списки) по характеру распределения элементов в памяти.

Указание типа данных четко определяет:

- размер памяти, отведенной под данную структуру и способ ее размещения в памяти;
- значения, допустимые для данного типа данных;
- операции, которые возможно над этими данными выполнять.

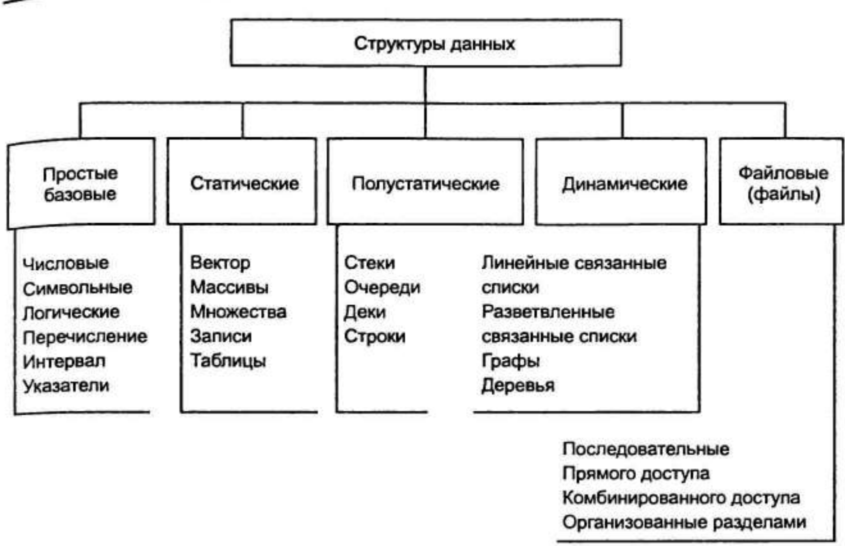


Рис. 3.1. Классификация структур данных

3.3.2. Простые структуры данных

Простые структуры данных служат основой для построения более сложных структур. Их называют также примитивными или базовыми структурами (типами данных). К ним относятся: числовые, битовые, логические, символьные, перечисляемые, интервальные, указатели. Структура простых типов данных для языка Pascal приведена на рис. 3.2 (в других языках программирования набор и размеры простых типов могут отличаться от приведенного на рисунке). Размер каждого типа данных указан на рисунке в байтах через запятую от названия типа.

Как уже было сказано, разные типы данных имеют различный формат представления их в машинной памяти.

На рис. 3.3—3.5 приведены примеры форматов числовых типов данных.

На рис. 3.4 S обозначает знаковый разряд числа (если $S = 0$, то число положительное, если $S = 1$ — число отрицательное).

Формат для представления чисел с плавающей точкой, приведенный на рис. 3.5, *а*, содержит поля мантиссы, порядка и знаков мантиссы и порядка фиксированной длины. Однако чаще вместо порядка используется характеристика, полученная

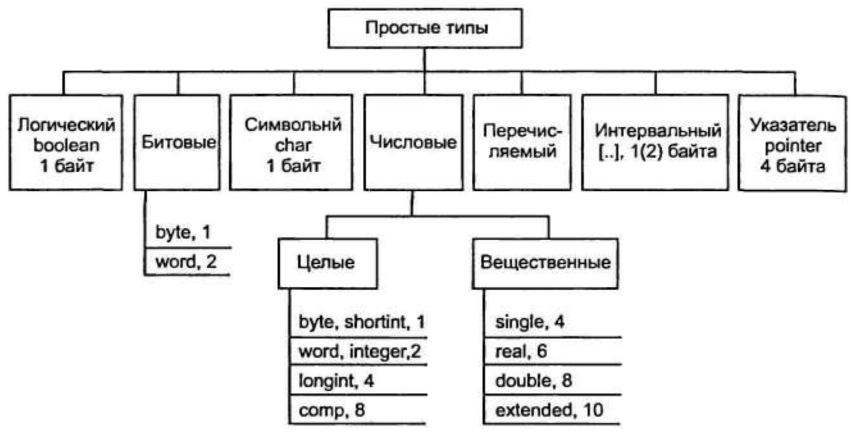


Рис. 3.2. Структура простых типов PASCAL

Рис. 3.3. Формат машинного представления беззнаковых чисел: *а* — тип byte; *б* — тип word

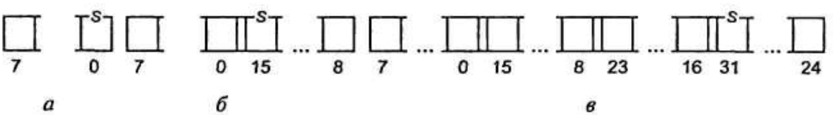


Рис. 3.4. Формат машинного представления чисел со знаком: *а* — shortint; *б* — integer; *в* — longint

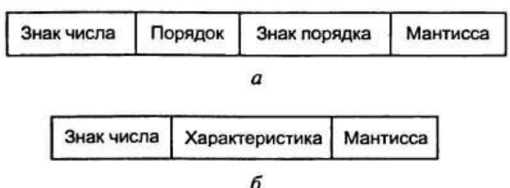


Рис. 3.5. Формат представления вещественных чисел: *а* — с порядком; *б* — с характеристикой

путем прибавления к порядку смещения, так чтобы характеристика была всегда положительной. При этом имеет место формат представления вещественных чисел такой, как на рис. 3.5, *б*.

3.3.3. Статические структуры данных

Статические структуры представляют собой структурированное множество примитивных структур. Например, вектор может быть представлен упорядоченным множеством чисел. Изменчивость несвойственна статическим структурам, т. е. размер памяти компьютера, отводимый для таких данных, постоянен и выделяется на этапе компиляции или выполнения программы.

Векторы

С логической точки зрения вектор (одномерный массив) представляет собой структуру данных с фиксированным числом элементов одного и того же типа. Каждый элемент вектора имеет свой уникальный номер (индекс). Обращение к элементу вектора выполняется по имени вектора и номеру элемента.

С физической точки зрения элементы вектора размещаются в памяти в подряд расположенных ячейках памяти (рис. 3.6). Под элемент вектора выделяется количество байт памяти, определяемое базовым типом элемента этого вектора. Тогда размер памяти, отводимой для размещения вектора, будет определяться следующим соотношением: $S = k$ * Sizeof(тип), где k — количество элементов (длина) вектора, а Sizeof(тип) — размер памяти, необходимой для хранения одного элемента вектора.

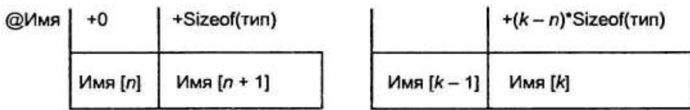


Рис. 3.6. Представление вектора в памяти:
@Имя — адрес вектора или адрес первого элемента вектора

Двумерные массивы

Двумерный массив (матрица) — это вектор, каждый элемент которого вектор. Поэтому то, что справедливо для вектора, справедливо и для матрицы (аналогично для *n*-мерных массивов).

Множества

Множеством является структура, представляющая собой набор неповторяющихся данных одного и того же типа. Множество может принимать все значения базового типа. Поскольку базовый тип не должен превышать 256 возможных значений, типом элементов множества могут быть byte, char и производные от них типы.

Множество в памяти (рис. 3.7) хранится как массив битов, в котором каждый бит указывает, является ли элемент принадлежащим объявленному множеству или нет. Таким образом, максимальное число элементов множества 256, а данные типа множество могут занимать не более 32 байт.

Смещение (байт)	Представление байта в машинной памяти (номера разрядов)							
@S + 0	7	6	5	4	3	2	1	0
@S + 31	255	254	253	252	251	250	249	248
	7							0

Рис. 3.7. Представление множества в памяти:
@S — адрес данного типа множество

Размер памяти (в байтах), выделяемых под множество, вычисляется по формуле: $S = (\max \text{ div } 8) - (\min \text{ div } 8) + 1$, где max и min — верхняя и нижняя границы базового типа данного множества, а div — целочисленное деление.

Записи

Запись — это комбинированный тип, значения которого представляют собой нетривиальную структуру данных. Они состоят из нескольких полей разного типа, доступ к которым осуществляется по их именам. Записи представляют собой средство для представления программных моделей реальных объектов предметной области, так как каждый такой объект обладает несколькими свойствами, которые могут описываться данными различных типов.

Пример записи — набор сведений о сотруднике кафедры.

Объект «сотрудник» может обладать следующими свойствами:

- табельный номер — целое положительное число;
- фамилия-имя-отчество — строка символов и т. д.;
- пол — символ;
- ученая степень — строка символов;
- заработная плата — вещественное число;
- и др.

В памяти эта структура может быть представлена в одном из двух видов:

- в виде последовательности полей, занимающих непрерывную область памяти (рис. 3.8). Чтобы получить доступ к любому элементу записи, нужно знать адрес начала записи и смещение относительно начала. При этом достигается экономия памяти компьютера, но затрачивается лишнее время на вычисление адресов полей;
- в виде связного списка с указателями на значения полей записи (рис. 3.9). При такой организации имеет место быстрый доступ к элементам, но очень неэкономичный расход памяти для хранения.

@rec	+0	+1	+21	+29	+37	+38	+39
	24	Иванов В. И.	АП	54	4	5	5

Рис. 3.8. Представление в памяти переменной типа запись в виде последовательности полей

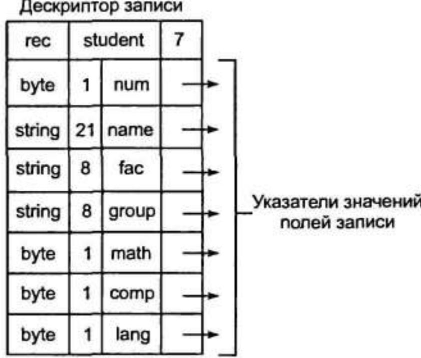


Рис. 3.9. Размещение в памяти переменной типа запись в виде указателей

3.3.4. Полустатические структуры данных

Свойства полустатических структур данных:

- они имеют переменную длину и простые способы ее изменения;
- изменение длины структуры происходит в определенных пределах, не превышая какого-то максимального (предельного) значения.

С логической точки зрения полустатическая структура представляет собой последовательность данных, связанную отношениями линейного списка (см. разд. 3.3.5). Доступ к элементу может осуществляться по его порядковому номеру.

Физически полустатические структуры представляются либо в виде вектора, т. е. располагаются в непрерывной области памяти, либо в виде однонаправленного связного списка, где каждый следующий элемент адресуется указателем, находящимся в текущем элементе.

К полустатическим структурам относятся стеки, очереди, деки, строки.

3.3.5. Динамические структуры данных

Динамические структуры не имеют постоянного размера, поэтому память под отдельные элементы таких структур выделяется в момент, когда они создаются в процессе выполнения программы, а не во время трансляции. Когда в элементе структуры больше нет необходимости, занимаемая им память освобождается (элемент «разрушается»).

Поскольку элементы динамической структуры располагаются в памяти не по порядку и даже не в одной области, адрес элемента такой структуры не может быть вычислен из адреса начального или предыдущего элемента. Связь между элементами динамической структуры устанавливается через указатели, содержащие адреса элементов в памяти. Такое представление данных в памяти называется связным.

Таким образом, кроме информационных полей, ради которых создается структура и которые являются видимыми для конечного пользователя ПО, динамические структуры содержат поля для связи с другими элементами, видимые только для программиста — разработчика ПО.

С помощью связного представления данных обеспечивается высокая изменчивость структуры. Достоинства динамических структур:

- размер структуры ограничивается только объемом памяти компьютера;
- при изменении логической последовательности элементов структуры (удалении, добавлении элемента, изменении порядка следования элементов) требуется только коррекция указателей.

С другой стороны, такие структуры обладают рядом недостатков:

- работа с указателями требует высокой квалификации программиста;
- на указатели расходуется дополнительная память;
- дополнительный расход времени на доступ к элементам связной структуры.

Связные линейные списки

Линейные связные списки являются простейшими динамическими структурами данных. Они являются упорядоченными множествами, содержащими переменное число элементов, на которые не накладываются ограничения по длине.

На рис. 3.10 приведена структура односвязного списка. Здесь поле INF — информационное поле, содержащее данные, NEXT — указатель на следующий элемент списка. Голова списка — указатель на начало списка. Указатель на следующий элемент последнего элемента списка содержит значение nil, это является признаком последнего элемента.

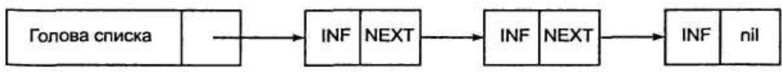


Рис. 3.10. Структура односвязного списка

Двусвязные списки

Обработка односвязного списка не всегда удобна, так как невозможно двигаться в противоположную сторону. Такую возможность обеспечивает двусвязный список, каждый элемент которого содержит два указателя: на следующий и предыдущий

элементы. Структура линейного двусвязного списка приведена на рис. 3.11, где поле NEXT — указатель на следующий элемент, поле PREV — указатель на предыдущий элемент. Первый и последний элементы такого списка содержат nil в указателе на предыдущий и последующий элементы соответственно.

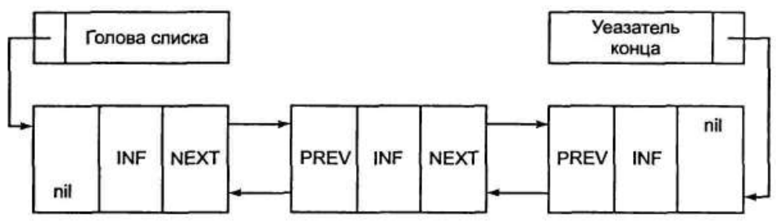


Рис. 3.11. Структура двусвязного списка

Для удобства обработки списка добавляют еще один особый элемент — указатель конца списка. Наличие двух указателей в каждом элементе усложняет список и приводит к дополнительным затратам памяти, но в то же время обеспечивает более эффективное выполнение некоторых операций над списком.

3.4. Модульное программирование

Для обеспечения технологичности разрабатываемого программного обеспечения применяется модульное программирование [37].

3.4.1. Понятие модуля

Приступая к разработке программы, следует иметь в виду, что она, как правило, является большой системой, поэтому необходимо принять меры для ее упрощения. Для этого программу разрабатывают по частям, которые называются ***программными модулями***. Такой метод создания программ называют модульным программированием.

Модульное программирование основано на понятии ***модуля*** — программы или функционально завершенного фрагмента программы.

Модуль характеризуют:

- один вход и один выход. На входе программный модуль получает определенный набор исходных данных, выполняет их обработку и возвращает один набор выходных данных;
- функциональная завершенность. Модуль выполняет набор определенных операций для реализации каждой отдельной функции, достаточных для завершения начатой обработки данных;
- логическая независимость. Результат работы данного фрагмента программы не зависит от работы других модулей;
- слабые информационные связи с другими программными модулями. Обмен информацией между отдельными модулями должен быть минимален;
- размер и сложность программного элемента в разумных рамках.

Таким образом, модули содержат описание исходных данных, операции обработки данных и структуры взаимосвязи с другими модулями.

Программный модуль является самостоятельным программным продуктом. Это означает, что каждый программный модуль разрабатывается, компилируется и отлаживается отдельно от других модулей программы. Более того, каждый разработанный программный модуль может включаться в состав разных программных систем при условии выполнения требований, предъявляемых к его использованию в документации к этому модулю. Таким образом, программный модуль может рассматриваться и как средство упрощения сложных программ, и как средство накопления и многократного использования программистских знаний.

3.4.2. Основные характеристики программного модуля

Как разработать хороший модуль, который действительно будет способствовать упрощению программы?

В литературе приводятся различные критерии оценки приемлемости модуля. Были предложены следующие критерии:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Предлагается использовать следующие характеристики программного модуля для оценки его приемлемости: размер модуля,

прочность модуля, сцепление с другими модулями и рутинность модуля.

Размер модуля измеряется числом содержащихся в нем операторов. Модуль не должен быть слишком маленьким или слишком большим. Большие модули, как правило, сложны для понимания и неудобны для внесения изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке. Маленькие модули усложняют общую структурную схему программы и могут не окупать накладных расходов, связанных с их оформлением. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

Прочность модуля — это мера его внутренних связей. Чем выше прочность модуля, тем больше связей скрыто от внешней по отношению к нему части программы и, следовательно, тем проще сама программа. Самой слабой степенью прочности обладает модуль, *прочный по совпадению.* В данном случае в программный модуль оформляется повторяющаяся в нескольких местах программы последовательность операторов. Если вдруг возникнет необходимость изменения этой последовательности в одном из контекстов, придется изменять сам модуль, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования.

Функционально прочный модуль — это модуль, реализующий одну какую-либо определенную функцию. При этом он может использовать и другие модули. Такой вид прочности модулей рекомендуется для использования.

Высшей степенью прочности обладает *информационно прочный модуль* — это модуль, выполняющий несколько операций над одной и той же структурой данных, которая неизвестна вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.

Сцепление модуля — это мера его зависимости по способу передачи данных от других модулей. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления существует шесть видов сцепления модулей по:

- данным;
- образцу;

- управлению;
- внешним ссылкам;
- общей области данных;
- содержимому.

Худшим видом сцепления модулей является *сцепление по содержимому.* Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо.

Не рекомендуется использовать также *сцепление по общей области* — это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти.

Сцепление по образцу предполагает, что модули обмениваются данными, объединенными в структуры. Этот тип обеспечивает неплохие характеристики по сравнению с предыдущими. Недостаток заключается в том, что конкретные передаваемые данные «спрятаны» в структуры, и потому уменьшается «прозрачность» связи между модулями. Кроме того, при изменении структуры передаваемых данных необходимо модифицировать все использующие ее модули.

При *сцеплении по управлению* один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней логикой модуля. Таким способом часто выполняют настройку режимов работы программного обеспечения. Подобные настройки также снижают наглядность взаимодействия модулей и потому обеспечивают не лучшие характеристики технологичности разрабатываемого программного обеспечения.

Сцепление по внешним ссылкам предполагает, что модули ссылаются на один и тот же глобальный элемент данных [1].

Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является *сцепление по данным* (параметрическое сцепление) — это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

Рутинность модуля — это его независимость от предыстории обращений к нему. Модуль будем называть *рутинным*, если ре-

зультат обращения к нему зависит только от значений его параметров и не зависит от результатов предыдущих обращений к нему. Модуль будем называть зависящим от предыстории, если результат обращения к нему зависит от внутреннего состояния этого модуля, хранящего следы предыдущих обращений к нему. В книге Майерса [45] не рекомендуется использовать зависящие от предыстории модули, так как они провоцируют появление в программах неуловимых ошибок. Однако во многих случаях именно зависящий от предыстории модуль является наиболее информационно прочным. Поэтому более приемлема следующая рекомендация:

- всегда следует использовать рутинный модуль, если это не приводит к плохим сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
- в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение данного модуля при разных последующих обращениях к нему.

Связность модулей — мера прочности соединения функциональных и информационных объектов внутри одного модуля. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи, в то время как помещение сильно связанных элементов в разные модули не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов также уменьшает технологичность модулей, делая их сложнее для понимания.

Различают следующие виды связности (в порядке убывания уровня) [1]:

- функциональную;
- последовательную;
- информационную (коммуникативную);
- процедурную;
- временную;
- логическую;
- случайную.

При *функциональной связности* модуль предназначен для выполнения одной функции. Его исходные данные и операции предназначены для решения одной конкретной задачи. Такой

модуль имеет максимальную связность и, как следствие, хорошую технологичность (простота компиляции, тестирования, сопровождения).

При *последовательной связности* модуля результат обработки данных одной функцией служит исходными данными для другой функции. Такой модуль реализует одну подпрограмму, выполняющую две функции. Модуль с последовательной связностью функций можно разбить на два модуля или более, как с последовательной, так и с функциональной связностью. При этом данные, используемые последовательными функциями, также связаны последовательно. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже с точки зрения понимания и тестирования.

Информационно связанными считают функции, обрабатывающие одни и те же данные. Информационно связанный модуль имеет неплохие показатели технологичности, так как все функции, работающие с одними и теми же данными, собраны в один модуль, что позволяет при изменении формата данных корректировать только его. Данные, которые обрабатываются одной функцией, также считают информационно связанными.

Процедурно связаны функции или данные, которые являются частями одного процесса. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими операции связаны лишь общим процессом, следовательно, технологичность такого модуля хуже, чем у предыдущих.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени. *Временная связность данных* означает, что они используются в некотором временном интервале. Отличительной особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса. Большая вероятность модификации функции еще больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущими, кроме того, содержание модуля с временной связностью функций может изменяться: в него могут включаться новые действия и/или исключаться старые.

Логическая связь строится на основе объединения данных или функций в одну логическую группу, например, логически связаны компоненты модуля, содержащего функции обработки

текстовой информации или данные одного и того же типа. При выполнении модуля с логически связанными компонентами всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению. Показатели технологичности таких модулей ниже предыдущих, так как сложно понять логику их работы.

Модуль, элементы которого имеют *случайную связность*, имеет самые низкие показатели технологичности, так как его элементы вообще не связаны.

В табл. 3.1 представлены характеристики различных видов связности по экспертным оценкам [1, 62, 63].

Таблица 3.1. **Сравнительные характеристики различных видов связности**

Вид связности	Сцепление, балл	Наглядность (понятность)	Возможность изменения	Сопровождаемость
Функциональная	10	Хорошая	Хорошая	Хорошая
Последовательная	9	Хорошая	Хорошая	Хорошая
Информационная	8	Средняя	Средняя	Средняя
Процедурная	5	Средняя	Средняя	Плохая
Временная	3	Средняя	Средняя	Плохая
Логическая	1	Плохая	Плохая	Плохая
Случайная	0	Плохая	Плохая	Плохая

Анализ табл. 3.1 показывает, что при проектировании программных модулей лучше всего использовать функциональную, последовательную и информационную связности.

3.4.3. Модульная структура программных продуктов

Модульная структура программы представляет собой древовидную структуру, в узлах которой размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей. Если в тексте модуля имеется ссылка на другой модуль, то их на структурной схеме соединяет дуга, которая исходит из первого и входит во второй модуль. Другими словами, каждый модуль может обращаться к подчиненным ему моду-

лям. При этом модульная структура программной системы, кроме структурной схемы, должна включать в себя еще и совокупность спецификаций модулей, образующих эту систему [37].

Функция верхнего уровня обеспечивается главным модулем; он управляет выполнением нижестоящих функций, которым соответствуют подчиненные модули.

При определении набора модулей, реализующих функции конкретного алгоритма, необходимо учитывать следующее:

1) модуль вызывается на выполнение вышестоящим по иерархии модулем и, закончив работу, возвращает ему управление;

2) принятие основных решений в алгоритме выносится на максимально высокий по иерархии уровень;

3) если в разных местах алгоритма используется одна и та же функция, то она оформляется в отдельный модуль, который будет вызываться по мере необходимости.

Состав, назначение и характер использования программных модулей в значительной степени определяются инструментальными средствами.

Например, при разработке СУБД используются следующие программные модули:

1) экранные формы ввода и/или редактирования информации базы данных;

2) отчеты;

3) макросы;

4) стандартные средства для обработки информации;

5) меню для выбора функции обработки и др.

3.4.4. Методы разработки при модульном программировании

Спецификация программного модуля состоит из функциональной спецификации модуля, описывающей семантику функций, выполняемых этим модулем по каждому из его входов, и синтаксической спецификации его входов, позволяющей построить на используемом языке программирования синтаксически правильное обращение к нему. Функциональная спецификация модуля определяется теми же принципами, что и функциональная спецификация программной системы.

Существуют разные методы разработки модульной структуры программы, в зависимости от которых определяется порядок

программирования и отладки модулей, указанных в этой структуре. Обычно в литературе обсуждаются два метода [42, 46]: метод восходящей разработки и метод нисходящей разработки.

Метод восходящей разработки

Сначала строится древовидная модульная структура программы. Затем поочередно проектируются и разрабатываются модули программы, начиная с модулей самого нижнего уровня, затем предыдущего уровня и т. д. То есть модули реализуются в таком порядке, чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же восходящем порядке. Достоинство метода заключается в том, что каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули. Недостатки метода восходящей разработки заключаются в следующем:

- на нижних уровнях модульной структуры спецификации могут быть еще определены не полностью, что может привести к полной переработке этих модулей после уточнения спецификаций на верхнем уровне;
- для восходящего тестирования всех модулей, кроме головного, который является модулем самого верхнего уровня, приходится создавать вызывающие программы, что приводит к созданию большого количества отладочного материала, но не гарантирует, что результаты тестирования верны;
- головной модуль проектируется и реализуется в последнюю очередь, что не дает продемонстрировать его заказчику для уточнения спецификаций.

Метод нисходящей разработки

Как и в предыдущем методе, сначала строится модульная структура программы в виде дерева. Затем проектируются и реализуются модули программы, начиная с модуля самого верхнего уровня — головного, далее разрабатываются модули уровнем ниже и т. д. При этом переход к программированию какого-либо модуля осуществляется только в том случае, если уже

запрограммирован модуль, который к нему обращается. Затем производится их поочередное тестирование и отладка в таком же нисходящем порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т. е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. Первым тестируется головной модуль программы, который представляет всю тестируемую программу, при этом все модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми «заглушками» [45]). Каждый *имитатор модуля* является простым программным фрагментом, реализующим сам факт обращения к данному модулю с необходимой для правильной работы программы обработкой значений его входных параметров и с выдачей, если это необходимо, подходящего результата. Далее производится тестирование следующих по уровню модулей. Для этого имитатор выбранного для тестирования модуля заменяется самим модулем, и добавляются имитаторы модулей, к которым может обращаться тестируемый модуль. При таком подходе каждый модуль будет тестироваться в «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования заменяется программированием достаточно простых имитаторов используемых в программе модулей.

Недостатком нисходящего подхода к программированию является необходимость абстрагироваться от реальных возможностей выбранного языка программирования и придумывать абстрактные операции, которые позже будут реализованы с помощью модулей. Однако способность к таким абстракциям является необходимым условием разработки больших программных средств.

Рассмотренные выше методы (нисходящей и восходящей разработок), являющиеся классическими, требуют, чтобы модульная древовидная структура была готова до начала программирования модулей. Как правило, точно и содержательно разработать структуру программы до начала программирования невозможно. При конструктивном и архитектурном подходах к разработке модульная структура формируется в процессе реализации модулей.

Конструктивный подход

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом (спецификация программы является одновременно спецификацией головного модуля). В процессе программирования головного модуля в случае, если эта программа достаточно большая, выделяются подзадачи (некоторые функции) и для них создаются спецификации реализующих эти подзадачи фрагментов программы. В дальнейшем каждый из этих фрагментов будет представлен поддеревом модулей (спецификация выделенной функции является одновременно спецификацией головного модуля этого поддерева).

Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя часть дерева, например, как на рис. 3.12.

Рис. 3.12. Первый шаг формирования модульной структуры программы при конструктивном подходе

По тому же принципу производится программирование следующих по уровню специфицированных, но еще не запрограммированных модулей в соответствии со сформированным дере-

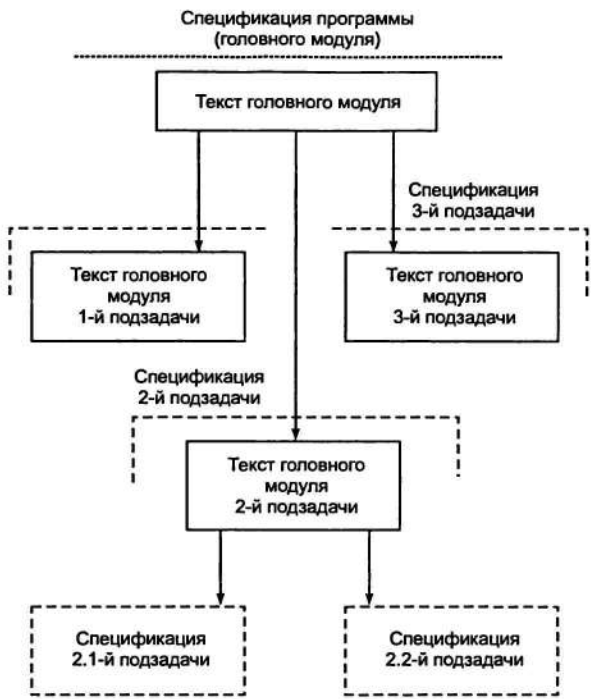


Рис. 3.13. Второй шаг формирования модульной структуры программы при конструктивном подходе

вом. В результате к дереву добавляются очередные уровни, как показано на рис. 3.13.

Архитектурный подход

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Целью разработки в данном методе является повышение уровня языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, специфицируются, а затем и программируются отдельные программные модули, выполняющие эти функции. Сначала в виде модулей реализуются более простые функции, а затем создаются модули, использующие уже имеющиеся функции, и т. д. Это позволяет

существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней уже имеющихся и проверенных на практике модульных структур нижнего уровня, что также позволяет бороться с дублированием в программировании. В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются для того, чтобы облегчить их применение настройкой параметров.

Нисходящая реализация

В классическом методе нисходящей разработки сначала все модули разрабатываемой программы программируются, а затем тестируются в нисходящем порядке. При этом тестирование и отладка модулей могут привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы. В результате может оказаться, что часть модулей вообще не нужна в данной структуре, а часть модулей придется переписывать. Более рационально каждый запрограммированный модуль тестировать сразу же до перехода к программированию другого модуля. Такой метод в литературе получил название *метода нисходящей реализации.*

Целенаправленная конструктивная реализация

В зависимости от того, в какой последовательности в процессе разработки программы обходятся узлы дерева, существуют разновидности описанных выше методов. Например, метод *целенаправленной конструктивной реализации,* при котором в рамках конструктивного подхода реализуются сначала модули, необходимые для самого простого рабочего варианта программы, остальные модули заменяются их имитаторами. Затем к имеющемуся варианту добавляются другие модули, обеспечивающие работу программы для других наборов данных, и так далее до полной реализации программы. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика.

Подводя итог сказанному, на рис. 3.14 представлена общая схема классификации рассмотренных методов разработки структуры программы.

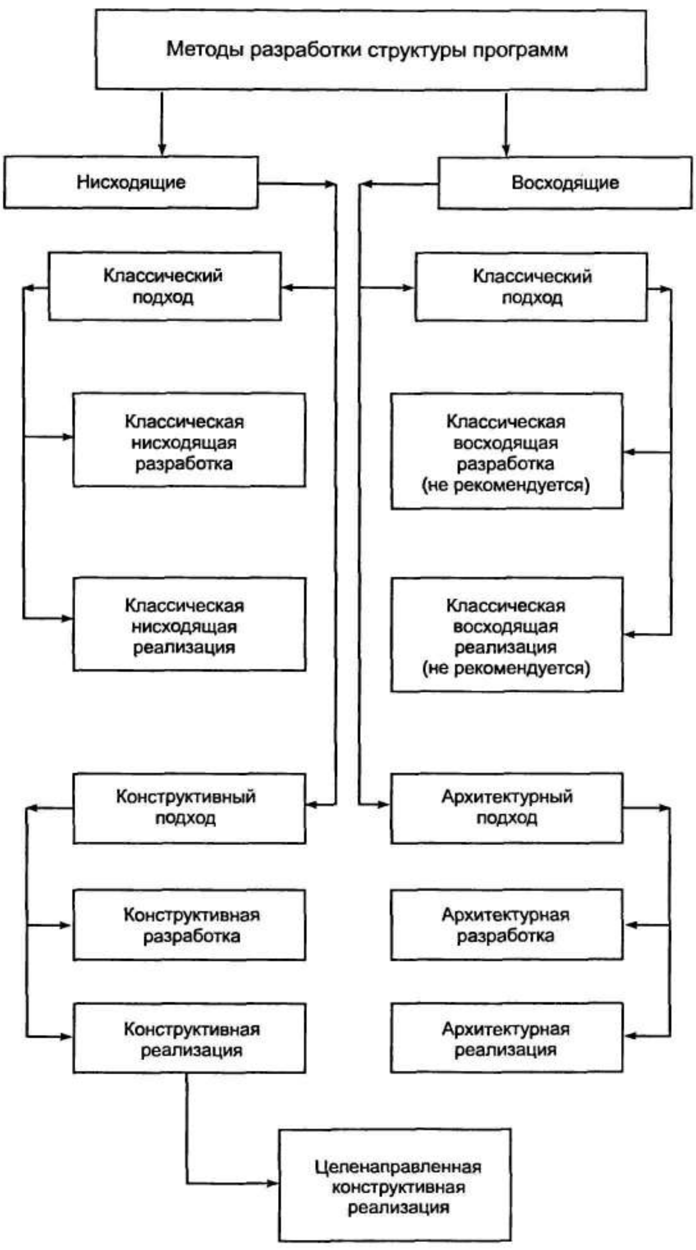


Рис. 3.14. Классификация методов разработки структуры программ

3.5. Анализ требований и определение спецификаций при структурном подходе

На этом этапе необходимо построить модели ПО во взаимодействии с окружающей средой. Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, рекомендуется использовать сразу несколько моделей и сопровождать их описаниями. Структурный подход к проектированию программных продуктов предполагает разработку следующих моделей [1, 53]:

- диаграмм потоков данных (DFD — Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе;
- диаграмм «сущность—связь» (ERD Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы;
- диаграмм переходов состояний (STD — State Transition Diagrams), характеризующих поведение системы во времени;
- функциональных диаграмм (методология SADT);
- спецификаций процессов;
- словаря терминов.

3.5.1. Спецификации процессов

Спецификации процессов могут быть представлены в виде псевдокодов, блок-схем алгоритмов, Flow-форм, диаграмм Насси — Шнейдермана или просто краткого текстового описания [1].

При структурном программировании различают три вида вычислительного процесса: линейный, разветвленный и циклический.

Линейная структура — выполнение операторов последовательно.

Разветвленная структура — в зависимости от выполнения некоторого условия выполняется та или иная последовательность операторов.

Циклическая структура — многократное выполнение одинаковой последовательности операторов.

Схемы алгоритмов

Для изображения схем алгоритмов разработан ГОСТ 19.701—90 (табл. 3.2).

Таблица 3.2. **Обозначение элементов схем алгоритмов**

Название	Обозначение	Назначение
Терминатор	Действие	Начало, завершение программы или подпрограммы
Процесс	Действие	Обработка данных (вычисления, пересылки и т. п.)
Данные	Данные	Операции ввода-вывода
Решение	Условие	Ветвление, выбор, поисковые и итерационные циклы
Подготовка	Действия	Счетные циклы
Граница цикла	Начало	Любые циклы
Предопределенный процесс	Имя	Вызов процедур
Соединитель	Имя	Маркировка разрывов линий
Комментарий	Комментарий	Пояснения к операциям

Любой, сколь угодно сложный, алгоритм можно представить с использованием трех основных конструкций, которые получили название базовых [1]:

- *следование.* Обозначает последовательное выполнение действий (рис. 3.15, *а*);
- *ветвление.* Соответствует выбору одного из двух вариантов действий (рис. 3.15, *б*);
- *цикл-пока.* Определяет повторение действий, пока не будет нарушено некоторое условие, выполнение которого проверяется в начале цикла (рис. 3.15, *в*).

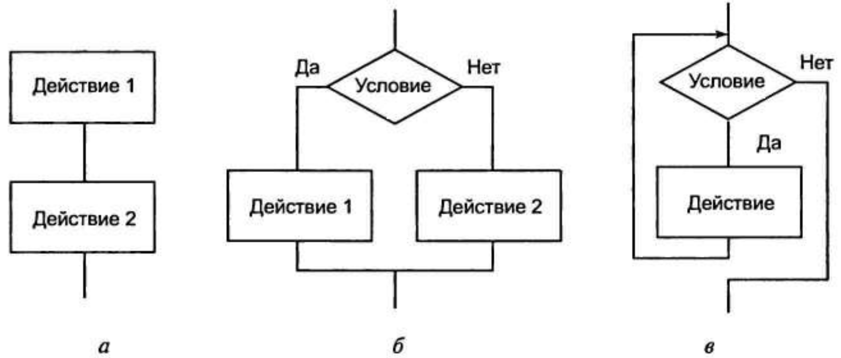


Рис. 3.15. Базовые алгоритмические структуры:
а — следование; *б* — ветвление; *в* — цикл-пока

Помимо базовых, процедурные языки программирования высокого уровня обычно используют еще три конструкции, которые можно составить из базовых:

- *выбор.* Обозначает выбор одного варианта из нескольких в зависимости от значения некоторой величины (рис. 3.16, *а*);
- *цикл-до.* Обозначает повторение некоторых действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле (рис. 3.16, *б*);
- *цикл с заданным числом повторений (счетный цикл).* Обозначает повторение некоторых действий указанное количество раз (рис. 3.16, *в*).

Любая из дополнительных конструкций легко реализуется с помощью базовых. Перечисленные шесть конструкций были положены в основу структурного программирования.

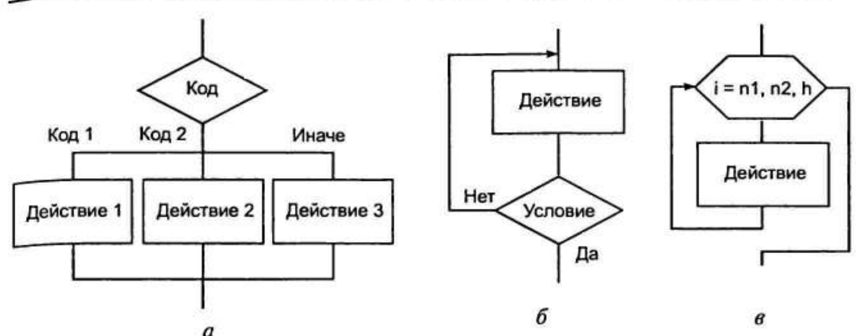


Рис. 3.16. Дополнительные структуры алгоритмов:
а — выбор; *б* — цикл-до; *в* — цикл с заданным числом повторений

Псевдокоды

Псевдокод — формализованное текстовое описание алгоритма (текстовая нотация). В литературе были предложены несколько вариантов псевдокодов. Один из них приведен в табл. 3.3.

Таблица 3.3. **Описания псевдокодов**

Структура	Псевдокод	Структура	Псевдокод
Следование	<Действие 1> <Действие 2>	Выбор	**Выбор** <код> <код 1>: <Действие 1> <код 2>: <Действие 2> Все-выбор
Ветвление	**Если** <Условие> **то** <Действие 1> **иначе** <Действие 2> Все-если	Цикл с заданным количеством повторений	**Для** <индекс> = <n>,<k>,<h> <Действие> Все-цикл
Цикл-пока	**Цикл-пока** <Условие> <Действие> Все-цикл	Цикл-до	Выполнять <Действие> **До** <Условие>

Flow-формы

Flow-формы представляют собой графическую нотацию описания структурных алгоритмов, которая иллюстрирует вложенность структур. Каждый символ Flow-формы имеет вид прямо-

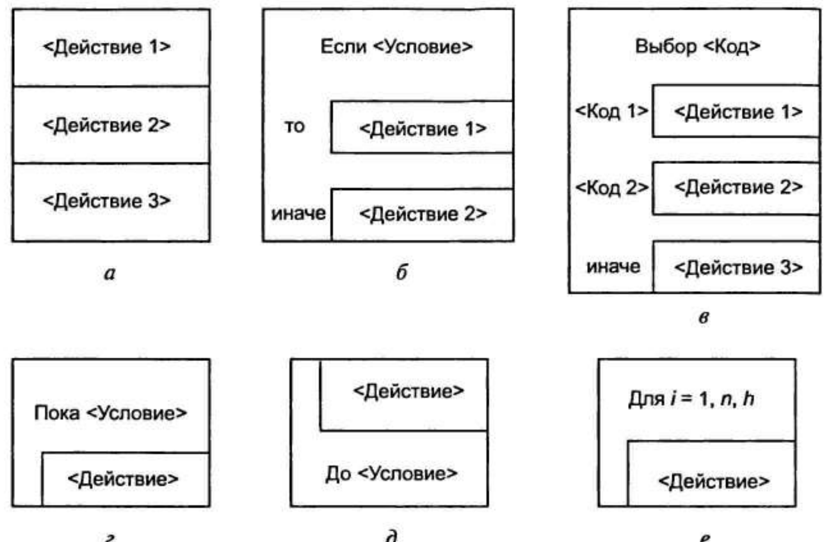


Рис. 3.17. Условные обозначения Flow-форм для основных конструкций: *а* — следование; *б* — ветвление; *в* — выбор; *г* — цикл-до; *д* — цикл-до; *е* — счетный цикл

угольника и может быть вписан в любой внутренний прямоугольник любого другого символа. Нотация Flow-форм приведена на рис. 3.17.

Диаграммы Насси — Шнейдермана

Диаграммы Насси — Шнейдермана являются продолжением Flow-форм. Отличие их от Flow-форм состоит в том, что область обозначения условий изображают в виде треугольников (рис. 3.18). Это обозначение обеспечивает большую наглядность представления алгоритма.

При использовании псевдокодов, Flow-форм и диаграмм Насси — Шнейдермана описать неструктурный алгоритм невозможно (для неструктурных передач управления в этих нотациях просто отсутствуют условные обозначения).

По сравнению с псевдокодами Flow-формы и диаграммы Насси — Шнейдермана, являясь графическими, лучше отображают вложенность конструкций.

Общим недостатком Flow-форм и диаграмм Насси — Шнейдермана является сложность построения изображений символов,

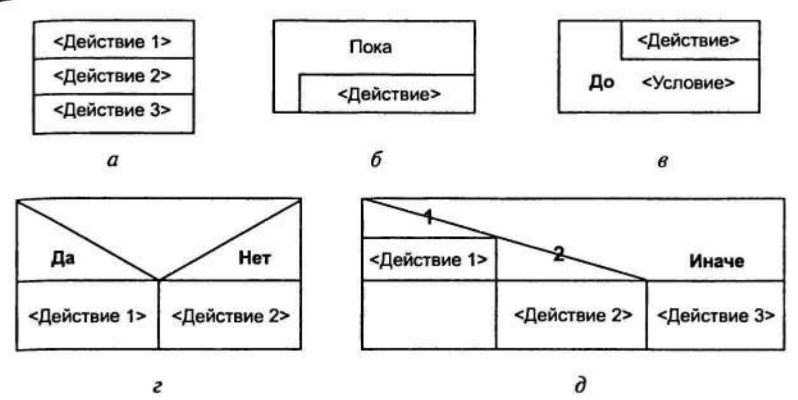


Рис. 3.18. Условные обозначения диаграмм Насси — Шнейдермана для основных конструкций:
а — следование; *б* — цикл-пока; *в* — цикл-до; *г* — ветвление; *д* — выбор

что затрудняет практическое применение этих нотаций для описания больших алгоритмов.

3.5.2. Словарь терминов

Словарь терминов представляет собой краткое описание основных понятий, используемых при составлении спецификаций. Он предназначен для повышения степени понимания предметной области и исключения риска возникновения разногласий при обсуждении моделей между заказчиками и разработчиками [1].

Обычно описание термина в словаре выполняют по следующей схеме:

- термин;
- категория (понятие предметной области, элемент данных, условное обозначение и т. д.);
- краткое описание.

Пример:

```
Термин   Web-сайт

Категория   Интернет-программирование

Описание  Совокупность Web-страниц с повторяющимся
дизайном, объединенных по смыслу, навигационно и
физически находящихся на одном сервере.
```

3.5.3. Диаграммы переходов состояний (SDT)

SDT демонстрирует поведение разрабатываемой программной системы при получении управляющих воздействий (извне).

В диаграммах такого вида узлы соответствуют состояниям динамической системы, а дуги — переходу системы из одного состояния в другое. Узел, из которого выходит дуга, является начальным состоянием, узел, в который дуга входит, — следующим. Дуга помечается именем входного сигнала или события, вызывающего переход, а также сигналом или действием, сопровождающим переход. Условные обозначения, используемые при построении диаграмм переходов состояний, показаны на рис. 3.19.

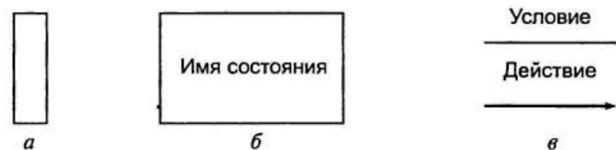


Рис. 3.19. Условные обозначения диаграмм переходов состояний: *а* — терминальное состояние; *б* — промежуточное состояние; *в* — переход

На рис. 3.20 представлена диаграмма переходов состояний программы, активно не взаимодействующей с окружающей средой, которая имеет примитивный интерфейс, производит некоторые вычисления и выводит простой результат [1].

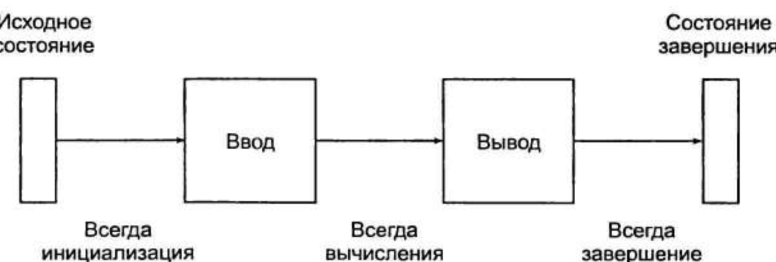


Рис. 3.20. Диаграмма переходов состояний программного обеспечения, активно не взаимодействующего с окружающей средой

На рис. 3.21 представлена диаграмма переходов торгового автомата, активно взаимодействующего с покупателем [55].

Рис. 3.21. Диаграмма переходов состояний торгового автомата

3.5.4. Функциональные диаграммы

Функциональными называют диаграммы, в первую очередь отражающие взаимосвязи функций разрабатываемого программного обеспечения [53].

Они создаются на ранних этапах проектирования систем, для того чтобы помочь проектировщику выявить основные функции и составные части проектируемой системы и, по возможности, обнаружить и устранить существенные ошибки. Современные методы структурного анализа и проектирования предоставляют разработчику определенные синтаксические и графические средства проектирования функциональных диаграмм информационных систем.

В качестве примера рассмотрим методологию SADT, предложенную Дугласом Россом. На ее основе разработана, в частности, известная методология IDEF0 (Icam DEFinition). Методология SADT представляет собой набор 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 меж-

ду этими действиями. Основные элементы этой методологии основываются на следующих концепциях:

- графическое представление блочного моделирования. На SADT-диаграмме функции представляются в виде блока, а интерфейсы входа-выхода — в виде дуг, соответственно входящих в блок и выходящих из него. Интерфейсные дуги отображают взаимодействие функций друг с другом;
- строгость и точность отображения.

Правила SADT включают:

- уникальность меток и наименований;
- ограничение количества блоков на каждом уровне декомпозиции;
- синтаксические правила для графики;
- связность диаграмм;
- отделение организации от функции;
- разделение входов и управлений.

Методология SADT может использоваться для моделирования и разработки широкого круга систем, удовлетворяющих определенным требованиям и реализующих требуемые функции. В уже разработанных системах методология SADT может быть использована для анализа выполняемых ими функций, а также для указания механизмов, посредством которых они осуществляются.

Диаграммы — главные компоненты модели, все функции программной системы и интерфейсы на них представлены как блоки и дуги. Место соединения дуги с блоком определяет тип интерфейса. Дуга, обозначающая управление, входит в блок сверху, в то время как информация, которая подвергается обработке, представляется дугой с левой стороны блока, а результаты обработки — дугами с правой стороны. Механизм (человек или автоматизированная система), который осуществляет операцию, представляется в виде дуги, входящей в блок снизу (рис. 3.22).

Рис. 3.22. Функциональный блок и интерфейсные дуги

Блоки на диаграмме размещают по «ступенчатой» схеме в соответствии с последовательностью их работы или *доминированием,* которое понимается как влияние, оказываемое одним блоком на другие. В функциональных диаграммах SADT различают пять типов влияний блоков друг на друга [1]:

- вход-выход блока подается на вход блока с меньшим доминированием, т. е. следующего (рис. 3.23, *а*);
- управление. Выход блока используется как управление для блока с меньшим доминированием (рис. 3.23, *б*);
- обратная связь по входу. Выход блока подается на вход блока с большим доминированием (рис. 3.23, *в*);
- обратная связь по управлению. Выход блока используется как управляющая информация для блока с большим доминированием (рис. 3.23, *г*);
- выход-исполнитель. Выход блока используется как механизм для другого блока (рис. 3.23, *д*).

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

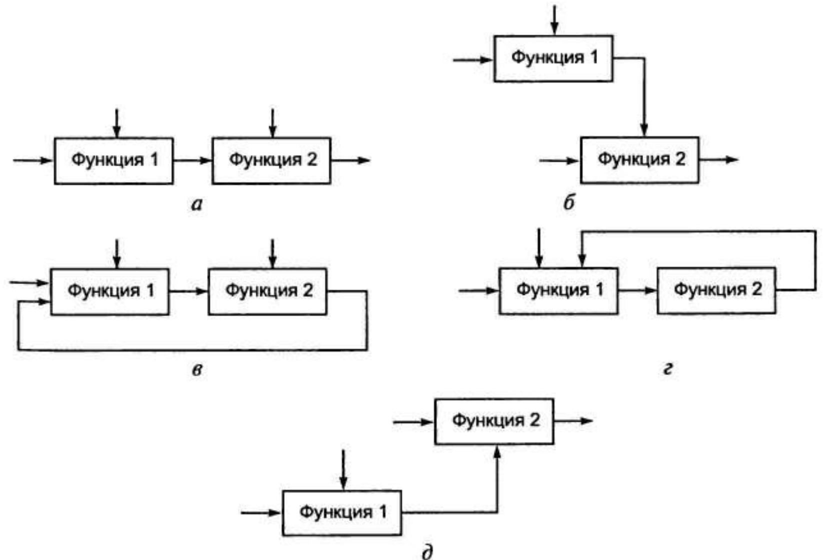


Рис. 3.23. Типы влияний блоков:
а — вход; *б* — управление; *в* — обратная связь по входу; *г* — обратная связь по управлению; *д* — выход-исполнитель

На рис. 3.24 приведены четыре диаграммы и их взаимосвязи, показывающие структуру SADT-модели. Каждый компонент модели может быть декомпозирован на другой диаграмме. Деталь-

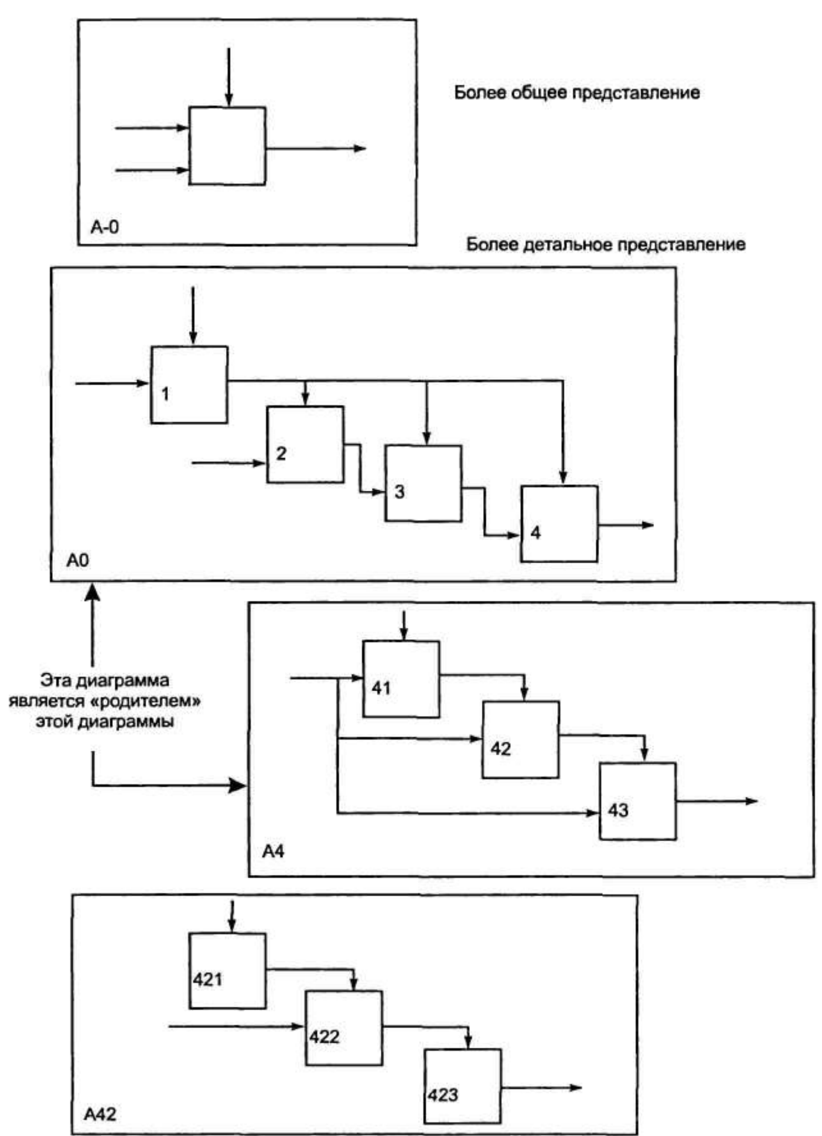


Рис. 3.24. Структура SADT-модели. Декомпозиция диаграмм

ная диаграмма иллюстрирует внутреннее строение блока «родительской» диаграммы.

Иерархия диаграмм

Прежде всего, вся система представляется в виде простейшей компоненты — одного блока и дуг, представляющих собой интерфейсы с внешними по отношению к данной системе функциями. Имя блока является общим для всей системы [53].

Затем блок, который представляет систему в целом, детализируется на следующей диаграмме. Он представляется в виде нескольких блоков, соединенных интерфейсными дугами. Каждый блок детальной диаграммы представляет собой подфункцию, границы которой определены интерфейсными дугами. Каждый из блоков детальной диаграммы может быть также детализирован на следующей в иерархии диаграмме. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Все диаграммы связывают друг с другом иерархической нумерацией блоков: первый уровень — АО, второй — A1, A2 и т. п., третий — A11, A12, A13 и т. п., где первые цифры — номер родительского блока, а последняя — номер конкретного блока детальной диаграммы.

Во всех случаях каждая подфункция может содержать только те элементы, которые входят в исходную функцию, причем никакие из них не могут быть опущены. То есть родительский блок и его интерфейсы обеспечивают контекст. К нему нельзя ничего добавить, и из него не может быть ничего удалено.

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. 3.25—3.27 представлены различные варианты выполнения функций и соединения дуг с блоками.

Последовательность операций, время их выполнения не указываются на SADT-диаграммах. 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. (см. рис. 3.27).

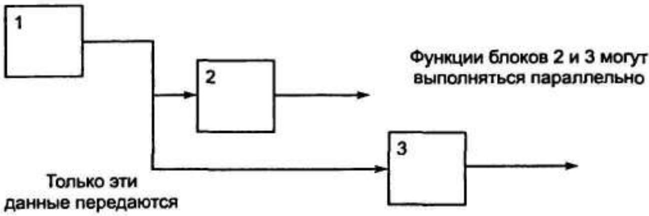
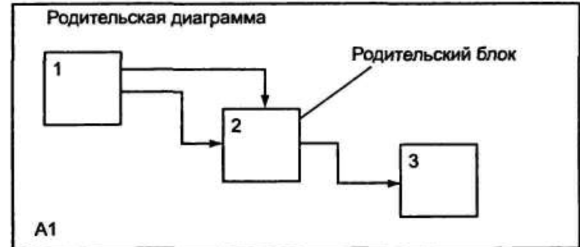


Рис. 3.25. Одновременное выполнение



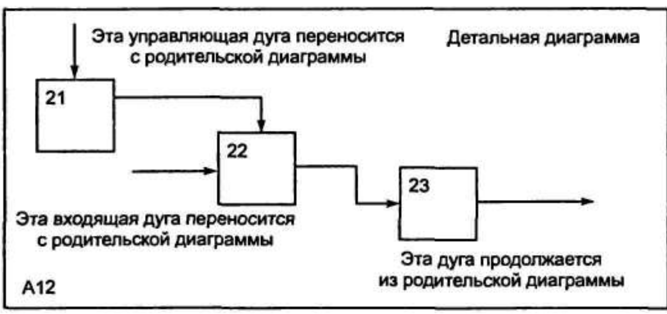


Рис. 3.26. Соответствие родительской и детальной диаграмм

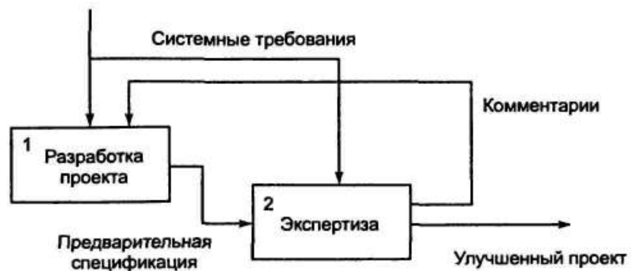


Рис. 3.27. Пример обратной связи

Пример 3.1. Разработку функциональных диаграмм продемонстрируем на примере уточнения спецификаций программы сортировки одномерного массива с использованием нескольких методов.

Диаграмма, представленная на рис. 3.28, *а,* является диаграммой верхнего уровня. Она иллюстрирует исходные данные программы и ожидаемые результаты.

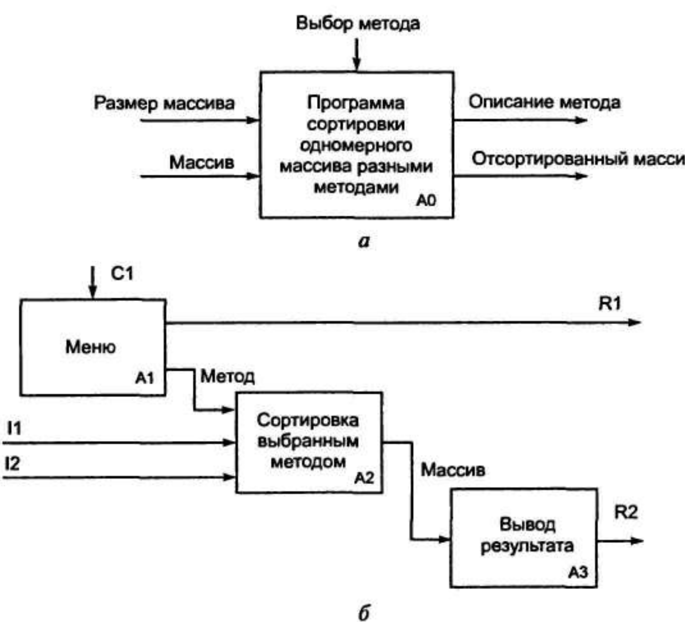


Рис. 3.28. Функциональные диаграммы для программы сортировки массива: *а* — диаграмма верхнего уровня; *б* — уточняющая диаграмма

Диаграмма, представленная на рис. 3.28, *б,* детализирует функции программы. На ней показаны три блока: **Меню**, **Сортировка**, **Вывод результата**. Для каждого блока определены исходные данные, управляющие воздействия и результаты. На детализирующей диаграмме используются следующие обозначения:

I1 — размер массива;
I2 — массив;
C1 — выбор метода;
R1 — вывод описания метода;
R2 — отсортированный массив.

3.5.5. Диаграммы потоков данных (DFD)

Такая диаграмма состоит из трех типов узлов: узлов обработки данных, узлов хранения данных и внешних узлов, представляющих внешние по отношению к используемой диаграмме источники или потребители данных. Дуги в диаграмме соответствуют потокам данных, передаваемых от узла к узлу. Они помечены именами соответствующих данных. Описание процесса, функции или системы обработки данных, соответствующих узлу диаграммы, может быть представлено диаграммой следующего уровня детализации, если процесс достаточно сложен [1, 53].

Для изображения диаграмм потоков данных традиционно используют два вида нотаций: нотации Йордана и Гейна — Сарсона (табл. 3.4).

Таблица 3.4. **Обозначения элементов диаграмм потоков данных**

Понятие	Описание	Нотация Йордана	Нотация Гейна — Сарсона
Внешняя сущность	Внешний по отношению к системе объект, обменивающийся с нею потоками данных	Имя внешнего объекта	
Функция	Действие, выполняемое моделируемой системой	Имя функции	Номер Имя Механизм
Поток данных	Объект, над которым выполняется действие. Может быть информационным (логическим) или управляющим. (Управляющие потоки обозначаются пунктирной линией со стрелкой)	Имя объекта	Имя объекта
Хранилище данных	Структура для хранения **информационных** объектов	Имя объекта	№ Имя объекта

Первым шагом при построении иерархии диаграмм потоков данных является построение контекстных диаграмм, показывающих, как система будет взаимодействовать с пользователями и другими внешними системами. При проектировании простых систем достаточно одной контекстной диаграммы, имеющей звездную топологию, в центре которой располагается основной процесс, соединенный с источниками и приемниками информации.

Для сложных систем строится иерархия контекстных диаграмм, которая определяет взаимодействие основных функциональных подсистем проектируемой системы как между собой, так и с внешними входными и выходными потоками данных и внешними объектами. При этом контекстная диаграмма верхнего уровня содержит набор подсистем, соединенных потоками данных. Контекстные диаграммы следующего уровня детализируют содержимое и структуру подсистем.

После построения контекстных диаграмм полученную модель следует проверить на полноту исходных данных и отсутствие информационных связей с другими объектами.

Для каждой подсистемы, присутствующей на контекстных диаграммах, выполняется ее детализация с помощью диаграмм потоков данных, при этом необходимо соблюдать следующие правила [53]:

- правило балансировки. Означает, что при детализации подсистемы можно использовать только те компоненты (подсистемы, процессы, внешние сущности, накопители данных), с которыми она имеет информационную связь на родительской диаграмме;
- правило нумерации. Означает, что при детализации подсистем должна поддерживаться их иерархическая нумерация. Например, подсистемы, детализирующие подсистему с номером 2, получают номера 2.1, 2.2, 2.3 и т. д.

При построении иерархии диаграмм потоков данных переходить к детализации процессов следует только после определения структур данных, которые описывают содержание всех потоков и накопителей данных. Структуры данных могут содержать альтернативы, условные вхождения и итерации. Условное вхождение означает, что соответствующие компоненты могут отсутствовать в структуре. Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация означает, что компонент может повторяться в структуре некоторое указанное число раз. Для каждого элемента данных может указываться его тип (непрерывный или дискретный). Для непрерывных данных может указываться единица измерения (кг, см и т. п.), диапазон значений, точность представления и форма физического кодирования. Для дискретных данных может указываться таблица допустимых значений.

Построенную модель системы необходимо проверить на полноту и согласованность. В полной модели все ее объекты (под-

системы, процессы, потоки данных) должны быть подробно описаны и детализированы. Выявленные недетализированные объекты следует детализировать, вернувшись на предыдущие шаги разработки. В согласованной модели для всех потоков данных и накопителей данных должно выполняться правило сохранения информации: все поступающие куда-либо данные должны быть считаны, а все считываемые данные должны быть записаны.

В соответствии с вышесказанным процесс построения модели разбивается на следующие этапы [39]:

1. Выделение множества требований в основные функциональные группы — процессы.
2. Выявление внешних объектов, связанных с разрабатываемой системой.
3. Идентификация основных потоков информации, циркулирующей между системой и внешними объектами.
4. Предварительная разработка контекстной диаграммы.
5. Проверка предварительной контекстной диаграммы и внесение в нее изменений.
6. Построение контекстной диаграммы путем объединения всех процессов предварительной диаграммы в один процесс, а также группирования потоков.
7. Проверка основных требований контекстной диаграммы.
8. Декомпозиция каждого процесса текущей DFD с помощью детализирующей диаграммы или спецификации процесса.
9. Проверка основных требований по DFD соответствующего уровня.
10. Добавление определений новых потоков в словарь данных при каждом их появлении на диаграммах.
11. Проверка полноты и наглядности модели после построения каждых двух-трех уровней.

Пример 3.2. Разработаем иерархию диаграмм потоков данных программы сортировки одномерных массивов.

Для начала построим контекстную диаграмму, для чего определим внешние сущности и потоки данных между программой и внешними сущностями. Внешней сущностью по отношению к программе является *Пользователь*. Он выбирает метод сортировки и вводит исходные данные, а затем получает от программы описание выбранного метода и отсортированный массив. На рис. 3.29 представлена контекстная диаграмма данной программы.

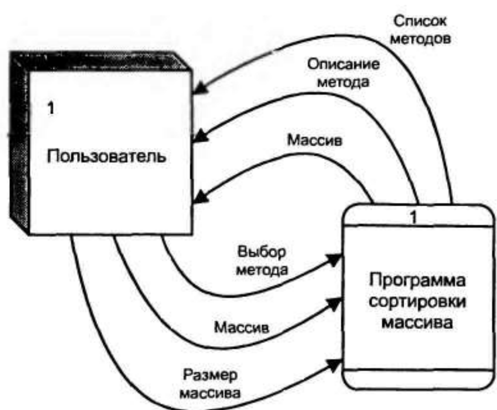


Рис. 3.29. Контекстная диаграмма программы сортировки массива

После детализации получилось три процесса: **Меню, Сортировка, Вывод результата**. Для хранения описаний алгоритмов служит **Хранилище алгоритмов**. Теперь определим потоки данных.

Детализирующая диаграмма потоков данных изображена на рис. 3.30. Как мы видим, она несколько отличается от функциональной диаграммы (см. рис. 3.28), например, на ней показано хранилище данных для хранения описаний алгоритмов. Это отличие является важным при проектировании баз данных.

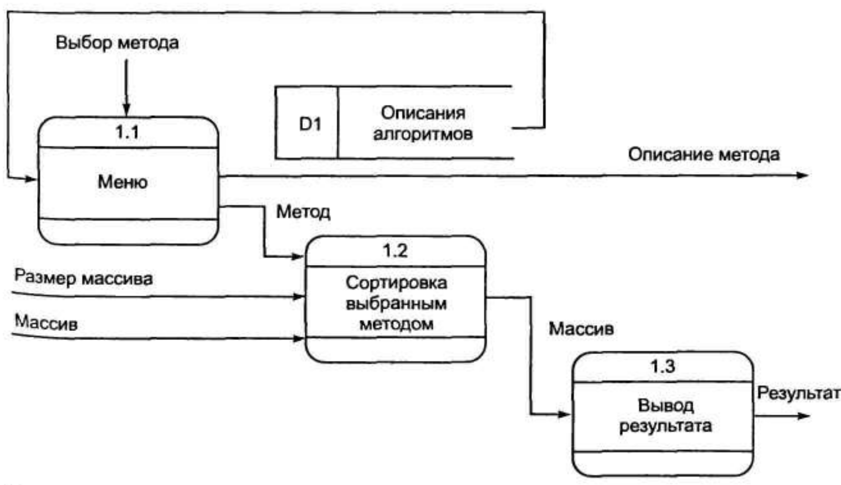


Рис. 3.30. Детализирующая диаграмма потоков данных программы сортировки одномерного массива (нотация Гейна — Сарсона)

3.5.6. Диаграммы сущность—связь

Базовыми понятиями ER-модели данных (ER — Entity-Relationship) являются сущность, атрибут и связь [55].

Первый вариант модели «сущность—связь» был предложен в 1976 г. Питером Пин-Шэн Ченом. В дальнейшем многими авторами были разработаны свои варианты подобных моделей (нотация Мартина, нотация IDEF1X, нотация Баркера и др.). Кроме того, различные программные средства, реализующие одну и ту же нотацию, могут отличаться своими возможностями. Все варианты диаграмм «сущность—связь» исходят из одной идеи — рисунок всегда нагляднее текстового описания. Все такие диаграммы используют графическое изображение сущностей предметной области, их свойств (атрибутов) и взаимосвязей между сущностями.

Поскольку нотация Баркера является наиболее распространенной, в дальнейшем будем придерживаться именно ее.

Основные понятия ER-диаграмм

Сущность — это класс однотипных объектов, информация о которых должна быть учтена в модели [55]. Сущность имеет наименование, выраженное существительным в единственном числе, и обозначается в виде прямоугольника с наименованием (рис. 3.31, *а*). Примерами сущностей могут быть такие классы объектов, как «Студент», «Сотрудник», «Товар».

Рис. 3.31. Обозначения сущности в нотации Баркера:
а — без атрибутов; *б* — с указанием атрибутов; *в* — с ключевым атрибутом

Экземпляр сущности — это конкретный представитель данной сущности. Например, конкретный представитель сущности «Студент» — «Максимов». Причем сущности должны иметь не-

которые свойства, уникальные для каждого экземпляра этой сущности, для того чтобы различать экземпляры.

Атрибут сущности — это именованная характеристика, являющаяся некоторым свойством сущности. Наименование атрибута должно быть выражено существительным в единственном числе (возможно, с описательными оборотами или прилагательными). Примерами атрибутов сущности «Студент» могут быть такие атрибуты, как «Номер зачетной книжки», «Фамилия», «Имя», «Пол», «Возраст», «Средний балл» и т. п. Атрибуты изображаются в прямоугольнике, обозначающем сущность (рис. 3.31, *б*).

Ключ сущности — это неизбыточный набор атрибутов, значения которых в совокупности являются уникальными для каждого экземпляра сущности. При удалении любого атрибута из ключа нарушается его уникальность. Ключей у сущности может быть несколько. На диаграмме ключевые атрибуты отображаются подчеркиванием (рис. 3.31, *в*).

Связь — это отношение одной сущности к другой или к самой себе. Возможно по одной сущности находить другие, связанные с ней. Например, связи между сущностями могут выражаться следующими фразами — «СОТРУДНИК может иметь несколько ДЕТЕЙ», «СОТРУДНИК обязан числиться точно в одном ОТДЕЛЕ». Графически связь изображается линией, соединяющей две сущности (рис. 3.32).

Рис. 3.32. Пример связи между сущностями

Каждая связь имеет одно или два наименования. Наименование обычно выражается неопределенной формой глагола: «Продавать», «Быть проданным» и т. п. Каждое из наименований относится к своему концу связи. Иногда наименования не пишутся ввиду их очевидности.

Связь может иметь один из следующих типов — рис. 3.33.

Связь типа *один-к-одному* означает, что один экземпляр первой сущности связан точно с одним экземпляром второй сущно-

Рис. 3.33. Типы связей

сти. Такая связь чаще всего свидетельствует о том, что мы неправильно разделили одну сущность на две.

Связь типа *один-ко-многим* означает, что один экземпляр первой сущности связан с несколькими экземплярами второй сущности. Это наиболее часто используемый тип связи. Пример такой связи приведен на рис. 3.32.

Связь типа *много-ко-многим* означает, что каждый экземпляр первой сущности может быть связан с несколькими экземплярами второй сущности, и наоборот. Тип связи много-ко-многим является временным типом связи, допустимым на ранних этапах разработки модели. В дальнейшем такую связь необходимо заменить двумя связями типа один-ко-многим путем создания промежуточной сущности.

Каждая связь может иметь одну из двух модальностей связи (рис. 3.34).

Может

Должен

Рис. 3.34. Модальности связей

Связь может иметь разную модальность с разных концов, как на рис. 3.32. Каждая связь может быть прочитана как слева направо, так и справа налево. Связь на рис. 3.32 читается так:

слева направо: «Сотрудник может иметь несколько детей»;

справа налево: «Ребенок должен принадлежать точно одному сотруднику».

Пример разработки простой ER-диаграммы

Необходимо разработать информационную систему по заказу некоторой оптовой торговой фирмы, которая должна выполнять следующие действия:

- хранить информацию о покупателях;

- печатать накладные на отпущенные товары;
- следить за наличием товаров на складе.

Выделим все существительные в этих предложениях — это будут потенциальные кандидаты на сущности и атрибуты, и проанализируем их (непонятные термины будем выделять знаком вопроса):

- *Покупатель* — явный кандидат на сущность.
- *Накладная* — явный кандидат на сущность.
- *Товар* — явный кандидат на сущность
- *(?)Склад* — а вообще, сколько складов имеет фирма? Если несколько, то это будет кандидатом на новую сущность.
- *(?)Наличие товара* — это, скорее всего, атрибут, но атрибут какой сущности?

Сразу возникает очевидная связь между сущностями — «покупатели могут покупать много товаров» и «товары могут продаваться многим покупателям». Первый вариант диаграммы выглядит, как показано на рис. 3.35.

Рис. 3.35. Первый вариант ER-диаграммы

Задав дополнительные вопросы менеджеру, мы выяснили, что фирма имеет несколько складов. Причем каждый товар может храниться на нескольких складах и быть проданным с любого склада.

Куда поместить сущности «Накладная» и «Склад» и с чем их связать? Спросим себя, как связаны эти сущности между собой и с сущностями «Покупатель» и «Товар»? Покупатели покупают товары, получая при этом накладные, в которые внесены данные о количестве и цене купленного товара. Каждый покупатель может получить несколько накладных. Каждая накладная обязана выписываться на одного покупателя. Каждая накладная обязана содержать несколько товаров (не бывает пустых накладных). Каждый товар, в свою очередь, может быть продан нескольким по-

купателям по нескольким накладным. Кроме того, каждая накладная должна быть выписана с определенного склада, и с любого склада может быть выписано много накладных. Таким образом, после уточнения диаграмма будет выглядеть следующим образом (рис. 3.36).

Рис. 3.36. Промежуточный вариант ER-диаграммы

Пора подумать об атрибутах сущностей. Беседуя с сотрудниками фирмы, мы выяснили следующее:

- каждый покупатель является юридическим лицом и имеет наименование, адрес, банковские реквизиты;
- каждый товар имеет наименование, цену, а также характеризуется единицами измерения;
- каждая накладная имеет уникальный номер, дату выписки, список товаров с количествами и ценами, а также общую сумму накладной. Накладная выписывается с определенного склада и на определенного покупателя;
- каждый склад имеет свое наименование.

Снова выпишем все существительные, которые будут потенциальными атрибутами, и проанализируем их:

- *Юридическое лицо* — термин риторический, мы не работаем с физическими лицами. Не обращаем внимания;
- *Наименование покупателя* — явная характеристика покупателя;

- *Адрес* — явная характеристика покупателя;
- *Банковские реквизиты* — явная характеристика покупателя;
- *Наименование товара* — явная характеристика товара;
- *(?)Цена товара* — похоже, что это характеристика товара. Отличается ли эта характеристика от цены в накладной?
- *Единица измерения* — явная характеристика товара;
- *Номер накладной* — явная уникальная характеристика накладной;
- *Дата накладной* — явная характеристика накладной;
- *(?)Список товаров в накладной* — список не может быть атрибутом. Вероятно, нужно выделить этот список в отдельную сущность;
- *(?)Количество товара в накладной* — это явная характеристика, но характеристика чего? Это характеристика не просто «товара», а «товара в накладной»;
- *(?)Цена товара в накладной* — опять же это должна быть не просто характеристика товара, а характеристика товара в накладной. Но цена товара уже встречалась выше — это одно и то же?
- *Сумма накладной* — явная характеристика накладной. Эта характеристика не является независимой. Сумма накладной равна сумме стоимостей всех товаров, входящих в накладную;
- *Наименование склада* — явная характеристика склада.

В ходе дополнительной беседы с менеджером удалось прояснить различные понятия цен. Оказалось, что каждый товар имеет некоторую текущую цену. Это цена, по которой товар продается в данный момент. Естественно, что эта цена может меняться со временем. Цена одного и того же товара в разных накладных, выписанных в разное время, может быть различной. Таким образом, имеется две цены — цена товара в накладной и текущая цена товара.

С возникающим понятием «Список товаров в накладной» все довольно ясно. Сущности «Накладная» и «Товар» связаны друг с другом отношением типа много-ко-многим. Такая связь, как мы отмечали ранее, должна быть расщеплена на две связи типа один-ко-многим. Для этого требуется дополнительная сущность. Этой сущностью и будет сущность «Список товаров в накладной». Связь ее с сущностями «Накладная» и «Товар» характеризуется следующими фразами — «каждая накладная обязана иметь несколько записей из списка товаров в накладной», «каж-

дая запись из списка товаров в накладной обязана включаться ровно в одну накладную», «каждый товар может включаться в несколько записей из списка товаров в накладной», «каждая запись из списка товаров в накладной обязана быть связана ровно с одним товаром». Атрибуты «Количество товара в накладной» и «Цена товара в накладной» являются атрибутами сущности «Список товаров в накладной».

Точно так же поступим со связью, соединяющей сущности «Склад» и «Товар». Введем дополнительную сущность «Товар на складе». Атрибутом этой сущности будет «Количество товара на складе». Таким образом, товар будет числиться на любом складе и количество его на каждом складе будет свое.

Теперь можно внести все это в диаграмму (рис. 3.37).

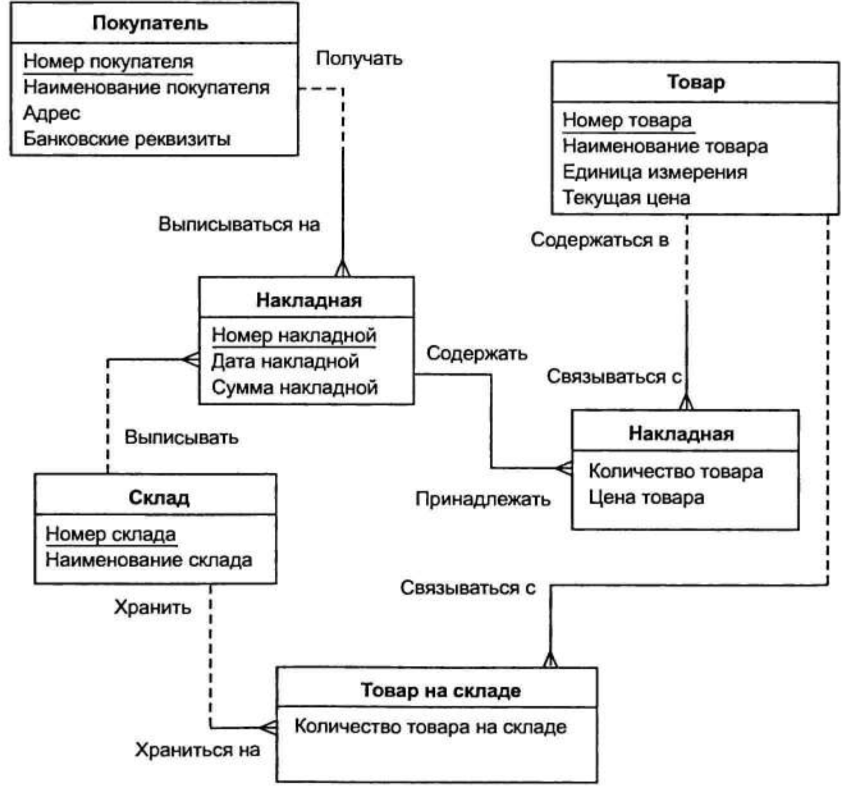


Рис. 3.37. Окончательный вариант ER-диаграммы

3.6. Анализ требований и определение спецификаций при объектном подходе

При объектном подходе к программированию модели разрабатываемой системы основываются на предметах и явлениях окружающего мира.

Модель — упрощенное представление реальности. С точки зрения программирования модель — это чертеж системы. Моделирование необходимо для решения следующих задач [4]:

1) визуализации системы;

2) определения ее структуры и поведения;

3) получения шаблона, позволяющего затем сконструировать систему;

4) документирования принимаемых решений, используя полученные модели.

Для решения этих задач при описании поведения проектируемого программного обеспечения в настоящее время используется UML (Unified Modeling Language) — унифицированный язык моделирования.

3.6.1. Некоторые теоретические сведения о UML — унифицированном языке моделирования

В основе объектного подхода к разработке программного обеспечения лежит объектная декомпозиция, т. е. представление разрабатываемого программного обеспечения в виде совокупности объектов, в процессе взаимодействия которых через передачу сообщений происходит выполнение требуемых функций.

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования, самым популярным из которых на сегодняшний день является UML.

Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей: логическую, использования, реализации, процессов, развертывания [1].

Модель использования содержит описание функций программного обеспечения с точки зрения пользователя.

Логическая модель описывает ключевые понятия моделируемого программного обеспечения (классы, интерфейсы и т. п.), т. е. средства, обеспечивающие его функциональность.

Модель реализации определяет реальную организацию программных модулей в среде разработки.

Модель процессов отображает организацию вычислений и позволяет оценить производительность, масштабируемость и надежность программного обеспечения.

И наконец, *модель развертывания* показывает, каким образом программные компоненты размещаются на конкретном оборудовании.

Все вместе указанные модели, каждая из которых характеризует определенную сторону проектируемого продукта, составляют относительно полную модель разрабатываемого программного обеспечения.

Всего UML предлагает девять дополняющих друг друга диаграмм, входящих в различные модели:

- диаграммы вариантов использования;
- диаграммы классов;
- диаграммы пакетов;
- диаграммы последовательностей действий;
- диаграммы кооперации;
- диаграммы деятельностей;
- диаграммы состояний объектов;
- диаграммы компонентов;
- диаграммы размещения.

На этапе анализа постановки задачи и требований к системе используют диаграммы прецедентов, диаграммы деятельностей для расшифровки содержания прецедентов, диаграммы состояний для моделирования поведения объектов со сложным состоянием, диаграммы классов для выделения концептуальных сущностей предметной области задачи и диаграммы последовательностей действий.

3.6.2. Определение прецедентов (вариантов использования)

Разработку спецификаций программного обеспечения начинают с анализа требований к функциональности, указанных в техническом задании. В процессе анализа выявляют внешних пользователей разрабатываемого программного обеспечения и перечень отдельных аспектов его поведения в процессе взаимодействия с конкретными пользователями.

Прецеденты (варианты использования — Use Cases) — это подробные процедурные описания вариантов использования системы всеми заинтересованными лицами, а также внешними системами, т. е. всеми, кто (или что) может рассматриваться как акторы (actors) — действующие лица. По сути, это своего рода алгоритмы работы с системой с точки зрения внешнего мира. Прецеденты являются основой функциональных требований к системе, позволяют описывать границы проектируемой системы, ее интерфейс, а затем выступают как основа для тестирования системы заказчиком с помощью приемочных тестов.

В зависимости от цели выполнения конкретной задачи различают следующие варианты использования [1]:

- основные, обеспечивают выполнение функций проектируемой системы;
- вспомогательные, обеспечивают выполнение настроек системы и ее обслуживание;
- дополнительные, служат для удобства пользователя (реализуются в том случае, если не требуют серьезных затрат каких-либо ресурсов ни при разработке, ни при эксплуатации).

Пример 3.3. Анализ функциональных требований и пользователей системы тестирования (модуль обучающей системы).

Система тестирования прежде всего требуется следующим заинтересованным лицам:

- обучаемому (студенту);
- составителю тестов (преподавателю);
- преподавателю, принимающему экзамен;
- сотруднику деканата, осуществляющему контроль за успеваемостью;
- администратору сети и баз данных учебного учреждения.

На начальном этапе создания системы мы можем ограничиться только двумя важными для нас ролями действующих лиц:

- студент (тестируемый);
- администратор (он же преподаватель, он же составитель тестов).

Соответственно основные прецеденты (варианты использования) для нашей системы следующие:

Прецедент для студента:

- П1 — пройти тестирование.

Прецеденты для администратора:

- П2 — создать/изменить тест;
- П3 — просмотреть результаты тестирования;
- П4 — добавить/изменить пользователей и др.

Вариант использования можно описать кратко или подробно. Краткая форма описания содержит название варианта использования, его цель, действующих лиц, тип варианта использования (основной, второстепенный или дополнительный) и его краткое описание [1].

Краткое описание варианта использования для данного примера:

Название варианта	Прохождение теста
Цель	Получение оценки
Действующие лица (актеры)	Студент
Краткое описание	Регистрация студента, запуск теста, выбор ответа из нескольких предложенных или ввод ответа, завершение теста, получение оценки
Тип варианта	Основной

Подробное описание варианта использования ***Прохождение теста***

Действия исполнителя	Отклик системы
1. Студент вводит свои данные (ФИО, Группа), т. е. регистрируется в системе	2. Система создает на диске файл с результатом тестирования и предлагает выбрать тест
3. Студент выбирает тест	4. Система запускает тест
5. Студент последовательно отвечает на вопросы	6. Система регистрирует правильные и неправильные ответы
7. Студент завершает тестирование	8. Система подсчитывает процент правильных ответов
9. Студент ожидает результат	10. Система демонстрирует результат и предлагает сохранить его
11. Студент решает, сохранять результат или нет	12. Если выбрано сохранение, система записывает результат в файл
13. Студент завершает работу	14. Система завершает работу

Для большей наглядности используют диаграммы вариантов использования.

Диаграммы вариантов использования

На рис. 3.38 приведены условные обозначения, которые применяют при изображении диаграмм прецедентов [48].

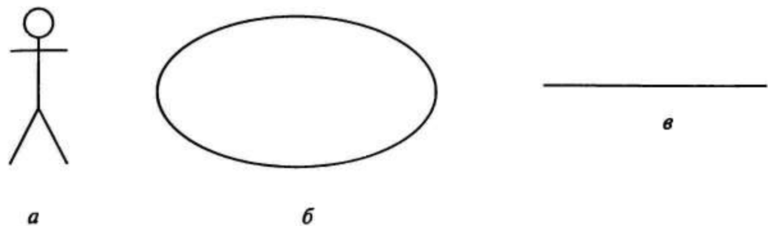


Рис. 3.38. Условные обозначения диаграмм прецедентов:
а — актор; *б* — вариант использования; *в* — связь

Приведем диаграмму прецедентов для вышеописанного примера (рис. 3.39).

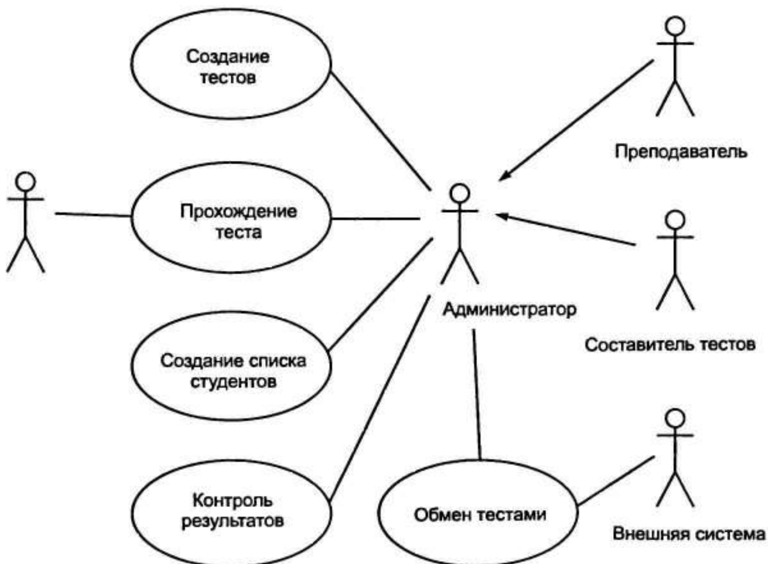


Рис. 3.39. Диаграмма вариантов использования тестовой системы

Естественно, все варианты использования определить, как правило, не удается: новые варианты фиксируют постоянно, даже в процессе эксплуатации. Но чем больше вариантов выявлено в процессе уточнения спецификаций, тем лучше, так как при этом получают более точную модель предметной области, что уменьшает вероятность ее пересмотра при добавлении функций.

3.6.3. Построение концептуальной модели предметной области

Диаграммы классов

Центральное место в объектно-ориентированном подходе к проектированию программного обеспечения занимает разработка логической модели системы в виде *диаграммы классов* (class diagram) [1, 48].

UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

- концептуальный уровень, на котором диаграммы классов отображают связи между основными понятиями предметной области;
- уровень спецификаций, на котором диаграммы классов отображают связи объектов этих классов;
- уровень реализации, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Каждую из перечисленных моделей используют на конкретном этапе разработки программного обеспечения:

- концептуальную модель — на этапе анализа;
- диаграммы классов уровня спецификации — на этапе проектирования;
- диаграммы классов уровня реализации — на этапе реализации.

Диаграмма классов может отражать различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. Диаграммы классов обычно содержат следующие сущности:

- классы;
- интерфейсы;

- кооперации;
- отношения зависимости, обобщения и ассоциации.

Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы, поэтому диаграмму классов принято считать графическим представлением таких взаимосвязей логической модели системы, которые не зависят от времени [48].

Класс

Класс (class) в языке UML служит для обозначения множества объектов, которые имеют одинаковую структуру, поведение и отношения с объектами из других классов. На диаграмме класс изображают в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на две или три секции (рис. 3.40). В этих разделах могут указываться имя класса, атрибуты (переменные) и операции (методы). Иногда в графическом изображении класса добавляется четвертая секция, содержащая описание исключительных ситуаций.

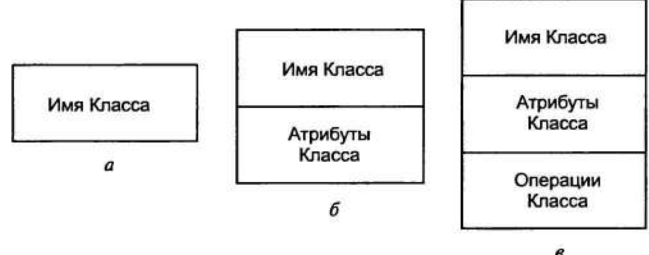


Рис. 3.40. Графическое изображение класса на диаграмме классов

Обязательным элементом обозначения класса является его имя. На начальных этапах разработки диаграммы класс может обозначаться простым прямоугольником с указанием только его имени (рис. 3.40, *а*). В дальнейшем диаграммы описания классов дополняются атрибутами (рис. 3.40, *б*) и операциями (рис. 3.40, *в*).

Чтобы сразу отличить класс от других элементов языка UML, секцию атрибутов и операций выделяют горизонтальной линией, даже если она является пустой. На рис. 3.41 приведены примеры графического изображения классов на диаграмме классов. В первом случае для класса «Прямоугольник»

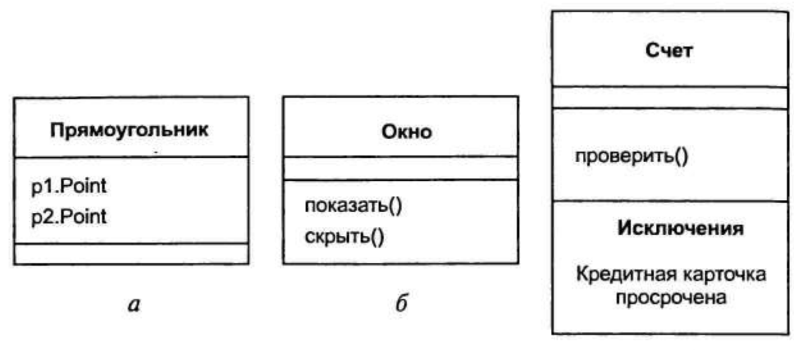


Рис. 3.41. Примеры графического изображения классов на диаграмме

(рис. 3.41, *а*) указаны только его атрибуты — точки на координатной плоскости, определяющие его местоположение. Для класса «Окно» (рис. 3.41, *б*) указаны только его операции (показать(), скрыть()), секция атрибутов оставлена пустой. Для класса «Счет» (рис. 3.41, *в*), кроме операции проверки кредитной карточки, дополнительно изображена четвертая секция, в которой указано исключение — отказ от обслуживания просроченной кредитной карточки.

Имя класса

Имя класса должно быть уникальным в пределах диаграммы или совокупности диаграмм классов пакета. Оно указывается в первой верхней секции прямоугольника, записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы. В качестве имен классов рекомендуется использовать одно или несколько существительных без пробелов между ними. Кроме того, в секции обозначения класса могут находиться ссылки на стандартные шаблоны или абстрактные классы, от которых образован данный класс и, соответственно, от которых он наследует свойства и методы.

Примерами имен классов могут быть такие существительные, как «Сотрудник», «Фирма», «Руководитель», «Покупатель», «Продавец» и многие другие, имеющие непосредственное отношение к моделируемой предметной области и функциональному назначению проектируемой системы.

Если класс не имеет экземпляров (объектов), то он называется *абстрактным классом*, его имя записывается курсивом так же, как любой текст, относящийся к абстрактному элементу.

Атрибуты класса

Во второй секции прямоугольника — графического изображения класса — записываются его *атрибуты* (attributes) или свойства. Стандартная запись атрибута в языке UML выглядит следующим образом:

```
<квантор видимости><имя атрибута>[кратность]
<тип атрибута>   <исходное значение>{строка-свойство}
```

Квантор видимости может быть опущен — это означает, что видимость атрибута не указывается либо же должна принимать одно из трех возможных значений:

- общедоступный (public) — обозначается «+»;
- защищенный (protected) — обозначается «#»;
- закрытый (private) — обозначается «–».

Имя атрибута — единственный обязательный элемент обозначения атрибута, представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и является уникальной в пределах данного класса.

Кратность атрибута показывает количество конкретных атрибутов данного типа, входящих в состав класса, и обозначается следующим образом:

```
[нижняя_граница1   верхняя_граница1, нижняя_граница2.
верхняя_граница2,      нижняя_границаk
верхняя_границаk],
```

где нижняя_граница и верхняя_граница являются положительными целыми числами, каждая пара которых служит для обозначения отдельного замкнутого интервала целых чисел. В качестве верхней границы может использоваться специальный символ «*», который означает произвольное положительное целое число, т. е. неограниченное сверху значение кратности соответствующего атрибута.

Если в качестве кратности указывается единственное число, то кратность атрибута принимается равной данному числу. Значения кратности из интервала следуют в монотонно возрастающем порядке без пропуска отдельных чисел, лежащих между

нижней и верхней границами, соответствующие нижние и верхние границы интервалов включаются в значение кратности. Если же указывается единственный знак «*», то это означает, что кратность атрибута может быть произвольным положительным целым числом или нулем.

Могут использоваться следующие варианты задания кратности атрибутов:

- [0..1] означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие значения для данного атрибута;
- [0..*] или просто [*] означает, что кратность атрибута может принимать любое положительное целое значение, большее или равное 0;
- [1..*] означает, что кратность атрибута может принимать любое положительное целое значение, большее или равное 1;
- [1..5] означает, что кратность атрибута может принимать любое значение из чисел 1, 2, 3, 4, 5;
- [1..3,7.. 10] означает, что кратность атрибута может принимать любое значение из чисел 1, 2, 3, 7, 8, 9, 10;
- [1..3,7..*] означает, что кратность атрибута может принимать любое значение из чисел 1, 2, 3, а также любое положительное целое значение, большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение, равное 1.

Тип атрибута указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс. Можно также определять тип атрибута в зависимости от языка программирования, который будет использоваться для реализации данной модели.

Например:

```
имя_студента [1..2]   String   здесь имя_студента
является именем атрибута, тип атрибута String (строка)
```

Исходное значение служит для задания некоторого начального значения для соответствующего атрибута в момент создания отдельного экземпляра класса.

Например:

```
имя_студента [1..2] : String = Иван.
```

Строка-свойство служит для указания фиксированных значений атрибута. Эти значения не могут быть изменены в программе при работе с данным типом объектов. При отсутствии строки-свойства значение соответствующего атрибута может быть изменено в программе. Например, строка-свойство в записи атрибута стипендия Integer = {$50} означает фиксированную сумму стипендии для всех объектов класса «Студент». Запись данного атрибута в виде стипендия Integer = $50 означает, что при создании нового экземпляра Студент для него устанавливается по умолчанию стипендия в 50 долл. Но по результатам очередной сессии некоторые студенты могут получать надбавки к стипендии и именные стипендии, а кто-то будет лишен стипендии вовсе, о чем необходимо позаботиться в программе.

Операция

Операцией класса (*методом класса*) называется именованный сервис, который предоставляется по требованию любым объектом данного класса. Другими словами, операция — это абстракция того, что можно делать с объектом. Класс может содержать любое число операций (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

Операции класса определяются в секции, расположенной ниже секции атрибутов. При этом можно ограничиться только указанием имен *операций*, оставив детальную спецификацию выполнения операций на более поздние этапы моделирования. Для именования операций рекомендуется использовать глаголы, соответствующие ожидаемому поведению объектов данного класса. Описание операции имеет следующий вид:

```
<квантор видимости><имя операции>(список параметров)
<выражение типа возвращаемого
значения>{строка-свойство}
```

Квантор видимости принимает такие же значения, как и в случае атрибутов класса, и может быть опущен. Вместо условных графических обозначений также можно записывать соответствующее ключевое слово: public, protected, private.

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей опера-

ции, и поэтому должна быть уникальной в пределах данного класса. Имя является единственным обязательным элементом синтаксического обозначения операции.

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде:

```
<вид параметра><имя параметра>:<выражение
типа>=<значение параметра по умолчанию>.
```

Вид параметра — это одно из ключевых слов *in, out* или *inout* со значением *in* по умолчанию. *Имя параметра* есть идентификатор соответствующего формального параметра. *Выражение типа* зависит от конкретного языка программирования и описывает тип возвращаемого значения для соответствующего формального параметра. *Значение по умолчанию* в общем случае представляет собой выражение для значения формального параметра.

Выражение типа возвращаемого значения также является зависимой от языка реализации спецификацией типа или типов значений параметров, которые возвращаются объектом после выполнения соответствующей операции. Операция может не возвращать никакого значения. Для указания кратности возвращаемого значения данная спецификация может быть записана в виде списка отдельных выражений.

Строка-свойство служит для определения значений свойств данного элемента. Строка-свойство может отсутствовать, если никакие свойства не специфицированы.

Операция, которая не может изменять состояние системы, обозначается строкой-свойством {запрос} ({query}). Другие операции могут изменять состояние системы, хотя и необязательно будут это делать.

Описание операции на самом верхнем уровне объявляет эту операцию на весь класс, при этом данная операция наследуется всеми потомками данного класса. Если в некотором классе операция не выполняется, то такая операция может быть помечена как *абстрактная* ({abstract}). Можно также записать сигнатуру операции курсивом, чтобы обозначить ее абстрактной. Подчиненное появление записи данной операции без свойства {абстрактная} указывает на тот факт, что соответствующий класс-потомок может выполнять данную операцию в качестве своего метода.

Пример:

+создать() — обозначает абстрактную операцию по созданию отдельного объекта класса, которая является общедоступной (public) и не содержит формальных параметров. Эта операция не возвращает никакого значения после своего выполнения.

3.6.4. Описание поведения системы. Диаграммы последовательностей, деятельности и состояний

Концептуальная модель характеризует статические свойства разрабатываемого программного обеспечения. Для описания особенностей его поведения, т. е. возможных действий системы, целесообразно использовать диаграммы последовательностей системы, системные события, системные операции, диаграммы деятельности, а при необходимости и диаграммы состояний объектов.

Диаграмма последовательностей системы (sequence diagram)

Диаграмма последовательностей системы — графическая модель, которая для определенного сценария варианта использования показывает динамику взаимодействия объектов во времени [1, 48].

Для построения диаграммы последовательностей системы необходимо:

- идентифицировать каждое действующее лицо (объект) и изобразить для него линию жизни. Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия (см. объект 1 на рис. 3.42). Правее изображается другой объект, который непосредственно взаимодействует с первым, и т. д.;
- из описания варианта использования определить множество системных событий и их последовательность;
- изобразить системные события в виде линий со стрелкой на конце между линиями жизни действующих лиц и системы, а также указать имена событий и списки передаваемых значений.

На диаграмме последовательности изображаются только те объекты, которые непосредственно участвуют во взаимодействии, и не показываются возможные статические ассоциации с

другими объектами. При этом диаграмма последовательности имеет как бы два измерения. Одно — слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни (рис. 3.42). Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием. При этом вся запись подчеркивается, что является признаком объекта, который представляет собой экземпляр класса.

Линия жизни объекта

Линия жизни объекта (object lifeline) служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. На диаграмме линия жизни изображается пунктирной вертикальной линией, ассоциированной с единственным объектом. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней (объекты 1 и 2 на рис. 3.42).

Если объекты разрушаются в какой-то момент для освобождения ресурсов системы, то их линия жизни обрывается в момент уничтожения. Для обозначения такого момента в языке UML используется специальный символ в форме латинской буквы «X» (объект 3 на рис. 3.42). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет и этот объект должен быть исключен из всех последующих взаимодействий.

Фокус управления

Объекты на диаграмме последовательности могут находиться в двух состояниях, активном — непосредственно выполняя какие-либо действия, и пассивном, ожидая сообщения от других объектов. Чтобы явно выделить подобную активность объектов, в языке UML применяется специальное понятие, получившее название *фокуса управления* (focus of control). Фокус управления изображается в форме вытянутого узкого прямоугольника (см. объект 1 на рис. 3.42), верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности),

а ее нижняя сторона — окончание фокуса управления (окончание активности).

Сообщения

Как уже было сказано, диаграмма последовательности описывает динамику взаимодействий между множеством объектов. Каждое взаимодействие описывается совокупностью *сообщений*, которыми участвующие в нем объекты обмениваются между собой. Сообщение (message) представляет собой законченный фрагмент информации, который отправляется одним объектом другому. Объект, принявший сообщение, должен отреагировать на него какой-либо последовательностью действий, направленных на решение поставленной задачи.

Графически сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности (рис. 3.42).

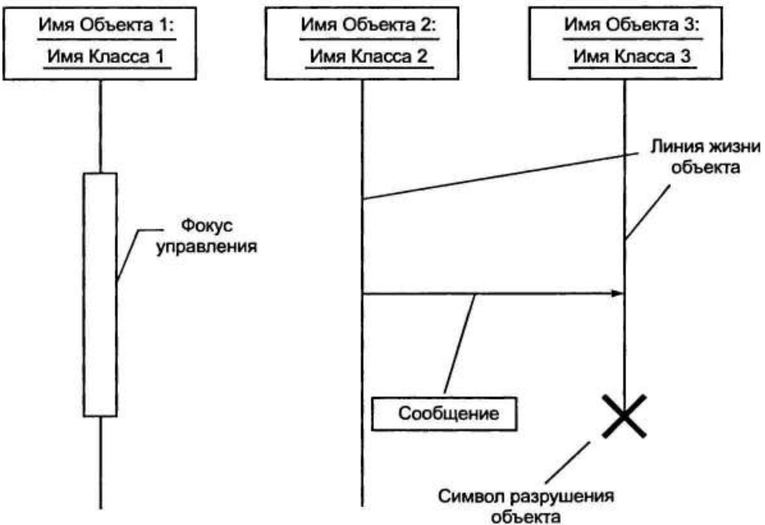


Рис. 3.42. Различные графические примитивы диаграммы последовательности

На рис 3.43 можно увидеть пример построения диаграммы последовательности для моделирования процесса телефонного разговора с использованием обычной телефонной сети. Объектами в этом примере являются: два абонента *a* и *b*, два телефон-

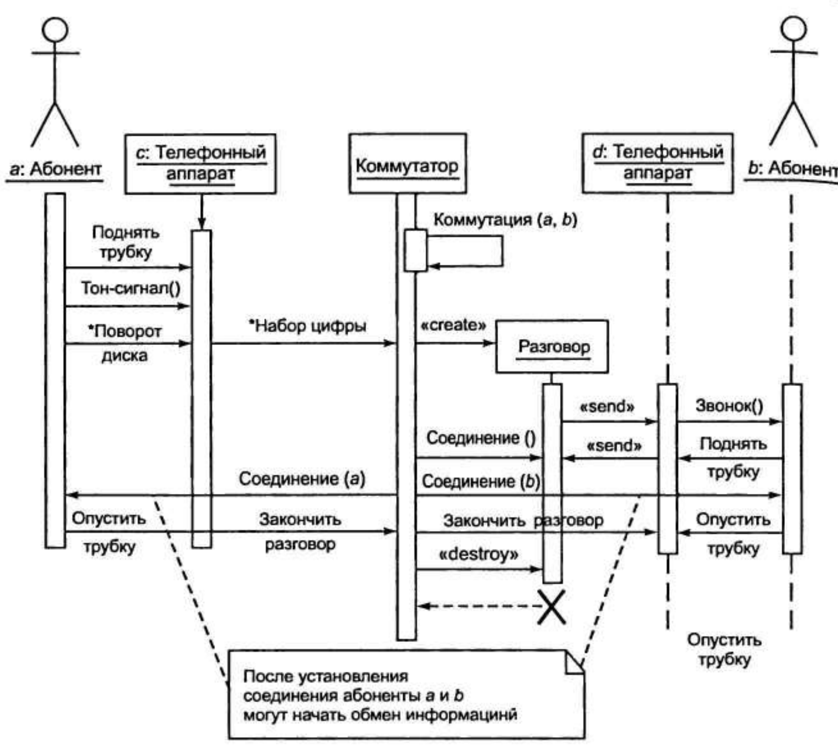


Рис. 3.43. Пример диаграммы последовательностей состояний

ных аппарата, коммутатор и сам разговор как объект моделирования.

Диаграммы деятельностей (activity diagram)

Для моделирования процесса выполнения операций в языке UML используются так называемые *диаграммы деятельности*. На этапе анализа требований и уточнения спецификаций диаграммы деятельностей позволяют конкретизировать основные функции разрабатываемого программного обеспечения.

Состояние действия

Под деятельностью в данном случае понимают задачу (операцию), которую необходимо выполнить вручную или с помощью средств автоматизации. Каждому варианту использования соот-

ветствует своя последовательность задач. В теоретическом плане диаграммы деятельности являются обобщенным представлением алгоритма, реализующего анализируемый вариант использования. Графически состояние действия представляется прямоугольником со скругленными углами (рис. 3.44). Внутри этой фигуры записывается выражение действия (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

Рис. 3.44. Графическое изображение состояния действия:
а — простое действие; *б* — выражение

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами (рис. 3.44, *а*). Если это возможно, то допускается запись действия на том языке программирования, на котором предполагается реализовывать конкретный проект (рис. 3.44, *б*).

Каждая диаграмма деятельности должна иметь единственное начальное и единственное конечное состояния (рис. 3.45). Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз. В этом случае начальное состояние будет изображаться в верхней части диаграммы, а конечное — в ее нижней части.

Рис. 3.45. Графическое изображение:
а — начальное состояние; *б* — конечное состояния

Переходы

При построении диаграммы деятельности используются только нетриггерные переходы, т. е. такие, которые срабатывают сразу после завершения деятельности или выполнения соответ-

ствующего действия. Этот переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.

Если из состояния действия выходит единственный переход, то он может быть никак не помечен. Если последовательно выполняемая деятельность должна разделиться на альтернативные ветви в зависимости от значения некоторого промежуточного результата (такая ситуация получила название ветвления, а для ее обозначения применяется специальный символ), то таких переходов несколько и сработать может только один из них. Именно в этом случае для каждого из таких переходов должно быть явно записано условие в квадратных скобках. При этом для всех выходящих из некоторого состояния переходов должно выполняться требование истинности только одного из них.

Графически ветвление на диаграмме деятельности обозначается небольшим ромбом, внутри которого нет никакого текста (рис. 3.46). В этот ромб может входить только одна стрелка от того состояния действия, после выполнения которого поток

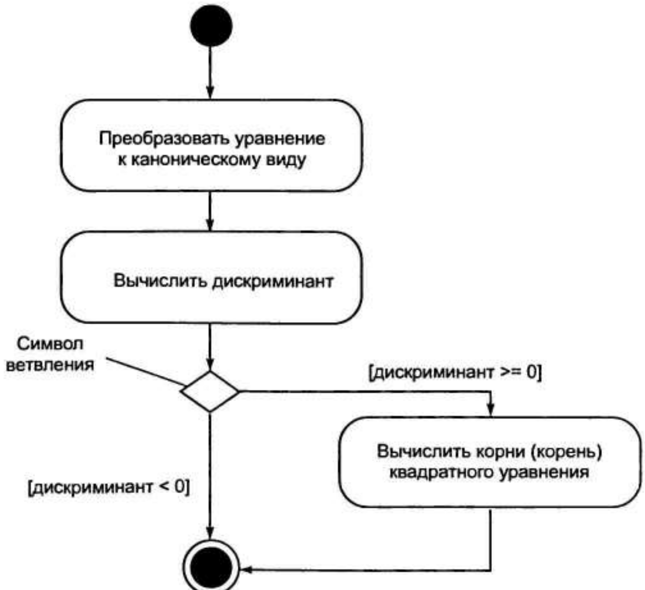


Рис. 3.46. Фрагмент диаграммы деятельности для алгоритма нахождения корней квадратного уравнения

управления должен быть продолжен по одной из взаимно исключающих ветвей. Принято входящую стрелку присоединять к верхней или левой вершине символа ветвления. Выходящих стрелок может быть две или более, но для каждой из них явно указывается соответствующее условие перехода в форме булевского выражения.

На рис. 3.46 представлен фрагмент известного алгоритма нахождения корней квадратного уравнения. В общем случае после приведения уравнения второй степени к каноническому виду $a\ x\ x + b\ \ x + c = 0$ в случае отрицательного дискриминанта уравнение не имеет решения на множестве действительных чисел, и дальнейшие вычисления должны быть прекращены. При неотрицательном дискриминанте уравнение имеет решение, корни которого могут быть получены на основе конкретной расчетной формулы.

Для представления параллельных процессов в языке UML используется специальный символ разделения и слияния параллельных вычислений или потоков управления. Таким символом является прямая черточка, которая изображается отрезком горизонтальной линии, толщина ее несколько шире основных сплошных линий диаграммы деятельности. При этом разделение (concurrent fork) имеет один входящий переход и несколько выходящих (рис. 3.47, *а*). Слияние (concurrent join), наоборот, имеет несколько входящих переходов и один выходящий (рис. 3.47, *б*).

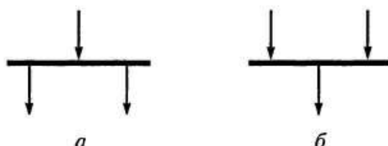


Рис. 3.47. Графическое изображение:
а — разделения; *б* — слияния параллельных потоков управления

На рис. 3.48 показан пример диаграммы деятельности процесса приготовления напитка.

Диаграммы состояний (statechart diagram)

Главное предназначение этой диаграммы — описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение элемента модели в течение

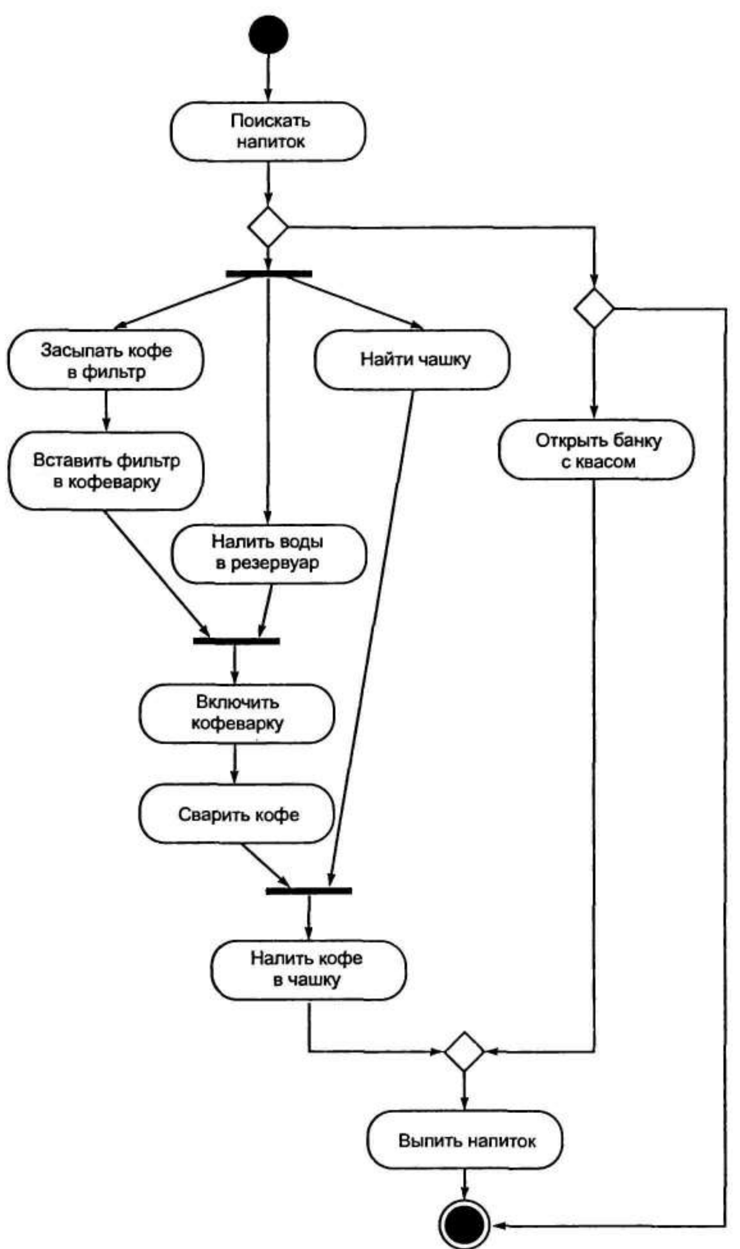


Рис 3.48. Пример диаграммы деятельности

его жизненного цикла. Диаграммы состояний используются для моделирования динамических аспектов системы. По большей части под этим подразумевается моделирование поведения реактивных объектов. Реактивным называется объект, поведение которого лучше всего характеризуется его реакцией на события, произошедшие вне его собственного контекста. У реактивного объекта есть четко выраженный жизненный цикл, когда текущее поведение обусловлено прошлым. Если внешние действия, изменяющие состояния системы, инициируются в произвольные случайные моменты времени, то говорят об асинхронном поведении модели.

Представьте себе моделирование поведения встроенной системы безопасности у себя дома. Она работает непрерывно, реагируя на внешние события, скажем, на разбитое окно. Порядок событий изменяет поведение системы. Например, обнаружение разбитого окна вызовет срабатывание сигнализации, только если система предварительно была активизирована. Поведение такой системы лучше всего описывается путем моделирования ее устойчивых состояний (например, Ожидание, Активна, Проверка и т. д.), событий, инициирующих смену состояния, и действий, выполняемых при каждой такой смене.

Автомат

Диаграмма состояний по существу является графом специального вида, который представляет некоторый автомат. Вершинами этого графа являются состояния, которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы более детального представления отдельных элементов модели.

Простейший пример автомата с двумя состояниями демонстрирует ситуация с исправностью компьютера. Здесь рассматриваются два самых общих состояния: «исправен» и «неисправен» и два перехода: «выход из строя» и «ремонт». Графически эта информация может быть представлена в виде изображенной диаграммы состояний компьютера (рис. 3.49).

Основными понятиями, описывающими автомат, являются состояние и переход. Предполагается, что система находится в

Рис. 3.49. Пример диаграммы состояний

каком-либо состоянии в течение некоторого времени, тогда как переход объекта из состояния в состояние происходит мгновенно.

Состояние

Состояние (State) — это ситуация в жизни объекта, на протяжении которой он удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает какого-то события.

Состояние на диаграмме изображается прямоугольником со скругленными вершинами (рис. 3.50), который может быть разделен горизонтальной линией на две секции.

Рис. 3.50. Графическое изображение состояний на диаграмме состояний

В прямоугольнике может располагаться «Имя состояния» (первая секция) и «Список внутренних действий в данном состоянии» (вторая секция). При этом под действием (action) в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения.

Имя состояния

Имя состояния — это строка текста, начинающаяся с заглавной буквы, которая раскрывает содержательный смысл данного состояния. Имя является необязательным элементом. Рекомендуется в качестве имени использовать глаголы в настоящем вре-

мени (звенит, печатает, ожидает) или соответствующие причастия (занят, свободен, передано, получено).

Список внутренних действий

Эта секция содержит перечень *внутренних действий* или *деятельностей*, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии. Каждое из действий записывается в виде отдельной строки и имеет следующий формат:

```
<метка-действия    выражение-действия>
```

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия. При этом выражение действия может использовать любые атрибуты и связи, которые принадлежат области имен или контексту моделируемого объекта. Если список выражений действия пустой, то разделитель в виде наклонной черты '/' может не указываться.

На рис. 3.51 показан пример состояния **Считывает запись** после открытия файла, содержащего несколько записей.

Считывает запись

символ / считать символ

конец _строки / проверить на конец строки

Рис. 3.51. Пример состояния с непустой секцией внутренних действий

Начальное и конечное состояния описаны в разделе описания диаграммы деятельности (см. рис. 3.45).

Переход

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта в первом состоянии может сопровождаться выполнением некоторых действий, а переход во второе состояние будет возможен после завершения этих действий, а

также после удовлетворения некоторых дополнительных условий. В этом случае говорят, что переход срабатывает. До срабатывания перехода объект находится в предыдущем от него состоянии, называемом исходным состоянием, или в источнике (не путать с начальным состоянием — это разные понятия), а после его срабатывания объект находится в последующем от него состоянии (целевом состоянии).

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая направлена в целевое состояние (например, «выход из строя» на рис. 3.49). Рядом с линией может находиться строка текста, описывающая событие-триггер, вызывающее переход (в этом случае переход будет триггерным), и сторожевое условие, по которому осуществляется переход. В примере на рис. 3.49 переход сработает при возникновении события — «выход из строя».

Событие

Событие (Event) — это спецификация существенного факта, который происходит во времени и пространстве. В контексте автоматов событие — это стимул, способный вызвать срабатывание перехода.

Сторожевое условие

Сторожевое условие (guard condition), если оно есть, всегда записывается в прямых скобках после события-триггера и представляет собой некоторое булевское выражение (выражение, результатом которого является «истина» или «ложь»).

Рис. 3.52. Диаграмма состояний для моделирования почтовой программы-клиента

Пример диаграммы состояний почтовой программы-клиента показан на рис. 3.52.

Контрольные вопросы

1. Приведите эксплуатационные требования к ПО.
2. Перечислите функциональные требования к ПО.
3. Чем определяется выбор архитектуры ПО?
4. Охарактеризуйте статические и полустатические структуры данных.
5. Охарактеризуйте динамические структуры данных.
6. Приведите понятие модуля. Характеристики модуля.
7. Какие существуют методы разработки модулей?
8. Что такое спецификации процессов?
9. Приведите пример диаграммы переходов состояний.
10. Какие бывают функциональные диаграммы?
11. Приведите пример диаграммы потоков данных.
12. Что такое диаграммы «сущность—связь»?
13. Охарактеризуйте понятие UML.
14. Опишите варианты использования системы.
15. Чем описывается поведение системы?

Глава 4
ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Существуют два стиля проектирования: эволюционное и предварительное проектирование [18].

Методология Extreme Programming (XP) бросила вызов многим устоявшимся представлениям о разработке программного обеспечения. Пожалуй, наиболее противоречивой идеей является отказ от предварительного проектирования в пользу более эволюционного подхода. Противники XP считают, что это возврат к разработкам типа «code and fix» («пишем и правим»). Для приверженцев же новой методологии это отказ от техник проектирования (например, UML), их принципов. Незачем беспокоиться о проектировании, считают они. Достаточно внимательно «вслушиваться» в свой код, и проектирование образуется само собой [17].

В большинстве случаев эволюционное проектирование — это нечто ужасное. В конце концов, все равно вместо дизайна системы вы получаете просто набор из специфических решений, каждое из которых затрудняет дальнейшие изменения в программном коде. Часто это вообще нельзя считать дизайном (и, уж конечно, такой дизайн никак нельзя назвать хорошим). Как говорит Кент, дизайн существует для того, чтобы дать возможность оперативно вносить в систему любые изменения. Если дизайн плох, то такая возможность исчезает. В результате вы будете иметь дело с энтропией программного продукта, и со временем и без того плохой дизайн системы станет еще хуже. Теперь вам будет не только сложнее вносить в систему изменения, но и отыскивать и исправлять ошибки, которые начинают множиться с катастрофической быстротой. Все это — кошмар

разработок в стиле «code and fix», когда с течением времени исправление ошибок обходится все дороже и дороже.

Предварительное проектирование — полная противоположность эволюционному. При разработке ПО проектировщики заранее продумывают все основные вопросы. При этом они не пишут программный код, поскольку не создают программный продукт, а только разрабатывают его дизайн. В своей работе они могут использовать такие техники, как UML, что позволяет им абстрагироваться от некоторых подробностей разработок, относящихся непосредственно к программированию. Как только проектный план готов, его можно передавать в другой отдел (или даже в другую компанию), где будут вестись работы по непосредственному созданию системы. Поскольку проектировщики работают на некотором уровне абстракции, им удается избежать принятия ряда тактических решений, ведущих к энтропии программного продукта. Программисты же могут руководствоваться проектным планом и (если они ему следуют) создавать качественно выстроенную систему [5].

Такой подход к разработке ПО не нов — им активно пользуется множество людей начиная с 1970-х годов. По многим показателям он гораздо лучше, чем эволюционное проектирование в стиле «code and fix», однако и у него есть существенные недостатки. Один из главных недостатков заключается в том, что невозможно заранее продумать все вопросы, с которыми придется столкнуться во время кодирования системы. Таким образом, в ходе работ непременно возникнет ситуация, когда у программистов появятся вопросы относительно спроектированного дизайна. А что, если проектировщики, закончив свою часть работы, уже переключились на другой проект? Тогда программисты начинают самостоятельно решать сложившуюся проблему, отступая от уже принятых проектных решений и внося при этом в программный продукт долю энтропии. И даже если проектировщик еще работает над проектом и может помочь, все равно ему потребуется довольно много времени, чтобы выяснить ситуацию, внести изменения в диаграммы и уже затем менять код. А при разработке, как правило, вопрос времени всегда стоит остро. Отсюда энтропия (опять-таки).

Кроме того, существует еще и проблема культур. Проектировщиками становятся благодаря высокому мастерству и большому опыту в программировании. Однако, став проектиров-

щиком, программист настолько поглощается новой работой, что просто не имеет физической возможности заниматься написанием программного кода. При этом инструментарий и материалы программных разработок постоянно меняются. А когда вы перестаете сами писать код, вы не только теряете возможность отслеживать новшества в этой области. Вы теряете уважение тех, кто продолжает заниматься написанием программного кода.

Однако такие проблемы все же можно как-то урегулировать. Может быть, можно что-то сделать с напряженностью в отношениях между людьми. Может быть, можно найти таких проектировщиков, которые могли бы разбираться в большинстве вопросов, и такой дисциплинированный процесс разработки, который позволял бы вносить изменения в диаграммы. Однако остается еще одна проблема — изменяющиеся требования. Именно изменяющиеся требования являются проблемой номер один.

Бороться с изменяющимися требованиями можно по-разному. Один из возможных путей — делать дизайн достаточно гибким, чтобы при изменениях в требованиях его можно было легко менять. Однако для этого требуется заранее знать, какого типа изменения следует ожидать. Да, при проектировании системы можно попытаться угадать те области, в которых наиболее вероятны изменения, и учесть их в дизайне. В этом случае вы, действительно, облегчите себе работу с ожидаемыми изменениями в требованиях, но ничуть не облегчите (а возможно, только ухудшите) ситуацию с изменениями неожиданными. Кроме того, чтобы заранее определить те области, в которых наиболее вероятны изменения, вы должны прекрасно понимать требования, что, по наблюдениям, очень непросто [3].

Впрочем, не все проблемы с изменениями в требованиях возникают из-за их непонимания. Множество людей напряженно работают над разработкой технических требований к системе в надежде, что это убережет их от дальнейших поправок при проектировании. Но и так вы далеко не всегда сможете решить проблему. Многие изменения в требованиях диктуются изменениями в экономике и том виде бизнеса, для которого предназначается система. Такие изменения предугадать невозможно, сколько бы вы ни сидели над разработкой требований.

4.1. Проектирование программного обеспечения при структурном подходе

При проектировании сложного программного обеспечения прежде всего необходимо определить структурные компоненты и связи между ними. Полученная в результате структура ПО должна быть представлена в виде структурной или функциональной схем и спецификаций ее компонентов [1].

4.1.1. Структурная схема разрабатываемого программного обеспечения

Структурной называют схему, отражающую состав и взаимодействие по управлению частей разрабатываемого программного обеспечения.

Структурная схема определяется архитектурой разрабатываемого ПО (см. разд. 3.2).

Разработку структурной схемы **программы** обычно выполняют методом пошаговой детализации (см. разд. 4.1.3).

Структурные схемы **пакетов программ** разрабатывают для каждой программы пакета по отдельности, поскольку организация программ в пакеты не предусматривает передачи управления между ними.

Компонентами структурной схемы **программной системы** или **программного комплекса** могут служить программы, подсистемы, базы данных, библиотеки ресурсов и т. п.

Пример структурной схемы программного комплекса для решения математических задач изображен на рис. 4.1.

Рис. 4.1. Пример структурной схемы программного комплекса

Как правило, для программных систем разрабатывается функциональная схема, которая дает более полное представление о проектируемом программном обеспечении с точки зрения взаимодействия его компонентов между собой и с внешней средой.

4.1.2. Функциональная схема

Функциональная схема (ГОСТ 19.701—90) — это схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств [1]. Для изображения функциональных схем используют специальные обозначения, установленные стандартом (табл. 4.1).

Таблица 4.1. **Обозначения элементов функциональных схем**

Название блока	Обозначение	Назначение блока
Сохраненные данные		Для обозначения таблиц и других структур данных, которые должны быть сохранены без уточнения типа устройства
Оперативное запоминающее устройство		Для обозначения таблиц и других структур данных, хранящихся в оперативной памяти
Запоминающее устройство с прямым доступом		Для обозначения таблиц и других структур данных, хранящихся на магнитных дисках
Документ		Для обозначения таблиц и других структур данных, выводимых на печать
Ручной ввод		Для обозначения ручного ввода данных с клавиатуры
Дисплей		Для обозначения данных, выводимых на дисплей компьютера

Функциональные схемы более информативны, чем структурные. На рис. 4.2 приведена функциональная схема программно-

Рис. 4.2. Пример функциональной схемы программного комплекса

го комплекса, реализующего различные методы сортировки массивов.

4.1.3. Метод пошаговой детализации при составлении алгоритмов

Метод пошаговой детализации реализует нисходящий подход к программированию и предполагает пошаговую разработку алгоритма. Можно выделить следующие этапы [38]:

1. Создается описание программы в целом. Определяются основные логические шаги, требуемые для решения задачи, даже если пока неизвестно, как их выполнить. Эти логические шаги могут отражать различные физические способы решения или могут быть удобными групповыми именами для тех действий, выполнение которых представляется довольно смутно. Последовательности шагов, требуемых для решения задачи, записываются на обычном языке или на псевдокоде (см. разд. 3.5.1).

2. В общих терминах детализируется описание шагов, введенных на этапе 1. В детализированное описание может входить обозначение циклических структур, в то время как действия внутри циклов могут по-прежнему оставаться неясными. Таким образом, выполняются только общие эскизы сложных действий.

3. На этом и последующих уровнях в виде последовательных итераций производятся те же действия, что описаны на этапе 2.

При каждой новой итерации уточняются детали, оставшиеся неясными после предыдущих итераций, и создаются более определенные описания. По мере выполнения итераций неопределенные детали становятся все проще и проще, так что на каком-то этапе могут быть полностью описаны.

4. Разработка завершена: в модульном виде получено описание требуемой программы. Перевод этого описания в программу на конкретном языке программирования должен быть достаточно простой задачей.

Пример 4.1. Пусть требуется определить наибольшее значение в некотором наборе данных и вывести эти данные, поделенные на наибольшее значение. Скажем, если данные представляют собой последовательность чисел:

5.0, –3.24, 10.0, –1.25, 8.33,

то вывод должен выглядеть следующим образом:

0.5, –0.324, 1, –0.125, 0.833

Уровень 1:

Программа
ввести данные
найти максимум введенных данных
вывести результаты
Конец.

Детализация 1.1. Ввод данных можно детализировать на псевдокоде следующим образом:

Ввести данные:
определить количество чисел
Цикл-пока: не все элементы введены
прочитать и запомнить значение элемента
Все-цикл

Детализация 1.2. Отыскание максимума можно детализировать следующим образом:

Найти максимум:
выбрать в качестве максимума первый элемент данных
сравнить все значения с максимумом, заменяя
текущий максимум на очередное значение, если оно
не превысило его

Детализация 1.3. Вывод результатов можно детализировать следующим образом:

Цикл-пока не все элементы выведены
 вывести значение элемента
Все-цикл

Уровень 2. Он включает в себя три детализованные выше части, из которых только детализация 1.2 требует дополнительного внимания. Ее можно детализировать на псевдокоде следующим образом:

Найти максимум:
 выбрать в качестве максимума первый элемент данных
 Цикл-пока не все элементы проверены
  сравнить все значения с максимумом
  Если текущее значение больше максимума
  Максимум = текущее значение
  Конец-если
 Конец-цикл

Задача в приведенном примере проста и не требует разбиения на модули.

При решении реальной задачи может потребоваться написание на псевдокоде многих уровней, чтобы довести все модули до такого состояния, при котором они окажутся готовыми для программирования.

4.1.4. Структурные карты Константайна

Методика структурных карт используется на этапе проектирования ПО для того, чтобы продемонстрировать, каким образом программный продукт выполняет системные требования. При этом наиболее часто применяются две техники: структурные карты Константайна (Constantine), предназначенные для описания отношений между модулями, и структурные карты Джексона (Jackson), предназначенные для описания внутренней структуры модулей [39].

Структуру программной системы составляют модули, которые в любом языке программирования имеют следующие общие свойства:

- модуль имеет имя, по которому к нему можно обращаться как к единому фрагменту;

- модуль состоит из множества операторов языка программирования, записанных последовательно;
- модуль может принимать и/или передавать данные как параметры в вызывающей последовательности или связывать данные через фиксированные ячейки или общие области.

Структурные карты Константайна представляют собой модель отношений между модулями программы. Узлы структурных карт соответствуют модулям и областям данных, потоки изображают межмодульные связи. На диаграмме специальными узлами изображаются циклические и условные вызовы модулей, а потоки проходят через эти специальные узлы. Потоки, изображающие межмодульные связи по данным и управлению, также изображаются на диаграмме специальными узлами, а стрелками указываются направления потоков. На рис. 4.3 приведены основные компоненты структурных карт Константайна.

Рис. 4.3. Элементы структурных карт:
а — модуль; *б* — вызов модуля; *в* — связь по данным; *г* — связь по управлению

Модуль является базовым элементом структурной карты. Различают следующие типы модулей (рис. 4.4):

- **модуль** (рис. 4.4, *а*);
- **подсистема** — детализированный модуль или программа. Может использоваться повторно любое число раз (рис. 4.4, *б*);
- **библиотека** — совокупность подпрограмм, размещенных в модуле отдельно от данной системы (рис. 4.4, *в*);
- **область данных** — описывает модули, содержащие исключительно области глобальных/распределенных данных (рис. 4.4, *г*).

Отдельные части программной системы (программы, подпрограммы) могут вызываться последовательно, параллельно или как сопрограммы (рис. 4.5).

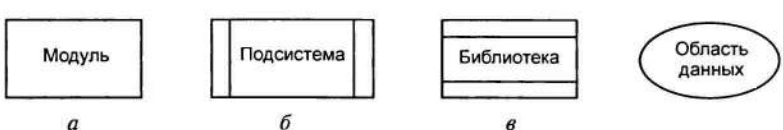


Рис. 4.4. Типы модулей

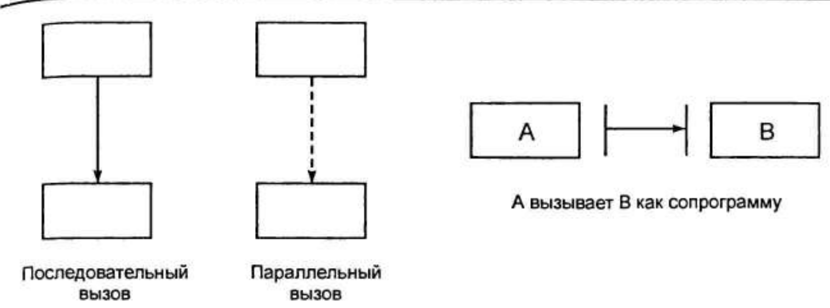


Рис. 4.5. Типы вызовов модулей

Для моделирования условных и циклических вызовов применяются следующие узлы (рис. 4.6):

- **условный узел** применяется для моделирования конструкций IF-THEN-ELSE (на диаграмме из узла выходят два потока) и IF-THEN (из узла выходит один поток). Условный узел изображается в виде ромба, потоки — альтернативные вызовы — изображаются выходящими из него;
- **итерационный узел** используется для того, чтобы показать, что вызываемый модуль выполняется в цикле. Он изображается полуокружностью со стрелкой с выходящими из него потоками.

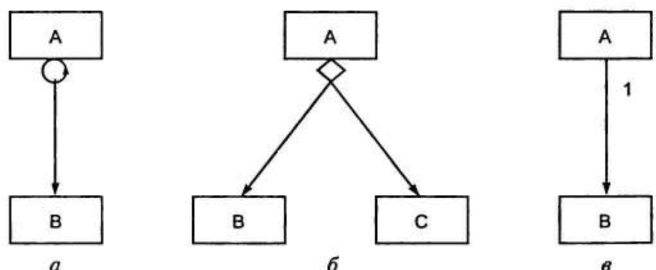


Рис. 4.6. Условные и циклические вызовы модулей:
а — циклический; *б* — условный; *в* — однократный

Если необходимо показать, что подчиненный модуль вызывается однократно, это осуществляется указанием цифры «1» рядом со стрелкой, обозначающей вызов модуля-наследника.

Связи по данным и управлению между модулями (передаваемые как параметры) обозначают стрелками, параллельными дуге вызова, которые показывают направления связей (рис. 4.7).

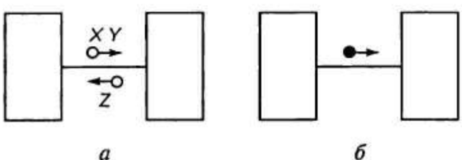


Рис. 4.7. Связи:
а — по данным; *б* — по управлению

Пример 4.2. Разработать структурную карту Константайна для задачи сортировки одномерного массива с помощью алгоритмов Пузырька, прямого выбора и Шелла.

Программа состоит из модулей **Меню, Методов сортировки** и **Вывода результата**. Пользователь выбирает нужный метод, вводит массив и получает в результате отсортированный массив.

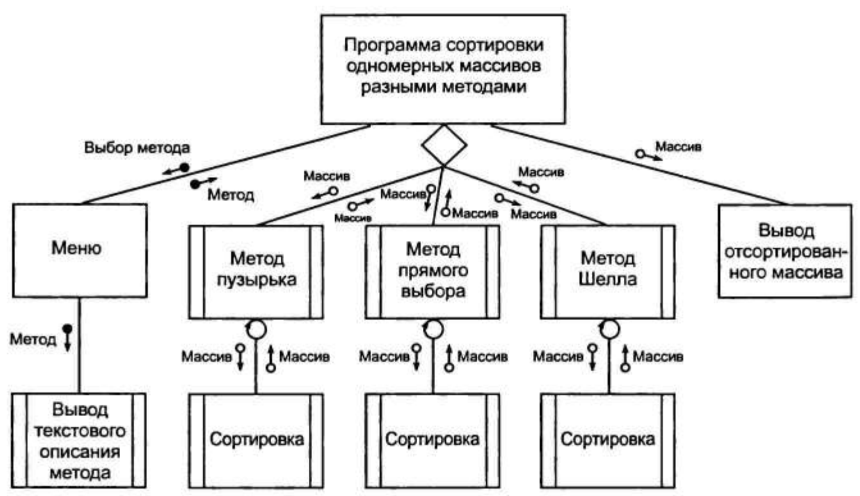


Рис. 4.8. Пример структурной карты Константайна

Результат приведен на рис. 4.8.

4.1.5. Структурные карты Джексона

Техника структурных карт Джексона основана на методе структурного программирования Джексона, который выявляет соответствие между структурой потоков данных и структурой

программы [39]. Основное внимание в методе сконцентрировано на соответствии входных и выходных потоков данных. Структуры на диаграммах Джексона строятся из четырех основных компонентов, представленных на рис. 4.9:

- операция — блок кодов, имеющий один вход и один выход (рис. 4.9, *а*);
- следование — последовательное выполнение операций слева направо (рис. 4.9, *б*);
- выбор — выполнение одной из операций в зависимости от выполнения условия (рис. 4.9, *в*);
- итерация — многократное выполнение блока (рис. 4.9, *г*).

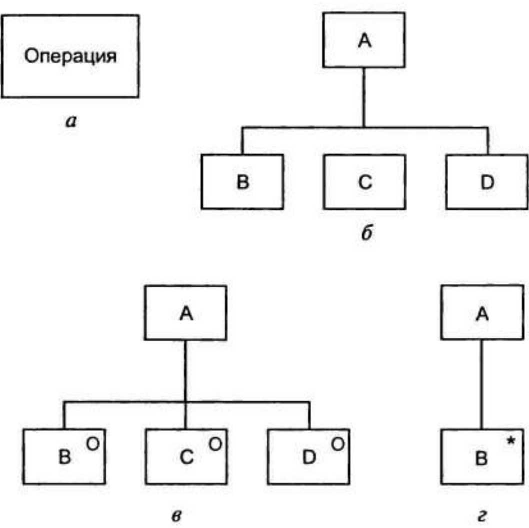


Рис. 4.9. Элементы структурных диаграмм Джексона

Пример 4.3. У менеджера торговой фирмы имеется файл, содержащий записи о принтерах со следующими полями: **фирма-производитель**, **марка**, **скорость печати**, **стоимость**, **количество единиц на складе**. Эти поля образуют структуру входных данных. По запросу менеджера программа выдает сведения о нужных покупателю принтерах в соответствии с критерием поиска. Критерием может быть: цена, скорость или фирма-производитель. Выходными данными является список, содержащий наименования выбранных принтеров.

С точки зрения структурного программирования Джексона алгоритм программы будет следующим:

Программа
    Цикл-пока не конец файла
        Прочитать запись
        Сравнить заданные поля с критерием поиска
        Если совпали
            Сохранить в выходной список
        Конец-если
    Конец-цикл
Вывод результирующего списка
Конец-программа

Рис. 4.10. Структурная карта Джексона

Полученная структурная карта Джексона приведена на рис. 4.10.

4.1.6. CASE-технологии

CASE-технологии (Computer-Aided Software/System Engineering — разработка программного обеспечения/систем с использованием компьютерной поддержки) — это реализованные в виде программных продуктов технологические системы, ориентированные на создание сложных программных систем и поддержку их полного жизненного цикла или его основных этапов. В настоящее время CASE-технологии используются не только для производства ПП, но и как мощный инструмент решения исследовательских и проектных задач (структурный анализ предметной области, моделирование деловых предложений с целью

решения задач оперативного и стратегического планирования и управления ресурсами) [53].

CASE-технологии начали развиваться в связи с развитием методологии структурного программирования. Их развитие стало возможным благодаря тому, что формализация в структурном программировании оказалась наиболее приемлемой для автоматизации. Таким образом, CASE-средства являются результатом эволюционного развития отрасли инструментальных (или технологических) средств.

CASE-средства обладают следующими основными достоинствами:

- повышают качество создаваемого ПО с помощью средств автоматического контроля;
- ускоряют процесс проектирования и разработки;
- позволяют за короткое время создавать прототип будущей системы, что позволяет на ранних этапах оценить ожидаемый результат;
- освобождают разработчика от рутинной работы, частично генерируя коды программ;
- поддерживают технологии повторного использования компонентов ПО;
- поддерживают развитие и сопровождение разработки.

При использовании CASE-технологий изменяются фазы жизненного цикла программного продукта, как показано в табл. 4.2.

Таблица 4.2. **Сравнительная характеристика этапов жизненного цикла ПО**

Традиционная технология	CASE-технология
Анализ	Прототипирование
Проектирование	Проектирование спецификаций
	Контроль проекта
Кодирование	Кодогенерация
Тестирование	Системное тестирование
Сопровождение	Сопровождение

Наиболее просто автоматизируемыми оказались стадии «контроль проекта» и «кодогенерация», хотя все остальные этапы жизненного цикла ПО также поддерживаются CASE-технологиями. Кроме изменения содержания фаз, существенно изменилось распределение трудозатрат по фазам, как показано в табл. 4.3.

Таблица 4.3. **Распределение трудозатрат по фазам жизненного цикла ПО**

Технология	Этапы разработки, %			
	Анализ	Проектирование	Кодирование	Тестирование
Традиционная	20	15	20	45
CASE-I	30	30	15	25
CASE-II	40	40	5	15

Таблица 4.4 содержит сравнительную характеристику целей и содержания этапов жизненного цикла ПО при традиционной разработке и с помощью CASE-средств.

Таблица 4.4. **Цели и содержание этапов жизненного цикла ПО**

№ п/п	Традиционная разработка	CASE-технология
1	Основные усилия — на кодирование и тестирование	Основные усилия — на анализ и проектирование
2	«Бумажные» спецификации	Быстрое итеративное прототипирование
3	Ручное кодирование	Автоматическая кодогенерация
4	Ручное документирование	Автоматическая генерация документации
5	Тестирование кодов	Автоматический контроль проекта
6	Сопровождение кодов	Сопровождение спецификаций проектирования

CASE-технология базируется на спиральной модели жизненного цикла ПО. На начальных этапах жизненного цикла (анализ требований, проектирование спецификаций, предварительное и детальное проектирование) проверяется и обосновывается реализуемость технических решений путем создания прототипов. Эта работа повторяется на каждом витке спирали, причем каждый следующий виток характеризуется более высокой степенью детализации создаваемого ПО. Окончанием витка является уточнение целей и характеристик проекта и планирование работ следующего витка спирали. Тем самым реализуется нисходящий принцип проектирования.

Чем же принципиально CASE-технология отличается от традиционной технологии разработки ПО? Девизом разработчиков

CASE-технологий является фраза «одна картинка стоит тысячи слов». Поэтому при использовании CASE-средств функционирование объекта (разрабатываемого ПО) отражается в различных схемах, таблицах, диаграммах, картах и т. п.

Большинство CASE-технологий основано на парадигме методология/метод/нотация/средство.

Методология на основе некоторого подхода определяет шаги работы, их последовательность, а также правила распределения и назначения методов.

Метод определяет способ достижения той или иной цели.

Нотацией называют систему обозначений, используемых для описания структуры системы, элементов данных, этапов обработки и других компонентов. Нотации могут быть графические (представление моделей в виде таблиц, графов, диаграмм, схем и т. п.) и текстовые (описания моделей на формальных и естественных языках).

Средства — инструментарий для поддержки методов. Эти инструменты обеспечивают работу пользователей-разработчиков при создании и редактировании проекта в интерактивном режиме, выполняют проверки соответствия компонентов и кодируют на некотором языке программирования модули ПО.

Наиболее часто и эффективно в методологии структурного анализа используются следующие средства:

- DFD (Data Flow Diagrams) — диаграммы потоков данных совместно со словарями данных и спецификациями процессов;
- ERD (Entity-Relationship Diagrams) диаграммы «сущность—связь»;
- STD (State Transition Diagrams) — диаграммы переходов состояний.

Современные структурные методологии анализа и проектирования классифицируются по следующим признакам:

- по типу целевых систем — для систем реального времени и для информационных систем;
- по отношению к школам — Software Engineering (SE) и Information Engineering (IE);
- по порядку построения моделей — процедурно-ориентированные, ориентированные на данные и информационно-ориентированные.

В табл. 4.5 приведены отличия информационных систем от систем реального времени.

Таблица 4.5. **Отличия информационных систем от систем реального времени**

Информационные системы	Системы реального времени
Управляются данными	Управляются событиями
Сложные структуры данных	Простые структуры данных
Большой объем входных данных	Малое количество входных данных
Интенсивный ввод-вывод	Интенсивные вычисления
Машинная независимость	Машинная зависимость

SE применяется при разработке как информационных систем, так и систем реального времени и реализует нисходящий подход к проектированию ПО. Эта дисциплина более апробирована, так как появилась раньше IE.

IE используется для проектирования информационных систем. Она новее, чем SE, и имеет более широкую область применения, поскольку является дисциплиной построения систем вообще, а не только систем ПО.

Различие в порядке построения моделей трактуется следующим образом. Традиционный процедурно-ориентированный подход регламентирует первичность проектирования функциональных компонентов по отношению к проектированию структур данных. При подходе, ориентированном на данные, вход и выход являются наиболее важными — структуры данных определяются первыми, а процедурные компоненты являются производными от данных. Информационно-ориентированный подход позволяет работать с неиерархическими структурами данных.

Ниже приводится деление CASE-средств по функциональным характеристикам.

Анализ и проектирование

Данные средства применяются для проектирования и создания спецификаций программной системы, поддерживают SE и IE:

- CASE-аналитик (Эйтекс);
- POSE (Computer Systems Advisers);
- Design/IDEF (Meta Software);
- BPWin (Logic Works);
- SELECT (Select Software Tools);
- CASE/4/0 (micro TOOl GmbH);
- и ряд других средств.

Проектирование баз данных и файлов

Технологии данной группы служат для логического моделирования данных, автоматического преобразования моделей в третью нормальную форму, автоматическую генерацию схем баз данных и описаний форматов файлов на уровне программного кода:

- ERWin (Logic Works);
- S-Designor (SPD);
- Designtr/2000 (Oracle);
- Sillverrun (Computer Systems Advisers).

Программирование

Данные средства позволяют получать из спецификаций полностью документированную выполняемую программу, поддерживают кодогенерацию и тестирование:

- COBOL 2/Workbench (Mikro Focus);
- DECASE (DEC);
- NETRON/CAP (Netron);
- APS (Sage Softwfre).

Сопровождение и реинжиниринг

К этим средствам относятся документаторы, анализаторы программ, средства реструктурирования:

- Adpac CASE Tools (Adpac);
- Scan/COBOL и SuperStructure (Computer Data Systems);
- Inshtctor/Recoder (language Tecnologe).

4.1.7. Ускорение разработки программного обеспечения. Методология RAD

В связи с развитием CASE-технологий в рамках спиральной модели жизненного цикла ПО в последнее время широкое распространение получила методология быстрой разработки приложений RAD (Rapid Application Development). Процесс разработки при этом содержит три элемента [53]:

- небольшую команду программистов (от 2 до 10 человек), что облегчает управление проектом;

- короткий, но тщательно проработанный производственный график (от 2 до 6 мес.), повышает эффективность работы;
- итерационный подход, при котором разработчики, по мере того как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Команда разработчиков представляет собой группу профессионалов, имеющих опыт в анализе, проектировании, генерации кода и тестировании ПО с использованием CASE-средств. Кроме того, разработчики должны уметь преобразовывать в рабочие прототипы предложения конечных пользователей.

Жизненный цикл ПО по методологии RAD состоит из четырех фаз:

- анализа и планирования требований;
- проектирования;
- реализации;
- внедрения.

На фазе анализа и планирования происходит определение требований к разрабатываемому ПО силами пользователей под руководством специалистов-разработчиков. Пользователи системы определяют функции, которые она должна выполнять, выделяют те, которые требуют проработки в первую очередь, описывают информационные потребности. Определяется возможность реализации данного проекта в установленных рамках финансирования, на данных аппаратных средствах и т. п. Затем определяются временные рамки самого проекта в каждой из последующих фаз. Результатом данной фазы должны быть состав и приоритеты функций будущей ИС, предварительные функциональные и информационные модели ИС.

На фазе проектирования часть пользователей под руководством специалистов-разработчиков принимает участие в техническом проектировании системы. Пользователи, непосредственно взаимодействуя с разработчиками, уточняют и дополняют требования к системе, которые не были выявлены на фазе анализа и планирования требований. Для быстрого получения работающих прототипов приложений используются CASE-средства. Анализируется и при необходимости корректируется функциональная модель. Определяются требования разграничения доступа к данным. Каждый процесс рассматривается детально, и при необходимости для каждого элементарного процесса создается частичный прототип: экран, диалог, отчет, устраняющий неяс-

ности или неоднозначности. Здесь же выясняется, какой набор документации необходим для эксплуатации будущей системы.

По результатам анализа процессов принимается решение о количестве, составляющих ИС подсистем, поддающихся разработке одной командой разработчиков за приемлемое для RAD-проектов время — порядка 2—3 мес.

Результатом данной фазы должны быть:

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

Использование CASE-средств позволяет избежать искажения данных при передаче информации с фазы на фазу. Кроме того, в подходе RAD каждый прототип не выбрасывается после выполнения своей задачи, а развивается в часть будущей системы. Поэтому на следующую фазу передается уже более полная и полезная информация.

На фазе реализации выполняется непосредственно сама быстрая разработка приложения. Программный код частично формируется с помощью автоматических генераторов CASE-средств. Для контроля за выполнением требований к ПО привлекаются конечные пользователи. Во время разработки осуществляется тестирование каждой подсистемы, что уменьшает стоимость исправления ошибок в коде программ по сравнению с тестированием уже готовой программной системы.

Автономно разрабатываемые подсистемы постепенно внедряются в общую систему. При подключении очередной части производится тестирование. Затем осуществляется тестирование всей системы в целом. Завершается физическое проектирование системы. При этом производится анализ использования данных, если необходимо, создаются базы данных и подключаются к системе, определяются требования к аппаратным ресурсам, завершается разработка документации ПО и определяются способы увеличения производительности.

Результатом фазы является готовая система, удовлетворяющая всем согласованным требованиям.

На этапе внедрения проводят обучение пользователей, организационные изменения и постепенный переход на новую сис-

тему. При этом параллельно с новой системой продолжается эксплуатация старой системы до полного внедрения новой.

Методология RAD не претендует на универсальность. Она хороша в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика, и неприменима для построения сложных расчетных программ, операционных систем или систем управления космическими кораблями, т. е. программ, требующих написания большого объема (сотни тысяч строк) уникального кода.

Основные принципы методологии RAD:

- итерационная разработка приложений;
- необязательность полного завершения работ на каждом из этапов жизненного цикла;
- применение CASE-средств, обеспечивающих целостность данных;
- участие конечных пользователей в процессе разработки ИС;
- разработка прототипов, позволяющая полнее выяснить и удовлетворить потребности конечного пользователя;
- тестирование, производимое параллельно с разработкой;
- разработка подсистем несколькими немногочисленными хорошо управляемыми командами профессионалов;
- четкое планирование и контроль выполнения работ.

4.2. Проектирование программного обеспечения при объектном подходе

Задачи проектирования включают в себя две составляющие: логическое и физическое проектирование программных продуктов. Логическое проектирование заключается в разработке классов для реализации их экземпляров — объектов. Для этого требуется подробное описание полей и методов классов, а также связей между ними. Для этого используются статические диаграммы классов и объектов, динамические — последовательностей состояний и кооперации. Физическое проектирование предполагает построение программных компонентов из ранее определенных классов и объектов и размещение их на конкретных вычислительных устройствах. Разрабатываемые на этом этапе диаграммы — компонентов и развертывания [1].

4.2.1. Разработка структуры программного обеспечения при объектном подходе

На этапе проектирования уточняются поля и методы классов, а также отношения между классами. Все это находит отражение на диаграмме классов.

Для уточнения содержания некоторых классов на диаграмме используют следующие обозначения:

- *управляющий класс* (control class) отвечает за координацию действий других классов и контролирует последовательность выполнения действий варианта использования для данного ПО. На каждой диаграмме *классов* должен быть хотя бы один управляющий класс (рис. 4.11, *а*).
- *класс-сущность* (entity class) — пассивный класс, информация о котором должна храниться постоянно. Как правило, этот класс соответствует отдельной таблице базы данных. В этом случае его *атрибуты* являются полями таблицы, а *операции* — присоединенными или хранимыми процедурами (рис. 4.11, *б*);
- *граничный класс* (boundary class) располагается на границе системы с внешней средой. К этому типу относят как классы, реализующие пользовательские интерфейсы, так и классы, обеспечивающие интерфейс с аппаратными средствами или программными системами (рис. 4.11, *в*).

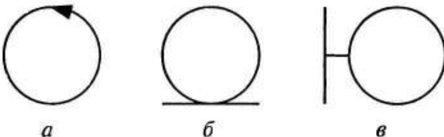


Рис. 4.11. Графическое изображение классов для моделирования программного обеспечения:
а — управляющий класс; *б* — класс-сущность; *в* — граничный класс

Отношения между классами

Кроме внутреннего устройства или структуры классов, на диаграмме классов необходимо отобразить различные отношения между ними. Основными отношениями или связями в языке UML являются [48]:

- отношение зависимости (dependency relationship);
- отношение ассоциации (association relationship);

- отношение обобщения (generalization relationship);
- отношение реализации (realization relationship).

Все эти отношения обозначаются по-своему на диаграмме и отражают различные типы взаимосвязей между классами и их объектами.

Отношение зависимости

Отношение зависимости используется в ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели.

Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов («→» или «←»). На диаграмме классов данное отношение связывает отдельные классы между собой, при этом стрелка направлена от класса-клиента зависимости к независимому классу или классу-источнику (рис. 4.12). На данном рисунке изображены два класса: Класс_А и Класс_Б, при этом Класс_Б является источником некоторой зависимости, а Класс_А — клиентом этой зависимости.

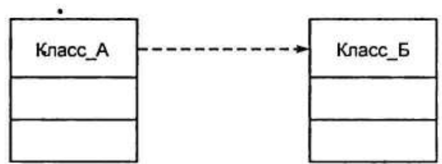


Рис. 4.12. Графическое изображение отношения зависимости на диаграмме классов

Отношение ассоциации

Отношение *ассоциации* обозначается сплошной линией с дополнительными специальными символами, которые характеризуют отдельные свойства конкретной ассоциации. Это могут быть имя ассоциации, а также имена и кратность классов-ролей ассоциации. *Имя ассоциации* является необязательным, но если оно задано, то записывается с прописной (заглавной) буквы рядом с линией соответствующей ассоциации.

Ассоциация, связывающая два класса (или класс с самим собой), называется бинарной. Для бинарной ассоциации на диаграмме может быть указан порядок следования классов с использованием треугольника в форме стрелки рядом с именем

данной ассоциации. Направление этой стрелки указывает на порядок классов, один из которых является первым (со стороны основания треугольника), а другой — вторым (со стороны вершины треугольника). Отсутствие данной стрелки рядом с именем ассоциации означает, что порядок следования классов в рассматриваемом отношении не определен.

На рис. 4.13 показано отношение бинарной ассоциации между классом «Группа» и классом «Студент». Они связаны между собой бинарной ассоциацией «Учеба», имя которой указано на рисунке над линией ассоциации. Порядок следования классов в данном отношении таков: первым является класс «Студент», а вторым — класс «Группа».

Рис. 4.13. Графическое изображение отношения бинарной ассоциации между классами

Можно, хотя это редко бывает необходимо, создавать ассоциации, связывающие сразу несколько классов; они называются *N*-арными. *N*-арная ассоциация графически обозначается ромбом, от которого к символам классов данной ассоциации ведут линии. В этом случае ромб соединяется с символами соответствующих классов сплошными линиями. Имя *N*-арной ассоциации записывается рядом с ромбом соответствующей ассоциации.

Пример тернарной ассоциации показан на рис. 4.14. Здесь изображено отношение между тремя классами: «Футбольная команда», «Год» и «Игра», которое может представлять информацию об играх футбольных команд в национальном чемпионате в течение нескольких последних лет.

Рис. 4.14. Графическое изображение тернарной ассоциации между тремя классами

Наиболее важные свойства ассоциации указываются на диаграмме рядом с этими элементами ассоциации и должны перемещаться вместе с ними.

К таким свойствам относятся:

- *имя роли* отдельного класса, входящего в ассоциацию, представляет собой строку текста рядом с концом ассоциации для соответствующего класса. Имя роли не является обязательным элементом обозначений и может отсутствовать на диаграмме;
- *кратность* отдельных классов, являющихся концами ассоциации. Интервал кратности записывается рядом с концом ассоциации и для N-арной ассоциации означает потенциальное число отдельных экземпляров или значений кортежей этой ассоциации, которые могут иметь место, когда остальные $N-1$ экземпляров или значений классов фиксированы.

В рассмотренном ранее примере (см. рис. 4.12) кратность «1» для класса «Группа» означает, что каждый студент может учиться только в одной группе. Кратность «1..*» для класса «Студент» означает, что в каждой группе могут учиться несколько студентов, общее число которых заранее неизвестно и ничем не ограничено, но всегда больше нуля.

На диаграмме классов может присутствовать так называемая исключающая ассоциация (Xor-association). Она означает, что из нескольких потенциально возможных вариантов данной ассоциации в каждый момент времени может использоваться только один ее экземпляр. Исключающая ассоциация изображается пунктирной линией, соединяющей две ассоциации и более, рядом с которой записывается строка-ограничение {xor}.

Например, счет в банке может быть открыт для клиента, в качестве которого может выступать физическое лицо или компания, что изображается с помощью исключающей ассоциации (рис. 4.15).

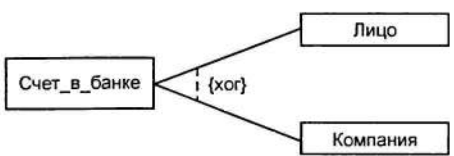


Рис. 4.15. Графическое изображение исключающей ассоциации между тремя классами

Отношение агрегации

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Данное отношение применяется для представления системных взаимосвязей типа «часть—целое». Раскрывая внутреннюю структуру системы, отношение агрегации показывает, из каких компонентов состоит система и как они связаны между собой. Это отношение по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость в последующем. При этом части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются вполне самостоятельными сущностями. Более того, части целого обладают своими собственными атрибутами и операциями, которые могут существенно отличаться от атрибутов и операций целого.

Агрегация является частным случаем ассоциации и изображается в виде пустой ассоциации с незакрашенным ромбом со стороны «целого» (рис. 4.16).

Рис. 4.16. Графическое изображение отношения агрегации в языке UML

Примером отношения агрегации может служить деление персонального компьютера на составные части: системный блок, монитор, клавиатуру и мышь (рис. 4.17).

Рис. 4.17. Диаграмма классов для иллюстрации отношения агрегации на примере структуры персонального компьютера

Отношение композиции

Отношение композиции является частным случаем отношения агрегации. Это отношение служит для описания специальной формы отношения «часть—целое», при которой составляющие части в некотором смысле находятся внутри целого. Причем части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой класс-композицию или «целое» (рис. 4.18).

Рис. 4.18. Графическое изображение отношения композиции в языке UML

Пример отношения композиции — окно интерфейса программы, которое может состоять из строки заголовка, кнопок управления размером, полос прокрутки, главного меню, рабочей области и строки состояния. В данном случае наглядно представлено отношение композиции.

В качестве дополнительных обозначений для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно, указание кратности класса ассоциации и имени данной ассоциации, которые не являются обязательными. Диаграмма классов для класса «Окно_программы», описанного выше, может иметь следующий вид (рис. 4.19).

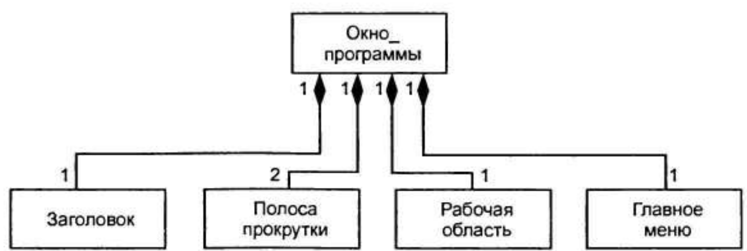


Рис. 4.19. Диаграмма классов для иллюстрации отношения композиции на примере класса окна программы

Отношение обобщения

Отношение обобщения является отношением между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком). Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка. Графически отношение обобщения изображается в виде линии с большой незакрашенной стрелкой, направленной на родителя (рис. 4.20).

Рис. 4.20. Графическое изображение отношения обобщения в языке UML

Пример отношения обобщения показан на рис 4.20. Здесь абстрактный класс «*Геометрическая фигура*» выступает в качестве суперкласса (класса-предка) для подклассов (классов-потомков), соответствующих конкретным геометрическим фигурам «Прямоугольник», «Окружность», «Эллипс» и др.

С целью упрощения обозначений на диаграмме классов совокупность линий, обозначающих одно и то же отношение обобщения, может быть объединена в одну линию. В этом случае данные отдельные линии изображаются сходящимися к единственной стрелке, имеющей с ними общую точку пересечения (рис. 4.21).

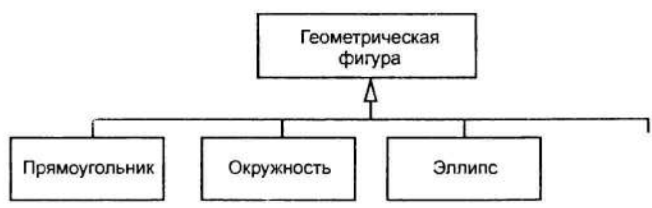


Рис. 4.21. Пример графического изображения обобщения классов

Многоточие вместо прямоугольника на диаграмме означает возможность наличия других классов-потомков, не включенных в обозначения представленных на диаграмме классов.

Для того чтобы проиллюстрировать описанные выше типы отношений, рассмотрим следующий пример:

Пример 4.4. Разработать диаграмму классов для некоей компании — класс «Компания», которая состоит из нескольких отделов — класс «Отдел», каждый из которых располагается в своем офисе — класс «Офис», имеет штаб-квартиру класс «Штаб-квартира» и содержит штат сотрудников — класс «Person», сведения о которых содержатся в системе кадрового учета. Каждый из вышеприведенных классов обладает своими атрибутами и операциями и связан с другими классами определенным типом отношений.

Полученная диаграмма приведена на рис. 4.22.

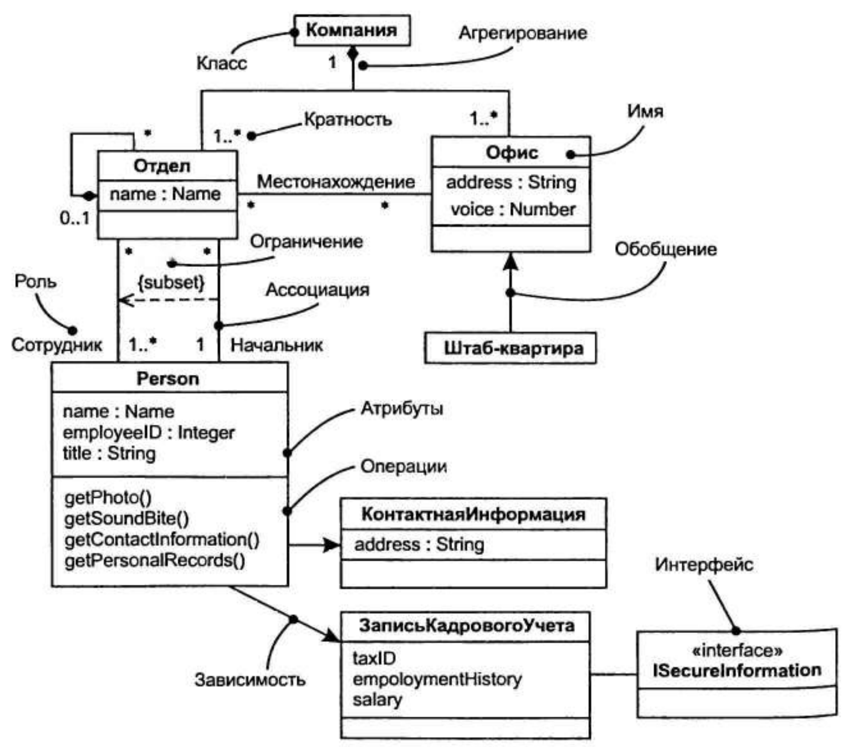


Рис. 4.22. Диаграммы классов

Кроме классов на диаграмме могут изображаться интерфейсы.

Интерфейсы

Интерфейс (interface) — именованное множество операций, характеризующих поведение отдельного элемента модели извне без указания их внутренней структуры.

В языке UML интерфейс является специальным случаем класса, у которого имеются *операции*, но отсутствуют *атрибуты*. Для обозначения интерфейса на диаграмме классов используется специальный графический символ — окружность, рядом с которой указывается имя интерфейса, или стандартный способ — прямоугольник класса с обозначением «Interface» (рис. 4.23).

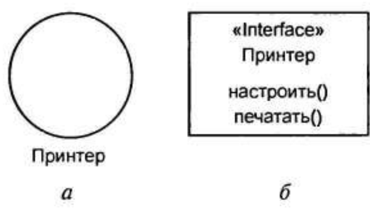


Рис. 4.23. Обозначения интерфейсов

Объекты

На этапе проектирования, кроме диаграмм классов, большое значение имеют *диаграммы объектов*, которые показывают взаимодействие между экземплярами определенных классов в некоторый момент времени.

Объект (object) — это отдельный экземпляр класса, который создается на этапе выполнения программы. Он имеет свое собственное имя и конкретные значения атрибутов. Имя объекта представляет собой строку текста «имя объекта»   «имя класса», разделенную двоеточием. Для графического изображения объектов используется такой же символ прямоугольника, как и для классов, но имя объекта в отличие от имени класса выделяется подчеркиванием. Пример обозначения объектов приведен на рис. 4.24.

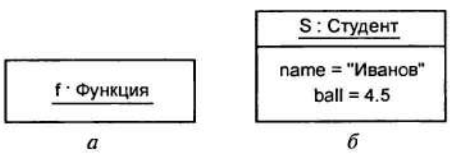


Рис. 4.24. Обозначения объектов

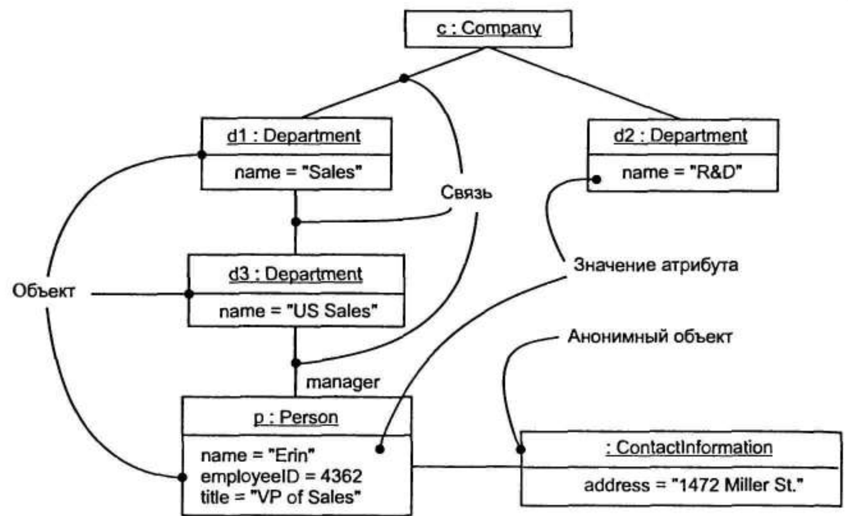


Рис. 4.25. Пример диаграммы объектов

Пример диаграммы объектов компании, состоящей из нескольких отделов — объекты d1, d2, d3 класса «Department», приведен на рис. 4.25.

4.2.2. Диаграммы кооперации

Диаграмма кооперации является альтернативным диаграмме последовательностей способом представления взаимодействия объектов. Она показывает потоки данных между объектами классов, что позволяет уточнить отношения между ними.

На диаграмме изображаются участвующие во взаимодействии объекты (в виде прямоугольников, содержащих имя объекта, имя его класса и значения атрибутов, если они имеются), а также указываются ассоциации между этими объектами, если необходимо, указывают имя ассоциации и роли объектов в данной ассоциации. Дополнительно могут быть изображены динамические связи — потоки сообщений. Они представляются также в виде соединительных линий между объектами, над которыми располагается стрелка с указанием направления, имени сообщения и порядкового номера в общей последовательности инициализации сообщений. Номера служат для синхронизации сообщений, так как на диаграмме кооперации прямо не указывается время.

Пример 4.5. Разработать диаграмму коопераций, иллюстрирующую взаимоотношения компании с составляющими ее отделами и сотрудниками.

Компания организует и расформировывает отделы, принимает и увольняет сотрудников, выдает задание отделам. Сотрудники выполняют работу и сдают отчеты начальникам отделов.

Решение приведено на рис. 4.26.

Рис. 4.26. Пример диаграммы кооперации

Заметим, что на приведенной диаграмме объекты классов «Отдел» и «Сотрудник» реализованы мультиобъектами.

Диаграммы компонентов и *развертывания* выполняются на этапе физического проектирования программных систем для того, чтобы привязать их к некоторому аппаратному обеспечению.

4.3. Экстремальное программирование

4.3.1. Основополагающие практики XP

В методологии XP имеется много спорных моментов. Одним из ключевых таких моментов является то, что она базируется на эволюционном, а не предварительном проектировании. А как

мы уже выяснили, использование эволюционного проектирования не может привести ни к чему хорошему из-за обилия сиюминутных проектировочных решений и энтропии программного продукта [3].

В основе этого утверждения лежит кривая стоимости изменений в программном продукте. Согласно этой кривой по мере развития проекта стоимость внесения изменений экспоненциально возрастает. Получается, что если экспоненциальная кривая верна, то эволюционное проектирование вообще нельзя использовать в работе. Отсюда же следует, что нельзя делать ошибки в предварительном проектировании — затраты на их исправление будут определяться все той же зависимостью.

В основе XP лежит предположение, что эту кривую можно сгладить до такой степени, чтобы можно было применять эволюционное проектирование. Такое сглаживание, с одной стороны, возникает при использовании методологии XP, а с другой — оно же в ней и используется. Это еще раз подчеркивает тесную взаимосвязь между практиками XP: нельзя использовать те части методологии, которые предполагают существование сглаживания, не используя те практики, которые это сглаживание осуществляют.

У практик, с помощью которых осуществляется сглаживание, есть множество составляющих. В основе всех их лежит Тестирование и Непрерывная интеграция. Именно надежность кода, которую обеспечивает тестирование, делает возможным все остальное в этой методологии. Непрерывная интеграция необходима для синхронной работы всех разработчиков, так чтобы любой человек мог вносить в систему свои изменения и не беспокоиться об интеграции с остальными членами команды. Взятые вместе, эти две практики могут оказывать существенное влияние на кривую стоимости изменений в программном продукте.

Подобный эффект имеет и рефакторинг (переработка написанного кода). Те, кто делают рефакторинг в той строгой манере, что принята в XP, отмечают значительное повышение его эффективности по сравнению с более бессистемной реструктуризацией.

Все эти основополагающие практики (непрерывная интеграция, тестирование и рефакторинг) создают новую среду, в которой эволюционное проектирование выглядит вполне убедительно.

4.3.2. Преимущества простого дизайна

В XP очень популярны два лозунга: «Do the Simplest Thing that Could Possibly Work» («Ищите самое простое решение, которое может сработать») и YAGNI («You Aren't Going to Need It» — «Это вам не понадобится»). Оба они олицетворяют собой одну из практик XP под названием Простой дизайн [3].

По принципу YAGNI вы не должны заниматься написанием кода сегодня, если он понадобится для того свойства программы, которое вы будете реализовывать только завтра. На первый взгляд в этом нет ничего сложного. Сложности начинаются, когда речь заходит о таких вещах, как программные каркасы для создания приложений, компоненты для повторного использования и гибкий дизайн. Надо сказать, что спроектировать их довольно сложно. Вы заранее добавляете к общей стоимости работ стоимость и такого проектирования и рассчитываете впоследствии вернуть эти деньги. При этом наличие заблаговременно встроенной в систему гибкости считается признаком хорошего проектирования.

Тем не менее XP не советует заниматься созданием гибких компонентов и каркасов до того, как понадобится именно эта функциональность. Лучше, если эти структуры будут наращиваться по мере необходимости. Если сегодня мне нужен класс Money, который обрабатывает сложение, а не умножение, то сегодня я буду встраивать в этот класс только сложение. Даже если я абсолютно уверен, что умножение понадобится мне уже в следующей итерации, и я знаю, как очень просто и быстро это сделать сейчас, все равно я должен оставить это на следующую итерацию, когда в нем появится реальная необходимость.

Такое поведение оправдано с экономической точки зрения. Занимаясь работой, которая понадобится только завтра, я тем самым расходую силы и время, предназначенные для задач, которые должны были быть сделаны сегодня. План выпуска программы четко указывает, над чем мне нужно работать в настоящий момент. Если я отклоняюсь от него, чтобы поработать над тем, что понадобится в будущем, я нарушаю свое соглашение с заказчиком. Кроме того, появляется риск не успеть сделать все записанное в требованиях для текущей итерации. И даже в том случае, если такой опасности нет и у вас появилось свободное время, то решать, чем вам заняться, — прерогатива заказчика, который может попросить заняться вовсе не умножением.

Таким образом, возможные препятствия экономического характера осложняются еще и тем, что мы можем ошибаться. Даже если мы абсолютно уверены в том, как работает эта функция, мы все равно можем ошибиться, особенно если у нас еще нет подробных требований заказчика. А чем раньше мы используем в работе над проектом ошибочные решения, тем хуже. Приверженцы методологии XP считают, что в такой ситуации гораздо легче принять неправильное решение.

Другая причина, по которой простой дизайн лучше сложного, — отказ от принципа «блуждающего огонька». Сложную конструкцию гораздо труднее понять, чем простую. Именно поэтому любая модификация системы делает ее все более сложной. Это опять-таки ведет к увеличению стоимости работ в период между тем временем, когда дизайн системы стал более сложным, и временем, когда это действительно стало необходимо [3].

Такой стиль работы многим кажется абсурдным, и надо сказать, что они правы. Правы при одном условии — абсурд получится, если эту практику начать применять в обычном процессе разработки, а все остальные практики XP игнорировать. Если же изменить существующий баланс между эволюционным и предварительным проектированием, то YAGNI становится очень полезным принципом (тогда и только тогда).

Подведем итог. Не стоит расходовать силы на то, чтобы внести в систему новую функциональность, если она не понадобится до следующей итерации. Даже если это практически ничего не стоит, вам не нужно это делать, так как это увеличит общую стоимость модификации. Однако для того, чтобы осознанно применять такой принцип на деле, вам нужно использовать XP или другую подобную методологию, которая снижает стоимость изменений.

4.3.3. Простой дизайн

Итак, необходимо, чтобы программный код был максимально прост. В конце концов, кому нужно, чтобы код был сложный и запутанный? Осталось только понять, что мы разумеем под словом «простой».

В книге Extreme Programming Explained Кент приводит четыре критерия простой системы [20]. Вот они в порядке убывания важности:

- система успешно проходит все тесты;

- код системы ясно раскрывает все изначальные замыслы;
- в ней отсутствует дублирование кода;
- используется минимально возможное количество классов и методов.

Успешное тестирование системы — довольно простой критерий. Отсутствие дублирования кода тоже вполне четкое требование, хотя большинство разработчиков нужно учить, как этого достичь. Самое сложное скрывается в словах «раскрывает изначальные замыслы». Что это значит?

Основное достоинство программного кода в данном случае — его ясность. XP всячески подчеркивает, что хороший код — это код, который можно легко прочесть. Скажите XP-шнику, что он пишет «заумный код», и будьте уверены, что обругали этого человека. Но понимание замыслов программиста, написавшего код, зависит также и от опыта и ума того, кто этот код пытается прочесть. Однако отметим, что не стоит думать над вопросом, как сделать дизайн максимально простым. В конце концов, позже вы сможете (и должны, и будете) заняться рефакторингом. В конце работы над проектом желание делать рефакторинг гораздо важнее, чем точное понимание того, какое решение является самым простым.

4.3.4. Рефакторинг и принцип YAGNI

Эта тема сравнительно недавно всплыла в списке рассылки, посвященном XP, и, коль скоро мы заговорили о роли проектирования, нам стоит ее обсудить.

Дело в том, что процесс рефакторинга требует времени, но не добавляет новой функциональности. С другой стороны, принцип YAGNI гласит, что надо проектировать только для текущей функциональности, а не для того, что понадобится в будущем. Не сталкиваемся ли мы здесь с противоречием?

Принцип YAGNI состоит в том, чтобы не делать систему более сложной, чем того требует реализация текущих задач. Это является частью практики «Простой дизайн». Рефакторинг же необходим для поддержания системы в максимально простом состоянии. Его нужно проводить сразу же, как только вы обнаружите, что можете что-либо упростить.

Простой дизайн одновременно задействует практики XP и сам по себе является основополагающей практикой. Только при

условии тестирования, непрерывной интеграции и рефакторинга можно говорить об эффективном использовании простого дизайна. Но в то же время простой дизайн абсолютно необходим для сглаживания кривой стоимости изменений. Любая излишне сложная конструкция затруднит внесение изменений в систему по всем направлениям, за исключением того из них, ради которого эта сложность в нее вносилась. Однако редко удается предсказать такое направление, поэтому лучше будет стремиться к простым решениям. И в то же время мало кому удается сделать все максимально просто с первого раза, так что вам придется заниматься рефакторингом, чтобы приблизиться к цели.

4.3.5. Наращивание архитектуры

Термин «архитектура» передает идею основных элементов системы, тех ее частей, которые трудно изменить. Они являются фундаментом, на котором можно построить все остальное [6].

Какую роль играет архитектура в эволюционном проектировании? Критики XP считают, что эта методология вообще не признает работы над архитектурой, что вся суть XP — сразу садиться за написание кода и уповать на то, что рефакторинг решит все проблемы с проектированием. Они правы, и, может быть, в этом заключается некоторая слабость XP Приверженцы XP — Кент Бек (Kent Beck), Рон Джеффриз (Ron Jeffries) и Боб Мартин (Bob Martin) — прикладывают очень много сил, чтобы вообще избежать любого предварительного проектирования архитектуры. Не добавляйте в систему базу данных, пока она вам действительно не понадобилась. Работайте поначалу с файлами, а база данных появится в следующей итерации, в результате рефакторинга.

Однако рекомендуется все-таки начинать работу с приблизительной оценки архитектуры системы. Если вы видите большое количество данных и множество различных пользователей, смело включайте в архитектуру базу данных. Если вы должны работать со сложной бизнес-логикой, используйте модель предметной области. Однако не забывайте об уважении к богам YAGNI и в сомнительных случаях отдавайте предпочтение более простым решениям. Кроме того, всегда будьте готовы выбросить кусок архитектуры, если видите, что он не приносит ничего полезного.

4.3.6. UML и XP

В идеале XP полностью отрицает проектирование системы, в частности методами UML. Тем не менее программисты все же часто используют на начальном этапе диаграммы UML. На самом деле диаграммы очень полезны для понимая разрабатываемого продукта, но чтобы они сделали процесс более длительным и трудоемким, необходимо их использовать правильно [2, 25].

Советы тем, кто хочет правильно использовать диаграммы.

Во-первых, пока рисуете диаграмму, не забывайте, для чего вы это делаете. Основное ее достоинство — коммуникация с людьми. Чтобы коммуникация была эффективной, нужно отображать на диаграмме только важные аспекты, не обращая внимания на все второстепенные. Такая избирательность — основа правильной работы с UML. Не надо отображать на диаграмме каждый класс — только самые важные. У классов не нужно задавать каждый атрибут или операцию — только самые важные. Не надо рисовать диаграммы последовательности для всех вариантов использования и сценариев — ну, и так далее. Самая распространенная проблема с использованием диаграмм — это то, что их пытаются сделать максимально всеобъемлющими. Однако самый лучший источник всеобъемлющей информации — это программный код, так как именно его легче всего синхронизировать с кодом. Для диаграммы же всеобъемлемость — враг удобопонятности.

Чаще всего диаграммы используются для того, чтобы проанализировать проектные решения еще до написания кода. Нередко при этом возникает чувство, что в XP этого делать нельзя. Это совсем не так. Многие полагают, что перед разработкой сложной задачи стоит ненадолго собраться всей командой для ее предварительного проектирования. Тем не менее, когда проводите такие собрания, не забывайте, что:

- они должны быть действительно недолгими;
- не нужно обсуждать все подробности (только самое важное);
- относитесь к полученному в результате проектному решению как к наброску, а не как к конечной версии, не подверженной изменениям.

Последний пункт стоит раскрыть подробнее. Когда вы занимаетесь предварительным проектированием, вы неизбежно обнаруживаете, что некоторые ваши решения неправильны. При-

чем обнаруживается это уже при кодировании. Разумеется, это не проблема, если вы после этого вносите соответствующие изменения. Проблемы начинаются тогда, когда вы полагаете, что с проектированием покончено, и не учитываете полученные сведения, сохраняя неверный дизайн [5].

Изменения в дизайне вовсе необязательно подразумевают изменения в диаграммах. Абсолютно разумным будет просто-напросто выбросить диаграмму, после того, как она помогла вам найти нужное решение. Нарисовав диаграмму, вы решили стоявшую перед вами проблему, и этого совершенно достаточно. Диаграмма и не должна существовать как некий постоянный артефакт. Надо сказать, что лучшие UML-диаграммы такими артефактами как раз не являются.

Кроме того, UML-диаграммы используются в качестве документации по проекту. Как правило, в своей обычной форме это модель, редактируемая с помощью некоторого CASE-инструмента. Идея здесь состоит в том, что ведение такой документации облегчает работу. На самом деле, чаще всего она вообще не нужна, поскольку:

- нужно постоянно тратить массу времени, чтобы не дать диаграммам устареть, в противном случае они не будут соответствовать программному коду;
- диаграммы находятся внутри сложного CASE-средства либо в толстенной папке, и никто туда не заглядывает.

Итак, если вы хотите иметь текущую документацию по проекту, учитывайте все вышеперечисленные проблемы:

- используйте только те диаграммы, которые вы можете поддерживать без особых усилий;
- помещайте диаграммы туда, где их все видят. Пусть остальные рисуют на ней ручкой все простые изменения, которые были внесены в изначальный вариант;
- посмотрите, обращают ли ваши разработчики на диаграммы хоть какое-то внимание, и если нет, выбросите их.

И наконец, последний аспект использования UML для документации — передача проекта в другие руки (например, от одной группы разработчиков другой). Согласно методологии XP создание документации — такая же задача, как и все остальные, а значит, ее приоритет должен быть определен заказчиком. В этой ситуации может пригодиться UML, разумеется, при условии избирательности диаграмм, которые создавались с целью облегчения коммуникации. Помните, что программный код —

это основной репозиторий подробной информации, а диаграммы служат для обобщенного представления основных аспектов системы.

4.3.7. Суть проектирования. Программирование и тестирование

Проектирование в XP требует от человека следующих качеств:

- постоянного желания сохранять программный код простым и понятным насколько это возможно;
- наличия навыков рефакторинга, так чтобы с уверенностью вносить в систему изменения, как только в этом возникнет необходимость;
- хорошего знания паттернов: рассматривать их не просто как готовые решения, а оценивать своевременность и использовать постепенно, от простого к сложному;
- умения объяснять при необходимости решения по конструированию системы (используя для этого программный код, диаграммы и, самое главное, личное общение).

Для того чтобы реализовать задачу, ответственный за нее программист прежде всего ищет себе партнера, поскольку окончательный код всегда пишется двумя людьми на одной машине. Если возникают вопросы о предмете или методах реализации, партнеры проводят короткую (15-минутную) встречу с заказчиком и/или программистами, осведомленными в вопросах кодирования задач, которые с наибольшей вероятностью будут связаны с кодом данной задачи в ходе реализации [1, 5].

По результатам этой встречи программисты составляют список тестовых примеров, которые необходимо прогнать до завершения реализации задачи. Из списка выбирается такой тест, в реализации которого программисты полностью уверены и с помощью которого они смогут лучше понять суть задачи. Пишется тестовая программа. Если она сразу нормально заработает, можно двигаться дальше. Однако, как правило, без проблем не обходится. В случае если тест не работает, возможна одна из следующих ситуаций:

- мы знаем простой способ заставить его работать, и мы действуем этим способом;

- мы знаем сложный и очень неприятный способ заставить его работать, но понимаем, как изменить архитектуру системы и добиться нормальной работы тестового примера без лишних усилий. Тогда мы решаемся на переработку системы;
- мы знаем сложный и неприятный способ заставить его работать и не видим никакой возможности переработать систему, поэтому мы идем этим сложным путем.

После того как тест заработал, мы, возможно, снова поймем, как усовершенствовать систему, что и сделаем.

Вполне вероятно, что в ходе реализации тестового примера мы найдем другой тестовый пример, который также должен работать. Мы заносим новый тест в свой список и продолжаем разработку. Возможно, мы обнаружим, что масштабы перестройки системы выходят за рамки требований текущего теста, тогда зафиксируем и этот факт и двинемся дальше. В конце концов, наша цель — сконцентрироваться на деталях и успешно справиться с конкретной проблемой, но одновременно не потерять общего представления о системе, которое формируется в процессе интенсивной работы над кодами.

Контрольные вопросы

1. Приведите пример структурной схемы ПО.
2. Опишите основные элементы функциональных схем ПО.
3. Охарактеризуйте метод пошаговой детализации.
4. Как составляются структурные карты Константайна?
5. Как составляются структурные карты Джексона?
6. Что такое CASE-технологии?
7. Что такое RAD-технологии?
8. Охарактеризуйте модель проектируемого ПО при объектном подходе.
9. Что такое экстремальное программирование?

Глава 5
ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММ

В целом разработчики различают *дефекты* программного обеспечения и *сбои*. В случае сбоя программа ведет себя не так, как ожидает пользователь. Дефект — это ошибка/неточность, которая может быть (а может и не быть) следствием сбоя.

Общепринятая практика состоит в том, что после завершения продукта и до передачи его заказчику независимой группой тестировщиков проводится тестирование ПО. Эта практика часто выражается в виде отдельной фазы тестирования (в общем цикле разработки ПО), которая часто используется для компенсирования задержек, возникающих на предыдущих стадиях разработки. Другая практика состоит в том, что тестирование начинается вместе с началом проекта и продолжается параллельно созданию продукта до завершения проекта. Второй путь обычно требует больших трудозатрат, но качество тестирования при этом будет выше.

Уровни тестирования:

- **модульное тестирование.** Тестируется минимально возможный для тестирования компонент, например отдельный класс или функция;
- **интеграционное тестирование.** Проверяется, есть ли какие-либо проблемы в интерфейсах и взаимодействии между интегрируемыми компонентами, например, не передается информация, передается некорректная информация;
- **системное тестирование.** Тестируется интегрированная система на ее соответствие исходным требованиям:
  - **альфа-тестирование** — имитация реальной работы с системой штатными разработчиками либо реальная работа с системой потенциальными пользователями/заказчиком на стороне разработчика. Часто альфа-тестирование применяется для законченного продукта в качестве внут-

реннего приемочного тестирования. Иногда альфа-тестирование выполняется под отладчиком или с использованием окружения, которое помогает быстро выявлять найденные ошибки. Обнаруженные ошибки могут быть переданы тестировщикам для дополнительного исследования в окружении, подобном тому, в котором будет использоваться ПО;
- **бета-тестирование** — в некоторых случаях выполняется распространение версии с ограничениями (по функциональности или времени работы) для некоторой группы лиц с тем, чтобы убедиться, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для того, чтобы получить обратную связь о продукте от его будущих пользователей.

5.1. Термины и определения

Выполнение программы с целью обнаружения ошибок называется ***тестированием***. Виды ошибок и способы их обнаружения приведены в табл. 5.1.

Таблица 5.1. **Виды программных ошибок и способы их обнаружения**

Виды программных ошибок	Способы их обнаружения
Синтаксические	Статический контроль и диагностика компиляторами и компоновщиком
Ошибки выполнения, выявляемые автоматически: а) переполнение, защита памяти; б) несоответствие типов; в) зацикливание	Динамический контроль: аппаратурой процессора; run-time системы программирования; операционной системой — по превышению лимита времени
Программа не соответствует спецификации	Целенаправленное тестирование
Спецификация не соответствует требованиям	Испытания, бета-тестирование

Эффективность контроля 1-го вида зависит и от языка, и от компилятора. Контроль 2-го вида осуществляется с помощью исключений — Exceptions и весьма полезен для проверки прав-

доподобности промежуточных результатов. *Тест* — это набор контрольных входных данных совместно с ожидаемыми результатами. В число входных данных времязависимых программ входят события и временные параметры. Ключевой вопрос — полнота тестирования: *какое* количество *каких* тестов гарантирует, возможно, более полную проверку программы? Исчерпывающая проверка на всем множестве входных данных недостижима. Пример: программа, вычисляющая функцию двух переменных: $Y = f(X, Z)$. Если X, Y, Z — real, то полное число тестов $(2^{32})^2 = 2^{64} \approx 10^{31}$ Если на каждый тест тратить 1 мс, то 2^{64} мс = = 800 млн лет. Следовательно:

- в любой нетривиальной программе на любой стадии ее готовности содержатся необнаруженные ошибки;
- тестирование — технико-экономическая проблема, основанная на компромиссе время — полнота. Поэтому нужно стремиться к возможно меньшему количеству хороших тестов с желательными свойствами.

Детективность: тест должен с большой вероятностью обнаруживать возможные ошибки

Покрывающая способность: один тест должен выявлять как можно больше ошибок.

Воспроизводимость: ошибка должна выявляться независимо от изменяющихся условий (например, от временных соотношений) — это труднодостижимо для времязависимых программ, результаты которых часто невоспроизводимы.

Только на основании выбранного *критерия* можно определить тот момент времени, когда конечное множество тестов окажется достаточным для проверки программы с некоторой полнотой (степень полноты, впрочем, определяется экспериментально). Используется два вида критериев (табл. 5.2):

- *функциональные* тесты составляются исходя из спецификации программы;
- *структурные* тесты составляются исходя из текста программы.

На рис. 5.1, *а* видно отличие тестирования команд (достаточен один тест) от С1 (необходимы два теста как минимум). Рисунок 5.1, *б* иллюстрирует различие С1 (достаточно двух тестов, покрывающих пути 1, 4 или 2, 3) от С2 (необходимо четыре теста для всех четырех путей). С2 недостижим в реальных програм-

Таблица 5.2. **Виды критериев и их функциональность**

Вид критерия	Функциональность тестов
Функциональные	
Тестирование классов входных данных	Содержать представителей всех классов входных или выходных классов и точки на границах классов
Тестирование классов выходных данных	
Тестирование функций	Каждая функция внешнего интерфейса должна быть проверена ≥1 раза
Структурные	
Тестирование команд	Каждая команда (оператор) должна быть выполнена ≥1 раза
Критерий C1 — тестирование ветвей	Каждая ветвь должна быть выполнена ≥1 раза
Критерий C2 — тестирование путей	Каждый путь в графе программы должен быть выполне более 1 раза

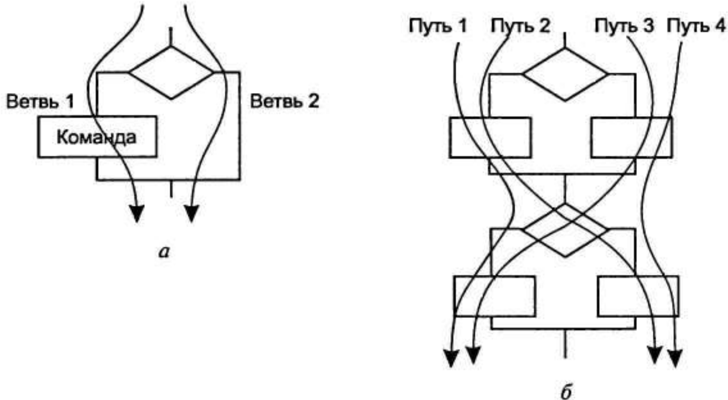


Рис. 5.1. Траектории вычислений при структурном тестировании

мах из-за их цикличности, поэтому ограничиваются тремя путями для каждого цикла: 0, 1 и N повторений цикла.

Остаются проблемы назначения классов входных/выходных данных для функционального тестирования и проектирования тестов для структурного тестирования. Классы, как правило, назначаются исходя из семантики решаемой задачи [6].

Рассмотрим **пример**. Найти минимальный набор тестов для программы нахождения вещественных корней квадратного уравнения $ax^2 + bx + c = 0$.

Решение представлено в табл. 5.3.

Таблица 5.3. **Поиск численного решения минимального набора тестов**

Номер теста	a	b	c	Ожидаемый результат	Что проверяется
1	2	–5	2	$x_1 = 2,\ x_2 = 0{,}5$	Случай вещественных корней
2	3	2	5	Сообщение	Случай комплексных корней
3	3	–12	0	$x_1 = 4,\ x_2 = 0$	Нулевой корень
4	0	0	10	Сообщение	Неразрешимое уравнение
5	0	0	0	Сообщение	Неразрешимое уравнение
6	0	5	17	Сообщение	Неквадратное уравнение (деление на 0)
7	9	0	0	$x_1 = x_2 = 0$	Корень из 0

Таким образом, для этой программы предлагается минимальный набор функциональных тестов, исходя из 7 классов выходных данных.

5.2. Тестирование «белого ящика» и «черного ящика»

В терминологии профессионалов тестирования (программного и некоторого аппаратного обеспечения) фразы тестирование «белого ящика» и тестирование «черного ящика» относятся к тому, имеет ли разработчик тестов доступ к исходному коду тестируемого ПО, или же тестирование выполняется через пользовательский интерфейс либо прикладной программный интерфейс, предоставленный тестируемым модулем.

При тестировании «белого ящика» (англ. *white-box testing*, также говорят — *прозрачного ящика*) разработчик теста имеет доступ к исходному коду и может писать код, который связан с библиотеками тестируемого ПО. Это типично для юнит-тестирования (англ. *unit testing*), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции работоспособны и устойчивы до определенной степени.

При тестировании «черного ящика» (англ. *black-box testing*) тестировщик имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов с уверенностью в том, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.

Если альфа- и бета-тестирование относятся к стадиям до выпуска продукта (а также, неявно, к объему тестирующего сообщества и ограничениям на методы тестирования), тестирование «белого ящика» и «черного ящика» имеет отношение к способам, которыми тестировщик достигает цели.

Бета-тестирование в целом ограничено техникой «черного ящика» (хотя постоянная часть тестировщиков обычно продолжает тестирование «белого ящика» параллельно бета-тестированию). Таким образом, термин бета-тестирование может указывать на состояние программы (ближе к выпуску, чем альфа) или может указывать на некоторую группу тестировщиков и процесс, выполняемый этой группой. Итак, тестировщик может продолжать работу по тестированию «белого ящика», хотя ПО уже «в бете» (стадия), но в этом случае он не является частью бета-тестирования (группы/процесса).

5.3. Порядок разработки тестов

По внешней спецификации разрабатываются тесты [3]:

- для каждого класса входных данных;
- для граничных и особых значений входных данных.

Контролируется, все ли классы выходных данных при этом проверяются, и добавляются при необходимости нужные тесты.

Разрабатываются тесты для тех функций, которые не проверяются в п. 1.

По тексту программы проверяется, все ли условные переходы выполнены в каждом направлении (C1). При необходимости добавляются новые тесты.

Аналогично проверяется, проходятся ли пути для каждого цикла: без выполнения тела, с однократным и максимальным числом повторений.

Готовятся тесты, проверяющие исключительные ситуации, недопустимые входные данные, аварийные ситуации.

Функциональное тестирование дополняется здесь структурным. Классы входных/выходных данных должны быть определены в плане тестирования уже во внешней спецификации. Согласно статистике 1-й и 2-й пункты обеспечивают степень охвата C1 в среднем 40—50 %. Проверка по C1 (пункт 3) обычно выявляет 90 % всех ошибок, найденных при тестировании. (Все программное обеспечение ВВС США принимается с проверкой по C1.)

Систематическое тестирование предполагает также ведение журнала отладки (Bug Book), в котором фиксируется ошибка (описание, дата обнаружения, автор модуля) и в дальнейшем — исправление (дата, автор).

Приведем так называемые **аксиомы тестирования**.

1. Тест должен быть направлен на обнаружение ошибки, а не на подтверждение правильности программы.

2. Автор теста — не автор программы.

3. Тесты разрабатываются одновременно или до разработки программы.

4. Необходимо предсказывать ожидаемые результаты теста до его выполнения и анализировать причины расхождения результатов.

5. Предыдущее тестирование необходимо повторять после каждого внесения исправлений в программу.

6. Следует повторять полное тестирование после внесения изменений в программу или после переноса ее в другую среду.

7. В те программы, в которых обнаружено много ошибок, необходимо дополнить первоначальный набор тестов [6].

5.4. Автоматизация тестирования

А. Автоматизация прогона тестов актуальна для 5-й и 6-й аксиом Майерса. Пишутся командные файлы для запуска программы с каждым тестом из набора и сравнением реального результата с ожидаемым. Существуют специальные средства (на-

пример система MIL-S для PL/1 фирмы IBM). Разрабатывается стандарт IEEE скриптового языка для описания тестовых наборов [3].

Б. Средства автоматизации подготовки тестов и анализа их результатов.

1. Генераторы случайных тестов в заданных областях входных данных.

2. Отладчики (для локализации ошибок).

3. Анализаторы динамики (profilers). Обычно входят в состав отладчиков; применяются для проверки соответствия тестовых наборов структурным критериям тестирования.

4. Средства автоматической генерации структурных тестов методом «символического выполнения» Кинга.

5.5. Модульное тестирование

Модульное тестирование — это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель модульного тестирования заключается в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. Модульное тестирование проводится по принципу «белого ящика», т. е. основывается на знании внутренней структуры программы и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие — для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне модульного тестирования проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т. п., обычно пропускаются на уровне мо-

дульного тестирования и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию модульного тестирования, т. е. расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая база данных (**Repository**) разработок, хранящая конкретные сведения о разработке предыдущих проектов: о версиях и сборках кода (**build**), зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т. п. Проведя анализ характеристик прежних проектов, подобных заказанному разработчику, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, поиск которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап модульного тестирования. Если анализ не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится поиск локальных дефектов, у которых код, ресурсы и информация, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на модульном уровне ошибки, связанные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули (а именно, спецификацию интерфейса), в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Являясь по способу исполнения структурным тестированием или тестированием «белого ящика», модульное тестирование характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

- на основе анализа потока управления. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на основе структурных критериев тестирования C0, C1, C2. К ним относятся вершины, дуги, пути управляющего графа программы (УГП), условия, комбинации условий и т. п.
- на основе анализа потока данных, когда элементы, которые должны быть покрыты, определяются на основе потока данных, т. е. информационного графа программы.

Тестирование на основе потока управления. Особенности использования структурных критериев тестирования С0, С1, С2 были рассмотрены в разд. 5.2. К ним следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевых) условий в программе. Критерии покрытия решений (ветвей — С1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относится критерий покрытия функций программы, согласно которому каждая функция программы должна быть вызвана хотя бы 1 раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы 1 раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (call pair coverage).

Тестирование на основе потока данных. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий потока данных). Как основа для стратегии тестирования поток данных впервые был описан в [14]. Предложенная там стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и определение переменной, на которую указывает ссылка (т. е. требуется покрытие дуг информационного графа программы). Недостаток стратегии в том, что она не включает критерий С1 и не гарантирует покрытия решений.

Стратегия требуемых пар [15] также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий СР, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима дуга w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не должна использоваться. Для покрытия еще одного популярного критерия Cdu достаточно тестировать пары (вершина, дуга), поскольку определение переменной происходит в вершине УГП, а ее использование — на дугах, исходящих из решений, или в вычислительных вершинах.

Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании.

Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

- конструирование УГП;
- выбор тестовых путей;
- генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования множества элементов, которые необходимо покрыть тестами.

На третьей фазе по известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей.

Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

- статические методы;
- динамические методы;
- методы реализуемых путей.

Статические методы. Самое простое и легко реализуемое решение — построение каждого пути посредством постепенного его удлинения за счет добавления дуг, пока не будет достигнута выходная вершина управляющего графа программы. Эта идея может быть усилена в так называемых адаптивных методах, которые каждый раз добавляют только один тестовый путь (входной тест), используя предыдущие пути (тесты) как руководство для выбора последующих путей в соответствии с некоторой стратегией. Чаще всего адаптивные стратегии применяются по отношению к критерию С1. Основной недостаток статических методов заключается в том, что не учитывается возможная нереализуемость построенных путей тестирования.

Динамические методы. Такие методы предполагают построение полной системы тестов, удовлетворяющих заданному критерию, путем одновременного решения задачи построения покрывающего множества путей и тестовых данных. При этом можно автоматически учитывать реализуемость или нереализуемость ранее рассмотренных путей или их частей. Основной идеей динамических методов является подсоединение к начальным реализуемым отрезкам путей дальнейших их частей так, чтобы: 1) не терять при этом реализуемости вновь полученных путей; 2) покрыть требуемые элементы структуры программы.

Методы реализуемых путей. Данная методика [16] заключается в выделении из множества путей подмножества всех реализуе-

— 7888

мых путей. После этого покрывающее множество путей строится из полученного подмножества реализуемых путей.

Достоинство статических методов состоит в сравнительно небольшом количестве необходимых ресурсов как при использовании, так и при разработке. Однако их реализация может содержать непредсказуемый процент брака (нереализуемых путей). Кроме того, в этих системах переход от покрывающего множества путей к полной системе тестов пользователь должен осуществить вручную, а эта работа достаточно трудоемкая. Динамические методы требуют значительно больших ресурсов как при разработке, так и при эксплуатации, однако увеличение затрат происходит в основном за счет разработки и эксплуатации аппарата определения реализуемости пути (символический интерпретатор, решатель неравенств). Достоинство этих методов заключается в том, что их продукция имеет некоторый качественный уровень — реализуемость путей. Методы реализуемых путей дают самый лучший результат [33].

5.6. Интеграционное тестирование

Интеграционное тестирование — это тестирование части системы, состоящей из двух и более модулей. Основная задача интеграционного тестирования — поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (**Stub**) на месте отсутствующих модулей. Основная разница между модульным и интеграционным тестированием состоит в целях, т. е. в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию выбора входных данных и методов анализа. В частности, на уровне интеграционного тестирования часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

На рис. 5.2 приведена структура комплекса программ K, состоящего из оттестированных на этапе модульного тестирования модулей M_1, M_2, M_{11}, M_{12}, M_{21}, M_{22}. Задача, решаемая методом интеграционного тестирования, — тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K. Интеграционное тестирование использует модель «белого ящика» на модульном уровне. Поскольку тестировщику текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправданно.

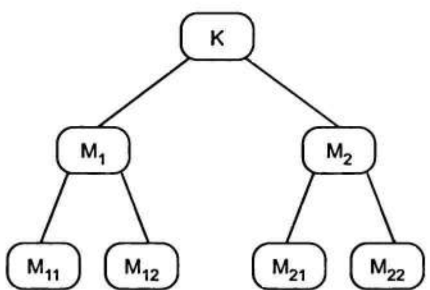


Рис. 5.2. Пример структуры комплекса программ

Интеграционное тестирование применяется на этапе сборки модульно оттестированных модулей в единый комплекс. Известны два метода сборки модулей:

- **монолитный**, характеризующийся одновременным объединением всех модулей в тестируемый комплекс;
- **инкрементальный**, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с **пошаговым тестированием** собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:
  - «сверху вниз» и соответствующее ему восходящее тестирование;
  - «снизу вверх» и соответственно нисходящее тестирование.

Особенности монолитного тестирования заключаются в следующем: для замены не разработанных к моменту тестирования модулей, кроме самого верхнего (K на рис. 5.2), необходимо дополнительно разрабатывать **драйверы (test driver)** и/или **заглушки**

(stub) [9], замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение монолитного и интегрального подходов дает следующие результаты.

Монолитное тестирование требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.

Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.

Монолитное тестирование предоставляет большие возможности распараллеливания работ, особенно на начальной фазе тестирования.

Особенности нисходящего тестирования заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих операции обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Например, порядок тестирования комплекса K (см. рис. 5.2) при нисходящем тестировании может быть таким, как показано в примере 5.3, где тестовый набор, разработанный для модуля Mi, обозначен как XYi = (X, Y)i.

1) $K \rightarrow XY_K$

2) $M_1 \rightarrow XY_1$

3) $M_{11} \rightarrow XY_{11}$

4) $M_2 \rightarrow XY_2$

5) $M_{22} \rightarrow XY_{22}$

6) $M_{21} \rightarrow XY_{21}$

7) $M_{12} \rightarrow XY_{12}$

Пример 5.1. Возможный порядок тестов при нисходящем тестировании (html, txt).

Недостатки нисходящего тестирования:

- проблема разработки достаточно «интеллектуальных» заглушек, т. е. заглушек, способных к использованию при

моделировании различных режимов работы комплекса, необходимых для тестирования;
- сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности;
- параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней.

Особенности восходящего тестирования в организации порядка сборки и перехода к тестированию модулей, соответствующему порядку их реализации.

Например, порядок тестирования комплекса K (см. рис. 5.2) при восходящем тестировании может быть следующим (см. пример 5.4).

1) $M_{11} \to XY_{11}$
2) $M_{12} \to XY_{12}$
3) $M_1 \to XY_1$
4) $M_{21} \to XY_{21}$
5) $M_2(M_{21}, Stub(M_{22})) \to XY_2$
6) $K(M_1, M_2(M_{21}, Stub(M_{22})) \to XY_K$
7) $M_{22} \to XY_{22}$
8) $M_2 \to XY_2$
9) $K \to XY_K$

Пример 5.2. Возможный порядок тестов при восходящем тестировании.

Недостатки восходящего тестирования:
- запаздывание проверки концептуальных особенностей тестируемого комплекса;
- необходимость в разработке и использовании драйверов [33].

5.7. Системное тестирование

Системное тестирование качественно отличается от интеграционного и модульного уровней. Системное тестирование рассматривает тестируемую систему в целом и оперирует на уровне

пользовательских интерфейсов, в отличие от последних фаз интеграционного тестирования, которое оперирует на уровне интерфейсов модулей. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы или отслеживать правильность работы конкретных функций. Основная задача системного тестирования — выявления дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в применении и т. п.

Системное тестирование производится над проектом в целом с помощью метода «черного ящика». Структура программы не имеет никакого значения, для проверки доступны только входы и выходы, видимые пользователю. Тестированию подлежат коды и пользовательская документация.

Категории тестов системного тестирования:

1. Полнота решения функциональных задач.
2. Стрессовое тестирование — на предельных объемах нагрузки входного потока.
3. Корректность использования ресурсов (утечка памяти, возврат ресурсов).
4. Оценка производительности.
5. Эффективность защиты от искажения данных и некорректных действий.
6. Проверка инсталляции и конфигурации на разных платформах.
7. Корректность документации.

Поскольку системное тестирование проводится на пользовательских интерфейсах, создается иллюзия того, что построение специальной системы автоматизации тестирования не всегда необходимо. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная автоматизация тестирования, что приводит к созданию тестовой системы, гораздо более сложной, чем система тестирования, применяемая на уровне тестирования модулей или их комбинаций.

5.8. Эффективность и оптимизация программ

Эффективными считаются программы, требующие минимального времени выполнения и/или минимального объема оперативной памяти. Особые требования к эффективности программного обеспечения предъявляют при наличии ограничений (на время реакции системы, на объем оперативной памяти и т. п.). В случаях, когда обеспечение эффективности не требует серьезных временных и трудовых затрат, а также не приводит к существенному ухудшению технологических свойств, необходимо это требование иметь в виду [7].

Разумный подход к обеспечению эффективности разрабатываемого программного обеспечения состоит в том, чтобы в первую очередь оптимизировать те фрагменты программы, которые существенно влияют на характеристики эффективности. Для уменьшения времени выполнения некоторой программы в первую очередь следует проанализировать циклические фрагменты с большим количеством повторений: экономия времени выполнения одной итерации цикла будет умножена на количество итераций.

Не следует забывать и о том, что многие способы снижения временных затрат приводят к увеличению емкостных и, наоборот, уменьшение объема памяти может потребовать дополнительного времени на обработку.

И тем более не следует «платить» за увеличение эффективности снижением технологичности разрабатываемого программного обеспечения. Исключения возможны лишь при очень жестких требованиях и наличии соответствующего контроля за качеством.

Частично проблему эффективности программ решают за программиста компиляторы.

Средства оптимизации, используемые компиляторами, делят на две группы:

- *машинно-зависимые*, т. е. ориентированные на конкретный машинный язык, выполняют оптимизацию кодов на уровне машинных команд, например, исключение лишних пересылок, использование более эффективных команд и т. п.
- *машинно-независимые* выполняют оптимизацию на уровне входного языка, например, вынесение вычислений константных (независящих от индекса цикла) выражений из циклов и т. п.

Естественно, нельзя вмешаться в работу компилятора, но существует много возможностей оптимизации программы на уровне команд.

Способы экономии памяти. Принятие мер по экономии памяти предполагает, что в каких-то случаях эта память неэкономно использовалась. Учитывая, что анализировать имеет смысл только операции размещения данных, существенно влияющие на характеристику эффективности, следует обращать особое внимание на выделение памяти под данные структурных типов (массивов, записей, объектов и т. п.).

Прежде всего при наличии ограничений на использование памяти следует выбирать алгоритмы обработки, не требующие дублирования исходных данных структурных типов в *процессе обработки*. Примером могут служить алгоритмы сортировки массивов, выполняющие операцию в заданном массиве, например хорошо известная сортировка методом «пузырька».

Если в программе необходимы большие массивы, используемые ограниченное время, то их можно размещать в динамической памяти и удалять при завершении обработки.

Также следует помнить, что при передаче структурных данных в подпрограмму «по значению» копии этих данных размещаются в стеке. Избежать копирования иногда удается, если передавать данные «по ссылке», но как неизменяемые (описанные const). В последнем случае в стеке размещается только адрес данных, например:

```
Type Mas.4iv   array [I.  100] of real;
function Summa (Const a:Massiv;   .)
```

Способы уменьшения времени выполнения. Как уже упоминалось выше, для уменьшения времени выполнения в первую очередь необходимо анализировать циклические участки программы с большим количеством повторений. При их написании необходимо по возможности:

- выносить вычисление константных, т. е. не зависящих от параметров цикла, выражений из циклов;
- избегать «длинных» операций умножения и деления, заменяя их сложением, вычитанием и сдвигами;
- минимизировать преобразования типов в выражениях;
- оптимизировать запись условных выражений — исключать лишние проверки;

- исключать многократные обращения к элементам массивов по индексам (особенно многомерных, так как при вычислении адреса элемента используются операции умножения на значение индексов), первый раз прочитав из памяти элемент массива, следует запомнить его в скалярной переменной и использовать в нужных местах;
- избегать использования различных типов в выражении и т. п.

Рассмотрим следующие примеры.

Пример 5.3. Пусть имеется цикл следующей структуры (Pascal):

```
for y:   0 to 99 do
for x:   0 to 99 do
a [320*x+y]    S [k,l];
```

В этом цикле операции умножения и обращения к элементу S[k] выполняются 10 000 раз.

Оптимизируем цикл, используя, что 320 = 28 + 26:

```
skl: =S [k,l]; {выносим обращение к элементу массива
   из цикла}
for x:   0 to 99 do (меняем циклы местами}
begin
    x shl 8 + x shl 6; {умножение заменяем на сдвиги
   и выносим из цикла}
for y;   0 to 99 do
a [i+y]  =skl;
end;
```

В результате вместо 10 000 операций умножения будут выполняться 200 операций сдвига, а их время приблизительно сравнимо со временем выполнения операции сложения. Обращение к элементу массива S[k] будет выполнено 1 раз.

Пример 5.4. Пусть имеется цикл, в теле которого реализовано сложное условие:

```
for k:   2 to n do
begin
ifx[k] > yk then S:   S+y[k]-x [k];
if (x [k]< = yk) and (y[k]<yk) then S: = S+yk-x[k];
end;...
```

В этом цикле можно убрать лишние проверки:

```
for k: =2 to n do
begin
ifx [k]>yk then S:=S+y[k]-x[k]
else
ify[k]<yk then S: =S+yk-x [k];
end;
```

Обратите внимание на то, что в примере 5.3 понять, что делает программа, стало сложнее, а в примере 5.4 — практически нет. Следовательно, оптимизация, выполненная в первом случае, может ухудшить технологичность программы, а потому не очень желательна.

5.9. Стиль программирования

Стиль программирования, под которым понимают стиль оформления программ и их «структурность», также существенно влияет на читаемость программного кода и количество ошибок программирования.

5.9.1. Основные принципы форматирования

Основная теорема форматирования гласит, что хорошее визуальное форматирование показывает логическую структуру программы. Создание красивого кода — хорошо, а демонстрация структуры кода — лучше. Если одна методика лучше показывает структуру кода, а другая выглядит красивей, следует использовать ту, которая лучше демонстрирует структуру.

Форматирование — это ключ к структуре программы. Компьютеру важна исключительно информация о скобках или операторах begin и end, а читатель-человек склонен делать выводы из визуального представления кода. Взгляните на фрагмент кода, приведенный ниже, схема отступов в котором заставляет человека думать, что все три выражения выполняются при каждом проходе цикла.

```
// меняем местами правые и левые элементы во всем массиве
for ( i = 0; i < MAX ELEMENTS; i++ )
```

```
leftElement   left[
left[   ]   right[
right[       leftElement;
```

Если код не содержит обрамляющих скобок, компилятор будет выполнять первое выражение MAXELEMENTS раз, а второе и третье — по одному разу. Отступы делают очевидным, что автор кода хотел выполнять все три выражения и собирался поместить скобки вокруг них. Компилятору это неочевидно [35].

Цели хорошего форматирования

Многие решения о том, как должно выглядеть хорошее форматирование, представляют собой субъективные эстетические оценки; часто можно достичь одной и той же цели по-разному. Говоря объективно, хорошая схема форматирования должна делать следующее:

- точно представлять логическую структуру кода. Для демонстрации логической структуры программисты обычно применяют отступы и другие неотображаемые символы;
- единообразно показывать логическую структуру кода. Некоторые стили форматирования состоят из правил с таким количеством исключений, что последовательно их соблюдать практически невозможно. Действительно хороший стиль подходит в большинстве случаев;
- улучшать читабельность. Стратегия использования отступов, соответствующая логике, но усложняющая процесс чтения кода, бесполезна. Схема форматирования, использующая пробелы и разделители только там, где они требуются компилятору, логична, но читать такой код невозможно. Хорошая структура форматирования упрощает чтение кода;
- выдерживать процедуру исправления. Лучшие схемы форматирования хорошо переносят модификацию кода. Исправление одной строки не должно приводить к изменению нескольких других.

В дополнение к этим критериям иногда во внимание принимается и задача минимизации количества строк кода, необходимых для реализации простого выражения или блока.

5.9.2. Способы форматирования

Можно получить хороший формат кода, по-разному используя несколько инструментов для форматирования.

Неотображаемые символы

Используйте неотображаемые символы для улучшения читаемости. Неотображаемые символы, к которым относятся пробелы, знаки табуляции, переводы строк и пустые строки, — это основное средство для демонстрации структуры программы.

Вам вряд ли придет в голову писать книгу без пробелов между словами, разбиения на абзацы и деления на главы. Может, такую книгу и можно прочитать от начала до конца, но практически невозможно просматривать ее в поисках какой-то мысли или важного момента. Еще хуже, что такой формат книги не позволит показать читателю, как автор намеревался организовать информацию. Структура, предлагаемая автором, дает подсказку о логической организации темы. Разбиение книги на главы, абзацы и предложения показывает читателю, как следует мысленно организовывать тему. Если эта организация неочевидна, читателю приходится самому ее домысливать, что налагает на него более тяжкое бремя и увеличивает вероятность никогда не узнать, как на самом деле организована данная тема.

Информация, содержащаяся в программе, сосредоточена еще плотней, чем информация в большинстве книг. Если страницу книги можно прочесть и понять за 1 или 2 минуты, то большинство программистов не могут читать и понимать листинг программы со скоростью, даже приблизительно сравнимой с этой. Программа должна давать гораздо больше подсказок о своей организации, чем книга.

Группировка взаимосвязанных выражений — еще один способ применения неотображаемых символов [35].

В литературе мысли группируются в абзацы. Хорошо написанный абзац содержит предложения, относящиеся только к определенной идее. Он не должен содержать посторонних предложений. Точно так же абзац кода должен содержать только взаимосвязанные операторы, выполняющие одно задание.

Пустые строки. Кроме необходимости группировать взаимосвязанные операторы, очень важно отделять несвязанные выражения друг от друга. Начало нового абзаца в книге обозначается

отступом или пустой строкой. Начало нового абзаца в коде нужно указывать с помощью пустой строки.

Пустые строки позволяют продемонстрировать организацию программы. Вы можете использовать их для деления групп взаимосвязанных операторов на абзацы, отделения методов друг от друга и выделения комментариев.

Хотя эту статистику тяжело применить на практике, но одно исследование показало, что оптимальное число пустых строк в программе составляет от 8 до 16 %. Если оно больше 16 %, то время, затрачиваемое на отладку, заметно увеличивается.

Отступы. Применяйте отступы для демонстрации логической структуры программы. Как правило, операторы выделяются отступами, когда они следуют после некоторого выражения, от которого они логически зависят. Оптимальными являются отступы из 2—4 пробелов [35].

Скобки

Используйте скобки чаще, чем вам это кажется необходимым. Применяйте скобки для разъяснения выражений, состоящих из двух и более членов. Возможно, в скобках нет нужды, но они добавляют ясности и ничего вам не стоят. Например, скажите, как вычисляется следующее выражение?

Вариант на C++: 12 + 4% 3 * 7 / 8.

Пришлось ли вам задуматься о том, как эти выражения вычисляются, вот в чем вопрос? Можете ли вы быть уверенными в своем ответе без обращения к справочной информации? Даже опытные программисты не отвечают с полной уверенностью, и именно поэтому следует использовать скобки, если есть хоть малейшее сомнение в том, как вычисляется выражение.

5.10. Надежность программного обеспечения

Это комплексное свойство включает две составляющие [3]:

- безошибочность (correctness) — соответствие спецификации;
- устойчивость (robustness) или отказоустойчивость (fault-tolerance) — способность продолжать правильно работать после отказов.

Улучшение первого качества достигается хорошей технологией, предупреждающей ошибки (fault-avoidance). Однако 100%-ное отсутствие ошибок недостижимо. Устойчивость должна быть относительно любых видов отказов, для ее поддержания создаются специальные программно-аппаратные средства.

Отказ (failure) по ГОСТу — нарушение работоспособности изделия и его соответствия требованиям технической документации (рис. 5.3). Применительно к программам (стандарт IEEE/ANSI) отказ есть неспособность функциональной единицы системы, зависящей от программы, выполнять требуемую функцию в заданных пределах.

Рис. 5.3. Диаграмма состояний и переходов при отказе

Классификация причин отказов:

1. Физические дефекты:
   - внутренние (старение, износ);
   - внешние воздействия (радиация, высокая температура).
2. Ошибки людей:
   - ошибки эксплуатации или взаимодействия;
   - проектные ошибки.

Повреждение (Fault) — неисправность уже появилась, но еще не проявилась вовне. Например, отказала ячейка памяти, но из нее еще не читали или в нее еще не писали. Отказ компонента — это повреждение системы. Время нахождения в неисправном, но работоспособном состоянии называется латентным (скрытым) периодом отказа.

Восстановление (Recovery) — возврат в исправное состояние путем:

а) ручного ремонта, замены, исправления;

б) автоматически — задействованием резервов;

в) самопроизвольно, обычно быстро.

В случае в) отказ называется ***сбоем***, т. е. сбой — это кратковременный самовосстанавливающийся отказ. Остальные отказы называются устойчивыми (по умолчанию отказ устойчивый). В электронной аппаратуре сбои происходят на порядок чаще устойчивых. Их причины — флюктуации питания, ситуации «гонок» сигналов, альфа-частицы и др. В программах аналогично сбоям ведут себя времязависимые ошибки — их иногда называют «мерцающими» (blinking bugs).

Типичный пример восстановления — с помощью резервной (backup) копии данных. Если выполнить восстановление в латентном периоде отказа, то он никогда не проявится вовне — в этом состоит идеальная отказоустойчивость.

5.10.1. Количественные характеристики надежности программ

Надежность нужно оценивать, измерять, предсказывать — обеспечивать заданные требования к надежности в проекте и проверять их выполнение в продукте [3].

«Внутренняя» характеристика надежности — количество оставшихся ошибок в программе — интересна больше разработчикам, чем потребителям. Для последних важны характеристики, традиционные для теории надежности, основанные на предположении о стохастическом (случайном во времени) процессе возникновения отказов: *среднее время безотказной работы* (MTBF — Mean Time Between Failures) и *коэффициент готовности*. Третья характеристика, взаимосвязанная с первой, — *интенсивность отказов* — среднее их количество в единицу времени.

В предположении *простейшего потока* отказов (отказы независимы, редки и их вероятность неизменна во времени) $P(t)$ — вероятность безотказной работы за время t — подчиняется закону Пуассона (экспоненциальному распределению вероятностей):

$$P(t) = e^{-\lambda t},$$

где λ — интенсивность отказов (обычно в 1/час).

Его первый момент — математическое ожидание $M(P)$ и есть $MTBF = 1/\lambda$.

В табл. 5.4 приведены средние значения MTBF для устойчивых отказов.

Таблица 5.4. **Средние значения MTBF для устойчивых отказов**

Вид компонента	Часы	Годы
Электромеханическая аппаратура	10^2—10^3	10^{-1}
Электронная аппаратура	10^3—10^4	1
Большие интегральные схемы	10^6—10^8	10^2—10^4
Обычные программы	10^1—10^3	10^{-3} —10^{-1}

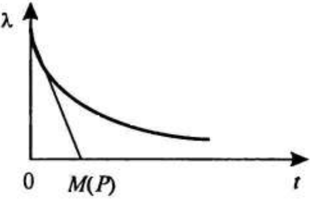


Рис. 5.4. Зависимость потока отказов от времени

Таким образом, программы вносят наибольший вклад в (не)надежность современных вычислительных систем. Между тем существуют столь ответственные (mission-critical) приложения, где требуется очень малая вероятность отказов. Например, для бортовой системы управления космическим зондом требуется $\lambda = 10^{-9}$, чтобы вероятность устойчивого отказа в первые 10 лет работы была не более 10^{-4} (или вероятность безотказной работы 0,9999), что означает MTBF = 100 тысяч лет!

Вообще говоря, λ не постоянна во времени. Для аппаратуры характерна зависимость, представленная на рис. 5.5.

Многие программные продукты (ПП) имеют аналогичный характер изменения надежности: А — период начальной экс-

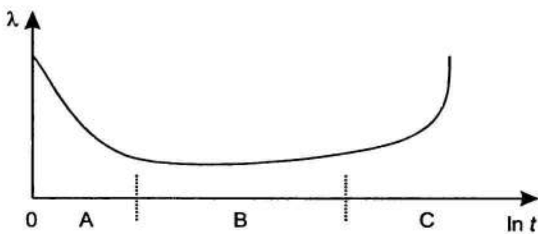


Рис. 5.5. Типичное изменение λ электронной аппаратуры во времени: А — период приработки («выжигание» дефектов); В — полезная жизнь; С — старение, износ

плуатации (расширенного бета-тестирования), С — накопление ошибок из-за модификаций.

Если отказ все же произошел, время восстановления должно быть минимальным. Это характеризуется показателем *ремонтопригодности* коэффициента готовности (availability):

$$k = (T - T_{пр})/T,$$

где T — общее время работы, $T_{пр}$ — время простоя из-за восстановлений. В ответственных системах требуется, чтобы значение k почти не отличалось от 1: для цифровых АТС — 2 часа простоя суммарно за 15 лет; для системы управления воздушным движением — 3 сек за год!

5.10.2. Методы оценки и измерения характеристик надежности

«Внутренняя» характеристика — число оставшихся ошибок — абсолютное или относительное. Считается, что в хорошо отлаженной программе — не более одной ошибки на 1 тыс. строк исходного кода. Для ответственных применений требуется то же на 10 тыс. строк и больше. Например, для СОИ этот показатель был задан как не более 2,5 ошибки на 10^5 операторов языка Ада.

Как проверить соответствие ПП таким требованиям?

1. Метод Миллса (IBM, 1972 г.) использует прием биологов, метод «меченых рыб». Группа тестирования засоряет программу искусственно ошибками и, продолжая тестирование, анализирует долю внесенных ошибок среди обнаруженных.

2. Пусть внесено s ошибок, обнаружено n собственных и v внесенных ошибок. Предполагая, что вероятность обнаружения внесенных и собственных ошибок одинакова, получим первоначальное количество оставшихся ошибок $N = sn/v$. Проверка гипотезы: осталось меньше k собственных ошибок. Вносится еще s ошибок и тестирование продолжается, пока все они не обнаружены. Пусть к этому моменту найдено n собственных. Тогда уровень значимости гипотезы (т. е. вероятность того, что она истинна, равна 1 при $n > k$ и $s/(s + k + 1)$ при $n \le k$.

3. В методе Руднера (1977 г.) тестирование осуществляется двумя независимыми группами тестеров параллельно, которые

обнаруживают m_1 и m_2 ошибок соответственно, из которых $m_{1,2}$ ошибок — общие. Используя гипергеометрическое распределение, методом максимального правдоподобия может быть найдена оценка числа первоначально содержавшихся в программе ошибок $N = m_1 m_2 / m_{1,2}$.

4. Фирма Motorola использует в качестве меры надежности своих ПП (управляющих программ реального времени) среднее число ошибок на 1000 строк исходного кода. Она разработала метод оценки времени тестирования без отказов (zero-failure method), подтверждающего заданную надежность. Метод основан на известной форме зависимости:

$$T = f(N, n, t),$$

где T — время окончательного тестирования — испытаний без отказов; N — допустимое число ошибок в коде; n — общее число ошибок, обнаруженных в ходе тестирования; t — время, потраченное на выявление всех n ошибок.

Если за время T ни одного отказа не произошло, считается, что в программе не более N ошибок и тестирование можно закончить. Если же отказ произошел через промежуток времени $t_r < T$, то испытание продолжается, причем функция T пересчитывается заново для $n + 1$ и $t + t_r$.

Функция T получена на базе одной из моделей надежности программ на основе оценок функции $\lambda(t)$, экстраполирующих динамику отладки (для этого и ведутся журналы отладки). Например, *модель Джелинского — Моранды* (1972 г.), использовавшаяся в ряде ответственных проектов, в том числе Apollo, основана на следующих допущениях:

1. Интенсивность обнаружения ошибок $R(t)$ пропорциональна текущему числу ошибок в программе, т. е. числу исходных ошибок минус обнаруженные.

2. Все ошибки равновероятны и их проявление не зависит друг от друга.

3. Ошибки постоянно исправляются без внесения новых.

4. $R(t)$ постоянна в промежутке между двумя смежными моментами обнаружения t_i и t_{i+1}.

Тогда $R(t)$ — ступенчатая монотонно убывающая функция вида (рис. 5.6).

Предполагается, что динамика проявления ошибок во время эксплуатации остается такой же, как при тестировании, т. е. по-

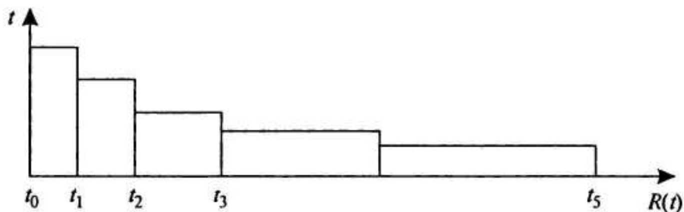


Рис. 5.6. Вид ступенчатой монотонно убывающей функции

стулируется тождество функций $R(t)$ и $\lambda(t)$ с тем отличием, что значение $\lambda(t)$ «замораживается» в момент завершения тестирования — прекращается исправление ошибок.

Если i — число ошибок, обнаруженных к моменту времени t, то до следующего обнаружения справедливо $R(t) = K(B - i)$, где B — неизвестное число исходных ошибок; K — неизвестный коэффициент пропорциональности. Полагая $X_i = t_{i+1} - t_i$ ($i = 0$, $n - 1$), можно утверждать, что X_i имеют экспоненциальное распределение: $P(X_i) = \exp(-K(B - i)X_i)$. Имея набор X_i из журнала отладки, можно решать задачу нахождения значений K и B, отвечающих принципу максимального правдоподобия. Однако проще решение для модифицированной модели, где поскольку $dN(t)/dt = R(t)$ (число ошибок, обнаруженных в единицу времени), получаем дифференциальное уравнение с начальными условиями:

$$\frac{dR(t)}{dt} = -K\frac{dN(t)}{dt}; \quad \frac{dR(t)}{dt} + KR(t) = 0; \quad N(0) = 0; \quad R(0) = KB.$$

Здесь снято допущение 4, и функция $R(t)$ перестает быть кусочно-постоянной: $R(t) = K(B - N(t))$.

Его решение: $R(t) = KBe^{-Kt}$ Обозначая $a = \ln(KB)$, $b = -K$, получаем запись решения в виде: $R(t) = \exp(a + bt)$. Логарифмируя обе части этого равенства и переходя к дискретному времени t_i, получаем систему уравнений

$$\ln R(t_i) = a + bt_i; \quad i = 1, \ldots, n.$$

Решая эту систему относительно двух неизвестных K и B методом наименьших квадратов, можно найти оценки величин K и B в соответствии с критерием максимального правдоподобия. Этот метод был применен для ряда конкретных проектов IBM и показал хорошее соответствие реальности.

5.10.3. Преимущества парного программирования

1. Парное программирование экономически оправданно. При использовании в организации XP штат программистов должен возрасти вдвое. Так как квалифицированные кадры являются достаточно дорогими, то, казалось бы, расходы на создание программного обеспечения должны значительно возрасти. И это действительно так — затраты возрастают на 15 %, но в то же время, как доказывают исследования, при парном программировании количество ошибок на 15 % меньше, чем при одиночном. Если учесть стоимость исправления ошибок, обнаруженных на разных стадиях жизненного цикла продукта, включая внедрение, то сэкономленные на этом деньги значительно превосходят затраты на увеличение штата сотрудников.

2. Быстрая обучаемость. При парном программировании специалисты перенимают приемы работы друг друга, значительно расширяют свои знания и передают свой опыт.

3. Работая вместе, люди чувствуют себя уверенней, так как каждый из них знает, что партнер контролирует все его действия. При этом увеличивается качество его работы.

4. В случае выбытия из проекта одного разработчика работавший с ним в паре сможет сразу же продолжить его деятельность, что приведет к экономии дорогого времени и сил второго разработчика.

5. Работа в паре улучшает коммуникабельность в коллективе, помогает наладить понимание и взаимоотношения, что является крайне важным фактором с точки зрения психологии [14].

5.11. Отладка программ

Локализация и исправление ошибок называется **отладкой** [3]. Отладка — это процесс обнаружения причин возникновения ошибки и ее последующего исправления (в отличие от тестирования, являющегося процессом обнаружения самого факта существования ошибки).

Отладка включает в себя элементы тестирования и разработки. На некоторых проектах отладка может занимать до 50 % всего времени разработки. Для многих программистов отладка — это самая трудная часть программирования. При соответствующем подходе количество ошибок, требующих отладки, должно

сократиться, и отладка должна стать самой легкой частью; при таком подходе все ошибки сводятся к небольшим недосмотрам или опечаткам.

Как и тестирование, отладка не является способом улучшения качества ПО. Отладка — это всего лишь способ исправления дефектов в программе. Качество программ должно обеспечиваться аккуратным анализом требований или прототипированием, грамотным проектированием и использованием лучших практик кодирования.

Ошибки синтаксиса языка

Как правило, в абсолютном большинстве случаев ловятся на стадии компиляции программы, или же, если вы работаете с интерпретируемым языком типа Perl или PHP, то при первом интерпретировании программы. Но есть один существенный момент — когда выражение допустимо, но зависит от конкретного компилятора или интерпретатора. Например, в языке Си вполне допустимыми по синтаксису, но не по смыслу, являются выражения: `s[i++]=i; printf("%d %d", i++, i++);`. Результат этих строк не определен, так как неизвестно, в каком порядке будет инкрементироваться и вычисляться значение переменной *i*. Переменная не может более 1 раза присутствовать в выражении, если ее значение изменяется в ходе вычисления этого выражения.

Ошибки во время выполнения

К ошибкам во время выполнения программы относятся ошибки в логике и алгоритмах программы. Ошибки в логике программы связаны с неправильной записью алгоритма, например, вместо && вы написали || или поставили точку с запятой там, где ее не должно быть. К этому же классу ошибок относится неправильное обращение с памятью. В языке Си неправильное обращение с памятью является наиболее распространенной ошибкой программистов, причем не только новички, но и профессионалы нередко допускают ошибки подобного типа. Для языка Си — это один из самых больных вопросов и больших недостатков. Ошибки в алгоритме связаны с неправильным проектированием программы или же с изначально неправильно заложенной бизнес-логикой и функциональными требованиями. Пользователь ждет от программы то, что изначально от нее не требовалось. Ошибки во время выполнения — наиболее часто

встречаемый тип ошибок, и, как правило, устранение таких ошибок не представляется чересчур сложной задачей.

Ошибки логики взаимосвязанных CGI-программ

Ошибки данного типа лежат во взаимосвязанных CGI-программах. Рассмотрим в качестве примера тестовую систему (см. сайт http://test.itsoft.ru). При сдаче теста в цикле работают два скрипта. Первый показывает вопрос, а второй проверяет правильность ответа. Если в тесте 10 вопросов, то эти CGI-скрипты вызываются парно в цикле 10 раз. Но что будет, если пользователь нажмет кнопку «Обновить» в броузере? Скрипт, который показывает вопрос, вызовется повторно. Что будет при разрыве модемного соединения? Отладка в таких системах значительно сложнее, так как вам придется наблюдать за выполнением ряда взаимосвязанных скриптов.

Ошибки многопользовательского доступа

Ошибки многопользовательских систем связаны с неправильным разграничением доступа к совместным ресурсам. Помимо файлов данных, записей в таблицах баз данных, общим ресурсов является генератор случайных чисел. На эти грабли нам пришлось наступить. Обязательно генерируйте случайные числа не только на основе времени, но и на основе уникального идентификатора процесса. В противном случае при выполнении в один и тот же момент времени двух копий одной CGI-программы вы получите одинаковые результаты для двух пользователей. Ошибки многопользовательского доступа сложны тем, что могут не проявлять себя очень долго, до тех пор, пока в один и тот же момент времени системой ни будет запущено несколько копий одной CGI-программы.

Невоспроизводимые ошибки

Невоспроизводимые ошибки представляют собой наиболее сложный тип ошибок. Например, в високосном году 29 февраля ваша система вдруг начала давать сбои, которые сами собой исчезают в невисокосном году. Но бывают ошибки, которые мистическим образом появляются и исчезают. В той же тестовой системе была непонятная ошибка, которая проявлялась 1 раз на несколько сот случаев. Непонятным образом некоторые студен-

ты после сдачи теста получали не результаты, а сбой системы. На исправление этой ошибки ушло два рабочих дня. Оказалось, что проблема в скрипте на JavaScript, который отправлял данные HTML-формы на сервер после истечения допустимого времени ответа на вопрос. Проблема в том, что если время подходило к концу и пользователь нажимал кнопку «Ответить», а в это же время уже начала работать функция JavaScript form.submit(), то отправка данных HTML-формы происходила дважды, т. е. скрипт проверки правильности ответа вызывался 2 раза. А это за собой тянуло ошибку во взаимосвязанных CGI-скриптах, и внешнее проявление сбоя системы мы наблюдали уже при подсчете результатов, а не непосредственно сразу после двойной отправки HTML-формы. Сам код JavaScript был написан верно, и с теоретической точки зрения даже если пользователь нажимает кнопку «Отправить» в последнюю секунду, HTML-форма должна была отправляться только 1 раз. Но на практике все оказалось совсем по-другому. На самом деле ничего мистического нет, или, как говорится, чудес на свете не бывает. Просто невозможно воспроизвести условия, в которых наблюдалась невоспроизводимая ошибка. Надо искать в программе случайности: одновременный доступ к одному ресурсу, генератор случайных чисел, неинициализированные переменные, некорректная работа с памятью или преобразование типов, которые могут проявлять себя не каждый раз.

Ошибки инструментария и других компонентов системы

Ошибки самого компилятора или интерпретатора очень редки, но и такие бывают.

После классификации ошибок давайте рассмотрим методы их поиска. В первую очередь, один простой и, казалось бы, очевидный совет: «надо думать, анализировать, почему программа не работает так, как было задумано, что надо в ней исправить». Никакой самый навороченный отладчик за вас ошибку не найдет. Никакая самая лучшая методика не найдет и не ускорит поиск ошибки, если вы не «включите» мозги по полной программе и не сосредоточитесь целиком и полностью на поимке ошибки. Итак, допустим, вами, пользователем или тестирующим, было зафиксировано некорректное поведение программы. Что делать? Ниже перечислены методы в порядке их приоритетности, которые используют многие программисты.

Локализация ошибки. Программа — это черный ящик, который что-то принимает на вход и выдает что-то на выходе. Первое, что вам необходимо сделать, — понять, где именно в этом черном ящике происходит сбой. Для этого уменьшайте количество параметров, передаваемых на вход программы. Добейтесь двух малоразличимых наборов входных параметров, чтобы при одном из них программа работала, а при другом нет.

Отключение ненужных модулей программы. Закомментируйте все лишнее. Тем самым вы упростите вашу программу. Имеет смысл комментировать отдельные куски программы до тех пор, пока ошибка не исчезнет.

Использование отладчика. Возможности современных отладчиков перечислены ниже:

- точки останова на конкретных строчках кода;
- остановка на *n*-й итерации цикла;
- остановка при изменении переменных;
- остановка при присваивании конкретного значения;
- прохождение кода строчка за строчкой;
- откат по программе (далеко не все);
- исследование всех данных в программе, включая типы, определенные пользователем;
- присваивание новых значений переменным;
- продолжение исполнения программы;
- многоязыковая отладка (язык1, язык2, ассемблер...);
- запоминание установок.

Контрольные вопросы

1. Какие виды ошибок существуют?
2. Что такое тест? Какими свойствами должен обладать тест?
3. Каковы критерии выбора тестов?
4. Дайте краткую характеристику каждому критерию выбора теста.
5. Опишите последовательность разработки тестов.
6. Что входит в понятие надежности ПО?
7. Какие виды отказов существуют?
8. Каковы количественные характеристики надежности программ?
9. Что представляют собой методы оценки и измерения характеристик надежности ПО?
10. Перечислите достоинства парного программирования.

Глава 6
СОПРОВОЖДЕНИЕ ПРОГРАММ

Составление программной документации — очень важный процесс. Стандарт, определяющий процессы жизненного цикла программного обеспечения, даже предусматривает специальный процесс, посвященный указанному вопросу. При этом на каждый программный продукт должна разрабатываться документация двух типов: для пользователей различных групп и для разработчиков. Отсутствие документации любого типа для конкретного программного продукта недопустимо.

При подготовке документации не следует забывать, что она разрабатывается для того, чтобы ее можно было использовать, и потому она должна содержать все необходимые сведения.

6.1. Виды программных документов

К программным относят документы, содержащие сведения, необходимые для разработки, сопровождения и эксплуатации программного обеспечения. Документирование программного обеспечения осуществляется в соответствии с Единой системой программной документации (ГОСТ 19.XXX). Так, ГОСТ 19.101—77 устанавливает виды программных документов для программного обеспечения различных типов. Ниже перечислены основные программные документы по этому стандарту и указано, какую информацию они должны содержать.

Спецификация должна содержать перечень и краткое описание назначения всех файлов программного обеспечения, в том числе и файлов документации на него, и является обязательной для программных систем, а также их компонентов, имеющих самостоятельное применение.

Ведомость держателей подлинников (код вида документа — 05) должна содержать список предприятий, на которых хранятся

подлинники программных документов. Необходимость этого документа определяется на этапе разработки и утверждения технического задания только для программного обеспечения со сложной архитектурой.

Текст программы (код вида документа — 12) должен содержать текст программы с необходимыми комментариями. Необходимость этого документа определяется на этапе разработки и утверждения технического задания.

Описание программы (код вида документа — 13) должно содержать сведения о логической структуре и функционировании программы.

Ведомость эксплуатационных документов (код вида документа — 20) должна содержать перечень эксплуатационных документов на программу, к которым относятся документы с кодами 30, 31, 32, 33, 34, 35, 46. Необходимость этого документа также определяется на этапе разработки и утверждения технического задания.

Формуляр (код вида документа — 30) должен содержать основные характеристики программного обеспечения, комплектность и сведения об эксплуатации программы.

Описание применения (код вида документа — 31) должно содержать сведения о назначении программного обеспечения, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств.

Руководство системного программиста (код вида документа — 32) должно содержать сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения.

Руководство программиста (код вида документа — 33) должно содержать сведения для эксплуатации программного обеспечения.

Руководство оператора (код вида документа — 34) должно содержать сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программного обеспечения.

Описание языка (код вида документа — 35) должно содержать описание синтаксиса и семантики языка.

Руководство по техническому обслуживанию (код вида документа — 46) должно содержать сведения для применения тесто-

вых и диагностических программ при обслуживании технических средств.

Программа и методика испытаний (код вида документа — 51) должны содержать требования, подлежащие проверке при испытании программного обеспечения, а также порядок и методы их контроля.

Пояснительная записка (код вида документа — 81) должна содержать информацию о структуре и конкретных компонентах программного обеспечения, в том числе схемы алгоритмов, их общее описание, а также обоснование принятых технических и технико-экономических решений.

Составляется на стадии эскизного и технического проектов.

Прочие документы (коды вида документа — 90—99) могут составляться на любых стадиях разработки, т. е. на стадиях эскизного, технического и рабочего проектов.

6.2. Пояснительная записка

Пояснительная записка должна содержать всю информацию, необходимую для сопровождения и модификации программного обеспечения: сведения о его структуре и конкретных компонентах, общее описание алгоритмов и их схемы, а также обоснование принятых технических и технико-экономических решений.

Содержание пояснительной записки по стандарту (ГОСТ 19.404—79) должно включать следующие разделы:

- введение;
- назначение и область применения;
- технические характеристики;
- ожидаемые технико-экономические показатели;
- источники, используемые при разработке.

В разделе *Введение* указывают наименование программы и документа, на основании которого ведется разработка.

В разделе *Назначение и область применения* указывают назначение программы и дают краткую характеристику области применения.

Раздел *Технические характеристики* должен содержать следующие подразделы:

- постановку задачи, описание применяемых математических методов и допущений и ограничений, связанных с выбранным математическим аппаратом;

- описание алгоритмов и функционирования программы с обоснованием принятых решений;
- описание и обоснование выбора способа организации входных и выходных данных;
- описание и обоснование выбора состава технических и программных средств на основании проведенных расчетов или анализов.

В разделе *Ожидаемые технико-экономические показатели* указывают технико-экономические показатели, обосновывающие преимущество выбранного варианта технического решения.

В разделе *Источники, использованные при разработке*, указывают перечень научно-технических публикаций, нормативно-технических документов и других научно-технических материалов, на которые есть ссылки в исходном тексте.

Пояснительная записка составляется профессионалами в области разработки программного обеспечения для специалистов того же уровня квалификации. Следовательно, в ней уместно использовать специальную терминологию, ссылаться на специальную литературу и т. п.

6.3. Руководство пользователя

В настоящее время часто используют еще один эксплуатационный документ, в который отчасти входит руководство системного программиста, программиста и оператора. Этот документ называют Руководством пользователя. Появление такого документа явилось следствием широкого распространения персональных компьютеров, работая на которых пользователи совмещают в своем лице трех указанных специалистов.

Составление документации для пользователей имеет свои особенности, связанные с тем, что пользователь, как правило, не является профессионалом в области разработки программного обеспечения. В книге С. Дж. Гримм [17] даны рекомендации по написанию подобной программной документации:

- учитывайте интересы пользователей — руководство должно содержать все инструкции, необходимые пользователю;
- излагайте ясно, используйте короткие предложения;
- избегайте технического жаргона и узкоспециальной терминологии, если все же необходимо использовать некоторые термины, то их следует пояснить;

- будьте точны и рациональны — длинные и запутанные руководства обычно никто не читает, например, лучше привести рисунок формы, чем долго ее описывать.

Руководство пользователя, как правило, содержит следующие разделы:

- общие сведения о программном продукте;
- описание установки;
- описание запуска;
- инструкции по работе (или описание пользовательского интерфейса);
- сообщения пользователю.

Раздел *Общие сведения о программе* обычно содержит наименование программного продукта, краткое описание его функций, реализованных методов и возможных областей применения.

Раздел *Установка* обычно содержит подробное описание действий по установке программного продукта и сообщений, которые при этом могут быть получены.

В разделе *Запуск*, как правило, описаны действия по запуску программного продукта и сообщений, которые при этом могут быть получены.

Раздел *Инструкции по работе* обычно содержит описание режимов работы, форматов ввода-вывода информации и возможных настроек.

Раздел *Сообщения пользователю* должен содержать перечень возможных сообщений, описание их содержания и действий, которые необходимо предпринять по этим сообщениям.

6.4. Руководство системного программиста

По ГОСТ 19.503—79 руководство системного программиста должно содержать всю информацию, необходимую для установки программного обеспечения, его настройки и проверки работоспособности. Кроме того, как указывалось выше, в него часто включают и описание необходимого обслуживания, которое раньше приводилось в руководстве оператора (ГОСТ 19.505—79) и/или руководстве по техническому обслуживанию (ГОСТ 19.508—79). В настоящее время данную схему используют для составления руководства системному администратору.

Руководство системного программиста должно содержать следующие разделы:

- общие сведения о программном продукте;
- структура;
- настройка;
- проверка;
- дополнительные возможности;
- сообщения системному программисту.

Раздел *Общие сведения* о программе должен включать описание назначения и функций программы, а также сведения о технических и программных средствах, обеспечивающих выполнение данной программы (например, объем оперативной памяти, требования к составу и параметрам внешних устройств, требования к программному обеспечению и т. п.).

В разделе *Структура программы* должны быть приведены сведения о структуре программы, ее составных частях, о связях между составными частями и о связях с другими программами.

В разделе *Настройка программы* должно быть приведено описание действий по настройке программы на условия практического применения.

В разделе *Проверка программы* должно быть приведено описание способов проверки работоспособности программы, например контрольные примеры.

В разделе *Дополнительные возможности* должно быть приведено описание дополнительных возможностей программы и способов доступа к ним.

В разделе *Сообщения системному программисту* должны быть указаны тексты сообщений, выдаваемых в ходе выполнения настройки и проверки программы, а также в ходе ее выполнения, описание их содержания и действий, которые необходимо предпринять по этим сообщениям [1].

Контрольные вопросы

1. Назовите виды программных документов.
2. Как составляется пояснительная записка?
3. Каким образом составляется руководство пользователя?
4. Как составляется руководство системного администратора?

Глава 7
РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.1. Инструментальные средства разработки программ

Инструментальное программное обеспечение (Software tools) — программное обеспечение, используемое в ходе разработки, корректировки или развития других программ: редакторы, компиляторы, отладчики, вспомогательные системные программы, графические пакеты и др.

Сюда входят языки программирования, интегрированные среды разработки программ, CASE-системы и др.

7.1.2. Выбор языка программирования

Существующие на сегодняшний день языки программирования можно выделить в следующие группы [1, 56]:

- универсальные языки высокого уровня;
- специализированные языки разработчика программного обеспечения;
- специализированные языки пользователя;
- языки низкого уровня.

В группе *универсальных языков высокого уровня* безусловным лидером на сегодня является язык C++. Действительно, он имеет ряд достоинств:

- масштабируемость. На языке C++ разрабатывают программы для самых различных платформ и систем;
- возможность работы на низком уровне с памятью, адресами, портами, что при неосторожном использовании может легко превратиться в недостаток;

- С++ имеет мощный препроцессор, унаследованный от С, но, как и любой другой мощный инструмент, требует осторожного использования;
- возможность создания обобщенных алгоритмов для разных типов данных, их специализация и вычисления на этапе компиляции, используя шаблоны.

При этом язык С++ обладает рядом существенных недостатков:

- подключение интерфейса внешнего модуля через препроцессорную вставку заголовочного файла (#include) серьезно замедляет компиляцию при подключении большого количества модулей;
- недостаток информации о типах данных во время компиляции;
- сложность для изучения и компиляции;
- некоторые преобразования типов неинтуитивны. В частности, операция над беззнаковым и знаковым числами выдает беззнаковый результат.

Для С++ существует большое количество библиотек классов, поддерживающих создание пользовательского интерфейса, клиент-серверных приложений, работу с базами данных и т. д., поэтому пока альтернативы С++ нет [40]. Для второстепенных проектов иногда используется Visual Basic. Язык Java рассматривался как альтернатива Basic, но из-за отсутствия визуального средства разработки форм он пока остается малопригодным. Современный Object Pascal, как и Pascal, предложенный Н. Виртом в середине 70-х годов XX в., остается наиболее привлекательным для обучения основам программирования в силу своей простоты, структурированности и обнаружения компилятором большого количества не только синтаксических, но и семантических ошибок.

В нынешнее время в отличие от 60-х годов XX в. языки программирования создаются крайне редко. За последние 15 лет можно отметить лишь две новинки, получившие широкое распространение — это Java (Sun Microsystems, 1995 г.), ставший популярным во многом благодаря технологии его использования в Интернете и появления такого понятия, как виртуальная Java-машина, и C# (Microsoft, 2000 г.), созданный на основе С++.

Создателем языка является сотрудник Microsoft Андреас Хейлсберг. Он стал известным в мире программистов задолго до

того, как пришел в Microsoft. Хейлсберг входил в число ведущих разработчиков одной из самых популярных сред разработки — Delphi. В Microsoft он участвовал в создании версии Java — J++, так что опыта в написании языков и сред программирования ему не занимать. Как отмечал сам Андреас Хейлсберг, C# создавался как язык компонентного программирования, и в этом одно из главных достоинств языка, направленное на возможность повторного использования созданных компонентов.

Другие достоинства языка C#:

- сохраняет лучшие черты популярных языков программирования C/C++, на основе которых он создан. В связи с этим облегчается переход программистов от C++ к C#;
- является проще и надежнее C++. Простота и надежность главным образом связаны с тем, что на C# хотя и допускаются, но не поощряются такие опасные свойства C++, как указатели, адресация, разыменование, адресная арифметика;
- является полностью объектно-ориентированным языком, где даже типы, встроенные в язык, представлены классами;
- реализует возможности наследования и универсализации;
- учитывает все возможности Framework .Net, так как C# создавался параллельно с данной средой;
- благодаря каркасу Framework .Net, ставшему надстройкой над операционной системой, программисты C# получают те же преимущества работы с виртуальной машиной, что и программисты Java. Эффективность кода даже повышается, поскольку исполнительная среда CLR представляет собой компилятор промежуточного языка, в то время как виртуальная Java-машина является интерпретатором байт-кода;
- мощная библиотека каркаса поддерживает удобство построения различных типов приложений на C#, позволяя легко строить Web-службы, другие виды компонентов, достаточно просто сохранять и получать информацию из базы данных и других хранилищ данных;
- является источником надежного и эффективного кода.

Кроме вышеописанных языков к группе универсальных принадлежат также Modula, Ada, COBOL, FORTRAN и некоторые другие. Каждый из вышеописанных языков имеет свои особенности и, соответственно, свою область применения. В на-

— 7888

стоящее время универсальные языки программирования применяются в самых различных областях человеческой деятельности, таких как:

- научные вычисления (языки C++, FORTRAN, Java);
- системное программирование (языки C++, Java);
- обработка информации (языки C++, COBOL, Java);
- искусственный интеллект (LISP, Prolog);
- издательская деятельность (Postscript, TeX);
- удаленная обработка информации (Perl, PHP, Java, C++);
- описание документов (HTML, XML).

С течением времени одни языки развивались, приобретали новые черты и остались востребованными, другие утратили свою актуальность и сегодня представляют в лучшем случае чисто теоретический интерес (Focal, PL/1 и др.). В значительной степени это связано с такими факторами:

- наличие среды программирования, поддерживающей разработку приложений на конкретном языке программирования;
- удобство сопровождения и тестирования программ;
- стоимость разработки с применением конкретного языка программирования;
- четкость и ортогональность конструкций языка;
- применение объектно-ориентированного подхода.

Специализированные языки разработчика используют для создания конкретных типов программного обеспечения. К ним относят:

- языки баз данных;
- языки создания сетевых приложений;
- языки создания систем искусственного интеллекта и т. д.

Специализированные языки пользователя обычно являются частью профессиональных сред пользователя, характеризуются узкой направленностью и разработчиками программного обеспечения не используются.

Языки низкого уровня позволяют осуществлять программирование практически на уровне машинных команд. При этом получают самые оптимальные как с точки зрения времени выполнения, так и с точки зрения объема необходимой памяти программы. Недостатком их является то, что они не поддерживают принципов структурного программирования [1].

В настоящее время языки типа ассемблера обычно используют:

- при написании сравнительно простых программ, для обращения к техническим средствам, например драйверов;
- в виде вставок в программы на языках высокого уровня, например, для ускорения преобразования данных в циклах с большим количеством повторений.

В большей степени выбор языка программирования определяется опытом разработчика, требованиями ведущей разработку организации или просто устоявшимся мнением.

7.1.3. Выбор среды программирования

Интегрированной средой разработки программного обеспечения называют систему программных средств, используемую программистами для разработки программного обеспечения [1, 56].

Обычно среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, компоновщик, отладчик и справочную систему. Иногда также содержит систему управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя. Многие современные среды разработки также включают инспектор объектов, браузер классов и диаграмму иерархии классов, которые используются для объектно-ориентированной разработки ПО. Обычно среда разработки предназначается для одного определенного языка программирования, как, например, Visual Basic или Deiphi, но существуют среды разработки, предназначенные для нескольких языков, такие как Eclipse или Microsoft Visual Studio.

Примеры сред разработки — Turbo Pascal, Borland C++, GNU toolchain, DrPython.

В последнее время, с развитием объектно-ориентированного программирования, широкое распространение получили упоминавшиеся ранее среды визуального программирования, в которых наиболее распространенные блоки программного кода представлены в виде графических объектов.

Наиболее часто используемыми являются визуальные среды Delphi, C++ Builder фирмы Borland (Inprise Corporation), Visual C++, Visual Basic фирмы Microsoft, Visual Ada фирмы IBM и др.

Большую популярность в наши дни получила технология .NET Framework, предложенная фирмой Microsoft в качестве платформы для создания как обычных программ, так и веб-приложений. Основным преимуществом .NET является совместимость различных служб, написанных на разных языках. Например, служба, написанная на C++ для .NET, может обратиться к методу класса из библиотеки, написанной на Delphi; на C# можно написать класс, наследующий от класса, написанного на Visual Basic .NET, а исключение, выброшенное методом, написанным на C#, может быть поймано и обработано в Delphi.

Так же как и в случае с выбором языка программирования, выбор среды программирования определяется характером проекта, привычками и навыками разработчика, веяниями времени, требованиями заказчика и просто общественным мнением: «Все подобные разработки должны выполняться в среде...

7.2. Технологии программирования

7.2.1. Объектно-ориентированное программирование

Рассмотрим сначала, как появилось ***объектно-ориентированное программирование***. Ключевое понятие, помогающее при программировании, — это абстракция. Она позволяет лучше понять сущность программированного объекта или среду. Пусть нужно совершить поездку в Мурманск. Возникают вопросы: «Каким образом это сделать? какой транспорт использовать? сколько это будет стоить?» и т. д. Нужно выделить главное и отбросить лишнее. Тут главным будет вид транспорта. Эту абстракцию и проще всего назвать классом при программировании. У транспорта есть данные (скорость, количество двигателей и др.) и методы (взлет, посадка для самолета). Класс группирует данные и методы в единую сущность. Данные обычно закрыты, и их изменение, как правило, производится посредством методов, т. е. они защищены корректной работой методов.

Первое использование классов как объектов произошло в 1967 г.: Бьерн Страуструп применил язык Simula в своей диссертации для программы, моделирующей компьютерные системы. Этот язык очень выразителен и позволяет работать с высоким уровнем абстракций. Однако при запуске программы оказалось,

что у нее очень низкая производительность и выполнить работу в срок не удастся, поэтому пришлось переписать программу на языке Си. В Си классов нет. Страуструп их добавил, и появился язык C++ [40]

Несколько лет назад в журнале Byte появилась статья «Объектно-ориентированное программирование умерло?». В ней говорилось о том, что объекты не оправдали возложенные на них надежды. Достичь главной цели — повторного использования кода — с помощью объектов сложно, хотя сам процесс программирования они упростили.

В этот же период Microsoft создает ***Visual Basic***. Главным нововведением в нем является возможность вставки управляющих элементов (кнопок, полей ввода) на форму. К каждому элементу можно добавить кусок кода для описания его деятельности. Оказалось очень удобно, и вскоре были созданы тысячи новых элементов — появилось расширение ***VBX (Visual Basic Extention).*** Статья Byte описала управляющие элементы VBX как наиболее успешную реализацию мечты о повторном использовании кода.

У многих поклонников ООП статья вызвала недовольство. Элементы VBX не являются объектно-ориентированными. В них нет даже концепции метода, нет наследования и полиморфизма. Тем не менее VBX может рассматриваться как пример ***программного компонента.*** Это часть бинарного кода, который может быть легко вставлен в различные приложения. VBX — это любопытная, но тупиковая ветвь эволюции технологии программирования. Однако она сыграла свою роль.

Как же развивалась эволюция программирования? Первоначально существовали статические библиотеки. Такие библиотеки компоновались в выполняемый файл, т. е. каждая программа содержала код библиотеки. Их легко представить в виде перфокарт, которые вставляли программисты в свои программы в «далекие времена».

Для того чтобы снизить затраты памяти, были созданы ***динамически компонуемые библиотеки DLL.*** При их применении несколькими приложениями в память загружалась только одна копия библиотеки, и все приложения использовали эту копию. Другое полезное свойство DLL — компоновка в процессе выполнения, т. е. новая версия DLL может быть использована без перестройки приложения. Если новая версия библиотеки совместима со старой, то код может оказаться эффективнее (если улучшены алгоритмы в библиотеки или исправлены ошибки),

если нет, приложение может оказаться неработоспособным. Второй случай называется DLL HELL.

Распределение памяти при использовании статических библиотек показано на рис. 7.1.

Рис. 7.1. Распределение памяти при использовании статических библиотек

Рис. 7.2. Распределение памяти при использовании DLL

Распределение памяти при использовании DLL отражено на рис. 7.2.

7.2.2. Платформа JAVA и .NET

На данный момент наиболее активно развиваются две конкурирующие линии технологий создания ПО на основе компонентов — технологии Java и .NET. Рассмотрим несколько элементов этих технологий, являющихся ключевыми в создании широко востребованного в настоящее время и достаточно сложного вида приложений. Это Web-приложения, т. е. распределенное программное обеспечение, использующее базовую инфраструктуру Интернета для связи между различными своими компонентами, а стандартные инструменты для навигации по Web-браузеры — как основу для своего пользовательского интерфейса.

Технологии Java представляют собой набор стандартов, инструментов и библиотек, предназначенных для разработки прило-

жений разных типов и связанных друг с другом использованием языка программирования Java. Торговая марка Java принадлежит компании Sun Microsystems, и эта компания во многом определяет развитие технологий Java, но в нем активно участвуют и другие игроки — IBM, Intel, Oracle, Hewlett-Packard, SAP, Bea и пр.

В этот набор входят следующие основные элементы:

- платформа *Java Platform Standard Edition (J2SE)* [1]. Предназначена для разработки обычных, в основном однопользовательских приложений;
- платформа *Java Platform Enterprise Edition (J2EE)* [2]. Предназначена для разработки распределенных веб-приложений уровня предприятия;
- платформа *Java Platform Micro Edition (J2ME)* [3]. Предназначена для разработки встроенных приложений, работающих на ограниченных ресурсах, в основном в мобильных телефонах и компьютеризированных бытовых устройствах;
- платформа *Java Card* [5]. Предназначена для разработки ПО, управляющего функционированием цифровых карт. Ресурсы, имеющиеся в распоряжении такого ПО, ограничены в наибольшей степени.

С некоторыми оговорками можно считать, что J2ME является подмножеством J2SE, а та, в свою очередь, — подмножеством J2EE. Java Card представляет собой, по существу, особый набор средств разработки, связанный с остальными платформами только поддержкой (в сильно урезанном виде) языка Java.

Язык Java — это объектно-ориентированный язык программирования, который транслируется не непосредственно в машинно-зависимый код, а в так называемый *байт-код*, исполняемый специальным интерпретатором, *виртуальной Java-машиной (Java Virtual Machine — JVM)*. Такая организация работы Java-программ позволяет им быть переносимыми без изменений и одинаково работать на разных платформах, если на этих платформах есть реализация JVM, соответствующая опубликованным спецификациям виртуальной машины.

Кроме того, интерпретация кода позволяет реализовывать различные политики безопасности для одних и тех же приложений, выполняемых в разных средах, — к каким ресурсам (файлам, устройствам и пр.) приложение может иметь доступ, а к каким нет, можно определять при запуске виртуальной машины. Таким способом можно обеспечить запускаемое пользователем вручную приложение (за вред, причиненный которым, будет от-

вечать этот пользователь) бóльшими правами, чем аплет, загруженный автоматически с какого-то сайта в Интернете.

Режим интерпретации приводит обычно к более низкой производительности программ по сравнению с программами, оттранслированными в машинно-специфический код. Для преодоления этой проблемы JVM может работать в режиме *динамической компиляции (just-in-time-compilation — JIT)*, в котором байт-код на лету компилируется в машинно-зависимый, а часто исполняемые участки кода подвергаются дополнительной оптимизации.

.NET представляет собой похожий набор стандартов, инструментов и библиотек, но разработка приложений в рамках .NET возможна с использованием различных языков программирования. Основой .NET являются виртуальная машина для *промежуточного языка (Intermediate Language — IL*, иногда встречается сокращение *Microsoft IL — MSIL)*, в который транслируются все .NET-программы, также называемая *общей средой выполнения (Common Language Runtime — CLR)*, и общая библиотека классов (.NET Framework class library), доступная из всех .NET-приложений.

Промежуточный язык является полноценным языком программирования, но он не предназначен для использования людьми. Разработка в рамках .NET ведется на одном из языков, для которых имеется транслятор, в промежуточный язык —Visual Basic.NET, C++, C#, Java (транслятор Java в .NET называется J#, и он не обеспечивает одинаковой работы программ на Java, оттранслированных в .NET и выполняемых на JVM) и пр. Однако разные языки достаточно сильно отличаются друг от друга, и чтобы гарантировать возможность из одного языка работать с компонентами, написанными на другом языке, необходимо при разработке этих компонентов придерживаться *общих правил (Common Language Specifications — CLS)*, определяющих, какими конструкциями можно пользоваться во всех .NET-языках без потери возможности взаимодействия между результатами. Наиболее близок к промежуточному языку C# — этот язык был специально разработан вместе с платформой .NET.

Некоторым отличием от Java является то, что код на промежуточном языке в .NET не интерпретируется, а всегда выполняется в режиме динамической компиляции (JIT).

Компания Microsoft инициировала разработку платформы .NET и принятие стандартов, описывающих ее отдельные эле-

менты (к сожалению, пока не все), и она же является основным поставщиком реализаций этой платформы и инструментов разработки. Благодаря наличию стандартов возможна независимая реализация .NET (например, такая реализация разработана в рамках проекта Mono), но, в силу молодости платформы и опасений по поводу монопольного влияния Microsoft на ее дальнейшее развитие, реализации .NET не от Microsoft используются достаточно редко.

Для Java-технологий базовым языком является Java, а для изучения правил построения компонентов для .NET используется язык C#. Он наиболее удобен при работе в этой среде и наиболее похож на Java.

Оба языка имеют мощные выразительные возможности объектно-ориентированных языков последнего поколения, поддерживающих автоматическое управление памятью и работу в многопоточном режиме. Они весьма похожи, но имеют большое число мелких отличий в деталях. Наиболее существенны для построения программ различия, касающиеся наличия в C# невиртуальных методов, возможности объявления и использования пользовательских типов значений и делегатных типов в C# и возможности передачи значений параметров в C# по ссылке.

7.3. Защита программных продуктов

7.3.1. Основные сведения о защите программных продуктов

Программные продукты и компьютерные базы данных являются предметом интеллектуального труда специалистов высокой квалификации. Процесс проектирования и реализации программных продуктов характеризуется значительными материальными и трудовыми затратами, основан на использовании наукоемких технологий и инструментария, требует применения и соответствующего уровня дорогостоящей вычислительной техники. Это обусловливает необходимость принятия мер по защите интересов разработчика программ и создателей компьютерных баз данных от несанкционированного их использования [56].

Программное обеспечение является объектом защиты также и в связи со сложностью и трудоемкостью восстановления его

работоспособности, значимостью программного обеспечения для работы информационной системы.

Защита программного обеспечения преследует цели:

1) ограничение несанкционированного доступа к программам или их преднамеренное разрушение и хищение;

2) исключение несанкционированного копирования (тиражирования) программ.

Программный продукт и базы данных должны быть защищены по нескольким направлениям от воздействия:

1) человека — хищение машинных носителей и документации программного обеспечения; нарушение работоспособности программного продукта и др.;

2) аппаратуры — подключение к компьютеру аппаратных средств для считывания программ и данных или их физического разрушения;

3) специализированных программ — приведение программного продукта или базы данных в неработоспособное состояние (например, вирусное заражение), несанкционированное копирование программ и базы данных и т. д.

Самый простой и доступный способ защиты программных продуктов и баз данных — *ограничение доступа*. Контроль доступа к программному продукту и базе данных устанавливается с помощью:

- защиты программ при их запуске паролем;
- использования ключевой дискеты для запуска программ;
- ограничения доступных пользователям программных модулей или данных, функций обработки и др.

Могут также использоваться *криптографические методы* защиты информации базы данных или головных программных модулей.

7.3.2. Криптографические методы защиты информации

Первоначальное значение термина «криптография» — «тайнопись», «тайное письмо».

Криптография — наука о защите информации от прочтения ее посторонними. Защита достигается шифрованием, т. е. преобразованием, которое делает защищенные входные данные трудночитаемыми без знания специальной информации — ключа.

Под ключом понимается легко изменяемая часть криптосистемы, хранящаяся в тайне и определяющая, какое шифрующее преобразование из возможных выполняется в данном случае. Криптосистема — семейство выбираемых с помощью ключа обратимых преобразований, которые преобразуют защищаемый открытый текст в шифрограмму, и обратно [56].

По характеру использования ключа известные криптосистемы можно разделить на два типа: симметричные (с секретным ключом) и несимметричные (с открытым ключом).

В первом случае в шифраторе отправителя и дешифраторе получателя используется один и тот же ключ. Шифратор образует шифр-текст, который является функцией открытого текста. Конкретный вид функции шифрования определяется секретным ключом. Дешифратор получателя сообщения выполняет обратное преобразование аналогичным образом. Секретный ключ хранится в тайне, и исключается незаконный перехват ключа. Обычно предполагается правило Кирхгофа: стойкость шифра определяется только секретностью ключа, т. е. криптоаналитику известны все детали процесса шифрования и дешифрования, кроме секретного ключа.

Открытый текст обычно имеет произвольную длину, если его размер велик и он не может быть обработан вычислительным устройством шифратора целиком, он разбивается на блоки фиксированной длины, и каждый блок шифруется в отдельности, независимо от его положения во входной последовательности. Такие криптосистемы называются системами блочного шифрования.

На практике обычно используют два общих принципа шифрования: рассеивание и перемешивание. Рассеивание заключается в распространении влияния одного символа открытого текста на много символов шифра-текста: это позволяет скрыть статистические свойства открытого текста. Развитием этого принципа является распространение влияния одного символа ключа на много символов шифрограммы, что позволяет исключить восстановление ключа по частям. Перемешивание состоит в использовании таких шифрующих преобразований, которые исключают восстановление взаимосвязи статистических свойств открытого и шифрованного текста. Распространенный способ достижения хорошего рассеивания состоит в использовании составного шифра, который может быть реализован в виде некоторой последовательности простых шифров, каждый из которых

вносит небольшой вклад в значительное суммарное рассеивание и перемешивание. В качестве простых шифров чаще всего используют простые подстановки и перестановки.

7.3.3. Программные системы защиты от несанкционированного копирования

Данные системы предотвращают нелицензионное использование программных продуктов и баз данных. Программа выполняется только при опознании некоторого уникального ключевого элемента.

Таким ключевым элементом могут быть:

- дискета, на которой записан не подлежащий копированию ключ;
- определенные характеристики аппаратуры компьютера;
- специальное устройство (электронный ключ), подключаемое к компьютеру и предназначенное для выдачи опознавательного кода.

Программные системы защиты от копирования программных продуктов:

- идентифицируют среду, из которой будет запускаться программа;
- устанавливают соответствие среды, из которой запущена программа, той, для которой разрешен санкционированный запуск;
- вырабатывают реакцию на запуск из несанкционированной среды;
- регистрируют санкционированное копирование;
- противодействуют изучению алгоритмов и программ работы системы.

Для идентификации запускающих дискет применяются следующие методы:

- нанесение повреждений на поверхность дискеты («лазерная дыра»), которая с трудом может быть воспроизведена в несанкционированной копии дискеты;
- нестандартное форматирование запускающей дискеты.

Идентификация среды компьютера обеспечивается за счет:

- закрепления месторасположения программ на жестком магнитном диске (так называемые неперемещаемые программы);

- привязки к номеру BIOS (расчет и запоминание с последующей проверкой при запуске контрольной суммы системы);
- привязки к аппаратному (электронному) ключу, вставляемому в порт ввода-вывода, и др.

7.3.4. Правовые методы защиты программных продуктов и баз данных

Правовые методы защиты программ включают:
- патентную защиту;
- закон о производственных секретах;
- лицензионные соглашения и контракты;
- закон об авторском праве.

Различают две категории прав:
- экономические права, дающие их обладателям право на получение экономических выгод от продажи или использования программных продуктов и баз данных;
- моральные права, обеспечивающие защиту личности автора в его произведении.

Во многих цивилизованных странах несанкционированное копирование программ в целях продажи или бесплатного распространения рассматривается как государственное преступление, карается штрафом или тюремным заключением. Но, к сожалению, само авторское право не обеспечивает защиту новой идеи, концепции, методологии и технологии разработки программ, поэтому требуются дополнительные меры их защиты.

Патентная защита устанавливает приоритет в разработке и использовании нового подхода или метода, примененного при разработке программ, удостоверяет их оригинальность.

Статус производственного секрета для программы ограничивает круг лиц, знакомых или допущенных к ее эксплуатации, а также определяет меру их ответственности за разглашение секретов. Например, используется парольный доступ к программному продукту или базе данных, вплоть до паролей на отдельные режимы (чтение, запись, корректировку и т. п.). Программы, как любой материальный объект большой стоимости, необходимо охранять от кражи и преднамеренных разрушений.

Лицензионные соглашения распространяются на все аспекты правовой охраны программных продуктов, включая автор-

ское право, патентную защиту, производственные секреты. Наиболее часто используются лицензионные соглашения на передачу авторских прав.

Лицензия — договор на передачу одним лицом (лицензиаром) другому лицу (лицензиату) права на использование имени, продукции, технологии или услуги. Лицензиар увеличивает свои доходы сбором лицензионных платежей, расширяет область распространения программного продукта или базы данных; лицензиат извлекает доходы за счет их применения.

В лицензионном соглашении оговариваются все условия эксплуатации программ, в том числе создание копий. На каждой копии программы должны быть те же отметки, что и на оригинале:

- знак авторского права (обычно ©) и название разработчика, год выпуска программы, прочие ее атрибуты;
- знак патентной защиты или производственного секрета;
- торговые марки, соответствующие использованным в программе другим программным изделиям (обычно — ™ и название фирмы — разработчика программного продукта);
- символ зарегистрированного права на распространение программного продукта (обычно ®).

Существует несколько типов лицензий на программные продукты.

1. *Исключительная лицензия* — продажа всех имущественных прав на программный продукт или базу данных, покупателю лицензии предоставляется исключительное право на их использование, а автор или владелец патента отказывается от самостоятельного их применения или предоставления другим лицам.

Это самый дорогой вид лицензии, к нему прибегают для монопольного владения с целью извлечения дополнительной прибыли либо с целью прекращения существования на рынке программных средств программного продукта.

2. *Простая лицензия* — это лицензия, когда лицензиар предоставляет право лицензиату использовать программный продукт или базу данных, оставляя за собой право применять их и предоставлять на аналогичных условиях неограниченному числу лиц (лицензиат при этом не может сам выдавать сублицензии, может лишь продать копии приобретенного программного продукта или базы данных).

Такой вид лицензии приобретают дилер (торговец) либо фирмы-производители, использующие купленные лицензии как сопутствующий товар к основному виду деятельности. Напри-

мер, многие производители и фирмы, торгующие компьютерной техникой, осуществляют продажу вычислительной техники с установленным лицензионным программным обеспечением (операционная система, текстовый редактор, электронная таблица, графические пакеты и т. д.).

3. *Этикеточная лицензия* — лицензия на одну копию программного продукта или базы данных. Данный тип лицензии применяется при розничной продаже. Каждый официальный покупатель заключает лицензионное соглашение с продавцом на их использование, но при этом сохраняется авторское право разработчика.

Закон об охране программных продуктов и компьютерных баз данных автором признает физическое лицо, в результате творческой деятельности которого они созданы. Автору независимо от его имущественных прав принадлежат личные авторские права: авторство, имя, неприкосновенность (целостность) программ или баз данных.

Авторское право действует с момента создания программного продукта или базы данных в течение всей жизни автора и 50 лет после его смерти. Автор может:

- выпускать в свет;
- воспроизводить в любой форме, любыми способами;
- распространять;
- модифицировать;
- осуществлять любое иное использование программного продукта или базы данных.

Авторское право не связано с правом собственности на материальный носитель.

Имущественные права на программный продукт или базу данных могут быть переданы частично или полностью другим физическим или юридическим лицам по договору. Имущественные права относятся к категории наследуемых. Если программный продукт или база данных созданы в порядке выполнения служебных обязанностей, имущественные права принадлежат работодателю.

Программные продукты и базы данных могут использоваться третьими лицами — *пользователями* на основании договора с правообладателем.

Лицо, правомерно владеющее экземпляром программы или базы данных, вправе, без получения дополнительного разрешения правообладателя, осуществлять любые действия, связанные

с функционированием программного продукта или базы данных в соответствии с ее назначением, в том числе:

- устанавливать один экземпляр, если не предусмотрено иное соглашение с правообладателем программного продукта или базы данных, на компьютер;
- исправлять явные ошибки;
- адаптировать программный продукт или базу данных;
- изготавливать страховые копии.

Контрольные вопросы

1. Назовите известные вам языки программирования и перечислите их особенности.
2. Как происходит выбор среды программирования?
3. Сравните технологию .NET с технологией Java.
4. Перечислите основные сведения о защите программных продуктов.
5. Назовите криптографические средства защиты.
6. Дайте определение программным средствам защиты.
7. Какие правовые средства защиты ПП вам известны?

Глава 8
КОЛЛЕКТИВНАЯ РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Из-за больших объемов проектов разработка программного обеспечения ведется коллективом специалистов. Работая в коллективе, отдельные специалисты должны взаимодействовать друг с другом, обеспечивая целостность проекта, что при отсутствии удовлетворительных средств описания поведения сложных систем, упоминавшемся выше, достаточно сложно. Причем чем больше коллектив разработчиков, тем сложнее организовать процесс работы [8].

8.1. Пакеты прикладных программ

Для визуализации, специфицирования, конструирования и документирования программных систем необходимо рассматривать их с различных точек зрения. Рассмотрим наиболее применяемые пакеты прикладных программ для коллективной разработки программного обеспечения.

8.1.1. Система контроля версий Microsoft Visual SourceSafe

Microsoft Visual SourceSafe (Visual SourceSafe, VSS) — программный продукт компании Майкрософт, файл-серверная система управления версиями, предназначенная для небольших команд разработчиков. VSS позволяет хранить в общем хранилище файлы, разделяемые несколькими пользователями, для каждого файла хранится история версий.

VSS входит в состав пакета Microsoft Visual Studio и интегрирован с продуктами этого пакета. Доступен только для платформы Windows. Версию для Unix поддерживает компания MainSoft.

Visual SourceSafe нацелен на индивидуальных разработчиков либо небольшие команды разработчиков. Там, где VSS недостаточно, ему на замену предлагается новый продукт Майкрософт — Visual Studio Team Foundation Server входящий в состав Visual Studio Team System.

8.1.2. Система контроля версий Subversion

Subversion — свободно распространяемая система управления версиями с открытым кодом. Subversion разработана специально для замены CVS, самой распространенной открытой системы управления версиями. Она обладает всеми основными функциями CVS (хотя некоторые из них выполняет другими способами) и лишена ряда ее недостатков.

Subversion часто называют **«svn»** — по названию клиентской программы, входящей в ее дистрибутив.

Subversion — централизованная система. Данные хранятся в едином хранилище. При сохранении новых версий используется дельта-компрессия, т. е. система находит отличия новой версии от предыдущей и записывает только их, избегая ненужного дублирования данных. Хранилище может располагаться на локальном диске или на сетевом сервере. К локальному хранилищу клиент Subversion обращается непосредственно. Для доступа к удаленному серверу может использоваться собственный сетевой протокол или стандартный протокол WebDAV, поддерживаемый с помощью специального модуля для веб-сервера Apache.

Клиенты копируют файлы из хранилища, создавая локальные рабочие копии, затем модифицируют их и публикуют изменения в хранилище. Несколько клиентов могут одновременно обращаться к хранилищу. При использовании доступа с помощью WebDAV опционально поддерживается прозрачное управление версиями — если любой клиент WebDAV открывает для записи и затем сохраняет файл, хранящийся на сетевом ресурсе, автоматически создается новая версия.

Коннтрольные вопросы

1. Что такое коллективная разработка ПО?
2. Что такое система контроля версий?
3. Расскажите об основных особенностях известных вам систем контроля версий.

Глава 9
ЭКОНОМИЧЕСКИЕ АСПЕКТЫ РАЗРАБОТКИ И ИСПОЛЬЗОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ

9.1. Оценка стоимости разработки программного обеспечения

Процесс оценки стоимости ПО является весьма сложной задачей. Он начинается на стадии анализа требований к программному обеспечению и продолжается даже на этапе реализации. Существует множество подходов к решению этой задачи, но до сих пор не существует универсальной методики, дающей гарантированный результат. Можно только утверждать, что чем точнее сформулированы требования к разрабатываемому программному продукту, тем вернее можно оценить его стоимость.

Одним из способов получения достоверного результата является применение разных методов на этапе анализа требований или проектирования, а затем усреднение результатов, полученных этими методами [57].

9.1.1. Линейный метод

Этот довольно старый метод, несмотря на свою простоту (и даже примитивность), активно применяется по сей день.

Стоимость разработки определяется следующим образом:

$$С = Т \times Ц, \tag{9.1}$$

где С — стоимость; Т — трудозатраты (например, в человеко-часах или человеко-месяцах); Ц — их удельная стоимость, которую

определяют, в основном исходя из заработной платы и связанных с ней начислений.

Трудозатраты вычисляют по следующей формуле:

$$Т = Р \times П. \tag{9.2}$$

Здесь Р — размер кода программы, чаще всего измеряется в строках (LOC — Lines Of Code); П — временная производительность.

Недостаток такого подхода кроется в способе, которым измеряется результат. Получается, что у программиста отсутствует стимул для повышения своего мастерства и написания лаконичного кода, более простого в отладке и соответственно более дешевого [8].

9.1.2. Метод функциональных точек

Этот метод используется для измерения производительности взамен устаревшего линейного подхода, где производительность измерялась количеством строк программного кода. Впервые функциональные точки (function points) были предложены сотрудником IBM Аланом Альбрехтом в 1979 г. [58].

Преимуществом данного метода является то, что поскольку применение функциональных точек основано на изучении требований, то оценка необходимых трудозатрат может быть выполнена на самых ранних стадиях работы над проектом. Для поддержки и развития данного метода в 1986 г. была создана Международная группа пользователей функционального измерения (IFPUG — International Function Point User Group).

Метод заключается в следующем.

Сначала выделяются функции разрабатываемого программного обеспечения, причем на уровне пользователей, а не программного кода. Например, рассмотрим программный комплекс, реализующий различные методы сортировки одномерных массивов. Одной из функций пользователя данного комплекса будет **выбор метода**, ее мы и будем описывать в качестве примера [61].

Следующим шагом метода будет подсчет количества факторов, приведенных ниже:

- *внешние входы.* Различаются только те входы, которые по-разному влияют на функцию. Функция **выбор метода** имеет один внешний вход;

- *внешние выходы.* Различными считаются выходы для различных алгоритмов. Представим, что наша функция выдает сообщение — текстовое описание выбранного метода, и вызывает другую функцию, непосредственно реализующую выбранный алгоритм сортировки, следовательно, она имеет два выхода;
- *внешние запросы.* В нашем примере таковых нет;
- *внутренние логические файлы* — группа данных, которая создается или поддерживается функцией, считается за единицу. В качестве внутреннего логического файла для нашей функции примем текстовый файл, содержащий описания алгоритмов;
- *внешние логические файлы* — пользовательские данные, находящиеся во внешних по отношению к данной функции файлах. Каждая группа данных принимается за единицу. Внешним по отношению к нашей функции является файл с результатом обработки.

Далее полученные значения умножаются на коэффициенты сложности для каждого фактора (по данным IFPUG) и суммируются для получения полного размера программного продукта. Значения этих коэффициентов приведены в табл. 9.1.

Таблица 9.1. **Значения коэффициентов сложности**

Параметр	Просто	Средне	Сложно
Внешние входы	3	4	6
Внешние выходы	4	5	7
Внешние запросы	3	4	6
Внутренние логические файлы	7	10	15
Внешние логические файлы	5	7	10

Для рассматриваемого нами примера возьмем значения, приведенные в табл. 9.2.

Размер нашей функции составит:

$$ФР = 1 \times 3 + 1 \times 4 + 1 \times 5 + 1 \times 7 + 1 \times 7 = 26.$$

Это число является предварительной оценкой и нуждается в уточнении.

Таблица 9.2. **Пример коэффициентов сложности**

Параметр	Просто		Средне		Сложно	
	Количество	Коэффициент	Количество	Коэффициент	Количество	Коэффициент
Внешние входы	1	3	0	4	0	6
Внешние выходы	1	4	1	5	0	7
Внешние запросы	0	3	0	4	0	6
Внутренние логические файлы	1	7	0	10	0	15
Внешние логические файлы	0	5	1	7	0	10

Следующим шагом в определении размера программного кода методом функциональных точек является присвоение веса (от 0 до 5) каждой характеристике проекта. Перечислим эти характеристики:

1. Требуется ли резервное копирование данных?
2. Требуется обмен данными?
3. Используются распределенные вычисления?
4. Важна ли производительность?
5. Программа выполняется на сильно загруженном оборудовании?
6. Требуется ли оперативный ввод данных?
7. Используется много форм для ввода данных?
8. Поля базы данных обновляются оперативно?
9. Ввод, вывод, запросы являются сложными?
10. Внутренние вычисления сложны?
11. Код предназначен для повторного использования?
12. Требуется преобразование данных и установка программы?
13. Требуется много установок в различных организациях?
14. Требуется поддерживать возможность настройки и простоту использования?

Значения для данных характеристик определяются следующим образом: 0 — никогда; 1 — иногда; 2 — редко; 3 — средне; 4 — часто; 5 — всегда.

Эти характеристики для примера функции сведены в табл. 9.3.

Таблица 9.3. **Пример характеристик проектов**

Характеристика	Значение в примере	Характеристика	Значение в примере
1	3	8	0
2	1	9	3
3	0	10	4
4	4	11	5
5	2	12	0
6	1	13	0
7	3	14	3

Определяется S — сумма всех весов.

И наконец, уточненный функциональный размер вычисляется по формуле

$$УФР = ФР \times (0{,}65 + 0{,}01 \times S). \tag{9.3}$$

Уточненный функциональный размер функции **выбор метода** будет следующим:

$$УФР = 26 \times \ (0{,}65 + 0{,}01 \times 29) = 17{,}19.$$

Получившийся результат показывает, что функция **выбор метода** достаточно проста и не требует больших трудозатрат. Полученные значения затем используются для оценки стоимости проекта.

В настоящее время существует несколько модификаций метода функциональных точек [57].

Точки свойств

В случае, если вышеописанные характеристики не отражают истинной сложности реализации (например, при разработке операционных систем), вместо метода функциональных точек применяют его усовершенствованный вариант, предложенный в 1988 г. Кейперсом Джонсом, — метод точек свойств. Этот метод корректирует оценки, полученные методом функциональных точек с учетом алгоритмической сложности программного продукта.

Метод Mark II

Метод Mark II был представлен Чарльзом Саймонсом также в 1988 г. Этот метод более пригоден для оценки сложных систем, чем классический метод функциональных точек. Он позволяет добиться одного и того же результата как при оценке системы в целом, так и при суммировании оценок, полученных для составляющих ее подсистем.

Трехмерные функциональные точки

В 1991 г. софтверным подразделением корпорации Boeing было предложено еще одно решение — метод трехмерных функциональных точек. Отличием от классического метода является то, что сложность программного продукта оценивается по трем направлениям — данные, функции и управление. Достоинством метода является его применимость не только к оценке программных проектов, но и к оценке трудоемкости задач в других сферах деятельности.

Объектные точки

Метод объектных точек адаптирует оригинальный метод функциональных точек к объектно-ориентированной технологии программирования.

9.1.3. Оценка с использованием эмпирических данных

Альтернативой методам, описанным выше, являются методы, основанные на уже имеющемся опыте, которые позволяют добиться хороших результатов при относительно невысоких затратах.

Метод Демарко

Метод, разработанный Томом Демарко в 1982 г., основан на использовании так называемой «бэнг-метрики». Этот простой, но эффективный подход к оценке стоимости ПО, близок по содержанию к методу функциональных точек с тем отличием, что полученные оценки корректируются с учетом хронологических данных по выполненным ранее проектам. Такой подход позво-

ляет получить не абстрактные показатели, а приближенные значения реальных затрат ресурсов и времени.

SLIM

В 1978 г. Лоуренс Патнам предложил нелинейную модель, использующую эмпирические данные при оценке стоимости ПО — SLIM (Software Life-cycle Model). Данная методика, созданная на основе реальных данных, собранных в Министерстве обороны США, предназначена для определения трудоемкости крупных программных проектов. Несмотря на то что широкого распространения эта модель не приобрела, некоторые фирмы до сих пор используют ее для расчета трудозатрат.

Согласно модели SLIM трудозатраты на разработку ПО вычисляются по следующей формуле:

$$\mathrm{T} = \left(\frac{\mathrm{P}}{\mathrm{C}}\right) \times t_d^{-4}, \qquad (9.4)$$

где P — размер программы, чаще всего измеряется в строках кода, хотя можно использовать и другие единицы измерения (функциональные точки, например);

C — технологическая константа, учитывающая как уровень существующих технологий, так и производительность персонала (команды разработчиков);

t_d — ограничение на срок поставки, измеряется в годах.

COCOMO

Модель COCOMO (COnstructive COst MOdel), предложенная в 1981 г. известным ученым Барри Боэмом [59], является на сегодняшний день самой популярной методикой для оценки стоимости разработки ПО. Для создания COCOMO были проанализированы статистические данные 63 проектов различных типов. Данная модель имеет три уровня детализации: базовый, промежуточный и подробный и предусматривает три режима использования в зависимости от размеров проекта и реализующей его команды (табл. 9.4).

Трудоемкость проекта на базовом уровне COCOMO определяется с помощью следующей формулы:

$$\mathrm{T} = a \times \mathrm{P} \times b, \qquad (9.5)$$

где a и b — константы, которые определяются режимом использования модели.

Таблица 9.4. **Режимы использования COCOMO**

Режим	Размер проекта	Описание
Органичный	До 50 KLOC[1]	Некрупный проект и небольшая команда, для которой нехарактерны нововведения, и среда остается стабильной
Сблокированный	50—300 KLOC	Относительно небольшая команда занимается проектом среднего размера, в процессе разработки необходимы определенные инновации, среда характеризуется незначительной нестабильностью
Внедренный	Более 300 KLOC	Большая команда разработчиков трудится над крупным проектом, необходим значительный объем инноваций, среда состоит из множества элементов, которые не характеризуются стабильностью

Из формулы (9.5) видно, что трудозатраты зависят от размера проекта и при смене режима модели скачкообразно изменяются (табл. 9.5).

Длительность выполнения проекта по модели COCOMO вычисляется по формуле

$$F = 2{,}5 \times \mathrm{T} \times k. \tag{9.6}$$

Из формулы (9.6) видно, что рост трудоемкости проекта не приведет к значительному увеличению его длительности, поскольку при этом изменяется значение константы k, зависящей от режима модели.

Таблица 9.5. **Значения коэффициентов в зависимости от режимов модели**

Режим	Коэффициент *a*	Коэффициент *b*	Коэффициент *k*
Органичный	2,4	1,05	0,38
Сблокированный	3,0	1,12	0,35
Внедренный	3,6	1,20	0,32

[1] KLOC (KiloLines Of Code) — тысяч строк кода.

Промежуточный и подробный уровни модели COCOMO добавляют в формулы (9.5) и (9.6) дополнительные коэффициенты, которые позволяют повысить точность оценок. Кроме того, в модели возможна корректировка оценок на основе хронологических данных из уже выполненных проектов.

COCOMO II

Модель COCOMO II пришла на смену устаревшей оригинальной модели в 1997 г. Выполненная на основе COCOMO, она адаптирована к современным технологиям разработки ПО. Так, если в COCOMO использовалась каскадная модель жизненного цикла, то COCOMO II предназначена для спиральной и итеративной моделей. Размер программного продукта в COCOMO II может измеряться как строками кода, так и функциональными и объектными точками.

COCOMO II имеет три варианта использования — фактически это три разные модели для решения различных задач, объединенные под одним общим названием (табл. 9.6).

Таблица 9.6. **Варианты использования модели COCOMO II**

Название модели	Описание
Композиционная прикладная	Ориентирована на проекты, создаваемые с применением современных инструментальных средств и UML, использует в качестве метрики объектные точки
Ранней разработки проекта	Применяется для получения приближенных оценок по проекту до определения его архитектуры, использует в качестве метрик количество строк кода или функциональные точки
Постархитектурная модель	Наиболее детализированная модель, используется после разработки архитектуры проекта и позволяет получить самые точные оценки, применяет в качестве метрик количество строк кода или функциональные точки

Таким образом, при сохранении основных принципов COCOMO модель стала намного гибче и при оценке трудоемкости и стоимости ПО учитывает намного больше различных факторов, влияющих на выполнение проекта.

9.2. Методы оценки эффективности ПО на этапе эксплуатации

Современная финансовая теория выделяет следующие показатели экономической эффективности внедрения программных проектов:

- **внутренняя норма дохода (IRR** — Internal Rate of Return);
- **чистая приведенная** (текущая) **стоимость (NPV** — Net Present Value);
- **срок окупаемости (PB** — Payback Period);
- **совокупная стоимость владения (TCO** — Total Cost of Ownership);
- **норма возврата инвестиций (ROI** — Return of Investment).

Наиболее часто для оценки эффективности информационных систем используют два последних показателя — TCO и ROI.

Под TCO понимается сумма всех затрат на внедрение и обеспечение функционирования ИС вплоть до момента ее вывода из эксплуатации. Существует две основные модели расчета совокупной стоимости владения: концепция, предложенная Gartner Group, и результат совместных усилий Microsoft и Interpose. По методике, предложенной Gartner, все затраты делятся на фиксированные и текущие.

Фиксированные затраты производятся один раз на этапе внедрения системы. К ним относят: стоимость разработки и внедрения проекта, первоначальные закупки необходимого для внедрения ИС аппаратного и программного обеспечения, привлечение внешних консультантов.

Текущие затраты — расходы, обеспечивающие функционирование системы. Это те затраты, которые требуются постоянно, пока система работает. К ним относятся:

- обновление и модернизация системы;
- управление системой в целом — администрирование, обучение администрации и конечных пользователей, заработная плата персонала, привлечение внешних ресурсов;
- «активность пользователя» — доработка ПО, дополнительные настройки ПО, работа с данными, последствия некомпетентных действий пользователя.

Казалось бы, все просто — нужно только подсчитать затраты по каждой из вышеперечисленных статей. Но на самом деле не

все расходы легко подсчитать — значительная их часть не только не закладывается заранее, но и нигде не учитывается. Причем 75 % затрат, входящих в состав TCO, обусловлены проблемами конечных пользователей. В модели TCO, разработанной Microsoft и Interpose, учитываются как раз эти затраты. Согласно их методике затраты делятся на прямые и косвенные.

Прямые затраты — те, которые предусматриваются бюджетом и планируются. К ним относятся расходы на аппаратное и программное обеспечение, управление (администрирование и проектирование), поддержку, разработку.

Косвенные затраты — составляют более 50 % средних расходов, которые не поддаются планированию и часто вообще не регистрируются. К ним относятся, прежде всего, пользовательские затраты (неформальное обучение, персональная поддержка, ошибки и просчеты) и простои из-за выхода оборудования из строя или плановых профилактических остановок.

Таким образом, совокупную стоимость владения можно подсчитать по простой формуле

$$\text{TCO} = \text{Пр} + \text{Кс}, \tag{9.7}$$

где Пр — прямые затраты;

Кс — косвенные затраты.

Эффективность вложений (возврат инвестиций) ROI — это показатель, характеризующий выгоду программного проекта. Он рассчитывается как отношение дисконтированных поступлений, ожидаемых от внедрения данного программного продукта, к начальной стоимости инвестиций:

$$\text{ROI} = \frac{\text{Эф}}{\text{И}}, \tag{9.8}$$

где Эф — эффект от внедрения, выраженный в денежных единицах;

И — инвестиции в ИС.

Модель ROI принадлежит Gartner Group.

Для оценки доходной части, как правило, анализируют цели бизнеса, которые нужно достичь путем внедрения программного проекта или появления каких-либо новых программных продуктов. Берут измеримые показатели бизнеса (например, сокраще-

ние операционных расходов, поддержку конкурентоспособного состояния, улучшение внутреннего контроля) и по ним делают оценки эффекта. В качестве расходной части чаще всего используют показатель TCO.

Контрольные вопросы

1. Перечислите методы оценки стоимости ПО.
2. Опишите линейный метод.
3. Опишите метод функциональных точек.
4. Какие существуют модификации метода функциональных точек?
5. Приведите методы оценки стоимости ПО с использованием эмпирических данных.
6. Охарактеризуйте COCOMO и COCOMO II.
7. Как производится оценка эффективности ПО на этапе эксплуатации?
8. Что такое показатели TCO и ROI?

Лабораторный практикум

ЛАБОРАТОРНАЯ РАБОТА № 1. Этапы разработки программного обеспечения при структурном подходе к программированию. Стадия «Техническое задание»

Цель работы: ознакомиться с правилами написания технического задания.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Этапы разработки программного обеспечения. Постановка задачи» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [1, 4].
3. Ознакомиться с разделами гл. 2 данного пособия.

Теоретическая часть. Разработка технического задания

Техническое задание представляет собой документ, в котором сформулированы основные цели разработки, требования к программному продукту, определены сроки и этапы разработки и

регламентирован процесс приемо-сдаточных испытаний. В разработке технического задания участвуют как представители заказчика, так и представители исполнителя. В основе этого документа лежат исходные требования заказчика, анализ передовых достижений техники, результаты выполнения научно-исследовательских работ, предпроектных исследований, научного прогнозирования и т. п.

Порядок разработки технического задания

Разработка технического задания выполняется в следующей последовательности. Прежде всего, устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных. Затем определяют перечень результатов, их характеристики и способы представления.

Далее уточняют среду функционирования программного обеспечения: конкретную комплектацию и параметры технических средств, версию используемой операционной системы и, возможно, версии и параметры другого установленного программного обеспечения, с которым предстоит взаимодействовать будущему программному продукту.

В случаях, когда разрабатываемое программное обеспечение собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы в случае сбоев оборудования и энергоснабжения.

1. Общие положения

1.1. Техническое задание оформляют в соответствии с ГОСТ 19.106—78 на листах формата А4 и А3 по ГОСТ 2.301—68, как правило, без заполнения полей листа. Номера листов (страниц) проставляют в верхней части листа над текстом.

1.2. Лист утверждения и титульный лист оформляют в соответствии с ГОСТ 19.104—78. Информационную часть (аннотацию и содержание), лист регистрации изменений допускается в документ не включать.

1.3. Для внесения изменений и дополнений в техническое задние на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

1.4. Техническое задание должно содержать следующие разделы:

- введение;
- наименование и область применения;
- основание для разработки;
- назначение разработки;
- технические требования к программе или программному изделию;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них. При необходимости допускается в техническое задание включать приложения.

2. Содержание разделов

2.1. Введение должно включать краткую характеристику области применения программы или программного продукта, а также объекта (например, системы), в котором предполагается их использовать. Основное назначение введения — продемонстрировать актуальность данной разработки и показать, какое место эта разработка занимает в ряду подобных.

2.2. В разделе «Наименование и область применения» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

2.3. В разделе «Основание для разработки» должны быть указаны:

- документ (документы), на основании которых ведется разработка. Таким документом может служить план, приказ, договор и т. п.
- организация, утвердившая этот документ, и дата его утверждения;
- наименование и (или) условное обозначение темы разработки.

2.4. В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программного изделия.

18 — 7888

2.5. Раздел «Технические требования к программе или программному изделию» должен содержать следующие подразделы:

- требования к функциональным характеристикам;
- требования к надежности;
- условия эксплуатации;
- требования к составу и параметрам технических средств;
- требования к информационной и программной совместимости;
- требования к маркировке и упаковке;
- требования к транспортированию и хранению;
- специальные требования.

2.5.1. В подразделе «Требования к функциональным характеристикам» должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т. п.

2.5.2. В подразделе «Требования к надежности» должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т. п.).

2.5.3. В подразделе «Условия эксплуатации» должны быть указаны условия эксплуатации (температура окружающего воздуха, относительная влажность и т. п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

2.5.4. В подразделе «Требования к составу и параметрам технических средств» указывают необходимый состав технических средств с указанием их технических характеристик.

2.5.5. В подразделе «Требования к информационной и программной совместимости» должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования. При необходимости должна обеспечиваться защита информации и программ.

2.5.6. В подразделе «Требования к маркировке и упаковке» в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

2.5.7. В подразделе «Требования к транспортированию и хранению» должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

2.5.8. В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность, предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

2.6. В разделе «Стадии и этапы разработки» устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, сроки разработки и определяют исполнителей.

2.7. В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

2.8. В приложениях к техническому заданию при необходимости приводят:

- перечень научно-исследовательских и других работ, обосновывающих разработку;
- схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке;
- другие источники разработки.

В случаях, если какие-либо требования, предусмотренные техническим заданием, заказчик не предъявляет, следует в соответствующем месте указать «Требования не предъявляются».

Пример 1.1. Разработать техническое задание на программный продукт, предназначенный для наглядной демонстрации школьникам графиков функций одного аргумента $y = f(x)$. Разрабатываемая программа должна рассчитывать таблицу значений и строить график функций на заданном отрезке по заданной формуле и менять шаг аргумента и границы отрезка. Кроме этого, программа должна запоминать введенные формулы.

Техническое задание к данному примеру смотри в приложении 2.

Пример 1.2. Разработать техническое задание на разработку «Модуля автоматизированной системы оперативно-диспетчерского управления теплоснабжением корпусов Московского института».

Техническое задание к данному примеру смотри в приложении 3.

Порядок выполнения работы

1. Разработать техническое задание на программный продукт (см. варианты заданий в приложении 1).

2. Оформить работу в соответствии с ГОСТ 19.106—78. При оформлении использовать MS Office.

3. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.

2. Технического задания на программный продукт.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Приведите этапы разработки программного обеспечения.
2. Что включает в себя постановка задачи и предпроектные исследования?
3. Перечислите функциональные и эксплуатационные требования к программному продукту.
4. Перечислите правила разработки технического задания.
5. Назовите основные разделы технического задания.

ЛАБОРАТОРНАЯ РАБОТА № 2.
Структурный подход к программированию. Стадия «Эскизный проект»

Цель работы: научиться создавать формальные модели и на их основе определять спецификации разрабатываемого программного обеспечения.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Этапы разработки программного обеспечения. Анализ требований и определение спецификаций программного обеспечения» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [1, 2, 39, 47, 53, 61].
3. Ознакомиться с разд. 3.5 данного учебного пособия.

Теоретическая часть. Разработка спецификаций

Разработка программного обеспечения начинается с анализа требований к нему. В результате анализа получают спецификации разрабатываемого программного обеспечения, строят общую модель его взаимодействия с пользователем или другими программами и конкретизируют его основные функции.

При структурном подходе к программированию на этапе анализа и определения спецификаций разрабатывают три типа моделей: модели функций, модели данных и модели потоков данных. Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, рекомендуется использовать сразу несколько моделей, разрабатываемых в виде диаграмм, и пояснять их текстовыми описаниями, словарями и т. п.

Структурный анализ предполагает использование следующих видов моделей:

- диаграмм потоков данных (DFD — Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе;
- диаграмм «сущность—связь» (ERD — Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы;
- диаграмм переходов состояний (STD — State Transition Diagrams), характеризующих поведение системы во времени;
- функциональных диаграмм (методика SADT);
- спецификаций процессов;
- словаря терминов.

Спецификации процессов

Спецификации процессов обычно представляют в виде краткого текстового описания, схем алгоритмов, псевдокодов, Flow-форм или диаграмм Насси — Шнейдермана (см. разд. 3.5.1).

Словарь терминов

Словарь терминов представляет собой краткое описание основных понятий, используемых при составлении спецификаций. Он должен включать определение основных понятий предметной области, описание структур элементов данных, их типов и форматов, а также всех сокращений и условных обозначений (см. разд. 3.5.2).

Диаграммы переходов состояний

С помощью *диаграмм переходов состояний* можно моделировать последующее функционирование системы на основе ее предыдущего и текущего функционирования. Моделируемая система в любой заданный момент времени находится точно в одном из конечного множества состояний. С течением времени она может изменить свое состояние, при этом переходы между состояниями должны быть точно определены (см. разд. 3.5.3).

Функциональные диаграммы

Функциональные диаграммы отражают взаимосвязи функций разрабатываемого программного обеспечения.

Они создаются на ранних этапах проектирования систем, для того чтобы помочь проектировщику выявить основные функции и составные части проектируемой системы и, по возможности, обнаружить и устранить существенные ошибки. Для создания функциональных диаграмм предлагается использовать методологию SADT (см. разд. 3.5.4).

Диаграммы потоков данных

Для описания потоков информации в системе применяются *диаграммы потоков данных* (DFD — Data flow diagrams). DFD позволяет описать требуемое поведение системы в виде совокупности процессов, взаимодействующих посредством связывающих их потоков данных. DFD показывает, как каждый из процессов преобразует свои входные потоки данных в выходные потоки данных и как процессы взаимодействуют между собой (см. разд. 3.5.5).

Диаграммы «сущность—связь»

Диаграмма сущность—связь — инструмент разработки моделей данных, обеспечивающий стандартный способ определения данных и отношений между ними. Она включает *сущности* и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные *сущности* и *связи* между ними, которые удовлетворяют требованиям, предъявляемым к ИС (см. разд. 3.5.6).

Порядок выполнения работы

1. На основе технического задания из лабораторной работы № 1 выполнить анализ функциональных и эксплуатационных требований к программному продукту.

2. Определить основные технические решения (выбор языка программирования, структура программного продукта, состав

функций ПП, режимы функционирования) и занести результаты в документ, называемый «Эскизным проектом» (см. приложение 4).

3. Определить диаграммы потоков данных для решаемой задачи.

4. Определить диаграммы «сущность—связь», если программный продукт содержит базу данных.

5. Определить функциональные диаграммы.

6. Определить диаграммы переходов состояний.

7. Определить спецификации процессов.

8. Добавить словарь терминов.

9. Оформить результаты, используя MS Office или MS Visio в виде эскизного проекта.

10. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Документа «Эскизный проект», содержащего:
   - выбор метода решения и языка программирования;
   - спецификации процессов;
   - все полученные диаграммы;
   - словарь терминов.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Назовите этапы разработки программного обеспечения.
2. Что такое жизненный цикл программного обеспечения?
3. В чем заключается постановка задачи и предпроектные исследования?
4. Назовите функциональные и эксплуатационные требования к программному продукту.
5. Перечислите составляющие эскизного проекта.
6. Охарактеризуйте спецификации и модели.

ЛАБОРАТОРНАЯ РАБОТА № 3. Структурный подход к программированию. Стадия «Технический проект»

Цель работы: изучить вопросы проектирования программного обеспечения.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Этапы разработки программного обеспечения. Проектирование программного обеспечения» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [1, 39, 47, 53].
3. Ознакомиться с разд. 4.1—4.3 настоящего пособия.

Теоретическая часть. Составляющие технического проекта

ПРОЕКТ ТЕХНИЧЕСКИЙ — образ намеченного к созданию объекта, представленный в виде его описания, схем, чертежей, расчетов, обоснований, числовых показателей.

Технический проект

Цель технического проекта — определение основных методов, используемых при создании информационной системы, и окончательное определение ее сметной стоимости.

Техническое проектирование подсистем осуществляется в соответствии с утвержденным техническим заданием.

Технический проект программной системы подробно описывает:

- выполняемые функции и варианты их использования;
- соответствующие им документы;

- структуры обрабатываемых баз данных;
- взаимосвязи данных;
- алгоритмы их обработки.

Технический проект должен включать данные об объемах и интенсивности потоков обрабатываемой информации, количестве пользователей программной системы, характеристиках оборудования и программного обеспечения, взаимодействующего с проектируемым программным продуктом.

При разработке технического проекта оформляются:

- ведомость технического проекта. Общая информация по проекту;
- пояснительная записка к техническому проекту. Вводная информация, позволяющая ее потребителю быстро освоить данные по конкретному проекту;
- описание систем классификации и кодирования;
- перечень входных данных (документов). Перечень информации, которая используется как входящий поток и служит источником накопления;
- перечень выходных данных (документов). Перечень информации, которая используется для анализа накопленных данных;
- описание используемого программного обеспечения. Перечень программного обеспечения и СУБД, которые планируется использовать для создания информационной системы;
- описание используемых технических средств. Перечень аппаратных средств, на которых планируется работа проектируемого программного продукта;
- проектная оценка надежности системы. Экспертная оценка надежности с выявлением наиболее благополучных участков программной системы и ее узких мест;
- ведомость оборудования и материалов. Перечень оборудования и материалов, которые потребуются в ходе реализации проекта.

Структурная схема

Структурной называют схему, отражающую состав и взаимодействие по управлению частями разрабатываемого программного обеспечения. Структурная схема определяется архитектурой разрабатываемого ПО (см. разд. 4.1.1).

Функциональная схема

Функциональная схема — это схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств (см. разд. 4.1.2).

Разработка алгоритмов

Метод пошаговой детализации реализует нисходящий подход к программированию и предполагает пошаговую разработку алгоритма (см. разд. 4.1.3).

Структурные карты

Методика структурных карт используется на этапе проектирования ПО для того, чтобы продемонстрировать, каким образом программный продукт выполняет системные требования. Структурные карты Константайна предназначены для описания отношений между модулями (см. разд. 4.2).

Техника структурных карт Джексона основана на методе структурного программирования Джексона, который выявляет соответствие между структурой потоков данных и структурой программы. Основное внимание в методе сконцентрировано на соответствии входных и выходных потоков данных (см. разд. 4.3).

Порядок выполнения работы

1. На основе технического задания из лабораторной работы № 1 и спецификаций из лабораторной работы № 2 разработать уточненные алгоритмы программ, составляющих заданный программный модуль. Использовать метод пошаговой детализации (см. разд. 4.1.3).

2. На основе уточненных и доработанных алгоритмов разработать структурную схему программного продукта (разд. 4.1.1).

3. Разработать функциональную схему программного продукта (разд. 4.1.2).

4. Представить структурную схему в виде структурных карт Константайна (см. разд. 4.2).

5. Представить структурную схему в виде структурных карт Джексона (см. разд. 4.3).

6. Оформить результаты, используя MS Office или MS Visio в виде технического проекта.

7. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Структурной схемы программного продукта.
2. Функциональной схемы.
3. Алгоритма программы.
4. Структурной карты Константайна.
5. Структурной карты Джексона.
6. Законченного технического проекта программного модуля.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Назовите этапы разработки программного обеспечения.
2. В чем заключается проектирование программного обеспечения?
3. Перечислите составляющие технического проекта.
4. Охарактеризуйте структурный подход к программированию.
5. Из чего состоят структурная и функциональная схемы?
6. Охарактеризуйте метод пошаговой детализации при составлении алгоритмов программ.
7. Приведите понятие псевдокода.
8. В чем заключается методика Константайна?
9. В чем заключается методика Джексона?

ЛАБОРАТОРНАЯ РАБОТА № 4. Этапы разработки программного обеспечения. Стадия «Реализация»

Цель работы: разработать программный продукт в соответствии с заданным вариантом.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Этапы разработки программного обеспечения. Структурный подход к программированию» учебной дисциплины «Технология разработки программного обеспечения».

2. Изучить соответствующие разделы в изданиях [1, 2, 5, 7, 40].

3. Ознакомиться с гл. 6 настоящего пособия.

Теоретическая часть. Составление программной документации

Важным этапом разработки программного продукта является составление программной документации. Жизненный цикл программного обеспечения содержит специальный процесс, посвященный этому вопросу. На каждый программный продукт должны составляться два типа документации — для разработчиков и для различных групп пользователей. Программная документация пользователей должна содержать все необходимые сведения по эксплуатации ПО. Аналогично, документация разработчика должна содержать сведения, необходимые для разработки и сопровождения программного обеспечения.

Виды программных документов

Документирование программного обеспечения осуществляется в соответствии с Единой системой программной документации (ГОСТ 19.XXX). ГОСТ 19.101—77 содержит виды программных документов для программного обеспечения различных типов. В данном ГОСТе перечислены документы следующих типов:

- *спецификация* должна содержать перечень и краткое описание назначения всех файлов программного обеспечения, в том числе и файлов документации на него, и является обязательной для программных систем, а также их компонентов, имеющих самостоятельное применение;
- *ведомость держателей подлинников* (код вида документа — 05) должна содержать список предприятий, на которых хранятся подлинники программных документов. Необходимость этого документа определяется на этапе разработки и утверждения технического задания только для программного обеспечения со сложной архитектурой;
- *текст программы* (код вида документа — 12) должен содержать текст программы с необходимыми комментариями. Необходимость этого документа определяется на этапе разработки и утверждения технического задания;
- *описание программы* (код вида документа — 13) должно содержать сведения о логической структуре и функционировании программы. Необходимость данного документа также определяется на этапе разработки и утверждения технического задания;
- *ведомость эксплуатационных документов* (код вида документа — 20) должна содержать перечень эксплуатационных документов на программу, к которым относятся документы с кодами 30, 31, 32, 33, 34, 35, 46. Необходимость этого документа также определяется на этапе разработки и утверждения технического задания;
- *формуляр* (код вида документа — 30) должен содержать основные характеристики программного обеспечения, комплектность и сведения об эксплуатации программы;
- *описание применения* (код вида документа — 31) должно содержать сведения о назначении программного обеспечения, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств;

- *руководство системного программиста* (код вида документа — 32) должно содержать сведения для проверки, обеспечения функционирования и настройки программы на условия конкретного применения;
- *руководство программиста* (код вида документа 33) должно содержать сведения для эксплуатации программного обеспечения;
- *руководство оператора* (код вида документа — 34) содержит сведения для обеспечения процедуры общения оператора с вычислительной системой в процессе выполнения программы;
- *описание языка* (код вида документа — 35) — описание синтаксиса и семантики языка программы;
- *руководство по техническому обслуживанию* (код вида документа — 46) содержит сведения для применения программы при обслуживании технических средств.

Порядок выполнения работы

1. По результатам лабораторных работ № 1—3 написать код программ для решения поставленной задачи на языке программирования, выбранном на этапе эскизного проектирования.

2. Отладить программный модуль.

3. Получить результаты работы.

4. Оформить документацию к разработанному программному обеспечению.

5. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Листингов программ.

2. Интерфейса пользователя.

3. Документации к программному обеспечению (руководство пользователя, руководство системного программиста, руководство программиста, руководство оператора).

4. Результатов работы программ.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. В чем состоит этап реализации и отладки программного обеспечения?
2. Какие существуют инструментальные средства разработки?
3. Охарактеризуйте этап стихийного программирования.
4. Охарактеризуйте этапы структурного и модульного программирования.
5. Что такое документация к программному обеспечению?

ЛАБОРАТОРНАЯ РАБОТА № 5. Тестирование программ методами «белого ящика»

Цель работы: изучить методы тестирования логики программы, формализованные описания результатов тестирования и стандарты по составлению схем программ.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Этапы разработки программного обеспечения. Тестирование и отладка программных продуктов» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [1, 2, 7, 8, 9, 37, 45].
3. Ознакомиться с разд. 5.2 данного пособия.

Теоретическая часть. Виды тестирования

Тестирование программного обеспечения включает в себя целый комплекс действий, аналогичных последовательности процессов разработки программного обеспечения. В него входят [7]:

- постановка задачи для теста;
- проектирование теста;
- написание тестов;
- тестирование тестов;
- выполнение тестов;
- изучение результатов тестирования.

Наиболее важным является проектирование тестов. Существуют разные подходы к проектированию тестов.

Первый состоит в том, что тесты проектируются на основе внешних спецификаций программ и модулей либо спецификаций сопряжения модуля с другими модулями, программа при

этом рассматривается как «черный ящик». Смысл теста заключается в том, чтобы проверить, соответствует ли программа внешним спецификациям. При этом содержание модуля не имеет значения. Такой подход получил название — стратегия «черного ящика».

Второй подход — стратегия «белого ящика», основан на анализе логики программы. При таком подходе тестирование заключается в проверке каждого пути, каждой ветви алгоритма. При этом внешняя спецификация во внимание не принимается.

Ни один из этих подходов не является оптимальным. Реализация тестирования методом «черного ящика» сводится к проверке всех возможных комбинаций входных данных. Невозможно протестировать программу, подавая на вход бесконечное множество значений, поэтому ограничиваются определенным набором данных. При этом исходят из максимальной отдачи теста по сравнению с затратами на его создание. Она измеряется вероятностью того, что тест выявит ошибки, если они имеются в программе. Затраты измеряются временем и стоимостью подготовки, выполнения и проверки результатов теста.

Тестирование методом «белого ящика» также не дает 100%-ной гарантии того, что модуль не содержит ошибок. Даже если предположить, что выполнены тесты для всех ветвей алгоритма, нельзя с полной уверенностью утверждать, что программа соответствует ее спецификациям. Например, если требовалось написать программу для вычисления кубического корня, а программа фактически вычисляет корень квадратный, то реализация будет совершенно неправильной, даже если проверить все пути. Вторая проблема — отсутствующие пути. Если программа реализует спецификации не полностью (например, отсутствует такая специализированная функция, как проверка на отрицательное значение входных данных программы вычисления квадратного корня), никакое тестирование существующих путей не выявит такой ошибки. И наконец, проблема зависимости результатов тестирования от входных данных. Одни данные будут давать правильные результаты, а другие нет. Например, если для определения равенства трех чисел программируется выражение вида:

$$\text{IF } (A + B + C)/3 = A,$$

то оно будет верным не для всех значений A, B и C (ошибка возникает в том случае, когда из двух значений B или C одно больше, а другое на столько же меньше A). Если концентрировать

внимание только на тестировании путей, нет гарантии, что эта ошибка будет выявлена.

Таким образом, полное тестирование программы невозможно, т. е. никакое тестирование не гарантирует полное отсутствие ошибок в программе. Поэтому необходимо проектировать тесты таким образом, чтобы увеличить вероятность обнаружения ошибки в программе.

Стратегия «белого ящика»

Существуют следующие методы тестирования по принципу «белого ящика»:

- покрытие операторов;
- покрытие решений;
- покрытие условий;
- покрытие решений/условий;
- комбинаторное покрытие условий.

Метод покрытия операторов

Целью этого метода тестирования является выполнение каждого оператора программы хотя бы один раз.

Пример.

Если для тестирования задать значения переменных $A = 2$, $B = 0$, $X = 3$, будет реализован путь *ace*, т. е. каждый оператор программы выполнится один раз (рис. Л5.1, *а*). Но если внести в алгоритм ошибки — заменить в первом условии *and* на *or*, а во втором $X > 1$ на $X < 1$ (рис. Л5.1, *б*), ни одна ошибка не будет обнаружена (табл. Л5.1). Кроме того, путь *abd* вообще не будет охвачен тестом, и если в нем есть ошибка, она также не будет обнаружена. В табл. Л5.1 ожидаемый результат определяется по блок-схеме на рис. Л5.1, *а*, а фактический — по рис. Л5.1, *б*.

Как видно из этой таблицы, ни одна из внесенных в алгоритм ошибок не будет обнаружена.

Таблица Л5.1. **Результат тестирования методом покрытия операторов**

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2$, $B = 0$, $X = 3$	$X = 2{,}5$	$X = 2{,}5$	Неуспешно

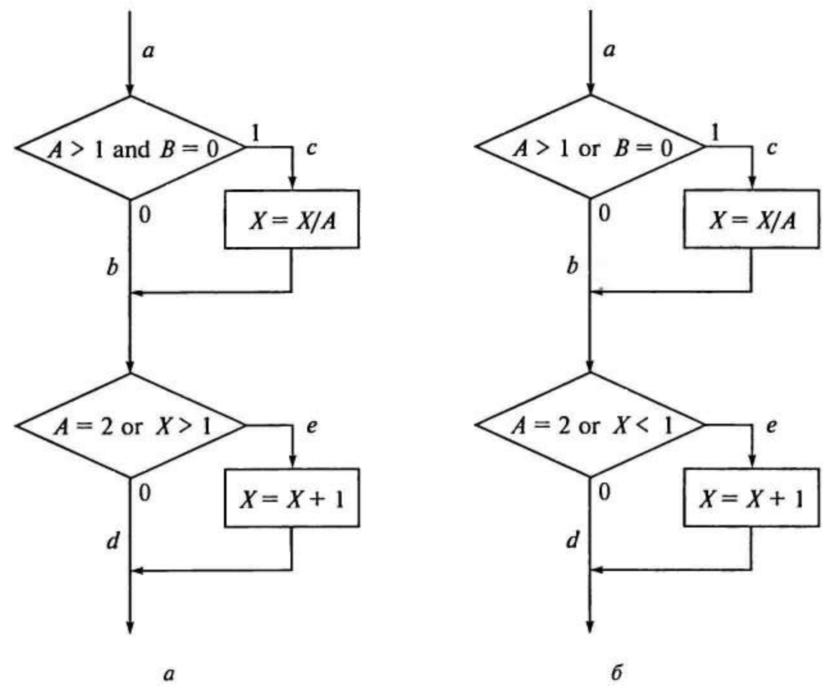


Рис. Л5.1. Пример алгоритма программы:
а — правильный; *б* — с ошибкой

Метод покрытия решений (покрытия переходов)

Согласно методу покрытия решений каждое направление перехода должно быть реализовано, по крайней мере, один раз. Этот метод включает в себя критерий покрытия операторов, так как при выполнении всех направлений переходов выполнятся все операторы, находящиеся на этих направлениях.

Для программы, приведенной на рис. Л5.1, покрытие решений может быть выполнено двумя тестами, покрывающими пути {*ace*, *abd*}, либо {*acd*, *abe*}. Для этого выберем следующие исходные данные: $\{A = 3,\ B = 0,\ X = 3\}$ — в первом случае и $\{A = 2,\ B = 1,\ X = 1\}$ — во втором. Однако путь, где X не меняется, будет проверен с вероятностью 50 %: если во втором условии вместо условия $X > 1$ записано $X < 1$, то ошибка не будет обнаружена двумя тестами.

Результаты тестирования приведены в табл. Л5.2.

Таблица Л5.2. **Результат тестирования методом покрытия решений**

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 3,\ B = 0,\ X = 3$	$X = 1$	$X = 1$	Неуспешно
$A = 2,\ B = 1,\ X = 1$	$X = 2$	$X = 1{,}5$	Успешно

Метод покрытия условий

Этот метод может дать лучшие результаты по сравнению с предыдущими. В соответствии с методом покрытия условий записывается число тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении выполнялись, по крайней мере, один раз.

В рассматриваемом примере имеем четыре условия: $\{A > 1,\ B = 0\}$, $\{A = 2,\ X > 1\}$. Следовательно, требуется достаточное число тестов, такое, чтобы реализовать ситуации, где $A > 1$, $A \leq 1$, $B = 0$ и $B \neq 0$ в точке a и $A = 2$, $A \neq 2$, $X > 1$ и $X \leq 1$ в точке b. Тесты, удовлетворяющие критерию покрытия условий (табл. Л5.3), и соответствующие им пути:

а) $A = 2,\ B = 0,\ X = 4$ *ace*;

б) $A = 1,\ B = 1,\ X = 0$ *abd*.

Таблица Л5.3. **Результаты тестирования методом покрытия условий**

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2,\ B = 0,\ X = 4$	$X = 3$	$X = 3$	Неуспешно
$A = 1,\ B = 1,\ X = 0$	$X = 0$	$X = 1$	Успешно

Метод покрытия решений/условий

Критерий покрытия решений/условий требует такого достаточного набора тестов, чтобы все возможные результаты каждого условия выполнялись по крайней мере один раз, все результаты каждого решения выполнялись по крайней мере один раз и, кроме того, каждой точке входа передавалось управление по крайней мере один раз.

Недостатки метода:

- не всегда можно проверить все условия;
- невозможно проверить условия, которые скрыты другими условиями;

- недостаточная чувствительность к ошибкам в логических выражениях.

Так, в рассматриваемом примере два теста метода покрытия условий

а) $A = 2$, $B = 0$, $X = 4$ *ace*;

б) $A = 1$, $B = 1$, $X = 0$ *abd*

отвечают и критерию покрытия решений/условий. Это является следствием того, что одни условия приведенных решений скрывают другие условия в этих решениях. Так, если условие $A > 1$ будет ложным, транслятор может не проверять условия $B = 0$, поскольку при любом результате условия $B = 0$ результат решения $((A > 1)\&(B = 0))$ примет значение *ложь.* То есть в варианте на рис. Л5.1 не все результаты всех условий выполнятся в процессе тестирования.

Рассмотрим реализацию того же примера на рис. Л5.2. Наиболее полное покрытие тестами в этом случае осуществляется

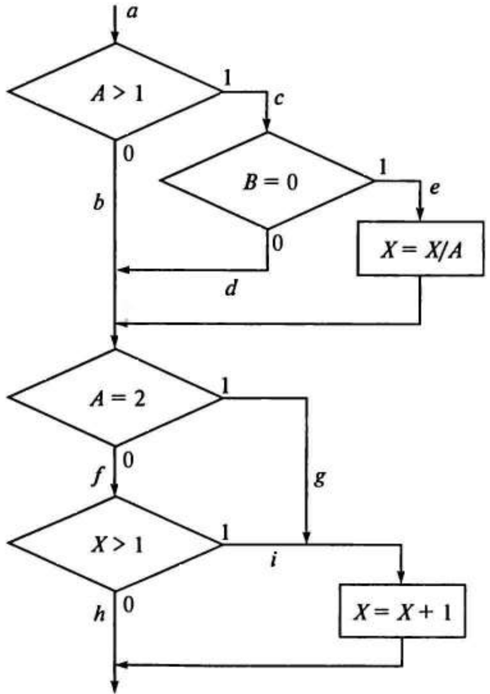


Рис. Л5.2. Пример алгоритма программы

так, чтобы выполнялись все возможные результаты каждого простого решения. Для этого нужно покрыть пути *aceg* (тест $A = 2$, $B = 0$, $X = 4$), *acdfh* (тест $A = 3$, $B = 1$, $X = 0$), *abfh* (тест $A = 0$, $B = 0$, $X = 0$), *abfi* (тест $A = 0$, $B = 0$, $X = 2$).

Протестировав алгоритм на рис. Л5.2, нетрудно убедиться в том, что критерии покрытия условий и критерии покрытия решений/условий недостаточно чувствительны к ошибкам в логических выражениях.

Метод комбинаторного покрытия условий

Критерий комбинаторного покрытия условий удовлетворяет также и критериям покрытия решений, покрытия условий и покрытия решений/условий.

Этот метод требует создания такого числа тестов, чтобы все возможные комбинации результатов условия в каждом решении выполнялись по крайней мере один раз. По этому критерию в рассматриваемом примере должны быть покрыты тестами следующие восемь комбинаций:

1. $A > 1$, $B = 0$.
2. $A > 1$, $B \neq 0$.
3. $A \leq 1$, $B = 0$.
4. $A \leq 1$, $B \neq 0$.
5. $A = 2$, $X > 1$.
6. $A = 2$, $X \leq 1$.
7. $A \neq 2$, $X > 1$.
8. $A \neq 2$, $X \leq 1$.

Для того чтобы протестировать эти комбинации, необязательно использовать все 8 тестов. Фактически они могут быть покрыты четырьмя тестами (табл. Л5.4):

- $A = 2$, $B = 0$, $X = 4$ {покрывает 1, 5};
- $A = 2$, $B = 1$, $X = 1$ {покрывает 2, 6};
- $A = 0{,}5$, $B = 0$, $X = 2$ {покрывает 3, 7};
- $A = 1$, $B = 0$, $X = 1$ {покрывает 4, 8}.

Таблица Л5.4. **Результаты тестирования методом комбинаторного покрытия условий**

Тест	Ожидаемый результат	Фактический результат	Результат тестирования
$A = 2$, $B = 0$, $X = 4$	$X = 3$	$X = 3$	Неуспешно
$A = 2$, $B = 1$, $X = 1$	$X = 2$	$X = 1{,}5$	Успешно
$A = 0{,}5$, $B = 0$, $X = 2$	$X = 3$	$X = 4$	Успешно
$A = 1$, $B = 0$, $X = 1$	$X = 1$	$X = 1$	Неуспешно

Порядок выполнения работы

1. Спроектировать тесты по принципу «белого ящика» для программы, разработанной в лабораторной работе № 4. Использовать схемы алгоритмов, разработанные и уточненные в лабораторных работах № 2, 3.

2. Выбрать несколько алгоритмов для тестирования и обозначить буквами или цифрами ветви этих алгоритмов.

3. Выписать пути алгоритма, которые должны быть проверены тестами для выбранного метода тестирования.

4. Записать тесты, которые позволят пройти по путям алгоритма.

5. Протестировать разработанную вами программу. Результаты оформить в виде таблиц (см. табл. Л5.1—Л5.4).

6. Проверить все виды тестов и сделать выводы об их эффективности.

7. Оформить отчет по лабораторной работе.

8. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Блок-схемы программ.
3. Тестов.
4. Таблиц тестирования программы.
5. Выводов по результатам тестирования (не забывайте, что целью тестирования является обнаружение ошибок в программе).

Контрольные вопросы

1. Охарактеризуйте этап реализации и тестирования программного продукта.
2. Какие существуют виды тестирования?
3. Назовите критерии выбора тестов.
4. Перечислите свойства тестов.
5. Приведите критерии надежности программ.
6. В чем заключается оценка надежности программ?

ЛАБОРАТОРНАЯ РАБОТА № 6. Использование технологий OLE, COM и ActiveX

Цель работы: научиться создавать формальные модели и на их основе определять спецификации разрабатываемого программного обеспечения.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Использование технологий OLE, COM и ActiveX» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [1—3, 10, 11].
3. Ознакомиться с разд. 6.2 настоящего учебного пособия.

Теоретическая часть. От OLE к ActiveX

В данной лабораторной работе рассматривается использование технологии OLE (Object Linking and Embedding — связывание и внедрение объектов), которую можно определить как объектно-ориентированный протокол совместного доступа к данным и программному коду из разных процессов. OLE позволяет программистам создавать приложения для работы с составными документами, представляющими собой динамические связанные структуры, отдельные части которых могут разрабатываться в различных программах.

ActiveX и OLE фирмы Microsoft — еще один шаг к более совершенным, т. е. более надежным и эффективным, программам.

В основе ActiveX и OLE лежит очень простая идея, но, как оказалось, она позволяет существенно повысить эффективность программирования.

Первоначально OLE была задумана как технология интеграции программных продуктов, входящих в комплект Microsoft Office. Предшественницей OLE является реализованная в Windows технология динамического обмена данными DDE (Dynamic Data Exchange), до сих пор широко применяемая в данной среде. Однако многие разработчики не без оснований считают, что DDE трудно использовать, поскольку это технология низкого уровня.

В качестве технологии более высокого уровня была реализована OLE 1.0 OLE 1. Она расширила возможности протокола DDE и, используя его как базовый механизм коммуникаций, позволила активизировать встроенный объект в документе, т. е. получить составной документ. Таким образом, OLE 1.0 унаследовала многие проблемы асинхронного протокола. Эта технология имела множество недостатков, а ее компоновка была слишком сложна для пользователей среднего уровня. Кроме того, установленные связи легко нарушались, например, в результате изменения маршрута доступа к файлу связанного объекта.

С помощью OLE 1 пользователь мог, например, объединить электронную таблицу, созданную Microsoft Excel, с текстовым документом «производства» Microsoft Word. Идея состояла в том, чтобы документно-ориентированная (document-centric) модель работы с компьютером позволила бы пользователю больше думать об информации и меньше о приложениях, ее обрабатывающих. Как следует из слов «связывание и внедрение», составные документы можно создать, либо связав два разных документа, либо полностью внедрив один документ в другой.

OLE 1, как и большинство первых версий программных продуктов, была несовершенна. Архитекторам следующей версии предстояло улучшить первоначальный проект. Вскоре они поняли, что составные документы лишь частный случай более общей проблемы: как разные программные компоненты должны предоставлять друг другу сервисы? Для решения этой проблемы архитекторы OLE создали группу технологий, область применения которых гораздо шире составных документов. Основу OLE 2 составляет важнейшая из этих технологий Модель многокомпо-

нентных объектов (Component Object Model — COM). Новая версия OLE не только обеспечивает поддержку составных документов лучше, чем первая, но и, несомненно, идет куда дальше простого объединения документов, созданных в разных приложениях. OLE 2 позволяет по-новому взглянуть на взаимодействие любых типов программ.

В начале 1996 г. Microsoft ввела в оборот новый термин — ActiveX. Сначала он относился к технологиям, связанным с Интернетом, и приложениям, выросшим из него, вроде WWW (World Wide Web). Поскольку большинство разработок Microsoft в данной области было основано на COM, то и ActiveX была непосредственно связана с OLE. Однако очень скоро новый термин стал захватывать территории, традиционно принадлежавшие OLE, и вот теперь все вернулось на круги своя: OLE, как встарь, обозначает только технологию создания составных документов связыванием и внедрением, а разнообразные технологии на основе COM, ранее объединенные под именем OLE, собраны под знаменем ActiveX. А некоторые технологии, название которых содержало слово «OLE», даже перекрестили — теперь это технологии ActiveX.

Понятие COM

Все технологии OLE и ActiveX, описанные ниже, построены на основании, обеспеченном COM. Итак, что же такое COM? Чтобы ответить на этот вопрос, зададимся сначала другим: «Каким образом одна часть программного обеспечения должна получать доступ к сервисам, предоставляемым другой частью?» На сегодняшний день ответ зависит от того, что представляют собой эти части.

Приложения, например, скомпонованные с библиотекой, могут пользоваться ее сервисами, вызывая функции из этой библиотеки.

Приложение также может использовать сервисы другого, являющегося совершенно отдельным процессом. В этом случае два таких локальных процесса взаимодействуют посредством некоего механизма связи, который обычно требует определения протокола между этими приложениями (набор сообщений, позволяющий одному приложению выдавать запросы, а другому соответствующим образом отвечать на них).

Например, приложение использует сервисы операционной системы или сервисы программного обеспечения, выполняемого на другой машине.

Проблема во всех этих случаях одна: одна часть программного обеспечения должна получить доступ к сервисам, предоставляемым другой частью. Но в каждом отдельном случае механизм доступа разный: вызовы локальных функций, передача сообщения средствами связи между процессами, системные вызовы (которые с точки зрения программиста выглядят практически так же, как и вызовы функций) или какая-то разновидность сетевых коммуникаций.

Этим и занимается COM. Она определяет стандартный механизм, с помощью которого одна часть программного обеспечения предоставляет свои сервисы другой и который работает во всех описанных выше случаях. Общая архитектура сервисов в библиотеках, приложениях, системном и сетевом программном обеспечении позволяет COM изменить подход к созданию программ.

В COM любая часть программного обеспечения реализует свои сервисы как один или несколько объектов COM (не следует путать объекты COM с объектами в языках программирования типа C++; несмотря на то что у них есть общие черты, это разные понятия; далее будет описано соотношение объектов COM и объектов других видов.). Каждый такой объект поддерживает один или несколько интерфейсов, состоящих из методов. Метод — это функция или процедура, которая выполняет некоторое действие и может быть вызвана программным обеспечением, использующим данный объект (клиентом объекта). Методы, составляющие каждый из интерфейсов, обычно определенным образом взаимосвязаны. Клиенты могут получить доступ к сервисам объекта COM только через вызовы методов интерфейсов объекта, у них нет непосредственного доступа к данным объекта.

Большинство объектов COM поддерживают более одного интерфейса. Сам объект всегда реализуется внутри некоторого сервера. Сервер может быть либо динамически подключаемой библиотекой (DLL), подгружаемой во время работы приложения, либо отдельным самостоятельным процессом.

Чтобы вызывать методы интерфейса объекта COM, клиент должен получить указатель на этот интерфейс. Обычно

СОМ-объект предоставляет свои сервисы посредством нескольких интерфейсов, и клиенту требуется отдельный указатель для каждого интерфейса, методы которого он намерен вызывать.

Любой СОМ-объект — это экземпляр определенного класса. Объекты одного класса могут, например, реализовывать сервисы корректировки орфографии и словаря синонимов, тогда как объекты другого класса — представлять банковские счета. Обычно знать класс объекта необходимо для запуска экземпляра этого объекта, выполняемого с помощью библиотеки СОМ. Эта библиотека присутствует на любой системе, поддерживающей СОМ, и имеет доступ к справочнику всех доступных на данной машине классов СОМ-объектов. Клиент может, например, вызвать функцию библиотеки СОМ, передав ей класс нужного ему СОМ-объекта и задав один из поддерживаемых объектом интерфейсов, указатель которого нужен клиенту в первую очередь. (Эти сервисы реализованы библиотекой СОМ в виде обычных вызовов функций, а не через методы интерфейса СОМ.) Затем библиотека СОМ запускает сервер, реализующий объекты данного класса. Кроме того, библиотека возвращает клиенту указатель требуемого интерфейса вновь созданного экземпляра объекта. Далее клиент может запросить указатели на другие необходимые ему интерфейсы непосредственно у самого объекта.

Получив указатель на нужный ему интерфейс выполняющегося объекта, клиент может использовать сервисы объекта, просто вызывая методы этого интерфейса. С точки зрения программиста, вызов метода аналогичен вызову локальной процедуры или функции. Но на самом деле код, выполняющийся по вызову метода, может быть частью или библиотеки, или отдельного процесса, или операционной системы и даже располагаться вообще на другом компьютере.

Благодаря СОМ, клиентам нет нужды учитывать данные отличия — доступ ко всему осуществляется единообразно. Для доступа к сервисам, предоставляемым любыми типами программного обеспечения, используется одна общая модель.

Microsoft применяет СОМ в большинстве продуктов; она используется для спецификации расширений Microsoft Windows и Microsoft Windows NT, а также для определения стандартных интерфейсов к различным типам сервисов. Выгоды применения СОМ в разработке всех типов программного обеспечения несомненны.

Разработку OLE-контейнеров и серверов проще всего вести в среде VisualC++ при помощи специальных мастеров (например, AppWizard), а также библиотеки MFC.

Порядок выполнения работы

1. Создать приложение Cnt, для этого:

- чтобы приступить к созданию нового проекта, выберите в окне компилятора MicrosoftVisualC++ в меню **File** команду New;
- в окне New выберите элемент **MFC AppWizard**(exe), в результате этого будет запущен мастер приложений, работа с которым осуществляется в шесть этапов:
  - — в первом окне установите опцию **Singledocument**;
  - — во втором окне не задавайте поддержку баз данных;
  - — в третьем окне установите опцию **Container**, указывающую на то, что приложение будет OLE-контейнером. В этом же окне следует включить поддержку элементов управления ActiveX;
  - — в следующем, четвертом окне необходимо оставить все опции, заданные по умолчанию;
  - — в пятом окне установите опцию **MFC Standard**, опцию включения комментариев в программу и опцию статической компоновки библиотеки MFC;
  - — наконец, в шестом окне просмотрите список классов, которые будут созданы автоматически, и щелкните по кнопке **Finish**;
- мастер приложений отобразит окно с отчетом о сделанных установках. Если все правильно, щелкните по кнопке OK, с тем чтобы запустить процесс генерации кода нового приложения;
- осталось только построить исполняемый файл приложения, выбрав для этого в меню **Build** команду **Rebuild All.** В результате в папку DEBUG будет добавлен файл CMT.EXE.

2. Проанализировать программный код.

Приложение включает пять основных исходных файлов, сгенерированных мастером AppWizard: CNT.CPP, MAINFRM.CPP, CNTDOC.CPP, CNTVIEW.CPP и CNTRITEM.CPP. Листинги файлов приведены в приложении 5.

3. Внести, где это требуется, изменения.
4. Отладить программы.
5. Проверить работоспособность контейнера, для этого:
   - запустите приложение Cnt (рис. Л6.1);

Рис. Л6.1. Окно приложения Cnt

   - выберите в меню Edit команду InsertNewObject..., в результате чего откроется стандартное диалоговое окно вставки объекта;
   - в этом окне выделите элемент, соответствующий электронной таблице Excel (рис. Л6.2).

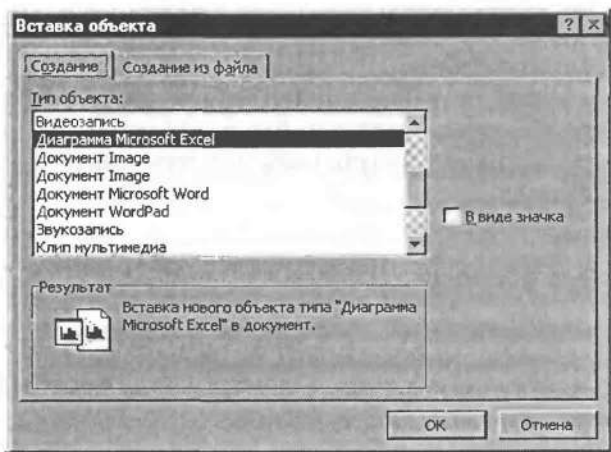


Рис. Л6.2. Выбор внедряемого объекта

6. Оценить результат (рис. Л6.3).

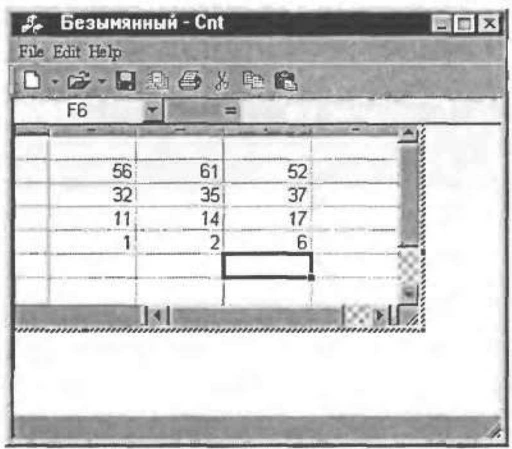


Рис. Л6.3. Внедрение электронной таблицы Excel в приложение Cnt

7. Создать приложение по выбору преподавателя.
8. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:
1. Листингов программ.
2. Результатов работы.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Охарактеризуйте технологии OLE, COM и ActiveX.
2. В чем заключаются преимущества и недостатки OLE?
3. В чем заключаются преимущества и недостатки COM?
4. Опишите развитие от OLE до ActiveX.
5. Как связаны VisualC++ и OLE?

ЛАБОРАТОРНАЯ РАБОТА № 7. Создание сетевых приложений на Delphi с использованием Windows Sockets API

Цель работы: изучить метод разработки сетевых приложений в среде Delphi на низком уровне с использованием Winsock API.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Сетевое программное обеспечение» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить программирование в среде Delphi.
3. Изучить программирование сетевых приложений.
4. Изучить соответствующие разделы в изданиях [1, 2, 4, 5].

Теоретическая часть. Сетевые приложения

Для поддержки сетевых приложений существует технология, названная «сокеты». Сокет — это модель одного конца соединения со всеми присущими ему свойствами и методами. По сути, это прикладной программный интерфейс, входящий в состав многих операционных систем (ОС) и призванный для поддержки сетевых возможностей ОС. В стандарте структуры протоколов семиуровневой модели OSI-сокеты лежат на так называемом транспортном уровне, ниже находится сетевой протокол ip, а выше — протоколы сеансового уровня, такие как ftp, pop3, smtp и т. д.

В Windows поддержка сокетов включена, начиная с версии 3.11, и названа winsock. Для написания приложений с сетевой поддержкой существует специальный winsock api.

Все сетевые приложения построены на технологии «клиент — сервер»; это значит, что в сети существует, по крайней мере, одно приложение, являющееся сервером, типичная задача

которого — это ожидание запроса на подключение от приложений-клиентов, которых может быть теоретически сколько угодно, и выполнение всевозможных процедур в ответ на запросы клиентов. Для клиент-серверной технологии абсолютно неважно, где расположены клиент и сервер — на одной машине или на разных. Конечно, для успешного соединения клиента с сервером клиенту необходимо иметь минимальный набор данных о расположении сервера — для сетей tcp/ip это ip-адрес компьютера, где расположен сервер, и адрес порта, на котором сервер ожидает запросы от клиентов.

Таким образом, пара адрес + порт представляет собой сокет-канал, по которому два компьютера обмениваются данными друг с другом. Только одно приложение на одном компьютере в одно и то же время может использовать конкретный порт, однако для серверных частей возможно создание нескольких сокетов на одном порту для работы с несколькими клиентами. Порт представляет собой число от 0 до 65 535.

Значение порта не обязательно должно совпадать на сервере и клиенте — клиенту для соединения важно только знать порт сервера, порт клиента может выбираться клиентом произвольно и становится известен серверу в момент запроса клиента на соединение. Когда соединение будет установлено, ОС создаст для серверного приложения соответствующий сокет, с которым и будет работать приложение, так что порт клиента для сервера совершенно не важен.

Механизм работы сокетов таков: на серверной стороне запускается серверный сокет, который после запуска сразу переходит в режим «прослушивания» (т. е. ожидания соединения клиентов). На стороне клиента создается сокет, для которого указывается ip-адрес и порт сервера и дается команда на соединение. Когда сервер получает запрос на соединение, ОС создает новый экземпляр сокета, с помощью которого сервер может обмениваться данными с клиентом. При этом сокет, который создан для «прослушивания», продолжает находиться в режиме приема соединений, таким образом, программист может создать сервер, работающий с несколькими подключениями от клиентов.

Работа с сокетами, по существу, это операции ввода-вывода, которые бывают синхронные и асинхронные. В терминологии сокетов работа в асинхронном режиме называется блокирующими сокетами, а в синхронном — неблокирующими сокетами. Попытка соединения или приема данных в блокирующем режи-

ме (отправка всегда синхронна, так как фактически является постановкой в очередь) означает, что пока программа не соединится или не примет данные, передачи управления на следующий оператор не произойдет.

Рассмотрим минимальный набор функций из winsock api, необходимых для написания элементарного клиента и сервера. Сами функции находятся в файле winsock.dll. Файл winsock.pas содержит необходимые объявления импортируемых функций winsock api и базовые структуры данных. К сожалению, этот файл импортирует не все необходимые нам функции, и позже мы напишем свой файл импорта.

```
function wsastartup(wversionrequired: word; var wsdata:
twsadata)  integer; stdcall;
```

Функция сообщает ОС, что в любом процессе приложения могут быть использованы функции winsock. Функция должна быть вызвана один раз при запуске приложения перед использованием любой функции winsock.

```
function wsacleanup: integer; stdcall;
```

Функция сообщает ОС, что приложение более не использует winsock. Данная функция должна быть вызвана перед завершением приложения.

```
function socket(af, struct, protocol: integer)
tsocket; stdcall;
```

Функция создает сокет. Порт и адрес задается в функции **bind** (сервер) или **connect** (клиент). Входящий параметр af — спецификация семейства сокетов (af_inet, af_ipx и др.), struct — спецификация типа нового сокета (принимает значение sock_stream или sock_dgram), protocol — специфический протокол, который будет использоваться сокетом. Если функция выполнена без ошибок, она возвращает дескриптор на новый сокет, если ошибки есть, возвращается invalid_socket.

```
function connect(s: tsocket; var name: tsockaddr;
namelen: integer)  integer; stdcall;
```

Функция соединения для клиента. Структура адреса содержит порт (необходимо привести функцией **htons**) и адрес (для

клиента необходимо привести из имени или спецификации ip4 — xxx.xxx.xxx.xxx).

```
function bind(s: tsocket; var addr: tsockaddr; namelen:
integer)  integer; stdcall;
```

Функция ассоциирует адрес с сокетом. Структура адреса содержит порт (необходимо привести функцией **htons**) и адрес (для сервера обычно указывается inaddr_any — любой).

```
function send(s: tsocket; var buf; len, flags:
integer)  integer; stdcall;
```

Функция отправки данных. Помещает в очередь сокета s кусок данных из buf длиной len. Последний параметр отвечает за вид передачи сообщения. Может быть проигнорирован (0).

```
function recv(s: tsocket; var buf; len, flags:
integer)  integer; stdcall;
```

Функция получения данных.

Итак, рассмотрим примеры элементарного сервера и клиента. Договоримся, что сервер будет работать в асинхронном (блокирующем режиме). Единственная функциональность сервера — это получение строк данных от клиента и вывод их на экран. Связь клиента с сервером разрывается после получения строки, состоящей из единственного символа 'q'

Для обеспечения возможности подключения к серверу множества клиентов сервер на каждое соединение запускает отдельный поток.

```
program winsock_server;
//Простейшее приложение-сервер.
//Сокеты работают в блокирующем режиме.
//На каждое соединение создается отдельный поток.
{$apptype console}
uses
sysutils,
winsock,
windows;
var
vwsadata  twsadata;
vlistensocket,vsocket  tsocket;
vsockaddr : tsockaddr;
```

```
trid   thandle;
const
cport   word(33);
csigexit   'q'
//Процедура отдельного потока для каждого клиента.
procedure socketthread;
var sockname   tsockaddr;
abuf   array of char;
vbuf   string;
vsize   integer;
s :tsocket;
bufsize   integer;
begin
s    vsocket;
if s   invalid_socket then exit;
vsize    sizeof(tsockaddr);
getpeername(s, sockname, vsize);
writeln(format('client accepted, remote address [%s]
[inet_ntoa (sockname.sin_addr)]));
//Определяем размер буфера чтения для сокета.
vsize    sizeof(bufsize);
getsockopt(s,sol_socket,so_rcvbuf,pchar(@
bufsize),vsize);
writeln(format('receive buffer size [%d]',[bufsize]));
setlength(abuf,bufsize);
repeat
//Получаем данные. Процедура работает в блокирующем
//режиме, таким образом, следующая строка кода
//не получит управление, пока не поступят данные
//от клиента.
vsize    recv(s,abuf[0],bufsize,0);
if vsize<=0 then break;
setlength(vbuf,vsize);
lstrcpyn(@vbuf[1],@abuf[0],vsize);
writeln(format('received from cleint: %s',[vbuf]));
until vbuf   'q'
writeln(format('client disconnected, remote address
[%s] ',[inet_ntoa(sockname.sin_addr)]));
setlength(abuf,0);
closesocket(s);
end;
begin
writeln('starting application...
```

```
//Объявляем, что программа будет использовать
//windows sockets.
if wsastartup($101,vwsadata)<>0 then halt(1);
writeln('using windows sockets.');
//Создаем прослушивающий сокет.
vlistensocket
socket(af_inet,sock_stream,ipproto_ip);
writeln(format('creating socket on port
[%d] ',[cport]));
if vlistensocket   invalid_socket then halt(1);
fillchar(vsockaddr,sizeof(tsockaddr),0);
vsockaddr.sin_family    af_inet;
vsockaddr.sin_port    htons(cport);
vsockaddr.sin_addr.s_addr    inaddr_any;
writeln('binding socket.  ');
//Привязываем адрес и порт к сокету.
if bind(vlistensocket,vsockaddr,sizeof(tsockaddr))    0
then halt(1);
//Начинаем прослушивать.
if listen(vlistensocket,somaxconn) <> 0
then halt(1);
writeln('socket status: listening.
repeat
//Ожидаем подключения.
vsocket    accept(vlistensocket,nil,nil);
//Клиент подключился, запускаем новый процесс на
соединение.
createthread(nil,0,@socketthread,0,0,trid);
until false;
closesocket(vlistensocket);
wsacleanup;
end
```

В тексте программы использована не описанная ранее функция **getpeername()**, которая возвращает информацию о канале, ассоциированном с сокетом. В данном контексте она нужна для получения информации о ip-адресе подключившегося клиента. Подробно об этой функции, как и о всех других функциях winsock, можно прочитать в windows sockets 2 application program interface, входящем в состав win32 programmer's reference.

Приведенный выше код можно использовать только как учебное пособие. Для того чтобы использовать его в качестве основы для настоящего приложения, необходимо некоторое коли-

чество доработок, так как многие вещи, которые могут привести в будущем к серьезным ошибкам, были сознательно опущены для уменьшения размера кода и акцентирования внимания именно на аспектах использования winsock. Если вы недостаточно хорошо знакомы с понятием потоков (threads) в windows, то вам лучше использовать класс tthread, существующий в delphi специально для поддержки многопоточных приложений.

Исходный код клиента представлен ниже:

```
program winsock_client;
{$apptype console}
uses
sysutils,
winsock;
const
cport   33;
csigexit   'q'
var
vwsadata   twsadata;
vsocket   tsocket;
vsockaddr   tsockaddr;
buf   string;
begin
if wsastartup($101,vwsadata)<>0 then halt(1);
vsocket    socket(af_inet,sock_stream,ipproto_ip);
if vsocket   invalid_socket then halt(1);
fillchar(vsockaddr,sizeof(tsockaddr),0);
vsockaddr.sin_family    af_inet;
vsockaddr.sin_port    htons(cport);
vsockaddr.sin_addr.s_addr    inet_addr('127.0.0.1');
if connect(vsocket,vsockaddr,sizeof(tsockaddr))
socket_error then halt(1);
repeat
readln(buf);
if send(vsocket,buf[1],length(buf),0)   socket_error
then break;
until buf   csigexit;
closesocket(vsocket);
wsacleanup;
end
```

Итак, мы рассмотрели, как работают сокеты в асинхронном режиме. Давайте посмотрим теперь, какие возможности winsock нам предоставляет для работы с неблокирующими сокетами.

Для того чтобы перевести сокет в неблокирующий режим, используется функция **ioctlsocket(...)**, позволяющая контролировать режимы работа сокета.

Так как теперь функция **recv** сервера будет возвращать управление сразу, независимо от наличия данных в буфере сокета, то теперь нам нужен механизм, позволяющий определять какие-либо события, происходящие с сокетом. Для этого существует несколько механизмов. Первый, который мы рассмотрим, это использование функции **select(...)**.

```
function select(nfds: integer; readfds, writefds,
exceptfds: pfdset; timeout: ptimeval)  longint;
stdcall;
```

Эта функция позволяет контролировать состояние набора сокетов.

Аргумент nfds игнорируется и оставлен только для совместимости. Должен быть равен 0. readfs, writefds, exceptfds — указатели на наборы сокетов, для которых нужно контролировать состояние чтения, отправки данных и ошибок соответственно. Наборы хранятся в структуре pfdset, управление которой осуществляется специальными макросами, описанными в winsock.pas:

```
procedure fs_zero(var fdset: tfdset);
```

Обнуляет структуру, устанавливает количество контролируемых сокетов в 0.

```
procedure fd_set(socket: tsocket; var fdset: tfdset);
```

Добавляет указанный сокет в структуру.

```
procedure fd_clr(socket: tsocket; var fdset: tfdset);
```

Удаляет указанный сокет из структуры.

```
function fd_isset(socket: tsocket; var fdset: tfdset)
boolean
```

Возвращает true, если указанный сокет является членом указанной структуры.

Аргумент timeout является ссылкой на структуру типа ptimeval, в которой можно указать время ожидания срабатыва-

ния функции **select**. В случае указания в качестве значения времени задержки 0 или nil в качестве аргумента timeout функция **select** будет ждать бесконечно, как при выполнении операции в блокирующем режиме.

Как видно из описания, функция может следить сразу за несколькими сокетами, таким образом, теперь мы можем либо также запускать отдельный процесс на каждый открытый сокет, который будет следить за конкретным сокетом, либо создать один процесс, который будет следить за всеми открытыми сокетами. В первом случае нам потребуются лишь незначительные изменения процедуры **socketthread**, во втором случае потребуются довольно значительные изменения.

Давайте рассмотрим логику работы во втором случае, так как первый случай банален.

Сначала в основной части программы необходимо перевести вновь созданный сокет в неблокирующий режим:

```
arg    1;
ioctlsocket(socket,fionbio,arg);
```

Внимание, перед закрытием сокета его необходимо будет вернуть в блокирующий режим:

```
arg    0;
ioctlsocket(socket, fionbio, arg);
```

Затем необходимо реализовать возможность сохранения каждого сокета в некоторый массив sockarray. Далее, нужно обеспечить, чтобы поток, который будет заниматься обработкой клиентов, запускался только один раз. Это сделать несложно, зная, что процедура **createthread** возвращает в переменную ссылку на вновь созданный поток, которую и нужно использовать для проверки, был ли создан поток. В потоке обработки нужно внести следующие изменения для использования функции **select**:

Введем новые переменные wfds :tfdset;   integer; tv   ttimeval.

Затем определим основной цикл, в котором будем обрабатывать данные:

```
repeat

until connum>0;
```

В этом цикле первым делом необходимо сформировать структуру wfds, содержащую набор контролируемых сокетов. Для этого мы переносим туда сокеты из массива sockarray:

```
fd_zero(wfds);
for  :=1 to connum do
begin
fd_set(sock[i],wfds);
end;
```

Далее, указываем в структуре tv время задержки для функции **select**:

```
tv.tv_sec    5;
tv.tv_usec    0;
```

Теперь можно вызывать функцию **select** (так как мы следим только за приемом данных, то в качестве writefds, exceptfds мы указываем nil):

```
select(0,@wfds,nil,nil,@tv);
```

Теперь, когда функция **select** возвратит управление переменной wfds, мы будем иметь набор сокетов, для которых необходимо произвести чтение, и можем обработать поступившие данные:

```
if wfds.fd_count=0 then continue;
for  :=0 to wfds.fd_count-1 do
begin
vsocket    wfds.fd_array[i];
//Обработка поступивших данных с сокета vsocket.
end;
```

Условие выхода из основного цикла — отсутствие открытых сокетов в массиве sockarray. Условия закрытия сокета мы оставляем на совести читателя, добавим только лишь, что сокет попадет в обработку **select** и при наступлении события разрыва связи (для обработки можно использовать то, что количество принятых байтов функцией recv будет равно нулю, а также функцию **wsagetlasterror**).

Давайте теперь разберемся, для чего мы указали время ожидания функции **send**, а не сделали его бесконечным? Дело в том,

что в текущей реализации при создании нового сокета он не попадет в обработку **select**, пока не будет установлен функцией **fs_set**, что, естественно, не пройдет в блокирующем режиме, пока **select** не возвратит управление по событию с одним из отслеживаемых клиентов. Установка значения timeout гарантирует, что сокет попадет в обработку независимо от состояния других сокетов.

Какой вариант использовать — поток на каждый сокет, либо один поток обрабатывает все соединения — вопрос, к которому нужно подходить по-разному в каждом конкретном случае. Если в процессе общения с клиентом серверу требуется значительное время на подготовку и обработку данных, то, конечно, следует создать разные потоки, чтобы другим клиентам не пришлось ожидать завершения обработки, а работать параллельно. В противном случае, если обработка минимальна, то создание большого количества процессов окажет влияние в худшую сторону на общую производительность системы.

Помимо функции **select** существует еще два метода работы с асинхронными сокетами:

```
function wsaasyncselect(s: tsocket; hwindow: hwnd;
wmsg: u_int; levent: longint)  integer; stdcall;
```

Эта функция связывает сокет с получением сообщений окна. При вызове этой функции, сообщения о соединении, чтении/записи данных в сокет и закрытии сокета можно обрабатывать в функции обработки сообщений от окна.

```
const

wm_mysocket   wm_user + 1;


type
tform1   class(tform)


private
procedure socket_proc(var msg:tmessage);message
wm_mysocket;


wsaasyncselect(vsocket,form1.handle,wm_
mysocket,fd_accept+fd_read);
```

```
procedure tform1.socket_proc(var msg: tmessage);
begin
if ((msg.msg   wm_mysocket)
and (msg.lparam   fd_accept))
then showmessage('connected');
end;
```

Рассмотрим еще один способ, который пригодится, если у вас нет окна приложения; этот способ основан на системных событиях (events). К сожалению, файл winsock.pas не импортирует соответствующие функции, в результате этого многие программисты пренебрегают возможностями событий. Напишем собственный импорт необходимых процедур:

```
function wsaeventselect(s: tsocket; event: thandle;
levent: longint):integer;stdcall;
external 'ws2_32.dll' name 'wsaeventselect'
function wsawaitformultipleevents(ncount: dword;
lphandles: pwohandlearray;
bwaitall: bool; dwmilliseconds: dword;
falertable:bool):integer;stdcall;
external 'ws2_32.dll' name 'wsawaitformultipleevents'
function wsacreateevent:thandle;stdcall;
external 'ws2_32.dll' name 'wsacreateevent'
function wsaresetevent(event   thandle):bool;stdcall;
external 'ws2_32.dll' name 'wsaresetevent'
function wsaenumnetworkevents(const s   tsocket;
const event   thandle; lpnetworkevents
lpwsanetworkevents)  longint
stdcall;far;
external 'ws2_32.dll' name 'wsaenumnetworkevents'
function wsacloseevent(event   thandle):integer;
stdcall; external 'ws2_32.dll' name 'wsacloseevent'
```

Также нам потребуется описание структуры wsanetworkevents.

```
const
fd_max_events   10;
type
twsanetworkevents   record
lnetworkevents: longint;
ierrorcode: array[0..fd_max_events-1] of integer;
end;
```

```
pwsanetworkevents   ^twsanetworkevents;
lpwsanetworkevents   pwsanetworkevents;
```

Принцип работы с этим набором функций состоит в создании специального объекта типа event, затем происходит связывание этого события с сокетом с помощью функции **wsaeventselect**, в которой также указывается набор отслеживаемых состояний сокета. Один сокет может быть связан только с одним объектом типа event.

Затем, в цикле обработки мы организуем ожидание поступления события от сокета; это реализуется с помощью api функций **waitforsingleobject** — для ожидания одного события либо **waitformultipleobjects** — для ожидания набора событий. При наступлении события функция возвращает управление. Для однозначной идентификации, от какого сокета пришло уведомление, используется функция **wsaenumnetworkevents**, возвращающая структуру типа twsanetworkevents.

```
var
fevent   thandle;
//Создаем серверный сокет

feventclose   wsacreateevent;
wsaeventselect(socket,fevent, fd_close   fd_read
repeat
waitforsingleobject(fevent,infinite);
wsaenumnetworkevents(fsocket,fevent,@ni);
case ni.lnetworkevents of
fd_close:break;
fd_read: begin
receivedata;
end;
end;
wsaresetevent(feventclose);
until false;
wsacloseevent(feventclose);
```

Вне зависимости от того, используем ли мы синхронные или асинхронные сокеты и какие методы выбраны для обработки событий, приходящих от сокетов, при отправке и приеме данных есть один подводный камень, на который попадают все начинающие программисты сетевых приложений.

Рассмотрим отправку данных. Дело в том, что, как мы уже говорили, отправка данных есть, по сути, постановка порции данных в очередь. Мы не можем управлять отдельными пакетами, более того, данные, попавшие в буфер, отправляются не сразу, а могут накапливаться для отправки в дальнейшем одним пакетом. Таким образом, последовательный вызов

```
send(vsocket,@buf1,length(buf1),0);
send(vsocket,@buf2,length(buf2),0);
```

фактически будет идентичен одному вызову **send** с объединенным буфером buf1+buf2.

Таким образом, при приеме данных, хотя мы послали две порции данных, мы получим одну. Совсем другой случай, когда мы посылаем порцию данных, бóльшую, чем буфер сокета, тогда функция **send** отправит только часть данных из указанного буфера, ровно столько, сколько влезло в буфер сокета. Для того чтобы отследить эту ситуацию и отправить необработанную часть буфера, нужно воспользоваться тем, что функция **send** возвращает количество фактически отосланных данных. Функцию **send** нужно запускать в цикле, условием завершения которого будет полная отправка всего буфера.

При чтении из сокета данных мы можем наблюдать как бы «склейку» порций данных либо, наоборот, фрагментацию (не путать с фрагментацией пакетов на уровне tcp/ip). Такие ситуации должна обрабатывать наша программа. Решить проблему можно добавлением сигнатуры признака конца блока данных. Это имеет смысл, если приложения часто обмениваются небольшими блоками данных, где чаще всего возникает эффект «склеивания», но неэффективны при больших объемах, так как сканирование большого буфера на предмет сигнатуры отнимает много времени. Обычно это решается таким способом — в начале каждого пакета добавляется 32-битовое число, определяющее длину порции данных в байтах. Таким образом, принимающая часть, зная размер каждого блока, может распознать «склейку» и фрагментацию.

Об остальных аспектах сетевого программирования с использованием библиотеки **winsock** вы можете узнать из справки «windows sdk» в разделе «windows sockets 2 application program interface».

Порядок выполнения работы

1. Написать сетевое приложение с использованием Winsock API в соответствии с заданным преподавателем вариантом. При этом один компьютер — сервер, другой — клиент.

2. Отладить программу.

3. Произвести обмен данными с соседним компьютером.

4. Изменить направление «клиент — сервер».

5. Еще раз обменяться данными.

6. Закончить работу с сокетами.

7. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Листингов программ.

2. Результатов работы.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Какое существует сетевое программное обеспечение?
2. Охарактеризуйте архитектуру «клиент — сервер».
3. Приведите понятие сокета.
4. Приведите понятие порта.
5. Какие существуют функции для работы с сокетами?
6. Охарактеризуйте синхронные и асинхронные операции ввода-вывода.

Варианты заданий

1. На базе примера написать чат. Программа должна передавать самой себе по WinSocket сообщения в обычном и кодированном виде. Использовать код Цезаря. Суть кода: все буквы сдвинуты на три позиции, то есть: «а» шифруется буквой «г»,

«б»—«д» и так далее, «э»—«а», «ю»—«б», «я»—«в». Аналогично сдвигается английский алфавит.

2. На базе примера написать чат. Программа должна передавать самой себе по WinSocket сообщения с присоединенными к ним файлами (бинарными в общем случае).

3. Написать интернет-игру. Программа должна передавать самой себе по WinSocket координаты точки. Эта точка должна или рисоваться, или должны выводиться ее координаты, либо указывать на ячейку таблицы Excel, либо отображаться каким-либо иным способом.

4. Написать программу, которая следит за использованием соединения и подсчитывает статистику передачи сообщений между двумя пользователями. Программа должна уметь работать на отдельном компьютере. В качестве программ пользователей использовать пример или программы из 1—3 вариантов.

5. Написать распределенную базу данных. Одна программа посылает запросы на получение данных и на сохранение изменений в этих данных. Другая программа работает с таблицей Excel, читает из нее запрашиваемые данные или записывает данные в таблицу.

Таблицу можно не отображать на экране.

6. Написать «защищенную сеть». По нажатию кнопки в диспетчерской программе все указанные соединения должны закрываться. В диспетчерской программе должен быть список открытых соединений. В качестве пользовательских программ использовать пример или программы из 1—3 вариантов.

ЛАБОРАТОРНАЯ РАБОТА № 8. Проектирование программной системы при объектном подходе к программированию

Цель работы: познакомить студентов с методом проектирования системы путем CRC-карт.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Проектирование программных продуктов при объектном подходе» учебной дисциплины «Технология разработки программного обеспечения».

2. Изучить соответствующие разделы в изданиях [1—3, 48, 49, 51].

3. Ознакомиться с разд. 3.6, 4.5 данного пособия.

Теоретическая часть. Основы UML-проектирования

Важным этапом создания программного обеспечения является проектирование. На этом шаге закладывается архитектура системы.

Одним из способов проектирования является метод CRC-карточек. Этот метод проектирования является составляющей UML-проектирования.

Шаг первый. Для первоначального понимания структуры программы строится диаграмма вариантов использования: выявляются акторы (люди или системы, между которыми происходит взаимодействие), прецеденты системы (действия, выполняемые системой для реализации общения акторов).

Пример «Банкомат».

Диаграмма вариантов использования для примера «Банкомат» приведена на рис. Л8.1.

Рис. Л8.1. Диаграмма вариантов использования «Банкомат»

На самом деле прецедентов может быть очень много. Допустим: проверить пароль, контролировать транзакции передачи данных, выдать информацию на экран и т. д.

Эта диаграмма дает понять, что будет делать система, как она будет функционировать. Диаграмма использования бывает также очень полезна для общения с заказчиком — она позволяет показать наиболее значимые действия системы и проверить, правильно ли вы поняли заказчика и значимость отдельных функций для него.

Шаг второй. На этом этапе выявляют классы, которые необходимо будет создать в программе для реализации системы. В случае банкомата это: *клиент*, *банк, служба безопасности банка*, сам *банкомат* и т. д.

Придумать можно много (*таймер, счетчик купюр, карточка* и т. д.).

Далее оформляются CRC-карты. Это листки бумаги 10 × 15. Они разделены на три части и выглядят следующим образом — рис. Л8.2.

На примере того же банкомата — рис. Л8.3.

Название класса	
Действия, которые он выполняет (всегда начинаются с глагола)	Классы, с которыми данный класс обменивается информацией

Рис. Л8.2. Оформление CRC-карты

Клиент	
1. Вставляет карточку в банкомат. 2. Вводит пароль. 3. Указывает тип операции (снять деньги, просмотреть остаток). 4. Вводит сумму. 5. Получает деньги. 6. Вынимает карточку	Банкомат

а

Банкомат	
1. Отображает информацию для клиента. 2. Передает информацию в банк. 3. Отсчитывает купюры. 4. Распечатывает счет	Клиент Банк Служба безопасности банка

б

Служба безопасности банка	
1. Проверяет пароль. 2. Проверяет подлинность карточки. 3. Идентифицирует клиента. 4. Следит за правильностью транзакций операций с деньгами	Банк Банкомат

в

Банк	
1. Проверяет возможность выдачи средств. 2. Сообщает о наличии денег. 3. Выдает информацию об остатке. 4. Хранит информацию о счете клиента	Банкомат Служба безопасности банка

Рис. Л8.3. Примеры CRC-карт

Шаг третий. Для проверки достаточности или избыточности придуманных классов, а также корректности их взаимодействия строится диаграмма взаимодействия (рис. Л8.4).

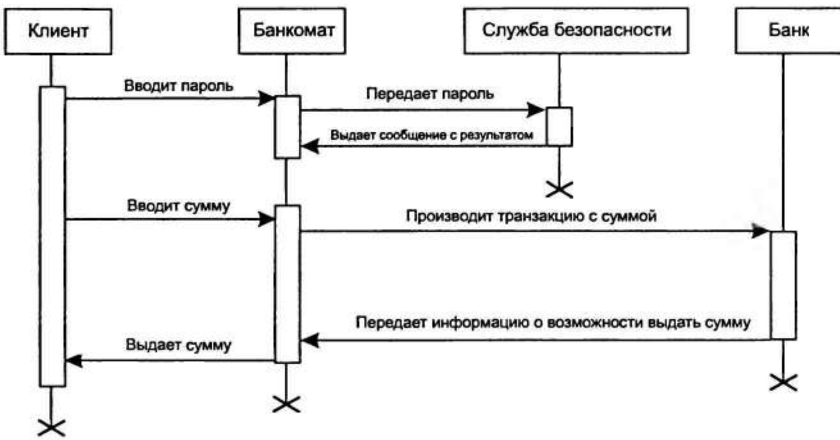


Рис. Л8.4. Диаграмма взаимодействия

Метод CRC-карточек позволяет провести также инсценировку работы системы. Для этого достаточно раздать карточки с классами участникам проекта. После этого начать ролевую игру. Первый участник встает и читает действие, совершаемое его классом. Другие участники, исходя из своих карточек, сообщают об ответной реакции других классов. Если в какой-то момент реакции не последует, то это признак несовершенства проекта системы. Такая игра может подсказать и об избыточности проекта.

Порядок выполнения работы

1. В соответствии с вариантом задания, предложенным преподавателем, определить действующих лиц (акторов) системы.

2. Определить варианты использования системы и описать их в краткой или полной форме (см. разд. 3.6.2).

3. Построить диаграмму вариантов использования системы (использовать MS Office или MS Visio).

4. Определить классы проектируемой системы.

5. Создать CRC-карты для всех классов системы (использовать MS Office или MS Visio).

6. Построить диаграмму взаимодействия (использовать MS Office или MS Visio).
7. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:
1. Постановки задачи.
2. Описания действующих лиц и прецедентов системы.
3. Диаграммы прецедентов.
4. CRC-карты.
5. Диаграммы взаимодействия.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Охарактеризуйте проектирование ПО при объектном подходе.
2. В чем заключается моделирование предметной области при проектировании ПО?
3. Язык UML. Его назначение, преимущества и недостатки.
4. Опишите варианты использования ПО.
5. Перечислите диаграммы в языке UML.
6. Приведите пример диаграммы прецедентов.
7 Приведите пример диаграммы взаимодействия.
8. В чем состоит назначение и использование CRC-карт?

Варианты заданий

1. Заказ билетов в аэропорту.
2. Электронный магазин.
3. Отправка sms.
4. Система охраны частного дома.
5. Система безопасности тюрьмы.
6. Система безопасности полета самолета.

ЛАБОРАТОРНАЯ РАБОТА № 9. Динамические структуры данных

Цель работы: научиться использовать динамические структуры данных при создании программного обеспечения.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Структура и формат данных» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [40, 41, 47].
3. Повторить программирование на C++ (Delphi).
4. Ознакомиться с разд. 3.3 данного пособия.

Теоретическая часть. Статические и динамические данные

При написании программ необходимо постоянно использовать переменные для хранения данных. При обычном объявлении переменной целочисленного массива на языке C++ (здесь и далее в качестве языка выбран C++, но аналогичные структуры данных есть и в других языках высокого уровня):

```
int mas[100];
```

на этапе компиляции программы происходит автоматическое резервирование памяти под эту переменную (рис Л9.1, *а*). Это имеет ряд серьезных недостатков:

- невозможность вернуть компьютеру неиспользуемую память. Например, большой массив будет использоваться только один раз, а память нельзя освободить до окончания работы программы;

- захват памяти, которая вообще может не использоваться программой. Эта ситуация, к сожалению, также не редкость. Зачастую трудно заранее определить размер массива, который понадобится для работы, и под него отводят память «с запасом».

Для решения описанных проблем используются динамические структуры данных. С помощью этого типа данных необходимую память можно выделять в ходе программы и возвращать компьютеру, когда она становится ненужной для работы. Например:

```
float *a;       //Объявляем указатель на переменную a
                //(переменная, в которой хранится
                //адрес ячейки данных)
a=new float;    //Выделяем память под переменную
&a=7.89;        //Присваиваем значение переменной
delete(a);      //Удаляем переменную и возвращаем память
```

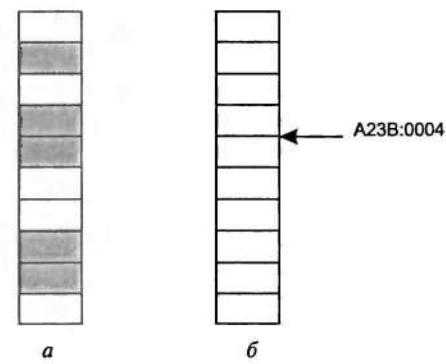


Рис. Л9.1. Распределение памяти при объявлении:
а — статических переменных; *б* — динамических переменных

Указатель — адрес ячейки памяти. При использовании оператора ***new*** компьютер выделяет необходимую память, и адрес этой ячейки памяти записывается в переменную-«указатель» (в примере — это переменная ***a***).

Создание списка

Список — последовательность ячеек памяти, хранящих элементы массива данных. Каждая ячейка такого списка состоит

как минимум из двух частей: в первой хранится элемент массива, во второй — указатель на следующую такую же ячейку памяти (рис. Л9.2).

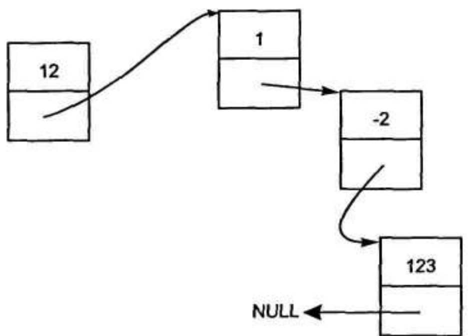


Рис. Л9.2. Пример динамического списка

Ниже приводится код программы для работы со списком на языке С++. Используя его, познакомимся с технологией создания списка.

Для работы со списками, как говорилось выше, используется структура, состоящая как минимум из двух элементов:

```
struct spisok
{
  int elem;          //Данные
       spisok *next; //Указатель, адрес следующего
элемента
 };
```

В отличие от простого объявления переменной, для динамических данных добавляется значение «звездочка» (оно означает, что это адрес памяти, где расположен элемент, но не он сам). Чтобы в ходе программы получить значение элемента по известному адресу или, наоборот, присвоить значение, используется знак ***& (&a=56)***.

1. Далее в программе следует строка:

```
spisok *l;
```

Это адрес первого элемента списка. Его необходимо знать всегда.

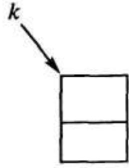
2. Рассмотрим построчно процедуру создания списка (для объяснения строки пронумерованы):

```
void FormSpisok()
{
(1)  spisok *m,*k;
(2)  int
(3)  randomize();
(4)  k=new spisok;
(5)  k->elem=random(5);
(6)  l=k;
(7)  for (i=1;i<10;
  {
(8)   m=new spisok;
(9)   m->elem=random(5);
(10)   k->next=m;
(11)   k=m;
  }
(12) k->next=NULL;
}
```

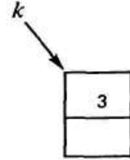
В строках (1) и (2) объявляются служебные переменные.

Строка (3) — инициализация генератора случайных чисел для последующего присвоения произвольных значений элементам списка.

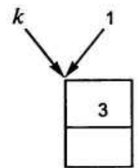
(4) k=new spisok; Выделяется ячейка память для переменной типа «список», и ее адрес сохраняется в переменной ***k***:



(5) k->elem=random(5); Присваивается значение элементу списка:

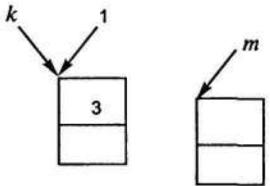


(6) l=k; Указатель на первый элемент списка запоминается в переменной ***l***:

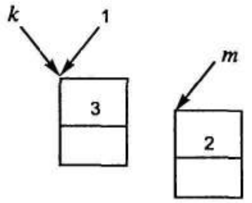


(7) for (i=1;i<10;i++). Для создания списка из 10 элементов используется цикл.

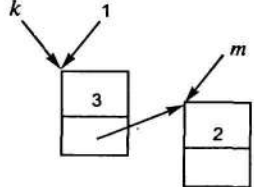
(8) m=new spisok; Выделяется еще один участок памяти под переменную spisok. Адрес памяти запоминается в переменную ***m***:



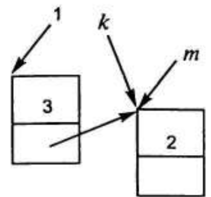
(9) m->elem=random(5); Заносим значение в переменную:



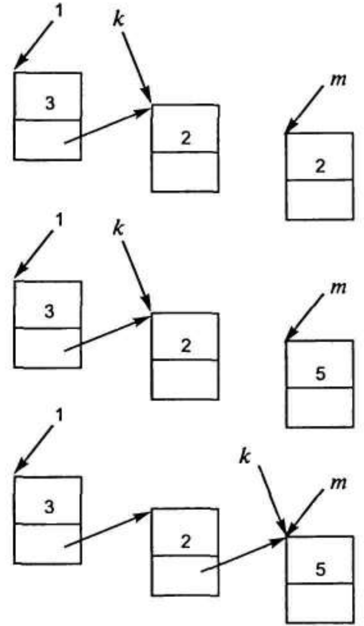
(10) k->next=m; Связываем существующие две ячейки:



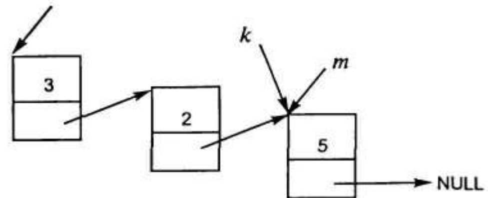
(11) k=m; Для создания остальных элементов меняем значение указателя ***k***, как показано на рисунке:



Далее действия повторяются в цикле:



(12) k->next=NULL; Для задания значения «последующий» для последнего элемента используется значение ***NULL***:



Пример программы по работе со списками на языке C++

```
#include<conio.h>
#include<stdlib.h>
#include<iostream.h>
#include<stdio.h>

     struct spisok
      {
      int elem;
      spisok *next;
      };
  spisok *l;

//Процедура создания списка
void FormSpisok()
{
 spisok *m,*k;
 int i;
 randomize();
     k=new spisok;
     k->elem=random(5);
     l=k;
     for (i=1;i<10;
 {
 m=new spisok;
 m->elem=random(5);
 k->next=m;
 k=m;
 }
     k->next=NULL;


void vivodSpisok()
{
 spisok *m;
     m=l;
     while (m!=NULL)
 {
 cout<<m->elem<<' ';
 m=m->next;
 }
```

```
void Vstavka()
{
 spisok *m,*k,*r;
   int i,a,s;
   m=l;
   cout<<endl<<"Введите позицию, после которой нужно
вставить новый элемент:
   cin>>a;
   cout<<"Введите значение элемента:
   cin>>s;

     for (i=1;i<a-1;     m=m->next;
     k=new spisok;
     k->elem=s;
     r=m->next;
     m->next=k;
     k->next=r;


void Del()
{
 spisok *m,*k,*r;
   int i,a;
   cout<<endl<<"Vvedite posiciju:
   cin>>a;

if (a==0)
       {
       m=l;
       l=l->next;
       delete(m);
       }
else
   {
   m=l;
     for(i=1;i<a-1;     m=m->next; //Поиск позиции
                                   // удаления
     k=m->next;
     r=k->next;

     m->next=r;
     delete(k);
```

```
void main()
{
clrscr();
FormSpisok();
vivodSpisok();
Vstavka();
vivodSpisok();
Del();
vivodSpisok();
getch();
}
```

Пример программы на Delphi по работе со списками

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs,
  StdCtrls, Buttons;

type
  TForm1   class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    vivod: TEdit;
    BitBtn1: TBitBtn;
    Label1: TLabel;
    Edit2: TEdit;
    Label2: TLabel;
    Edit3: TEdit;
    Label3: TLabel;
    Edit1: TEdit;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
  private
    { Private declarations }
  public
```

```
    { Public declarations
  end;

var
  Form1: TForm1;

implementation

{$R  .DFM}
const n=10;
type spisok=^zveno;
     zveno   record
      elem:integer;
      next:spisok;
             end;
var l:spisok;

procedure TForm1.Button1Click(Sender: TObject);
var m,k:spisok;
     :integer;
begin
randomize;
     new(k);
     k^.elem:=random(5);
     l:=k;
     for  :=1 to n-1 do begin
 new(m);
 m^.elem:=random(5);
 k^.next:=m;
 k:=m;
    end;
     k^.next:=nil;
end;

procedure TForm1.Button2Click(Sender: TObject);
var m:spisok;
    str:string[100];
begin
     str:=''              //Добавил
     m:=l;
     while m<>nil do begin
        str:=str+IntToStr(m^.elem)+' ';
// write(m^.elem,  ');
 m:=m^.next;
```

```
    end;
     vivod.Text:=str;
end;

procedure TForm1.Button3Click(Sender: TObject);
var m,k,r:spisok;
     :integer;
begin
   m:=l;
     for  :=1 to StrToInt(Edit3.Text)-1 do m:=m^.next;
//Поиск позиции вставки

     new(k);
     k^.elem:=StrToInt(Edit2.Text); //Создали элемент

     r:=m^.next;
     m^.next:=k;
     k^.next:=r;
end;

procedure TForm1.Button4Click(Sender: TObject);
var m,k,r:spisok;
     :integer;
begin
if Edit1.Text='0' then
     begin
       m:=l;
       l:=l^.next;
       dispose(m);
       end
else begin
   m:=l;
     for i:=1 to StrToInt(Edit1.Text)-1 do m:=m^.next;
//Поиск позиции удаления
     k:=m^.next;
     r:=k^.next;

     m^.next:=r;
     dispose(k);
     end;
end;

end.
```

Порядок выполнения работы

1. В соответствии с вариантом задания написать программу обработки списка на языке C++ или Delphi.

2. Отладить программу.

3. Получить результат.

4. Показать текст программы и результат ее работы преподавателю.

5. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Блок-схемы программы.
3. Текста программы на языке программирования.
4. Результатов.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Приведите классификацию данных.
2. Что такое простые типы данных?
3. Что такое статические и полустатические структуры данных?
4. Какие существуют динамические структуры данных?
5. Как осуществляется размещение переменных в памяти компьютера?

Варианты задания

1. Создать список. Поменять местами максимальный и минимальный элементы.

2. Создать список. Удалить из него повторяющиеся элементы.

3. Создать два списка. Создать 3-й список, состоящий из элементов, которые есть как в первом, так и во втором списке.

4. Создать два списка. Создать 3-й список, объединяющий первый и второй списки. Удалить максимальный элемент в новом списке.

5. Создать список и отсортировать его по убыванию.

6. Создать два списка одинаковой длины. Произвести поэлементное вычитание из первого списка элементов второго списка.

7. Создать два списка. Создать 3-й список, состоящий из неповторяющихся элементов первых двух списков.

8. Создать список. Вставить в него после максимального элемента копию минимального элемента.

9. Создать список. После каждого отрицательного элемента вставить элемент, равный 0.

10. Создать список. Посчитать сумму всех элементов. Полученный результат вставить после минимального элемента списка.

11. Создать список. Удалить из него все отрицательные элементы.

12. Создать два списка. Создать 3-й список, состоящий из положительных элементов двух первых списков.

ЛАБОРАТОРНАЯ РАБОТА № 10. Объектно-ориентированное программирование (ООП)

Цель работы: познакомиться с принципами объектно-ориентированного программирования.

Лабораторная работа рассчитана на 4 академических часа.

Подготовка к лабораторной работе

1. Ознакомиться с лекционным материалом по теме «Объектно-ориентированное программирование» учебной дисциплины «Технология разработки программного обеспечения».
2. Изучить соответствующие разделы в изданиях [1, 40, 41].
3. Повторить программирование на C++ (Delphi).
4. Ознакомиться с разд. 6.1 данного пособия.

Теоретическая часть. Основные понятия

Класс — абстрактный тип данных, включающий в себя свойства объекта (поля) и методы. Класс позволяет упростить процесс программирования, так как человеку проще представлять любой объект из реальности, обладающий некоторыми характеристиками (свойствами) и действиями, которые может совершать объект или которые можно совершать над ним.

Класс — это тип данных. ***Объект класса*** — переменная типа «класс».

Из определения класса следует первое свойство ООП — ***инкапсуляция***. Инкапсуляция данных означает, что данные являются не глобальными — доступными всей программе, а локальными — доступными только малой ее части. Инкапсуляция автоматически подразумевает защиту данных. Для этого в структуре `class` используется спецификатор раздела `private`, содержа-

щий данные и методы, доступные только для самого класса. Если данные и методы содержатся в разделе public, они доступны извне класса. Раздел protected содержит данные и методы, доступные из класса и любого его производного класса:

```
class Stek
  {
  private:
   spisok *l;
  public:
  Stek();
  void PrintSpisok();
  void AddElem(int pos,int element);
  void DelElem(int pos);
  };
```

Вторым по значимости понятием является **наследование**. Новый (производный) класс может быть создан на основе уже имеющегося (базового) класса. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки. Если объект наследует свои свойства от одного родителя, то говорят об *одиночном наследовании*. Если же объект наследует атрибуты от нескольких базовых классов, то говорят о *множественном наследовании*:

```
class PotomokSteka:  public Stek

     private:
  spisok *z;
     public:
  PotomokSteka();
 void SortSpisok();
     };
```

Третьим по значимости понятием является ***полиморфизм***. Полиморфной называется функция, независимо определенная в каждой из группы производных классов и имеющая в них общее имя. Полиморфная функция обладает тем свойством, что при отсутствии полной информации о том, объект какого из произ-

водных классов в данный момент обрабатывается, она тем не менее корректно вызывается в том виде, в каком она была определена для данного конкретного класса.

Теперь поговорим о реализации классов в программе. Для инициализации объекта класса принято создавать дополнительный метод, называемый ***конструктором.*** Эта функция должна называться так же, как и класс. Возможно создать несколько конструкторов для разных методов первоначальной инициализации объекта:

```
class Stek
  {
  private:
   spisok *l;
  public:
  Stek();       //По умолчанию создается список
                //из 10 элементов
  Stek(int q); //Количество элементов в списке
                //передается в виде параметра


Stek::Stek()
{
 spisok *m,*k;
 int
// randomize();
     k=new spisok;
     k->elem=random(5);
     l=k;
     for (i=1;i<10;
 {
 m=new spisok;
 m->elem=random(5);
 k->next=m;
 k=m;
 }
     k->next=NULL;


Stek::Stek(int g)
{
 spisok *m,*k;
 int i;
```

```
    randomize();
      k=new spisok;
      k->elem=random(5);
      l=k;
      for     ;i<g;
 {
 m=new spisok;
 m->elem=random(5);
 k->next=m;
 k=m;
 }
      k->next=NULL;


void main()
{

Stek st1,st2(7); //Создание объектов класса в программе
```

Существует также понятие ***деструктора*** класса. Это метод класса, предназначенный для уничтожения объекта класса. Допустим, он может быть полезен, если нужно удалить динамический список, освободив память:

```
class Stek
  {
  private:
   spisok *l;
  public:
  Stek();
  Stek(int q);
  ~Stek();  //Деструктор


Stek::~Stek()
{
 spisok *m;
      m=l;
      while (m!=NULL)
 {
  l=m->next;
```

```
  delete(m);
  m=l;

  cout<<"Stek was deleted";
```

Работу деструктора можно увидеть, например, если создать указатель на объект класса «динамический список». Тогда при выполнении операции ***delete*** будет вызван деструктор класса:

```
void main()
{

Stek *st3;
st3=new Stek;
cout<<endl<<"Creat  new steck:";
st3->PrintSpisok();
delete(st3);
```

Пример программы на языке C++

```
#include<conio.h>
#include<stdlib.h>
#include<iostream.h>
#include<stdio.h>

     struct spisok
      {
      int elem;
      spisok *next;
      };

class Stek
  {
  private:
   spisok
  public:
  Stek();
  Stek(int q);
  ~Stek();
  void PrintSpisok();
```

```
  void AddElem(int pos,int element);
  void DelElem(int pos);
  };

Stek::Stek()
{
 spisok *m,*k;
 int i;
// randomize();
     k=new spisok;
     k->elem=random(5);
     l=k;
     for (i=1;i<10;
 {
 m=new spisok;
 m->elem=random(5);
 k->next=m;
 k=m;
 }
     k->next=NULL;


Stek::Stek(int g)
{
 spisok *m,*k;
 int i;
// randomize();
     k=new spisok;
     k->elem=random(5);
     l=k;
     for (i=1;i<g;
 {
 m=new spisok;
 m->elem=random(5);
 k->next=m;
 k=m;
 }
     k->next=NULL;


void Stek::PrintSpisok()
{
 spisok *m;
```

```
      m=l;
      while (m!=NULL)
  {
  cout<<m->elem<<'
  m=m->next;
  }
 cout<<endl;
 }

 Stek::~Stek()
 {
  spisok *m;
      m=l;
      while (m!=NULL)
  {
   l=m->next;
   delete(m);
   m=l;
  }
   cout<<"Stek was deleted";


 void Stek::AddElem(int pos,int element)
 {
  spisok *m,*k,*r;
    int
    m=l;
      for (i=1;i<pos-1;     m=m->next;
      k=new spisok;
      k->elem=element;
      r=m->next;
      m->next=k;
      k->next=r;


 void Stek::DelElem(int pos)
 {
  spisok *m,*k,*r;
    int i,a;

 if (pos==0)
        {
        m=l;
```

```
        l=l->next;
        delete(m);
        }
else
   {
   m=l;
     for(i=1;i<pos-1;      m=m->next; //Поиск позиции
                                      //удаления
     k=m->next;
     r=k->next;

     m->next=r;
     delete(k);



class PotomokSteka: public Stek
     {
     private:
  spisok *z;
     public:
  PotomokSteka();
     };

PotomokSteka::PotomokSteka()
{
 spisok *m,*k;
 int i;
 randomize();
     k=new spisok;
     k->elem=random(5);
     z=k;
     for (i=1;i<10;
 {
 m=new spisok;
 m->elem=random(5);
 k->next=m;
 k=m;
 }
     k->next=NULL;
```

```
void main()
{
clrscr();

Stek st1,st2(7);
cout<<endl<<"st1:
st1.PrintSpisok();
cout<<endl<<"st2:
st2.PrintSpisok();

cout<<endl<<"Add element in 1-st spisok"<<endl;
st1.AddElem(3,1000);
st1.PrintSpisok();

cout<<endl<<"Delete element from 1-st spisok"<<endl;
st1.DelElem(2);
st1.PrintSpisok();

Stek *st3;
st3=new Stek;
cout<<endl<<"Creat  new steck:";
st3->PrintSpisok();
delete(st3);

PotomokSteka pst4;
cout<<endl<<endl<<"pst4:
pst4.PrintSpisok();

getch();
}
```

Порядок выполнения работы

1. В соответствии с вариантом задания из лабораторной работы № 9 написать программу обработки списка на языке C++ или Delphi.

2. Отладить программу.

3. Получить результат.

4. Показать текст программы и результат ее работы преподавателю.

5. Сдать и защитить работу.

Защита отчета по лабораторной работе

Отчет по лабораторной работе должен состоять из:

1. Постановки задачи.
2. Блок-схемы программы.
3. Текста программы на языке программирования.
4. Результата.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов (на экране монитора), демонстрации полученных навыков и ответах на вопросы преподавателя.

Контрольные вопросы

1. Опишите понятия объектно-ориентированного программирования:
    - класс;
    - объект;
    - инкапсуляция;
    - наследование;
    - полиморфизм.
2. Что такое поля и методы классов?
3. Как производится объявление переменных и методов public, protected, private?
4. Что такое конструктор и деструктор?

Приложения

Приложение 1

Варианты заданий

Лабораторные работы № 1—5 выполняются для одного и того же варианта.

1. Разработать программный модуль «Учет успеваемости студентов». Программный модуль предназначен для оперативного учета успеваемости студентов в сессию деканом, заместителями декана и сотрудниками деканата. Сведения об успеваемости студентов должны храниться в течение всего срока их обучения и использоваться при составлении справок о прослушанных курсах и приложений к диплому.

2. Разработать программный модуль «Личные дела студентов». Программный модуль предназначен для получения сведений о студентах сотрудниками деканата, профкома и отдела кадров. Сведения должны храниться в течение всего срока обучения студентов и использоваться при составлении справок и отчетов.

3. Разработать программный модуль «Решение комбинаторно-оптимизационных задач». Модуль должен содержать алгоритмы поиска цикла минимальной длины (задача коммивояжера), поиска кратчайшего пути и поиска минимального связывающего дерева.

4. Разработать приложение Windows «Органайзер». Приложение предназначено для записи, хранения и поиска адресов и те-

лефонов физических лиц и организаций, а также расписания, встреч и др. Приложение предназначено для любых пользователей компьютера.

5. Разработать приложение Windows «Калькулятор». Приложение предназначено для любых пользователей и должно содержать все арифметические операции (с соблюдением приоритетов) и желательно (но не обязательно) несколько математических функций.

6. Разработать программный модуль «Кафедра», содержащий сведения о сотрудниках кафедры (ФИО, должность, ученая степень, дисциплины, нагрузка, общественная работа, совместительство и др.). Модуль предназначен для использования сотрудниками отдела кадров и деканата.

7. Разработать программный модуль «Лаборатория», содержащий сведения о сотрудниках лаборатории (ФИО, пол, возраст, семейное положение, наличие детей, должность, ученая степень). Модуль предназначен для использования сотрудниками профкома и отдела кадров.

8. Разработать программный модуль «Автосервис». При записи на обслуживание заполняется заявка, в которой указываются ФИО владельца, марка автомобиля, вид работы, дата приема заказа и стоимость ремонта. После выполнения работ распечатывается квитанция.

9. Разработать программный модуль «Учет нарушений правил дорожного движения». Для каждой автомашины (и ее владельца) в базе хранится список нарушений. Для каждого нарушения фиксируется дата, время, вид нарушения и размер штрафа. При оплате всех штрафов машина удаляется из базы.

10. Разработать программный модуль «Картотека агентства недвижимости», предназначенный для использования работниками агентства. В базе содержатся сведения о квартирах (количество комнат, этаж, метраж и др.). При поступлении заявки на обмен (куплю, продажу) производится поиск подходящего варианта. Если такого нет, клиент заносится в клиентскую базу и оповещается, когда вариант появляется.

11. Разработать программный модуль «Картотека абонентов АТС». Картотека содержит сведения о телефонах и их владельцах. Фиксирует задолженности по оплате (абонентской и повре-

менной). Считается, что повременная оплата местных телефонных разговоров уже введена.

12. Разработать программный модуль «Авиакасса», содержащий сведения о наличии свободных мест на авиамаршруты. В базе должны содержаться сведения о номере рейса, экипаже, типе самолета, дате и времени вылета, а также стоимости авиабилетов (разного класса). При поступлении заявки на билеты программа производит поиск подходящего рейса.

13. Разработать программный модуль «Книжный магазин», содержащий сведения о книгах (автор, название, издательство, год издания, цена). Покупатель оформляет заявку на нужные ему книги, если таковых нет, он заносится в базу и оповещается, когда нужные книги поступают в магазин.

14. Разработать программный модуль «Автостоянка». В программе содержится информация о марке автомобиля, его владельце, дате и времени въезда, стоимости стоянки, скидках, задолженности по оплате и др.

15. Разработать программный модуль «Кадровое агентство», содержащий сведения о вакансиях и резюме. Программный модуль предназначен как для поиска сотрудника, отвечающего требованиям руководителей фирмы, так и для поиска подходящей работы.

Примечание. При разработке программы не ограничиваться функциями, приведенными в варианте, добавить несколько своих функций. Обязательно использование структурного и модульного подходов к программированию. Желательно использование объектного подхода.

Приложение 2

Пример разработки технического задания на программный продукт

Министерство образования Российской Федерации

Московский государственный институт электронной техники
(технический университет)

Кафедра информатики и программного обеспечения
вычислительных систем

УТВЕРЖДАЮ
Зав. кафедрой ИПОВС,
д.т.н., проф.________Гагарина Л. Г
«____»__________2007 г.

ПРОГРАММА СОРТИРОВКИ ОДНОМЕРНОГО МАССИВА

Техническое задание на лабораторную работу

Листов 3

Руководитель, к.т.н., доцент_________Петров А. А.
Исполнитель, студент гр. МП 33_____Власов С. Е.

МОСКВА, 2007

Рис. П2.1. Пример оформления титульного листа технического задания на учебный программный продукт

1. Введение

Настоящее техническое задание распространяется на разработку программы сортировки одномерного массива методами пузырька, прямого выбора, Шелла и быстрой сортировки, предназначенной для использования школьниками старших классов при изучении курса школьной информатики.

2. Основание для разработки

2.1. Программа разрабатывается на основе учебного плана кафедры «Информатика и программное обеспечение вычислительных систем».

2.2. Наименование работы:

«Программа сортировки одномерного массива».

2.3. Исполнитель: компания BestSoft.

2.4. Соисполнители: нет.

3. Назначение

Программа предназначена для использования школьниками при изучении темы «Обработка одномерных массивов» в курсе «Информатика».

4. Требования к программе или программному изделию

4.1. Требования к функциональным характеристикам

4.1.1. Программа должна обеспечивать возможность выполнения следующих функций:

- ввод размера массива и самого массива;
- хранение массива в памяти;
- выбор метода сортировки;
- вывод текстового описания метода сортировки;
- вывод результата сортировки.

4.1.2. Исходные данные:

- размер массива, заданный целым числом;
- массив.

4.1.3. Организация входных и выходных данных

Входные данные поступают с клавиатуры.

Выходные данные отображаются на экране и при необходимости выводятся на печать.

4.2. Требования к надежности

Предусмотреть контроль вводимой информации.

Предусмотреть блокировку некорректных действий пользователя при работе с системой.

4.3. Требования к составу и параметрам технических средств.

Система должна работать на IBM-совместимых персональных компьютерах.

Минимальная конфигурация:

- тип процессора. Pentium и выше;
- объем оперативного запоминающего устройства 32 Мб и более;
- объем свободного места на жестком диске 40 Мб.

Рекомендуемая конфигурация:

- тип процессора. Pentium II 400;
- объем оперативного запоминающего устройства 128 Мб;
- объем свободного места на жестком диске 60 Мб.

4.4. Требования к программной совместимости.

Программа должна работать под управлением семейства операционных систем Win 32 (Windows 95/98/2000/ME/XP и т. п.).

5. Требования к программной документации

5.1. Разрабатываемые программные модули должны быть самодокументированы, т. е. тексты программ должны содержать все необходимые комментарии.

5.2. Разрабатываемая программа должна включать справочную информацию о работе программы, описания методов сортировки и подсказки учащимся.

5.3. В состав сопровождающей документации должны входить:

5.3.1. Пояснительная записка на пяти листах, содержащая описание разработки.

5.3.2. Руководство пользователя.

Приложение 3

Пример технического задания на разработку

«Утверждаю»

Профессор кафедры ВС

_____________(Иванов И. И.)

«___» ________________ 200 г.

Техническое задание

на разработку «Модуля автоматизированной системы оперативно-диспетчерского управления теплоснабжением корпусов Московского института»

Москва, 200_

1. Введение

Работа выполняется в рамках проекта «Автоматизированная система оперативно-диспетчерского управления электротеплоснабжением корпусов Московского института».

2. Основание для разработки

2.1. Основанием для данной работы служит договор № 1234 от 10 марта 2003 г.

2.2. Наименование работы:

«Модуль автоматизированной системы оперативно-диспетчерского управления теплоснабжением корпусов Московского института».

2.3. Исполнители: ОАО «Лаборатория создания программного обеспечения».

2.4. Соисполнители: нет.

3. Назначение разработки

Создание модуля для контроля и оперативной корректировки состояния основных параметров теплообеспечения корпусов Московского института.

4. Технические требования

4.1. Требования к функциональным характеристикам.

4.1.1. Состав выполняемых функций.

Разрабатываемое ПО должно обеспечивать:

- сбор и анализ информации о расходовании тепла, горячей и холодной воды по данным теплосчетчиков SA-94 на всех тепловых выходах;
- сбор и анализ информации с устройств управления системами воздушного отопления и кондиционирования типа РТ1 и РТ2 (разработки кафедры СММЭ и ТЦ);
- предварительный анализ информации на предмет нахождения параметров в допустимых пределах и сигнализирование при выходе параметров за пределы допуска;
- выдачу рекомендаций по дальнейшей работе;

- отображение текущего состояния по набору параметров — циклически постоянно (режим работы круглосуточный), при сохранении периодичности контроля прочих параметров;
- визуализацию информации по расходу теплоносителя:
  - — текущую, аналогично показаниям счетчиков;
  - — с накоплением за прошедшие сутки, неделю, месяц — в виде почасового графика для информации за сутки и неделю;
  - — суточный расход — для информации за месяц.

Для устройств управления приточной вентиляцией текущая информация должна содержать номер приточной системы и все параметры, выдаваемые на собственный индикатор.

По отдельному запросу осуществляются внутренние настройки.

В конце отчетного периода система должна архивировать данные.

4.1.2. Организация входных и выходных данных.

Исходные данные в систему поступают в виде значений с датчиков, установленных в помещениях института. Эти значения отображаются на компьютере диспетчера. После анализа поступившей информации оператор диспетчерского пункта устанавливает необходимые параметры для устройств, регулирующих отопление и вентиляцию в помещениях. Возможна также автоматическая установка некоторых параметров для устройств регулирования.

Основной режим использования системы — ежедневная работа.

4.2. Требования к надежности.

Для обеспечения надежности необходимо проверять корректность получаемых данных с датчиков.

4.3. Условия эксплуатации и требования к составу и параметрам технических средств.

Для работы системы должен быть выделен ответственный оператор.

Требования к составу и параметрам технических средств уточняются на этапе эскизного проектирования системы.

4.4. Требования к информационной и программной совместимости.

Программа должна работать на платформах Windows 98/NT/2000.

4.5. Требования к транспортировке и хранению.

Программа поставляется на лазерном носителе информации. Программная документация поставляется в электронном и печатном виде.

4.6. Специальные требования:

- программное обеспечение должно иметь дружественный интерфейс, рассчитанный на пользователя (в плане компьютерной грамотности) квалификации;
- ввиду объемности проекта задачи предполагается решать поэтапно, при этом модули ПО, созданные в разное время, должны предполагать возможность наращивания системы и быть совместимы друг с другом, поэтому документация на принятое эксплуатационное ПО должна содержать полную информацию, необходимую для работы программистов с ним;
- язык программирования — по выбору исполнителя, должен обеспечивать возможность интеграции программного обеспечения с некоторыми видами периферийного оборудования (например, счетчик SA-94 и т. п.).

5. Требования к программной документации

Основными документами, регламентирующими разработку будущих программ, должны быть документы Единой Системы Программной Документации (ЕСПД): руководство пользователя, руководство администратора, описание применения.

6. Технико-экономические показатели

Эффективность системы определяется удобством использования системы для контроля и управления основными параметрами теплообеспечения помещений Московского института, а также экономической выгодой, полученной от внедрения аппаратно-программного комплекса.

7. Порядок контроля и приемки

После передачи Исполнителем отдельного функционального модуля программы Заказчику последний имеет право тестировать модуль в течение 7 дней. После тестирования Заказчик дол-

жен принять работу по данному этапу или в письменном виде изложить причину отказа принятия. В случае обоснованного отказа Исполнитель обязуется доработать модуль.

8. Календарный план работ

№ этапа	Название этапа	Сроки этапа	Чем заканчивается этап
1	Изучение предметной области. Проектирование системы. Разработка предложений по реализации системы	01.02.200_—28.02.200_	Предложения по работе системы. Акт сдачи-приемки
2	Разработка программного модуля по сбору и анализу информации со счетчиков и устройств управления. Внедрение системы для одного из корпусов МИЭТ	01.03.200_-31.08.200_	Программный комплекс, решающий поставленные задачи для пилотного корпуса МИЭТ. Акт сдачи-приемки
3	Тестирование и отладка модуля. Внедрение системы во всех корпусах МИЭТ	01.09.200_—30.12.200_	Готовая система контроля теплообеспечения МИЭТ, установленная в диспетчерском пункте. Программная документация. Акт сдачи-приемки работ

Руководитель работ Сидоров С. В.

Приложение 4

Пример эскизного проекта

УТВЕРЖДАЮ

Руководитель (заказчика ИС)
Личная подпись ________ Расшифровка подписи ________________
Печать
Дата «___»______________ 2004 г.

УТВЕРЖДАЮ

Руководитель (разработчика ИС)
Личная подпись________ Расшифровка подписи________________
Печать
Дата «___»______________ 2004 г.

Эскизный проект на создание
информационной системы

Система Управления Базой Данных
(наименование вида ИС)

БИБЛИОТЕЧНЫЙ ФОНД РОССИЙСКОЙ ФЕДЕРАЦИИ
(наименование объекта информатизации)

СУБД «Библиотека»
(сокращенное наименование ИС)

На 8 листах

Действует с «___»______________ 2004 г.

Содержание

Содержание 361

Ведомость эскизного проекта 362

Пояснительная записка к эскизному проекту 363

Общие положения. 363

Основные технические решения 363

Решения по структуре системы 363

Решения по режимам функционирования, работы системы 365

Решения по численности, квалификации и функциям персонала АС. 365

Состав функций комплексов задач, реализуемых системой 365

Решения по составу программных средств, языкам деятельности, алгоритмам процедур и операций и методам их реализации 366

Источники разработки 367

Ведомость эскизного проекта

На предыдущих стадиях разработки СУБД «Пенсионный Фонд» были составлены и утверждены следующие документы:

- Техническое задание на создание информационной системы СУБД «Пенсионный Фонд», разработанное на основании ГОСТ 34.602—89 на написание ТЗ на автоматизированные системы управления от 01.01.1990 г.

Пояснительная записка к эскизному проекту

Общие положения

Данный документ является эскизным проектом на создание Системы Управления Базой Данных для Библиотечного Фонда Российской Федерации (СУБД «Библиотека»).

Перечень организаций, участвующих в разработке системы, сроки и стадии разработки, а также ее цели и назначение указаны в техническом задании на создание информационной системы.

Основные технические решения

Решения по структуре системы

СУБД «Библиотека» будет представлять собой персональную систему управления локальной базой данных, работающей на одном компьютере.

Система будет управлять реляционной базой данных, представляющей собой набор связанных между собой таблиц в формате Paradox, доступ к которым осуществляется с помощью ключей или индексов. Сведения в одной таблице могут отражать сведения из другой, и при изменении сведений в первой таблице эти изменения немедленно отображаются во второй. Таким образом будет достигнута непротиворечивость данных.

Общая структура базы данных:

- Анкеты организации, которые зарегистрированы в данном ПФ:
  - Тип предприятия (Российская организация, Физическое лицо, Иностранная организация, Обособленное подразделение).
  - Вид предприятия (Адвокаты, Бюджетное, Единый налог 6 %, Единый налог 15 %, СельхозПродукция, Службы занятости, Фермерское хозяйство, Прочее).
  - Регистрационный номер работодателя в ПФР (3 — 3 — 6).
  - Свидетельство: серия, номер.
  - Дата выдачи свидетельства (число_месяц_год).
  - ИНН.
  - КПП.

- Наименование.
- Юридический адрес:
  - Почтовый индекс.
  - Регион.
  - Район.
  - Город.
  - Населенный пункт.
  - Улица.
  - Дом.
  - Корпус.
  - Квартира.
- Адрес постоянно действующего органа (при отличии от юридического).

- Анкеты сотрудников этих организаций:
  - Фамилия.
  - Имя.
  - Отчество.
  - Пол (М/Ж).
  - Дата рождения (Дата).
  - Страховой номер.
  - Место рождения (Страна, Регион, Район, Город, Населенный пункт).
    Гражданство.
  - Адрес регистрации (Страна, Почтовый индекс, Регион, Район, Город, Населенный пункт, Улица, Дом, Корпус, Квартира).
  - Адрес места жительства фактический (Страна, Почтовый индекс, Регион, Район, Город, Населенный пункт, Улица, Дом, Корпус, Квартира).
  - Телефон домашний.
  - Телефон служебный.
  - Документ (Удостовер. личность).
  - Дата выдачи (Дата).
  - Кем выдан ().
  - Дата заполнения (Дата).
  - ИНН.

- Сведения о стаже сотрудников этих организаций:
  - Страховой номер.
  - Фамилия.

— Имя.
— Отчество.
— Дата рождения.
Территориальные условия проживания на
— Таблица периодов работы со следующей структурой:
  - Начало периода (дата).
  - Конец периода (дата).
  - Вид деятельности (работа, служба соцстрах, уход-дети, безр, реабилит, уход-инвд, профзаб, пересмотр).
  - Наименование организации.
  - Должность.
  - Территориальные условия.

Решения по режимам функционирования, работы системы

СУБД «Библиотека» будет функционировать в однопользовательском режиме, а также будет способна:

- просматривать записи базы данных (в том числе и при помощи фильтров);
- добавлять новые записи;
- удалять записи;
- при входе в систему будет запрашиваться пароль.

Решения по численности, квалификации и функциям персонала АС

Указанные решения должны удовлетворять требованиям, приведенным в техническом задании на разработку системы.

Состав функций комплексов задач, реализуемых системой

Автоматизированная система должна выполнять следующие функции:

- сделать запись о пенсионном удостоверении;
- удалить информацию о пенсионном удостоверении;
- выдать справку о всех пенсионных удостоверениях;
- зарегестрировать новое предприятие в ПФ РФ;
- удалить предприятие из базы данных;
- выдать справку обо всех предприятиях, зарегистрированных в ПФ РФ;

- подсчитать пенсию для работников предприятий на основании стажа;
- выдать справку о пенсионных накоплениях работника.

Решения по составу программных средств, языкам деятельности, алгоритмам процедур и операций и методам их реализации

Для реализации АС будет использоваться среда программирования Boland Delphi 7.0 и язык программирования Object Pascal.

Для подсчета пенсии будет использоваться следующий алгоритм.

Вначале определяется стажевый коэффициент пенсионера. Он полагается равным 0,55 за общий трудовой стаж до текущей даты не менее 25 лет мужчинам и 20 лет женщинам. За каждый полный год стажа сверх указанного стажевый коэффициент увеличивается на 0,01, но не более чем на 0,20.

Затем определяется отношение заработка пенсионера к среднемесячной заработной плате в стране. Этот заработок может быть взят за этот отсчетный период или за любые 60 месяцев работы подряд, или тот, из которого была исчислена пенсия на момент реформы. Среднемесячная зарплата в стране берется за тот же самый период.

Отношение заработков учитывается в размере не свыше 1,2. Для пенсионеров, проживающих на Крайнем Севере, учитываемое соотношение выше: от 1,4 до 1,9 в зависимости от установленного в централизованном порядке районного коэффициента к зарплате.

Затем стажевый коэффициент умножается на соотношение заработков и на 1671 руб. — утвержденную для расчетов среднемесячную зарплату в стране за III квартал 2001 г. Это и будет пересчитанный размер трудовой пенсии по новому законодательству в обычном случае. Если он оказался менее 660 руб., то размер пенсии «доводится» до этого гарантированного минимума.

Если пенсионер является инвалидом I группы или достиг к 1 января 2002 г. возраста 80 лет и более, рассчитанный в этом порядке размер пенсии по старости увеличивается на 450 руб.

Если у пенсионера имеются лица, находящиеся на его иждивении, то рассчитанный размер пенсии увеличивается на 150 руб. на каждого иждивенца, но не более чем на трех в общей сложности.

Источники разработки

Данный документ разрабатывался на основании ГОСТ 34.698—90 на написание ТЗ на автоматизированные системы управления от 01.01.1992 г.

Приложения

СОСТАВИЛИ

Должность исполнителя ______________________________
Фамилия, имя, отчество ______________________________
Подпись ________
Дата «___»______________ 2007 г.

Должность исполнителя ______________________________
Фамилия, имя, отчество ______________________________
Подпись ________
Дата «___»______________ 2007 г.

Должность исполнителя ______________________________
Фамилия, имя, отчество ______________________________
Подпись ________
Дата «___»______________ 2007 г.

Приложение 5

Листинги программ к лабораторной работе № 6

1. Файл CNT.CPP

```
// Cnt.cpp: определяет работу приложения.
//
#include "stdafx.h"
#include "Cnt.h"
#include "MainFrm.h"
#include "CntDoc.h"
#include "CntView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE_;
#endif


// CCntApp
BEGIN_MESSAGE_MAP (CCntApp, CWinApp) //{(AFX_MSG_MAP
(CCntApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)

   ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять
   здесь макросы схемы сообщений.
   НЕ РЕДАКТИРУЙТЕ то, что здесь находится.

//}}AFX_MSG_MAP

// Стандартные операции с документами
```

```
ON_COMMAND(ID_FILE_NEW, CWinApp    OnFileNew)
ON__COMMAND(ID_FILE_OPEN, CWinApp: :OnFileOpen)
```

Стандартная команда задания установок принтера

```
ON_COmAND(ID_FILE_PRINT_SETUP, CWinApp:
:OnFilePrintSetup)
END_MESSAGE_MAP()
```

Конструктор класса ***CcntApp***

```
CCntApp::CCntApp() {

// Здесь добавьте код конструктора.
// Все наиболее важные команды инициализации
// разместите в методе Initlnstance.
}
///////////////////////////////////////////////////////////
```

Единственный объект класса ***CCntAppCCntApptheApp;***

Инициализация класса ***CcntApp***

```
BOOLCCntApp::Initlnstance() {

// Инициализация библиотек OLE

if(!AfxOleInit() )
{
AfxMessageBox(IDP_OLE_INIT_FAILED); return FALSE; }
AfxEnableControlContainer ();

// Стандартная инициализация.
// Если вам не нужны используемые здесь возможности
// и вы хотите сократить размер исполняемого файла,
// удалите ненужные команды.

tifdef _AFXDLL
Enable3dControls();
```

Эта функция вызывается при
динамической компоновке ***MFC***

```
#else
Enable3dControlsStatic();

   эта функция вызывается при
   статической компоновке MFC

#endif

// Измените раздел реестра, где будут храниться
// параметры программы.

SetRegistryKey(_T("Local AppWizard-Generated
Applications"));
LoadStdProfileSettings();

   загрузка параметров из INI-файла
   Регистрация шаблонов документов приложения

CSingleDocTemplate* pDocTemplate; pDocTemplate = new
CSingleDocTemplate (
IDR_MAINFRAME,
RUNTIME_CLASS (CCntDoc)
RUNTIME_CLASS (CMainFrame)   // основное SDI-окно>>
RUNTIME_CLASS (CCntView) )
pDocTemplate->SetContainer!nfo(IDR_CNTR_INPLACE)
AddDocTemplate (pDocTemplate)

   Анализ командной строки на предмет поиска системных
команд, DDE-команд или команд открытия файлов.

CCommandLinelnfo cmdlnfo;
ParseCommandLine (cmdlnf o);

   Обработка команд, указанных в командной строке

if (   ProcessShellCommand (cmdlnfo) ) return FALSE;

   Отображение окна приложения

_pMainWnd->ShowWindow (SW_SHOW)
m_pMainWnd->UpdateWindow() ;
return TRUE;
}
```

Класс **CAboutDlg**, управляющий окном **About**

```
class CAboutDlg   public Cdialog
 { public:CAboutDlg ( )

// Данные диалогового окна
//{{AFX_DATA (CAboutDlg) enum { IDD = IDD_ABOUTBOX };
//}}AFX_DATA
// Виртуальные функции, сгенерированные
// мастером ClassWizard
//{{AFX_VIRTUAL (CAboutDlg)

protected:
virtual void DoDataExchange (CDataExchange* pDX)

//}}AFX_VIRTUAL
// Реализация

protected:

// { { AFX_MSG (CAboutDlg)
// Обработчики сообщений отсутствуют
//})AFXMSG

DECLARE_MESSAGE_MAP()
};
CAboutDlg:   CAboutDlg ()   CDialog (CAboutDlg:   IDD)
{

//{fAFX_DATA_INIT (CAboutDlg)
//}}AFX DATA INIT

}
void CAboutDlg::DoDataExchange(CDataExchange* pDX) {
CDialog::DoDataExchange(pDX);

//{{AFX_DATA_MAP(CAboutDlg)
// } } AFX_DATA_MAP }

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//{{AFX_MSG_MAP(CAboutDlg)
// Обработчики сообщений отсутствуют
//}}AFX_MSG_MAPEND_MESSAGE_MAP()
```

```
    Функция, управляющая выводом окна About
void CCntApp::OnAppAbout()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal(); }


    Другие функции класса CCntApp
```

Данный листинг содержит один фрагмент, заслуживающий особого внимания. В OLE используется концепция непосредственного редактирования (in-placeediting). Это означает, что после двойного щелчка на объекте, внедренном в документ контейнера, такой как наш, строка меню и панели инструментов соответствующего OLE-сервера замещают меню и панели инструментов контейнера. Например, если в документ приложения Cnt будет внедрена электронная таблица Excel, то после двойного щелчка на ней строка меню и панели инструментов программы Excel появятся в окне программы Cnt. Смена меню происходит автоматически и обрабатывается библиотекой MFC с помощью средств OLE. Этот процесс становится возможным благодаря тому, что в приложение добавляется два ресурса меню: idr_mainfraimeи idr_cntr_inplace (имя последнего идентификатора уникально для данного приложения). По умолчанию отображается меню IDR_MAINFRAIME. Но когда внедренный объект активизируется для непосредственного редактирования, загружается меню idr_cntr_in-PLACE.

2. Файл MAINFRM.CPP

```
// MainFrm.cpp: реализация класса CMainFrame
//
#include "stdafx.h"
#include "Cnt.h"
#include "MainFrm.h" tifdef _DEBUG
#define new DEBUG_NEW
tundef THIS_FILE
Static char THIS_FILE[]   _FILE_;
#endif
```

```
   CmainFrame

IMPLEMENT_DYNCREATE (CMainFrame, CFrameWnd)
i BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)

//{{AFX_MSG_MAP (CMainFrame)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять
// и удалять здесь макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то,что здесь находится.
//}}AFX_MSG_MAPEND_MESSAGE_MAP ( )

static UINT indicators!]   {
ID_SEPARATOR, // поля строки состояния
ID_INDICATOR_^CAPS,
ID_INDICATOR_NUM,
ID_INDICATOR_SCRL,
};


IIКонструктор и деструктор класса CMainFrame
CMainFrame::CMainFrame() {

    Здесь добавьте код конструктора.

}
CMainFrame::-CMainFrame() f
int CMainFrame: :OnCreate (LPCREATESTRUCT
IpCreateStruct) {
if (CFrameWnd: :OnCreate (IpCreateStruct)    -1) return -1;
if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD
WS_VISIBLE I CBRS_TOP | CBRS_GRIPPER I CBRSJTOOLTIPS I
CBRS_FLYBY | CBRS_SIZE_DYNAMIC) |
!m_wndToolBar.LoadToolBar(IDR_MAINFRAME)
TRACEO ("Failed to create toolbar\n")
return-1; // создать панель инструментов не удалось
}
if (!m_wndStatusBar.Create(this)||
!m_wndStatusBar.Setlndicators(indicators,
sizeof(indicators)/sizeof(UINT))) (
TRACEO("Failedto create status bar\n"); return -1;
// создать
```

```
строку состояния не удалось}
// Удалите следующие три строки, если вы не хотите,
// чтобы панель инструментов была перемещаемой.

m_wndToolBar  EnableDocking (CBRS_ALIGN_ANY)
EnableDocking(CBRS_ALIGN_ANY)
DockControlBar (Sm_wndToolBar)
return 0;
}
BOOL CMainFrame:  PreCreateWindow (CREATESTRUCT Ses)
if(ICFrameWnd:  PreCreateWindow (cs)) return FALSE;

   Здесь можно модифицировать класс окна, изменяя
   поля структуры cs.

return TRUE;


   Диагностика класса CmainFrame

#ifdef _DEBUG
void CMainFrame: :AssertValid   const
{
CFrameWnd: :AssertValid()
}
void CMainFrame::Dump(CDumpContext Sdc) const
{
CFrameWnd::Dump(dc);
#endif
//_DEBUG


   Обработчики сообщений класса CMainFrame
```

3. Файл CNTDOC.CPP

Файл CNTDOC.CPP содержит ряд дополнительных фрагментов, с которыми ранее мы не встречались.

```
   CntDoc.cpp: реализация класса CCntDoc//

#include "stdafx.h" #include "Cnt.h"
```

```
# include "CntDoc.h"
#include "Cntrltem.h"
tifdef _DEBUG
#define new DEBUG_NEW
lundef THIS_FILE
static char THIS_FILE[]     FILE
#endif

////////////////////////////////////////////////////////////
// CcntDoc

IMPLEMENT_DYNCREATE (CCntDoc, COleDocument)
BEGIN_MESSAGE_MAP (CCntDoc, COleDocument)

//{{AFX_MSG_MAP (CCntDoc)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять
// и удалять здесь макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAP
// Используется стандартная реализация OLE-контейнера

ON_UPDATE_COMMAHD_UI ( ID_EDIT_PASTE,
COleDocument: :OnUpdatePasteMenu) ON_OPDATE_COMMAMD_OI
(
ID_EDIT_PASTE_LINK
COleDocument: :OnUpdatePasteLinkMenu)
ON_UPDATE_COMMMJD_UI (
ID_OLE_EDIT_CONVERT
COleDocument: :OnUpdateObjectVerbMenu) ON_COMMAND
(ID_OLE_EDIT_CONVERT,
COleDocument: :OnEditConvert) ON_OPDATE_COMMAHD_OI
(ID_OLE_EDIT_LINKS,COleDocument:
:OnUpdateEditLinksMenu)
ON_COMMAND (ID_OLE_EDIT_LINKS
COleDocument: :OnEditLinks)
ON_0PDATE_COMMAND_UI_RftNGE   ID_OLE_VERB_FIRST
ID_OLE_VERB_IAST,
COleDocument: :OnUpdateObjectVerbMenu) END_MESSAGE_MAP
()


// Конструктор и деструктор класса CCntDoc '
```

```
CCntDoc:   CCntDoc(   {
// Включается поддержка составных файлов

EnablecompoundFile

// Здесь добавьте код конструктора.

}
CCntDoc::-CCntDoc()
{
}
BOOLCCntDoc:   OnNewDocument( )
{
if (   COleDocument     OnNewDocument        return
FALSE;

// Здесь добавьте код повторной инициализации
// (специфика SDI-приложений)

return TRUE;
}

//////////////////////////////////////////////////////

// Сериализация класса CcntDoc

void CCntDoc::Serialize(CArchive sar)
if (ar. IsStoringO )
{
// Здесь добавьте код сохранения.
}
else.

// здесь добавьте код загрузки.

}

// Вызов функции базового класса COleDocument
// обеспечивает сериализацию объектов COleClientltem,
// содержащихся в документе контейнера.

COleDocument : : Serialize (ar ) ;
}
```

```
   Диагностика класса CcntDoc

tifdef _DEBUG
void CCntDoc::AssertValid() const
{
COleDocument::AssertValid(); }
void CCntDoc:   Dump (CDumpContext .&dc) const {
COleDocument: :Dump(dc)   } #endif
//_DEBUG


   Другие функции класса CcntDoc
```

Обратите внимание на функцию EnableCompoundFile(), вызываемую в конструкторе. Эта функция осуществляет поддержку составных файлов, что позволяет сохранять документы с внедренными объектами в специальном структурированном формате.

4. Файл CNTVIEW.CPP

```
   CntView.cpp   реализация класса CCntView


#include "stdafx.h"
#include "Cnt.h"
#include "CntDoc.h"
#include "Cntrltem.h"
#include "CntView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[]   _FILE_;
#endif


   CcntView

IMPLEMENT_DYNCREATE(CCntView, CView)
```

```
BEGIN_MESSAGE_MAP(CCntView, CView)

 //{{AFX_MSG_MAP(CCntView)
 // ПРИМЕЧАНИЕ: мастер классов будет добавлять
 // и удалять здесь макросы схемы сообщений.
 // НЕ РЕДАКТИРУЙТЕ то, что здесь находится.

 ON_WM_DESTROY()
ON_WM_SETFOCUS() ON_WM_SIZE()
ON_COMMAND (ID_OLE_INSERT_NEW, OnlnsertObject)
ON_COMMAND(ID_CANCEL_EDIT_CNTR, OnCancelEditCntr)
//}}AFX_MSG_MAP //
Стандартные команды печати
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW,
CView::OnFilePrintPreview)
END_MESSAGE_MAP ()


   Конструктор и деструктор класса CcntView

CCntView::CCntView()
m_pSelection   NULL;

   здесь добавьте код конструктора.

}
CCntView::-CCntView()
}
BOOL CCntView::PreCreateWindow(CREATESTRUCT Ses)

   Здесь можно модифицировать класс окна,
   изменяя поля структуры сз.

return CView::PreCreateWindow(cs);


   Отображение документа

void CCntView::OnDraw(CDC* pDC)
CCn.tDoc* pDoc   GetDocument () ;
ASSERT_VALID(pDoc);
```

```
   Здесь добавьте код для отображения собственных
   данных.
   Должны отображаться все OLE-объекты,
   содержащиеся в документе.
   Выделенный элемент может быть нарисован
   в произвольном месте.
   Этот код следует удалить в том случае, если
   вы вводите собственный код рисования. Указанные
   ниже координаты в точности соответствуют
   координатам, возвращаемым объектом CCntCntrltem,
   что создает эффект непосредственного
   редактирования.
   удалите следующий код, если реализуете
   собственный код рисования.

if (m_pSelection    NULL)
{
POSITION pos   pDoc-X3etStartPosition()
m_pSelection   (CCntCntrItem*)
pDoc-XSetNextClientltern(pos);   if
(m_pSelection    NULL)
m_pSelection->Draw(pDC, CRect(10, 10,210,210));
}
void CCntView::OnInitialUpdate()
CView::OnInitialUpdate();

   Удалите следующий код, если реализуете собственный
   код инициализации.

m_pSelection   NULL;

   инициализация переменной, содержащей
   указатель на выделенный объект




// Печать документа
BOOL CCntView: :OnPreparePrinting (CPrintlnf o* plnfo)
{

   Стандартные действия по подготовке к печати

return DoPreparePrinting (plnfo) ;
```

```
}
void CCntView::OnBeginPrinting (CDC* /*pDC*/,
CPrintlnfo* /*plnfo*/) {

   Добавьте код дополнительной инициализации
   перед печатью.


void CCntView::OnEndPrinting(CDC* /*pDC*/, CPrintlnfo*
/*plnfo*/)
{

   Добавьте код очистки после печати.

}
void CCntView::OnDestroy()
{

   Деактивировать объект при удалении; это важно,
   если используется режим разделения окна просмотра.

CView::OnDestroy(); COleClientltem* pActiveltem
GetDocument()->
GetlnPlaceActiveltem(this); if (pActiveltem    NULL &&
pActive!tem->
GetActiveView()    this) {
pActive!tem->Deactivate0; ASSERT(GetDocument()->
GetlnPlaceActiveltem(this)    NULL);
}
}



// Поддержка OLE-клиентов

BOOL CCntView::IsSelected(const CObject* pDocItem)
const
{

// Представленная ниже реализация подходит, если
// выделенными являются только объекты CCntCntrltem.
// В противном случае данный код следует заменить.
// Реализуйте функцию, которая проверяет тип
// выделенного объекта.
```

```
return pDocItem    m_pSelection;
void CCntView::OnlnsertObject()

// Вызов стандартного диалогового окна InsertObject
// для получения информации о новом объекте
CCntCntrltem.

COlelnsertDialog dig;
if (dlg.DoModaK)    IDOK) return;
BeginWaitCursor()
CCntCntrltem* pltem   NULL;
TRY

   Создаем новый объект, связанный с этим документом.

CCntDoc* pDoc   GetDocument();
ASSERT_VALID(pDoc);
pltem   new CCntCntrltem(pDoc);
ASSERT_VALID(pltem);

   Инициализация объекта на основании данных,
   полученных из диалогового окна.

if (!dig.Createltem(pltem))
AfxThrowMemoryException();

   Подойдет исключение любого типа

ASSERT_VALID(pltem);

   Если объект был выбран из списка классов,
   а не загружен из файла, запускаем сервер
   для редактирования объекта,

if (dlg.GetSelectionType()
COlelnsertDialog::createNew!tem)
p!tem->DoVerb(OLEIVERB_SHOW, this);
ASSERT_VALID(pltem);

// Последний введенный объект выделяется.
// Введите код, соответствующий требованиям
// вашего приложения.
```

```
m_pSelection   pltem;

// Указатель устанавливается на
// последний введенный объект

pDoc->UpdateAHViews (NULL)
CATCH(CException, e) if (pltem    NULL)
ASSERT_VALID(pltem)   p!tem->Delete()
AfxMessageBox(IDP_FAILED_TO_CREATE); END_CATCH
EndWaitCursor(); >

// Следующий обработчик позволяет с помощью клавиатуры
// прерывать сеанс непосредственного редактирования.
// Инициируется это контейнером, а не сервером,

voidCCntView::OnCancelEditCntr()

   Редактируемый объект закрывается.

COleClientltem* pActiveltem= GetDocument()->
GetlnPlaceActiveltem(this); if (pActiveltem    NULL)
pActive!tem->Close();
ASSERT(GetDocument()->GetInPlaceActiveItem(this)
NULL);
}

   Обработчики OnSetFocusи OnSize требуются контейнеру
   в случае непосредственного редактирования объекта.

void CCntView: :OnSetFocus (CWnd* pOldWnd)
{
COleClientltem* pActiveltem   GetDocument (
GetlnPlaceActiveltem(this)   if (pActiveltem    NULL &&
pActive!tem->GetItemState ()
COleClientltem: :activeUIState)

   Фокус необходимо установить на объект, если он
   находится в той же области просмотра.

CWnd* pWnd   pActiveltem- >GetInPlaceWindow(
if (pWnd    NULL)
{
pWnd->SetFocus( ) ; // метод SetFocus базового класса
                    // не вызывается
```

```
return;
}}
CView::OnSetFocus(pOldWnd); }
void CCntView: :OnSize (UINT nType, int ex, int cy)
{
CView: :OnSize (nType, ex, cy)
COleClientltem* pActiveltem   GetDocument
GetlnPlaceActiveltem(this)   if (pActiveltem    NULL)
pActiveItem->SetItemRects
}
```

```
   Диагностика класса CcntView

tifdef _DEBUG
void CCntView::AssertValid() const
{
CView::AssertValid();
void CCntView::Dump(CDumpContext Sdc) const
CView::Dump(dc); }
CCntDoc* CCntView::GetDocument() // отладочная версия
__________ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS
(CCntDoc))); return
(CCntDoc*)m_pDocument;
#endlf
//_DEBUG
```

```
   Обработчики сообщений класса CCntView
```

Чтобы приложение могло отображать внедренные объекты, были внесены изменения в функцию OnDraw( ). Обратите внимание на такой фрагмент:

```
if (m_pSelection !=NULL)
m_pSeleotion->Draw(pDC, CRectdO, 10,210, 210));
}
```

По умолчанию объект размещается в окне приложения в заранее заданной области, которая определяется конструктором CRect( ). Указанные координаты можно изменить вручную.

В файл были также добавлены функции OnlnitialUpdate(), IsSelectedO, nlnsertObject(), OnCancelEditCntr(), OnSetFocusOИ OnSize(). В частности, функция OnlnsertObject() вызывает стандартное диалоговое окно класса COlelnsertDialog, предназначенное для вставки объектов.

5. Файл CNTRITEM.CPP

Файл CNTRITEM.CPP, листинг которого приведен ниже, содержит реализацию класса CCntCntrltem.

```
Cntrltem.cpp: реализация класса CCntCntrltem


#include "stdafx.h"
#include "Cnt.h"
#include "CntDoc.h"
#include "CntView.h"
#include "Cntrltem.h"
#ifdef _DEBUG
#define new DEBUG_NEW
tundef THIS_FILE
static char THIS_FILE[]     FILE
#endif


// CcntCntrltem

IMPLEMENT_SERIAL (CCntCntrltem, COleClientltem, 0)
CCntCntrltem:   CCntCntrltem (CCntDoc* pContainer)
  COleClientltem (pContainer)

   здесь добавьте код конструктора.

CCntCntrltera::~CCntCntrItem

   здесь добавьте код очистки.
```

```
void CCntCntrltem::OnChange(OLE_NOTIFICATION nCode,
DWORD dwParam)
ASSERT_VALID(this)  COleClientltem::OnChange(nCode,
dwParam);
```

Редактируемому объекту посылается уведомление OnChange, **свидетельствующее об изменении его состояния или внешнего вида.**
Обозначьте рабочую область объекта как недействительную путем вызова функции UpdateAllViews.

```
GetDocument()->UpdateAllViews(NULL)
BOOL CCntCntrltem::OnChangeItemPosition(const CRect
SrectPos)
ASSERT_VALID(this);
```

В процессе непосредственного редактирования данная функция вызывается сервером для изменения позиции окна редактирования.
По умолчанию вызывается метод базового класса, который, в свою очередь, задействует функцию COleClientltem::SetItemRects для перемещения объекта в новую позицию.

```
if (!COleClientltem::OnChangeItemPosition(rectPos))
return FALSE;
```

обновите всю кэшированную информацию, связанную с размерами объекта.

```
return TRUE; )
void CCntCntrltem::OnGetItemPosition(CRect SrPosition)
ASSERT_VALID(this);
```

В процессе непосредственного редактирования данная функция позволяет определить координаты области, занимаемой объектом.
По умолчанию заданы фиксированные координаты.
Запишите правильные координаты (в пикселях) в переменную rPosition.

```
rPosition.SetRect(10, 10,210, 210);
```

```
void CCntCntrltem: :OnActivate()
```

Допускается только один сеанс редактирования в окне приложения.

```
CCntView* pView   GetActiveView
ASSERT_VALID(pView)
COleClientltem* pltem   GetDocument ( )
GetlnPlaceActiveltem(pView)   if (pltem   NULL
ss-pltem   this)
p!tem->Close ()
COleClientltem:   OnAct ivate
}
void CCntCntrltem: :OnDeactivateUI (BOOL bUndoable)
COleClientltem: :OnDeactivateUI (bUndoable)
```

Скрывает объект, если он был активизирован обычным способом

```
DWORDdwMisc   0;
m_lpObject->GetMiscStatus (GetDrawAspect      SdwMisc)
if (dwMisc & OLEMISC_INSIDEOUT)
DoVerb (OLEIVERB_HIDE, NULL)
}
void CCntCntrltem:   Serialize (CArchive Sar)
ASSERT_VALID(this)
```

Вызов функции базового класса для считывания данных, связанных с объектом COleClientltem. **При этом инициализируется указатель** m_j>Document, **возвращаемый функцией** CCntCntrltem: :GetDocument, **поэтому функцию базового класса лучше вызывать вначале.**

```
COleClientltem:   Serialize (ar);
```

Теперь сериализуем данные, специфичные для объекта CCntCntrltem.

```
if (ar.IsStoringO
{
```

здесь добавьте код сохранения.

```
else{

   здесь добавьте код загрузки.




//Диагностика класса CcntCntrltem

#if<fef _DEBUG
void CCntCntrltem: :AssertValid   const
COleClientltem::&ssertValid0;
}
void CCntCntrltem: :Dump(CDumpContext' &dc) const
COleClientltem::Dump(dc);
#endif
```

Основное назначение этого файла состоит в отслеживании координат и размеров области, занимаемой внедренным объектом. Обратите внимание на фрагмент, выделенный полужирным шрифтом. Именно здесь задаются координаты, о которых говорилось при рассмотрении файла CNTVIEW.CPP.

Литература

1. *Иванова Г. С.* Технология программирования. М.: Изд-во МГТУ им. Баумана, 2002.

2. *Вельбицкий И. В.* Технология программирования. Киев, 1984.

3. *Грейди Буч, Джеймс Рамбо, Айвар Джекобсон.* UML руководство пользователя. М.: ДМК, 2000.

4. *Дин Леффингуэлл, Дон Уидриг.* Принципы работы с требованиями к программному обеспечению. М.: Вильямс, 2002.

5. *Дэвид С. Платт.* Знакомство с Microsoft .NET. М.: Русская редакция, 2001.

6. *Липаев В. В.* Проектирование программных средств. М.: Высшая школа, 1990.

7. *Майерс Г.* Искусство тестирования программ. М.: Финансы и статистика, 1982.

8. *Брукс Ф.* Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс, 1999.

9. *Роберт Дж. Орберг.* COM+ технология. Основы и программирование М.: Вильямс, 2000. 478 с.

10. *E. Yourdon.* Who Has the Right Stuff? // Software Development. 1997. № 9.

11. *Gause, Weinberg.* Exploring Requirements: Quality Before Design, 1989

12. *Аджиев В.* // Открытые системы. 1998. № 1.

13. *Батенко Л. П.* // Менеджмент и менеджер. 2003. № 3.

14. *Марио Аписелла* // Computerworld. 2000. № 16—17.

15. *Davis.* Fifteen Principles of Software Engineering // IEEE Software. 1994. Vol. 11. № 6. P. 94—101.

16. *Boehm A.* Spiral Model of Software Development and Enhancement // Computer. 1988. Vol. 21. № 5. P. 61—72.

17. Good-Enough Software // Application Development Strategies. 1996. № 1.

18. *Алистэр Коуберн, Лори Вильямс.* Парное программирование: преимущества и недостатки.

19. *Cusumano M., Selby R.* How Microsoft Builds Software // Communications of the ACM. 1997. Vol. 40. № 6. P. 53—61.

20. *Cusumano M., Selby R.* Microsoft's Weaknesses in Software Development // American Programmer. 1997. Vol. 10. № 10.

21. *Knepper S., Ichbiah D.* The Making Microsoft. Prima Publishing, 1993

22. www.maxkir.com.

23. http://www.martinfowler.com/articles/designDead.html.

24. http://www.softportal.com/articles/item_txt.php?id=78 Кент Бек.

25. http://ergl.ru/archive/cs/tp/01.htm#P4 Лекции ВМиК. Технология программирования.

26. *Ожегов С. И.* Словарь русского языка. М.: Советская Энциклопедия, 1975.

27. Советский энциклопедический словарь. М.: Советская Энциклопедия, 1979.

28. Политехнический словарь / Гл. ред. акад. А. Ю. Ишлинский. 2-е изд. М.: Советская Энциклопедия, 1980.

29. *Терри Кватрани.* Визуальное моделирование с помощью Rational Rose 2002 и UML. М.: Вильямс, 2003.

30. http://www.zsksoft.ru/zsync-doc/usage/team-development.htm.

31. *Voas J.* The Software Quality Certification Triangle. Crosstalk, Nov. 1998. P 12—14.

32. http://www.spc-consulting.ru/standart/cmm.htm.

33. http://www.interface.ru/home.asp?artId=3987.

34. http://www.intuit.ru/department/se/testing/.

35. *McCabe T. J., Butler Ch. W* Design complexity measurement and testing Communications of the ACM. 32, 12 (Dec. 1989). P. 1415—1425.

36. *Макконнелл С.* Совершенный код. СПб.: Питер, 2006.

37. *Жоголев Е. А.* Введение в технологию программирования (конспект лекций). М.: ДИАЛОГ-МГУ, 1994.

38. *Уолш Б.* Программирование на Бейсике. М.: Радио и связь, 1988.

39. *Калянов Г. Н.* Консалтинг при автоматизации предприятий (подходы, методы, средства). М.: СИНТЕГ, 1997.

40. *Страуструп Б.* Язык программирования C++. Киев: ДиаСофт, 1993.

41. Модели и структуры данных / В. Д. Далека, А. С. Деревянко, О. Г. Кравец, Л. Е. Тимановская. Харьков: ХГПУ, 2000.

42. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию. М.: Мир, 1980. С. 29—71.

43. *Жоголев Е. А.* Технологические основы модульного программирования // Программирование. 1980. № 2. С. 44—49.

44. *Holt R. C.* Structure of Computer Programs: A Survey // Proceedings of the IEEE. 1975. 63(6). P. 879—893.

45. *Майерс Г.* Надежность программного обеспечения. М.: Мир, 1980. С. 92—113.

46. *Зелковец М. Шоу А., Гэннон Дж.* Принципы разработки программного обеспечения. М.: Мир, 1982. С. 65—71.

47. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование. М.: Мир, 1975. С. 7—19.

48. *Леоненков А. В.* Самоучитель UML. СПб.: BHV, 2006.

49. *Крачтен Ф.* Rational Unified Process: введение: третье издание. Издательство Addison-Wesley Professional, 2003.

50. *Басс Л., Клементс П., Рик Кацман Р.* Практическая архитектура программного обеспечения: второе издание. Addison Wesley, 2003.

51. Object Management Group Inc. Спецификация унифицированного языка моделирования OMG версия 1.5. Документ номер 03-03-01. Март 2003.

52. *Мак-Говерн Джеймс и др.* Практическое руководство по архитектуре корпораций. Prentice Hall, 2004.

53. *Вендров А. М.* CASE-технологии. Современные методы и средства проектирования информационных систем. М.: Финансы и статистика, 1998.

54. *Пушников А. Ю.* Введение в системы управления базами данных. Источник материала — Пушников А. Ю. Введение в системы управления базами данных. Часть 2. Нормальные формы отношений и транзакции: учеб. пособие. Изд-е Башкирского ун-та, 1999.

55. Структурный анализ при разработке программного обеспечения систем реального времени / В. А. Матьяш, А. В. Никандров, В. А. Путилов, А. Е. Федоров, В. В. Фильчаков. Апатиты: КФ ПетрГУ, 1997.

56. http://se.math.spbu.ru/.

57. *Колдовский В.* Разработка ПО: оценка результата // Компьютерное обозрение. 2006. 21 сент.

58. *Albrecht A. J.* Measuring Application Development Productivity? Proceedings of the Joint SHARE/GUIDE/IBM Application Develop-. ment Simposium? 1979. October. P 83—92.

59. *Боэм Б.* Инженерное проектирование программного обеспечения. М.: Радио и связь, 1985.

60. Visual C++ 6. Руководство разработчика.

61. *Брауде Э. Д.* Технология разработки программного обеспечения. СПб.: Питер, 2004.

62. *Зиндер Е. З.* Бизнес-реинжиниринг и технологии системного проектирвания. М.: Центр информационных технологий, 1996.

63. *Канер С., Фолк Д., Нгуен Е. К.* Тестирование программного обеспечения: пер. с англ. Киев: ДиаСофт, 2000.

Предметный указатель

A

Application *17*

C

CASE-технология *28*

O

OS/2 *23*

OS/2 Warp *23*

P

Problem definition *18*

Problem, task *17*

Program *84*

Program, routine *17*

Programming *18*

R

Routine *84*

S

Software *17*

Spiral *66*

System Software *20*

W

Windows NT *24*

А

Алгоритм *18*

Архитектура *84*

Архитектура программного обеспечения *83*

Архитектура программного обеспечения системы *84*

Архитектура программы *83*

Архитектура ПС *83*

Б

Базовое ПО *21*

В

Векторы *89*

Г

Генераторы отчетов *30*

Д

Двумерные массивы *89*

Двусвязные списки *93*

З

Задача *17*

Записи *90*

И

Инструментальная среда пользователя *27*

Интегрированные пакеты *31*

Интегрированные среды разработки программ *27*

Итерационная модель *65*

К

Каскадная модель *64*

Л

Логическая структура данных *86*

М

Методо-ориентированные ППП *31*
Множества *90*
Модульное программирование *94*

О

Операционная система *21*
Операционные оболочки *24*
Органайзеры *31*
ОС UNIX *23*
Офисные ППП *31*

П

Постановка задачи *18*
ППП автоматизированного проектирования *29*
ППП общего назначения *29*
Прикладное ПО *20*
Приложение *17*
Проблемно-ориентированные ППП *28*
Программа *17*, *84*
Программирование *18*
Программное обеспечение *17*
Программные комплексы *84*
Программные модули *94*
Программные продукты *19*
Программные продукты: freeware *19*
Программные продукты: shareware *19*
Программные системы *85*
Простой тип данных *86*
Прочность модуля *96*

Р

Размер модуля *96*
Распознавание текста *31*
Рутинность модуля *97*

С

Связность модулей *98*
Связные линейные списки *93*
Серверы БД *30*
Сервисное ПО *21*
Сетевая ОС *24*
Системное ПО *20*
Системы программирования *27*
Системы управления базами данных *29*
Сопровождение программы *18*
Спиральная модель *66*
Средства для создания приложений *26*
Средства презентационной графики *30*
Средства создания WWW-страниц *31*
Средства электронной почты *31*
Структура данных *85*
Сцепление модуля *96*

Т

Технология программирования *17*

У

Утилиты *25*

Ф

Физическая структура данных *86*

Э

Эксплуатационные требования *81*
- адаптируемость *83*
- аппаратная совместимость *82*
- защищенность *82*
- надежность *81*
- повторная входимость *83*
- правильность *81*
- проверяемость *82*
- программная совместимость *82*
- реентерабельность *83*
- точность результатов *82*
- универсальность *81*
- эффективность *82*

Я

Язык программирования *26*

Оглавление

Введение 3

Глава 1. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ 5

1.1. Технология программирования в историческом аспекте 5
1.2. Основные понятия и определения 17
1.3. Классификация программного обеспечения 20
1.3.1. Системное программное обеспечение 21
1.3.2. Инструментарий технологии программирования 26
1.3.3. Пакеты прикладных программ 28

Глава 2. ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНЫХ ПРОДУКТОВ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ И ПОДХОДЫ 33

2.1. Особенности создания программного продукта 33
2.1.1. Принципы работы с требованиями к программному обеспечению. Проблематика проектирования 33
2.1.2. Оценка стоимости ошибок 34
2.1.3. Управление требованиями 36
2.1.4. Последовательность работы с требованиями. Анализ проблемы 37
2.1.5. Преграды на пути выявления требований 41
2.2. Оценка качества процессов создания программного обеспечения 46
2.2.1. Серия стандартов ISO 9000 47
2.2.2. CMM 53
2.2.3. Процесс сертификации программ на базе информации об их использовании 55

2.3. Жизненный цикл программы 60
2.3.1. Понятие технологии разработки программы 60
2.3.2. Основа разработки программного обеспечения 62
2.3.3. Модели жизненного цикла 64
2.3.4. Rational Objectory Process — модель жизненного цикла (методология объектно-ориентированного программирования) 66
2.3.5. Жизненный цикл UML (Rational Objectory Process) 67
2.3.6. Специфицирование и планирование 70
2.3.7. Процесс разработки 72
2.3.8. Выпуск продукта и механизмы обратной связи 76

Глава 3. АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 79

3.1. Определение требований к программным продуктам 79
3.1.1. Функциональные требования 79
3.1.2. Эксплуатационные требования 81
3.2. Выбор архитектуры программного обеспечения 83
3.3. Структура и формат данных. Статические, полустатические и динамические структуры 85
3.3.1. Классификация структур данных 86
3.3.2. Простые структуры данных 87
3.3.3. Статические структуры данных 89
3.3.4. Полустатические структуры данных 92
3.3.5. Динамические структуры данных 92
3.4. Модульное программирование 94
3.4.1. Понятие модуля 94
3.4.2. Основные характеристики программного модуля 95
3.4.3. Модульная структура программных продуктов 100
3.4.4. Методы разработки при модульном программировании 101
3.5. Анализ требований и определение спецификаций при структурном подходе 108
3.5.1. Спецификации процессов 108

3.5.2. Словарь терминов 113
3.5.3. Диаграммы переходов состояний (SDT) 114
3.5.4. Функциональные диаграммы 115
3.5.5. Диаграммы потоков данных (DFD) 122
3.5.6. Диаграммы сущность—связь 126
3.6. Анализ требований и определение спецификаций при объектном подходе 133
3.6.1. Некоторые теоретические сведения о UML — унифицированном языке моделирования 133
3.6.2. Определение прецедентов (вариантов использования) 134
3.6.3. Построение концептуальной модели предметной области 138
3.6.4. Описание поведения системы. Диаграммы последовательностей, деятельности и состояний 145

Глава 4. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 158

4.1. Проектирование программного обеспечения при структурном подходе 161
4.1.1. Структурная схема разрабатываемого программного обеспечения 161
4.1.2. Функциональная схема 162
4.1.3. Метод пошаговой детализации при составлении алгоритмов 163
4.1.4. Структурные карты Константайна 165
4.1.5. Структурные карты Джексона 168
4.1.6. CASE-технологии 170
4.1.7. Ускорение разработки программного обеспечения. Методология RAD 175
4.2. Проектирование программного обеспечения при объектном подходе 178
4.2.1. Разработка структуры программного обеспечения при объектном подходе 179
4.2.2. Диаграммы кооперации 188
4.3. Экстремальное программирование 189
4.3.1. Основополагающие практики XP 189

4.3.2. Преимущества простого дизайна 191
4.3.3. Простой дизайн 192
4.3.4. Рефакторинг и принцип YAGNI 193
4.3.5. Наращивание архитектуры 194
4.3.6. UML и XP 195
4.3.7. Суть проектирования. Программирование и тестирование 197

Глава 5. ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММ 199

5.1. Термины и определения 200
5.2. Тестирование «белого ящика» и «черного ящика» 203
5.3. Порядок разработки тестов 204
5.4. Автоматизация тестирования 205
5.5. Модульное тестирование 206
5.6. Интеграционное тестирование 210
5.7. Системное тестирование 213
5.8. Эффективность и оптимизация программ 215
5.9. Стиль программирования 218
5.9.1. Основные принципы форматирования 218
5.9.2. Способы форматирования 220
5.10. Надежность программного обеспечения 221
5.10.1. Количественные характеристики надежности программ 223
5.10.2. Методы оценки и измерения характеристик надежности 225
5.10.3. Преимущества парного программирования 228
5.11. Отладка программ 228

Глава 6. СОПРОВОЖДЕНИЕ ПРОГРАММ 233

6.1. Виды программных документов 233
6.2. Пояснительная записка 235
6.3. Руководство пользователя 236
6.4. Руководство системного программиста 237

Глава 7. РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 239

7.1. Инструментальные средства разработки программ 239
7.1.2. Выбор языка программирования 239
7.1.3. Выбор среды программирования 243

7.2. Технологии программирования 244
7.2.1. Объектно-ориентированное программирование 244
7.2.2. Платформа JAVA и .NET 246
7.3. Защита программных продуктов 249
7.3.1. Основные сведения о защите программных продуктов 249
7.3.2. Криптографические методы защиты информации 250
7.3.3. Программные системы защиты от несанкционированного копирования 252
7.3.4. Правовые методы защиты программных продуктов и баз данных 253

Глава 8. КОЛЛЕКТИВНАЯ РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 257

8.1. Пакеты прикладных программ 257
8.1.1. Система контроля версий Microsoft Visual SourceSafe 257
8.1.2. Система контроля версий Subversion 258

Глава 9. ЭКОНОМИЧЕСКИЕ АСПЕКТЫ РАЗРАБОТКИ И ИСПОЛЬЗОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ 259

9.1. Оценка стоимости разработки программного обеспечения 259
9.1.1. Линейный метод 259
9.1.2. Метод функциональных точек 260
9.1.3. Оценка с использованием эмпирических данных 264
9.2. Методы оценки эффективности ПО на этапе эксплуатации 268

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа № 1. Этапы разработки программного обеспечения при структурном подходе к программированию. Стадия «Техническое задание» 271
Лабораторная работа № 2. Структурный подход к программированию. Стадия «Эскизный проект» 277

Лабораторная работа № 3. Структурный подход к программированию. Стадия «Технический проект» 281
Лабораторная работа № 4. Этапы разработки программного обеспечения. Стадия «Реализация» 285
Лабораторная работа № 5. Тестирование программ методами «белого ящика» 289
Лабораторная работа № 6. Использование технологий OLE, COM и ActiveX 297
Лабораторная работа № 7. Создание сетевых приложений на Delphi с использованием Windows Sockets API 305
Лабораторная работа № 8. Проектирование программной системы при объектном подходе к программированию 321
Лабораторная работа № 9. Динамические структуры данных 326
Лабораторная работа № 10. Объектно-ориентированное программирование (ООП) 339

ПРИЛОЖЕНИЯ

Приложение 1. Варианты заданий 349
Приложение 2. Пример разработки технического задания на программный продукт 352
Приложение 3. Пример технического задания на разработку 355
Приложение 4. Пример эскизного проекта 360
Приложение 5. Листинги программ к лабораторной работе № 6 368

Литература 388

Предметный указатель 392

Гагарина Лариса Геннадьевна
Кокорева Елена Викторовна
Виснадул Белла Дмитриевна

Технология разработки программного обеспечения

Учебное пособие

Редактор *А. В. Волковицкая*
Корректор *О. Н. Картамышева*
Компьютерная верстка *И. В. Кондратьевой*
Оформление серии *К. В. Пономарева*

Сдано в набор 10.06.2007. Подписано в печать 12.07.2007. Формат $60 \times 90^{1}/_{16}$
Печать офсетная. Гарнитура «Таймс». Усл. печ. л. 25,0. Уч.-изд. л. 25,5.
Бумага офсетная. Тираж 2000 экз. Заказ № 7888.

ЛР № 071629 от 20.04.98
Издательский Дом «ФОРУМ»
101000, Москва — Центр, Колпачный пер., д. 9а
Тел./факс: (495) 625-39-27
E-mail: forum-books@mail.ru

ЛР № 070824 от 21.01.93
Издательский Дом «ИНФРА-М»
127282, Москва, Полярная ул., д. 31в
Тел.: (495) 380-05-40
Факс: (495) 363-92-12
E-mail: books@infra-m.ru
Http://www.infra-m.ru

По вопросам приобретения книг обращайтесь:

Отдел продаж «ИНФРА-М»
127282, Москва, ул. Полярная, д. 31в
Тел.: (495) 363-42-60
Факс: (495) 363-92-12
E-mail: books@infra-m.ru

Центр комплектования библиотек
119019, Москва, ул. Моховая, д. 16
(Российская государственная библиотека, кор. К)
Тел.: (495) 202-93-15

Магазин «Библиосфера» (розничная продажа)
109147, Москва, ул. Марксистская, д. 9
Тел.: (495) 670-52-18, (495) 670-52-19

Отпечатано с готовых диапозитивов в ОАО ордена «Знак Почета»
«Смоленская областная типография им. В. И. Смирнова».
214000, г. Смоленск, проспект им. Ю. Гагарина, 2.

ISBN 978-5-8199-0342-1
9 785819 903421