

Г.С. Иванова

Технология программирования

Допущено Министерством образования
Российской Федерации
в качестве учебника для студентов
высших учебных заведений, обучающихся по направлению
«Информатика и вычислительная техника»,
специальностям: «Вычислительные машины, комплексы,
системы и сети», «Автоматизированные системы обработки
информации и управления», «Программное обеспечение
вычислительной техники и информационных систем»

Москва
Издательство МГТУ имени Н.Э. Баумана
2002

Серия основана в 2000 году

РЕДАКЦИОННАЯ КОЛЛЕГИЯ:

Д-р техн. наук И.Б. Федоров — главный редактор

д-р техн. наук И.П. Норенков — зам. главного редактора

д-р техн. наук Ю.М. Смирнов — зам. главного редактора

д-р техн. наук В.В. Девятков

д-р техн. наук В.В. Емельянов канд. техн. наук И.П. Иванов

д-р техн. наук В.А. Матвеев канд. техн. наук Н.В. Медведев

д-р техн. наук В.В. Сюезв

д-р техн. наук Б.Г. Трусов

д-р техн. наук В.М. Черненький д-р техн. наук В.А. Шахнов

УДК 681.3.06(075.8)
ББК 32.973-018 И201

Рецензенты:

кафедра «Компьютерные системы и технологии»
Московского государственного инженерно-физического института
(зав. кафедрой профессор Л.Д. Забродин);
кафедра «ЭВМ, комплексы и сети»
Московского государственного авиационного института
(зав. кафедрой профессор О.М. Брехов)

Иванова Г.С.

И201 . Технология программирования: Учебник для вузов. - М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. - 320 с.: ил. (Сер. Информатика в техническом университете.)
ISBN 5-7038-2077-4

Подробно рассмотрены основные методы и нотации, применяемые при разработке сложного программного обеспечения. При этом особое внимание уделено проектированию программных систем с использованием структурного, объектного и компонентного подходов. Детально разобраны основные приемы обеспечения требуемых технологических свойств. Приведена классификация и проанализированы принципы проектирования пользовательских интерфейсов программного обеспечения. Изложение материала иллюстрируется большим количеством примеров и поясняющих рисунков.

Содержание учебника соответствует курсу лекций, который автор читает в МГТУ им. Н.Э. Баумана.

Для студентов вузов, обучающихся по специальностям, связанным с информатикой. Может быть полезен всем изучающим программирование самостоятельно.

УДК 681.3.06(075.8)
ББК 32.973-018

ISBN 5-7038-2077-4

© Г.С. Иванова, 2002
© МГТУ им. Н.Э. Баумана, 2002

Оглавление

Предисловие

Введение

1.Технология программирования. Основные понятия и подходы

- 1.1.Технология программирования и основные этапы ее развития
- 1.2.Проблемы разработки сложных программных систем
- 1.3.Блочный-иерархический подход к созданию сложных систем
- 1.4.Жизненный цикл и этапы разработки программного обеспечения
- 1.5.Эволюция моделей жизненного цикла программного обеспечения
- 1.6.Ускорение разработки программного обеспечения. Технология RAD
- 1.7.Оценка качества процессов создания программного обеспечения

2.Приемы обеспечения технологичности программных продуктов

- 2.1.Понятие технологичности программного обеспечения
- 2.2.Модули и их свойства
- 2.3.Нисходящая и восходящая разработка программного обеспечения
- 2.4.Структурное и «неструктурное» программирование. Средства описания структурных алгоритмов
- 2.5.Стиль оформления программы
- 2.6.Эффективность и технологичность
- 2.7.Программирование «с защитой от ошибок»
- 2.8.Сквозной структурный контроль

3.Определение требований к программному обеспечению и исходных данных для его проектирования

- 3.1. Классификация программных продуктов по функциональному признаку
- 3.2. Основные эксплуатационные требования к программным продуктам
- 3.3.Предпроектные исследования предметной области
- 3.4.Разработка технического задания
- 3.5.Принципиальные решения начальных этапов проектирования

4.Анализ требований и определение спецификаций программного обеспечения при структурном подходе

- 4.1.Спецификации программного обеспечения при структурном подходе
- 4.2.Диаграммы переходов состояний
- 4.3.Функциональные диаграммы
- 4.4.Диаграммы потоков данных
- 4.5.Структуры данных и диаграммы отношений компонентов данных
- 4.6.Математические модели задач, разработка или выбор методов решения

5.Проектирование программного обеспечения при структурном подходе

- 5.1. Разработка структурной и функциональной схем
- 5.2. Использование метода пошаговой детализации для проектирования структуры программного обеспечения
- 5.3.Структурные карты Константайна
- 5.4.Проектирование структур данных
- 5.5.Проектирование программного обеспечения, основанное на декомпозиции данных
- 5.6.Случаи-технологии, основанные на структурных методологиях анализа и проектирования

6.Анализ требований и определение спецификаций программного обеспечения при объектном подходе

- 6.1.UML - стандартный язык описания разработки программных продуктов с использованием объектного подхода

- 6.2.Определение «вариантов использования»
- 6.3.Построение концептуальной модели предметной области
- 6.4.Описание поведения. Системные события и операции
- 7. Проектирование программного обеспечения при объектном подходе**
- 7.1. Разработка структуры программного обеспечения при объектном подходе
- 7.2.Определение отношений между объектами
- 7.3.Уточнение отношений классов
- 7.4.Проектирование классов
- 7.5.Компоновка программных компонентов
- 7.6.Проектирование размещения программных компонентов
для распределенных программных систем
- 7.7. Особенность спиральной модели разработки. Реорганизация проекта
- 8. Разработка пользовательских интерфейсов**
- 8.1.Типы пользовательских интерфейсов и этапы их разработки
- 8.2.Психофизические особенности человека, связанные с восприятием, запоминанием и обработкой информации
- 8.3.Пользовательская и программная модели интерфейса
- 8.4.Классификации диалогов и общие принципы их разработки
- 8.5.Основные компоненты графических пользовательских интерфейсов
- 8.6.Реализация диалогов в графическом пользовательском интерфейсе
- 8.7.Пользовательские интерфейсы прямого манипулирования
и их проектирование
- 8.8.Интеллектуальные элементы пользовательских интерфейсов
- 9.Тестирование программных продуктов**
- 9.1.Виды контроля качества разрабатываемого программного обеспечения
- 9.2.Ручной контроль программного обеспечения
- 9.3.Структурное тестирование
- 9.4.Функциональное тестирование
- 9.5.Тестирования модулей и комплексное тестирование
- 9.6.Оценочное тестирование
- 10. Отладка программного обеспечения**
- 10.1. Классификация ошибок
- 10.2. Методы отладки программного обеспечения
- 10.3. Методы и средства получения дополнительной информации
- 10.4. Общая методика отладки программного обеспечения
- 11. Составление программной документации**
- 11.1.Виды программных документов
- 11.2.Пояснительная записка
- 11.3.Руководство пользователя
- 11.4.Руководство системного программиста
- 11.5.Основные правила оформления программной документации
- 11.6.Правила оформления расчетно-пояснительных записок при курсовом
проектировании
- Приложение. Система условных обозначений унифицированного
языка моделирования UML**
- Список литературы**

ПРЕДИСЛОВИЕ

До последнего времени элементы технологии разработки программного обеспечения студенты изучали в таких курсах, как «Алгоритмические языки и программирование» и «Системное программирование», параллельно с основным материалом, что не позволяло сконцентрироваться на указанных вопросах. Однако сравнительно недавно в учебных планах специальностей, связанных с информатикой, появился курс «Технология программирования», полностью посвященный этой теме. В предлагаемом учебнике сделана попытка обобщения и методического осмысления опыта, накопленного специалистами в области разработки программного обеспечения на протяжении всей истории существования.

Приведенные сведения могут быть полезны при выполнении учебных проектов и небольших программных продуктов.

В создании данного учебника в той или иной степени участвовало много заинтересованных лиц. Хочется от души поблагодарить: редактора издательства Овчеренко Н.Е., поддержавшую идею написания данной книги; заведующих кафедрами «Компьютерные системы и сети» Сюзева В.В. и «Программирования и информационные технологии» Трусова Б.Г. - за всестороннее содействие; моего мужа- профессора кафедры «Компьютерные системы и сети» Овчинникова В.А. - за помощь и поддержку; преподавателей университета: Борисова С.В., Курова А.В. - за предоставленную литературу изданий прошлых лет; Романову Т.Н., Пугачева Е.К., Ничушкину Т.Н., Волосатову Т.М. и Балдина А.В. - за материалы и советы по содержанию учебника. Автор также глубоко признательна рецензентам: коллективу кафедры «Компьютерные системы и технологии» МИФИ во главе с д-ром техн. наук, профессором Забродиным Л.Д. и коллективу кафедры «ЭВМ, комплексы и сети» МАИ во главе с д-ром техн. наук, профессором Бреховым О.М., чьи ценные замечания позволили улучшить качество книги.

ВВЕДЕНИЕ

Создание программной системы - весьма трудоемкая задача, особенно в наше время, когда обычный объем программного обеспечения превышает сотни тысяч операторов. Будущий специалист в области разработки программного обеспечения должен иметь представление о методах анализа, проектирования, реализации и тестирования программных систем, а также ориентироваться в существующих подходах и технологиях.

Изложение материала учебника строится в соответствии с основными этапами разработки программного обеспечения. Исключением являются первые главы, в которых рассмотрены общие вопросы технологии программирования.

В первой главе проанализирована история развития технологии программирования, показано, что в основе разработки программного обеспечения лежит блочно-иерархический подход, рассмотрены особенности применения этого подхода к разработке программных продуктов.

Вторая глава содержит описание приемов обеспечения качества программного обеспечения: основных положений структурного, модульного и защитного программирования. В ней также приведены некоторые рекомендации, например, по стилю оформления программ.

В третьей главе рассматриваются проблемы, связанные с постановкой задачи: от классификации программных продуктов до разработки технического задания и принятия основных решений начального этапа проектирования, например, выбора подхода, среды и языка программирования.

Четвертая и пятая главы посвящены особенностям разработки программного обеспечения при структурном подходе; четвертая - анализу различных моделей разрабатываемого программного обеспечения, используемых на этапе уточнения спецификаций, а пятая - методикам проектирования.

Шестая и седьмая главы содержат аналогичный материал для объектного подхода. В качестве основного языка описания моделей анализа и проектирования при объектном подходе используется UML, как мощное и практически стандартное средство описания объектных разработок.

В восьмой главе подробно рассмотрены проблемы проектирования пользовательского интерфейса и предлагаются соответствующие модели.

Девятая глава посвящена тестированию программных продуктов как по частям, так и в целом, десятая - методам, средствам и методикам отладки разрабатываемого программного обеспечения.

В одиннадцатой главе приведены сведения и рекомендации по разработке программной документации.

Материал сопровождается большим количеством сравнительно простых примеров, причем по возможности использованы три примера разработки, для которых рассмотрены различные аспекты проектирования.

Курс обучения целесообразно завершать курсовым проектом или курсовой работой, целью которых должно быть создание небольшого, но завершенного программного продукта (в отличие от небольших и, как правило, недокументированных программ, которые студенты пишут на лабораторных работах при изучении основ программирования и/или конкретных языков). Проект должен начинаться с составления и утверждения технического задания и сопровождаться подготовкой необходимой программной документации.

1. ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ.

ОСНОВНЫЕ ПОНЯТИЯ И ПОДХОДЫ

Программирование — сравнительно молодая и быстро развивающаяся отрасль науки и техники. Опыт ведения реальных разработок и совершенствования имеющихся программных и технических средств постоянно переосмысливается, в результате чего появляются новые методы, методологии и технологии, которые, в свою очередь, служат основой более современных средств разработки программного обеспечения. Исследовать процессы создания новых технологий и определять их основные тенденции целесообразно, сопоставляя эти технологии с уровнем развития программирования и особенностями имеющихся в распоряжении программистов программных и аппаратных средств.

1.1. Технология программирования и основные этапы ее развития

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология, технология программирования представляет собой набор технологических инструкций, включающих:

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т. п. (рис. 1.1).

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Различают технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки.

В основе первых, как правило, лежит ограниченно применимый *метод*, позволяющий решить конкретную задачу. В основе вторых обычно лежит базовый метод или *подход*,

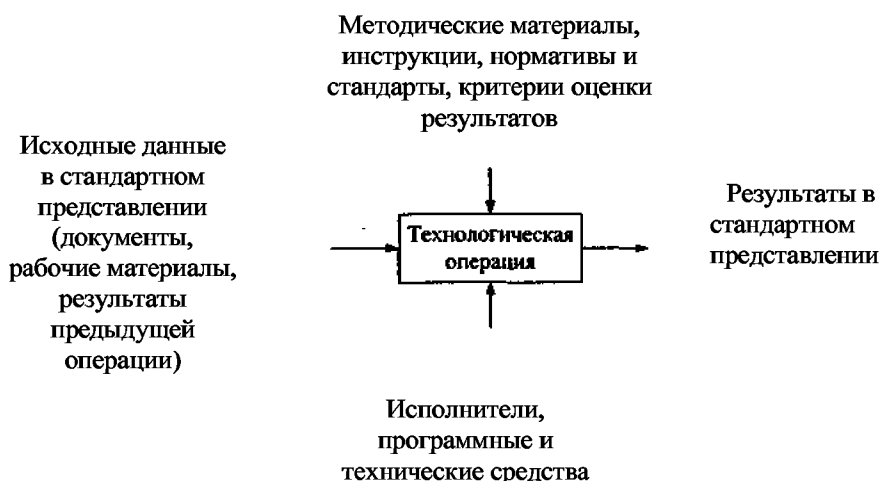


Рис. 1.1. Структура описания технологической операции

определяющий совокупность методов, используемых на разных этапах разработки, или *методологию*.

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. 1.2). Сложность программ в машинных кодах ограничивалась способностью программиста одновременно мысленно отслеживать последовательность выполняемых операций и местонахождение данных при программировании.

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.

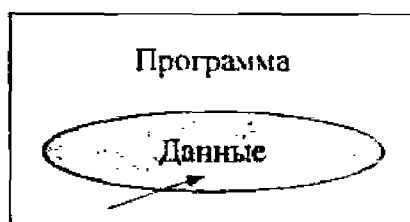


Рис. 1.2. Структура первых программ

(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. 1.3).

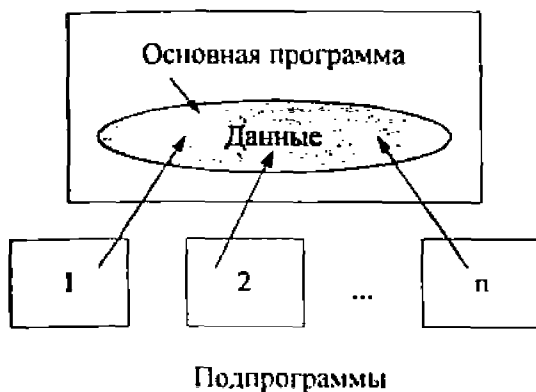


Рис. 1.3. Архитектура программы с глобальной областью данных

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, было предложено в подпрограммах размещать *локальные данные* (рис. 1.4).

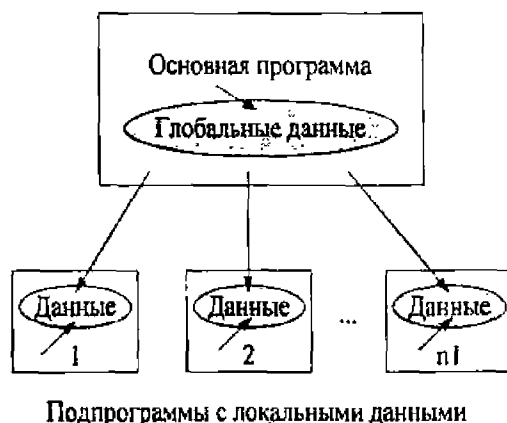


Рис. 1.4. Архитектура программы, использующей подпрограммы с локальными данными

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 отсутствии 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 усложняло 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 *структурирования 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.

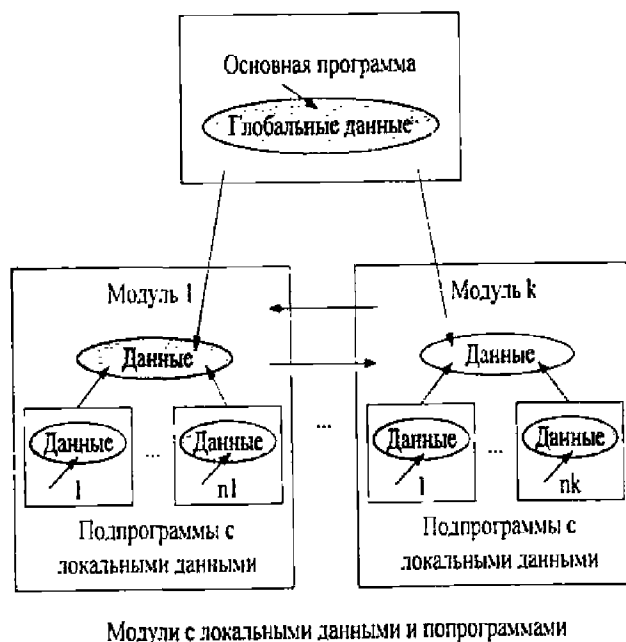


Рис. 1.5. Архитектура программы, состоящей из модулей

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. 1.5). 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 C (C++), языки Ада и Modula.

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. 1.6).

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) программы. 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:

- 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 одного

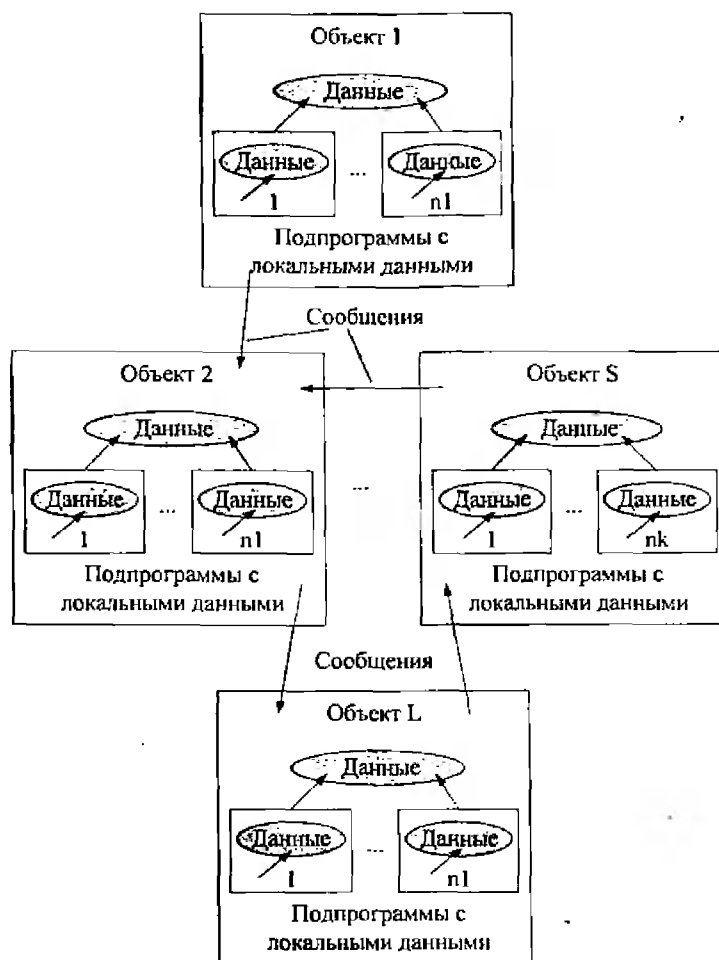


Рис. 1.6. Архитектура программы
при объектно-ориентированном программировании

языка программирования высокого уровня и одного компилятора, а значит, требует наличия исходных кодов используемых библиотек классов;

- 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 *стандартизированные 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.

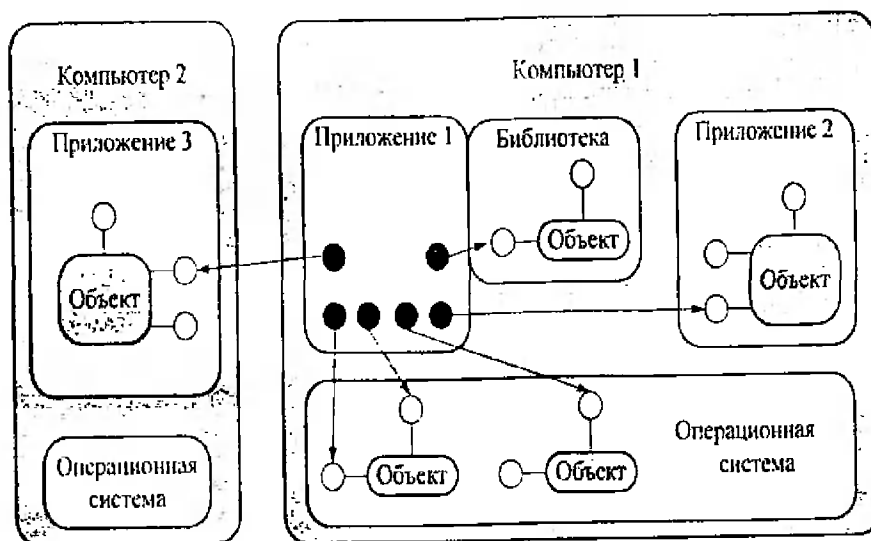


Рис. 1.7. Взаимодействие программных компонентов различных типов

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. 1.7). 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 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:

- **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;

- локальный сервер — создается отдельным процессом (модулем, exe), который работает на одном компьютере с клиентом;
- 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 *диспитейфейса* (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 Интернет.

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

- 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;
- открытость и мобильность - спецификации технологии недавно были переданы в Open Group как основа открытого стандарта;
- возможность написания приложений с использованием знакомых средств разработки, например, Visual Basic, Visual C++, Borland Delphi, Borland C++ и любых средств разработки на Java;
- 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);
- стандартность - технология ActiveX основана на широко используемых стандартах Internet (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 OMC (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 программных 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. Проблемы разработки сложных программных систем

Большинство современных программных систем объективно очень сложны. Эта сложность обуславливается многими причинами, главной из которых является *логическая сложность решаемых ими задач*.

Пока вычислительных установок было мало, и их возможности были ограничены, ЭВМ применяли в очень узких областях науки и техники, причем, в первую очередь, там, где решаемые задачи были хорошо детерминированы и требовали значительных вычислений. В наше время, когда созданы мощные компьютерные сети, появилась возможность переложить на них решение сложных ресурсоемких задач, о компьютеризации которых раньше никто, и не думал. Сейчас в процесс компьютеризации вовлекаются совершенно новые предметные области, а для уже освоенных областей усложняются уже сложившиеся постановки задач.

Дополнительными факторами, увеличивающими сложность разработки программных систем, являются [10]:

- сложность формального определения требований к программным системам;
- отсутствие удовлетворительных средств описания поведения дискретных систем с большим числом состояний при недетерминированной последовательности входных воздействий;
- коллективная разработка;
- необходимость увеличения степени повторяемости кодов.

Сложность определения требований к программным системам. Сложность определения требований к программным системам обуславливается двумя факторами. Во-первых, при определении требований необходимо учесть большое количество различных факторов. Во-вторых, разработчики программных систем не являются специалистами в автоматизируемых предметных

областях, а специалисты в предметной области, как правило, не могут сформулировать проблему в нужном ракурсе.

Отсутствие удовлетворительных средств формального описания поведения дискретных систем. В процессе создания программных систем используют языки сравнительно низкого уровня. Это приводит к ранней детализации операций в процессе создания программного обеспечения и увеличивает объем описаний разрабатываемых продуктов, который, как правило, превышает сотни тысяч операторов языка программирования. Средств же позволяющих детально описывать *поведение* сложных дискретных систем на более высоком уровне, чем универсальный язык программирования, не существует.

Коллективная разработка. 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].

Необходимость увеличения степени повторяемости кодов. На сложность разрабатываемого программного продукта влияет и то, что для увеличения производительности труда компании стремятся к созданию библиотек компонентов, которые можно было бы использовать в дальнейших разработках. Однако в этом случае компоненты приходится делать более универсальными, что в конечном итоге увеличивает сложность разработки.

Вместе взятые, эти факторы существенно увеличивают сложность процесса разработки. Однако очевидно, что все они напрямую связаны со сложностью объекта разработки - программной системы.

1.3. Блочный-иерархический подход к созданию сложных систем

Практика показывает, что подавляющее большинство сложных систем как в природе, так и в технике имеет иерархическую внутреннюю структуру. Это связано с тем, что обычно связи элементов сложных систем различны как по типу, так и по силе, что и позволяет рассматривать эти системы как некоторую *совокупность взаимозависимых подсистем*. Внутренние связи элементов таких подсистем сильнее, чем связи между подсистемами. Например, компьютер состоит из процессора, памяти и внешних устройств, а Солнечная система включает Солнце и планеты, вращающиеся вокруг него.

В свою очередь, используя то же различие связей, можно каждую подсистему разделить на подсистемы и т. д. до самого нижнего «элементарного» уровня, причем выбор уровня, компоненты которого следует считать элементарными, остается за исследователем. На элементарном уровне система, как правило, состоит из немногих типов подсистем, по-разному скомбинированных и организованных. Иерархии такого типа получили название «целое-часть».

Поведение системы в целом обычно оказывается сложнее поведения отдельных частей, причем из-за более сильных внутренних связей особенности системы в основном обусловлены отношениями между ее частями, а не частями как таковыми.

В природе существует еще один вид иерархии - иерархия «простое-сложное» или иерархия развития (усложнения) систем в процессе эволюции. В этой иерархии любая функционирующая система является результатом развития более простой системы. Именно данный вид иерархии реализуется механизмом наследования объектно-ориентированного программирования.

Будучи в значительной степени отражением природных и технических систем, программные системы обычно являются иерархическими, т. е. обладают описанными выше свойствами. На этих свойствах иерархических систем строится *блочный-иерархический подход* к их исследованию или созданию. Этот подход предполагает сначала создавать части таких объектов (блоки, модули), а затем собирать из них сам объект.

Процесс разбиения сложного объекта на сравнительно независимые части получил название *декомпозиции*. При декомпозиции учитывают, что связи между отдельными частями должны быть

слабее, чем связи элементов внутри частей. Кроме того, чтобы из полученных частей можно было собрать разрабатываемый объект, в процессе декомпозиции необходимо определить все виды связей частей между собой.

При создании очень сложных объектов процесс декомпозиции выполняется многократно: каждый блок, в свою очередь, декомпозируют на части пока не получают блоки, которые сравнительно легко разработать. Данный метод разработки получил название *пошаговой детализации*.

Существенно и то, что в процессе декомпозиции стараются выделить аналогичные блоки, которые можно было бы разрабатывать на общей основе. Таким образом, как уже упоминалось выше, обеспечивают увеличение степени повторяемости кодов и, соответственно, снижение стоимости разработки.

Результат декомпозиции обычно представляют в виде схемы *иерархии*, на нижнем уровне которой располагают сравнительно простые блоки, а на верхнем - объект, подлежащий разработке. На каждом иерархическом уровне описание блоков выполняют с определенной степенью детализации, *абстрагируясь* от несущественных деталей. Следовательно, для каждого уровня используют свои формы документации и свои модели, отражающие сущность процессов, выполняемых каждым блоком. Так для объекта в целом, как правило, удается сформулировать лишь самые общие требования, а блоки нижнего уровня должны быть специфицированы так, чтобы из них действительно можно было собрать работающий объект. Другими словами, *чем больше блок, тем более абстрактным должно быть его описание* (рис. 1.8).

При соблюдении этого принципа разработчик сохраняет возможность осмысления проекта и, следовательно, может принимать наиболее правильные решения на каждом этапе, что называют локальной оптимизацией (в отличие от глобальной оптимизации характеристик объектов, которая для действительно сложных объектов не всегда возможна).

Примечание. Следует иметь в виду, что понятие сложного объекта по мере совершенствования технологий изменяется, и то, что было сложным вчера, не обязательно останется сложным завтра.

Итак, в основе блочно-иерархического подхода лежат декомпозиция и иерархическое упорядочение. Важную роль играют также следующие принципы:

- непротиворечивость — контроль согласованности элементов между собой;
- полнота - контроль на присутствие лишних элементов;
- формализация - строгость методического подхода;
- повторяемость - необходимость выделения одинаковых блоков для удешевления и ускорения разработки;
- локальная оптимизация - оптимизация в пределах уровня иерархии.

Совокупность языков моделей, постановок задач, методов описаний некоторого иерархического уровня принято называть *уровнем проектирования*.

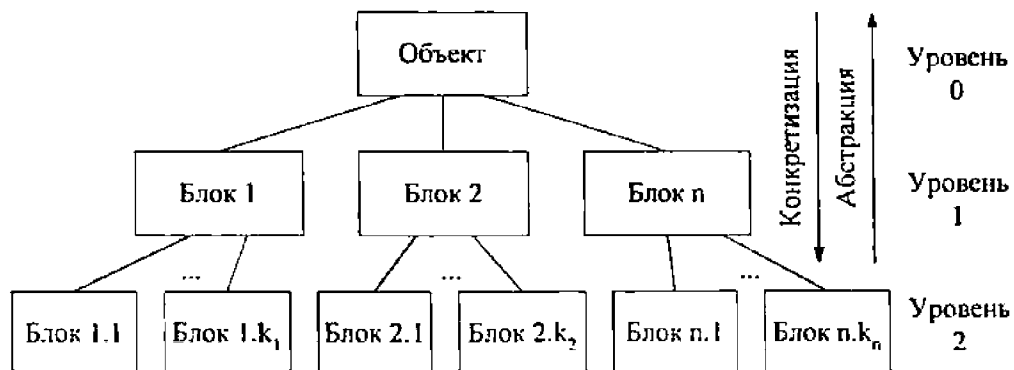


Рис. 1.8. Соотношение абстрактного и конкретного в описании блоков при блочно-иерархическом подходе

Каждый объект в процессе проектирования, как правило, приходится рассматривать с нескольких сторон. Различные взгляды на объект проектирования принято называть *асpekтами проектирования*.

Помимо того, что использование блочно-иерархического подхода делает возможным создание сложных систем, он также:

- упрощает проверку работоспособности, как системы в целом, так и отдельных блоков;
- обеспечивает возможность модернизации систем, например, замены ненадежных блоков с сохранением их интерфейсов.

Необходимо отметить, что использование блочно-иерархического подхода применительно к программным системам стало возможным только после конкретизации общих положений подхода и внесения некоторых изменений в процесс проектирования. При этом структурный подход учитывает только свойства иерархии «целое-часть», а объектный - использует еще и свойства иерархии «простое-сложное».

1.4. Жизненный цикл и этапы разработки программного обеспечения

Жизненным циклом программного обеспечения называют период от момента появления идеи создания некоторого программного обеспечения до момента завершения его поддержки фирмой-разработчиком или фирмой, выполнявшей сопровождение.

Состав процессов жизненного цикла регламентируется международным стандартом ISO/IEC 12207: 1995 «Information Technology - Software Life Cycle Processes» («Информационные технологии - Процессы жизненного цикла программного обеспечения»). ISO - International Organization for Standardization - Международная организация по стандартизации. IEC - International Electrotechnical Commission - Международная комиссия по электротехнике.

Этот стандарт описывает структуру жизненного цикла программного обеспечения и его процессы. *Процесс* жизненного цикла определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. На рис. 1.9 представлены процессы жизненного цикла по указанному стандарту. Каждый процесс характеризуется определенными задачами и методами их решения, а также исходными данными и результатами.

Процесс разработки (development process) в соответствии со стандартом предусматривает

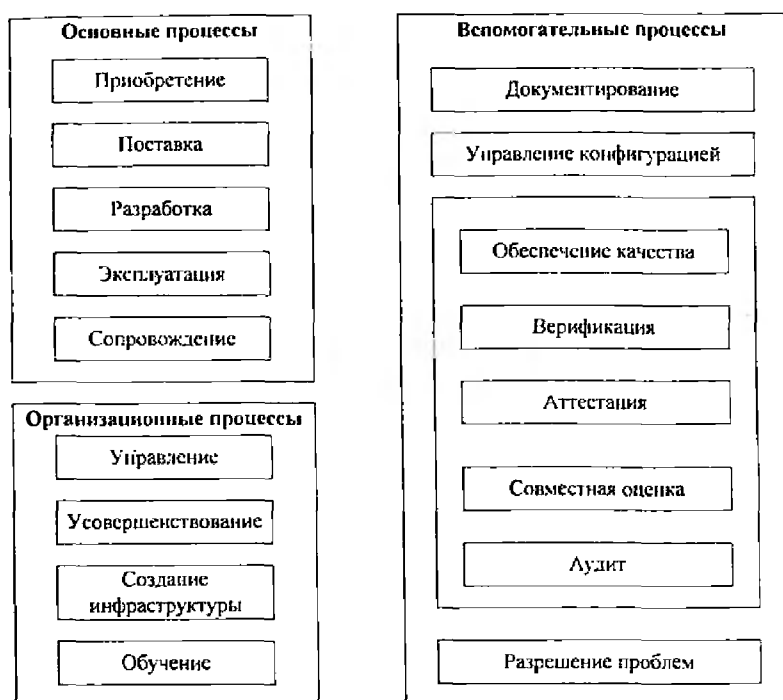


Рис. 1.9. Структура процессов жизненного цикла программного обеспечения

действия и задачи, выполняемые разработчиком, и охватывает работы по созданию программного обеспечения и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, а также подготовку материалов, необходимых для проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала, и т. д.

По стандарту процесс разработки включает следующие действия:

- *подготовительную работу* - выбор модели жизненного цикла (см. далее), стандартов, методов и средств разработки, а также составление плана работ;

- *анализ требований к системе* - определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т. д.;

- *проектирование архитектуры системы* - определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;

- *анализ требований к программному обеспечению* - определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надежности и безопасности, эргономических требований, требований к используемым данным, установке, приемке, пользовательской документации, эксплуатации и сопровождению;

- *проектирование архитектуры программного обеспечения* - определение структуры программного обеспечения, документирование интерфейсов его компонентов, разработку предварительной версии пользовательской документации, а также требований к тестам и плана интеграции;

- *детальное проектирование программного обеспечения* – подробное описание компонентов программного обеспечения и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования компонентов программного обеспечения, обновление плана интеграции компонентов;

- *кодирование и тестирование программного обеспечения* – разработку и документирование каждого компонента, а также совокупности тестовых процедур и данных для их тестирования, тестирование компонентов, обновление пользовательской документации, обновление плана интеграции программного обеспечения;

- *интеграцию программного обеспечения* - сборку программных компонентов в соответствии с планом интеграции и тестирование программного обеспечения на соответствие квалификационным требованиям, представляющих собой набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт, как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;

- *квалификационное тестирование программного обеспечения* - тестирование программного обеспечения в присутствии заказчика для демонстрации его соответствия требованиям и готовности к эксплуатации; при этом проверяется также готовность и полнота технической и пользовательской документации

- *интеграцию системы* - сборку всех компонентов системы, включая программное обеспечение и оборудование;

- *квалификационное тестирование системы* - тестирование системы на соответствие требованиям к ней и проверка оформления и полноты документации;

- *установку программного обеспечения* - установку программного обеспечения на оборудовании заказчика и проверку его работоспособности;

- *приемку программного обеспечения* - оценку результатов квалификационного тестирования программного обеспечения и системы в целом и документирование результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

Указанные действия можно сгруппировать, условно выделив следующие основные этапы разработки программного обеспечения [10] (в скобках указаны соответствующие *стадии разработки* по ГОСТ 19.102-77 «Стадии разработки»):

- постановка задачи (стадия «Техническое задание»);
- анализ требований и разработка спецификаций (стадия «Эскизный проект»);
- проектирование (стадия «Технический проект»);
- реализация (стадия «Рабочий проект»).

Традиционно разработка также включала этап *сопровождения* (началу этого этапа соответствует стадия «Внедрение» по ГОСТ). Однако по международному стандарту в соответствии с изменениями, произошедшими в индустрии разработки программного обеспечения, этот процесс теперь рассматривается отдельно.

Условность выделения этапов связана с тем, что на любом этапе возможно принятие решений, которые потребуют пересмотра решений, принятых ранее (см. § 1.5).

Постановка задачи. В процессе *постановки задачи* четко формулируют назначение программного обеспечения и определяют основные требования к нему. Каждое требование представляет собой описание необходимого или желаемого свойства программного обеспечения. Различают *функциональные требования*, определяющие функции, которые должно выполнять разрабатываемое программное обеспечение, и *эксплуатационные требования*, определяющие особенности его функционирования.

Требования к программному обеспечению, имеющему *прототипы*, обычно определяют по аналогии, учитывая структуру и характеристики уже существующего программного обеспечения. Для формулирования требований к программному обеспечению, не имеющему аналогов, иногда необходимо провести специальные исследования, называемые *предпроектными*. В процессе таких исследований определяют разрешимость задачи, возможно, разрабатывают методы ее решения (если они новые) и устанавливают наиболее существенные характеристики разрабатываемого программного обеспечения. Для выполнения предпроектных исследований, как правило, заключают договор на выполнение научно-исследовательских работ. В любом случае этап постановки задачи заканчивается разработкой *технического задания*, фиксирующего принципиальные требования, и принятием основных проектных решений (см. гл. 3).

Анализ требований и определение спецификаций. *Спецификациями* называют точное формализованное описание функций и ограничений разрабатываемого программного обеспечения. Соответственно различают *функциональные* и *эксплуатационные* спецификации. Совокупность спецификаций представляет собой *общую* логическую модель проектируемого программного обеспечения.

Для получения спецификаций выполняют анализ требований технического задания, формулируют содержательную постановку задачи, выбирают математический аппарат формализации, строят модель предметной области, определяют подзадачи и выбирают или разрабатывают методы их решения. Часть спецификаций может быть определена в процессе предпроектных исследований и, соответственно, зафиксирована в техническом задании.

На этом этапе также целесообразно сформировать тесты для поиска ошибок в проектируемом программном обеспечении, обязательно указав ожидаемые результаты.

Проектирование. Основной задачей этого этапа является определение *подробных* спецификаций разрабатываемого программного обеспечения. Процесс проектирования сложного программного обеспечения обычно включает:

- проектирование общей структуры - определение основных компонентов и их взаимосвязей;
- декомпозицию компонентов и построение структурных иерархий в соответствии с рекомендациями блочно-иерархического подхода;
- проектирование компонентов.

Результатом проектирования является *детальная модель* разрабатываемого программного обеспечения вместе со спецификациями его компонентов всех уровней. Тип модели зависит от выбранного подхода (структурный, объектный или компонентный) и конкретной технологии проектирования. Однако в любом случае процесс проектирования охватывает как проектирование программ (подпрограмм) и определение взаимосвязей между ними, так и проектирование данных, с которыми взаимодействуют эти программы или подпрограммы.

Принято различать также два аспекта проектирования:

- логическое проектирование, которое включает те проектные операции, которые непосредственно не зависят от имеющихся технических и программных средств, составляющих среду функционирования будущего программного продукта;

- физическое проектирование - привязка к конкретным техническим и программным средствам среды функционирования, т. е. учет ограничений, определенных в спецификациях.

Реализация. Реализация представляет собой процесс поэтапного написания кодов программы на выбранном языке программирования (кодирование), их тестирование и отладку.

Сопровождение. Сопровождение - это процесс создания и внедрения новых версий программного продукта. Причинами выпуска новых версий могут служить:

- необходимость исправления ошибок, выявленных в процессе эксплуатации предыдущих версий;

- необходимость совершенствования предыдущих версий, например, улучшения интерфейса, расширения состава выполняемых функций или повышения его производительности;

- изменение среды функционирования, например, появление новых технических средств и/или программных продуктов, с которыми взаимодействует сопровождаемое программное обеспечение.

На этом этапе в программный продукт вносят необходимые изменения, которые так же, как в остальных случаях, могут потребовать пересмотра проектных решений, принятых на любом предыдущем этапе. С изменением модели жизненного цикла программного обеспечения (см. далее) роль этого этапа существенно возросла, так как продукты теперь создаются итерационно: сначала выпускается сравнительно простая версия, затем следующая с большими возможностями, затем следующая и т. д. Именно это и послужило причиной выделения этапа сопровождения в отдельный процесс жизненного цикла в соответствии с стандартом ISO/IEC 12207.

Рассматриваемый стандарт только называет и определяет процессы жизненного цикла программного обеспечения, не конкретизируя в деталях, как реализовывать или выполнять действия и задачи, включенные в эти процессы. Эти вопросы регламентируются соответствующими методами, методиками и т. п. Прежде, чем перейти к подробному рассмотрению последних, проанализируем эволюцию схем разработки программного обеспечения от момента их появления до настоящего времени.

1.5. Эволюция моделей жизненного цикла программного обеспечения

На протяжении последних тридцати лет в программировании сменились три модели жизненного цикла программного обеспечения; каскадная, модель с промежуточным контролем и спиральная.

Каскадная модель. Первоначально (1970-1985 годы) была предложена и использовалась *каскадная схема разработки программного обеспечения* (рис. 1.10), которая предполагала, что переход на следующую стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии. Достоинствами такой схемы являются:

- получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;

- простота планирования процесса разработки.

Именно такую схему и используют обычно при блочно-иерархическом подходе к разработке сложных *технических* объектов, обеспечивая очень высокие параметры эффективности разработки. Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования. Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречается крайне редко.

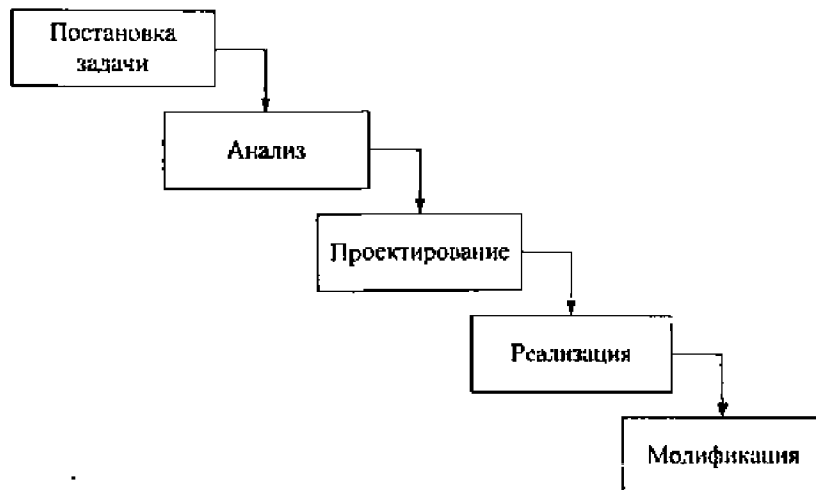


Рис. 1.10. Каскадная схема разработки программного обеспечения

В целом необходимость возвратов на предыдущие стадии обусловлена следующими причинами:

- неточные спецификации, уточнение которых в процессе разработки может привести к необходимости пересмотра уже принятых решений;
- изменение требований заказчика непосредственно в процессе разработки;
- быстрое моральное устаревание используемых технических и программных средств;
- отсутствие удовлетворительных средств описания разработки на стадиях постановки задачи, анализа и проектирования.

Отказ от уточнения (изменения) спецификаций приведет к тому, что законченный продукт не будет удовлетворять потребности пользователей. При отказе от учета смены оборудования и программной среды пользователь получит морально устаревший продукт. А отказ от пересмотра неудачных проектных решений приводит к ухудшению структуры программного продукта и, соответственно, усложнит, растянет по времени и удорожит процесс его создания. Реальный процесс разработки, таким образом, носит итерационный характер.

Модель с промежуточным контролем. Схема, поддерживающая итерационный характер процесса разработки, была названа *схемой с промежуточным контролем* (рис. 1.11). Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

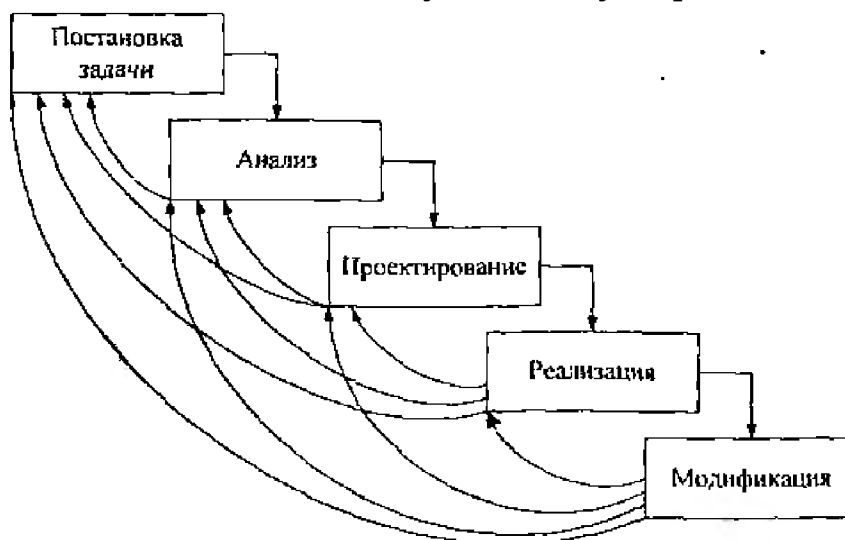


Рис. 1.11. Схема разработки программного обеспечения с промежуточным контролем

Примечание. Народная мудрость в подобных случаях говорит «лучшее - враг хорошего». Осталось только понять, что можно считать «хорошим» и как все-таки добиться лучшего...

Спиральная модель. Для преодоления перечисленных проблем в середине 80-х годов XX в. была предложена *спиральная схема* (рис. 1.12). В соответствии с данной схемой программное обеспечение создается не сразу, а итерационно с использованием метода прототипирования, базирующегося на создании прототипов. Именно появление прототипирования привело к тому, что процесс модификации программного обеспечения перестал восприниматься, как «необходимое зло», а стал восприниматься как отдельный важный процесс.

Прототипом называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемого программного обеспечения.

На первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй - добавляют некоторый ограниченный набор функций. На последующих этапах этот набор расширяют, наращивая возможности данного продукта.

Основным достоинством данной схемы является то, что, начиная с некоторой итерации, на которой обеспечена определенная функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

- сократить время до появления первых версий программного продукта;
- заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;

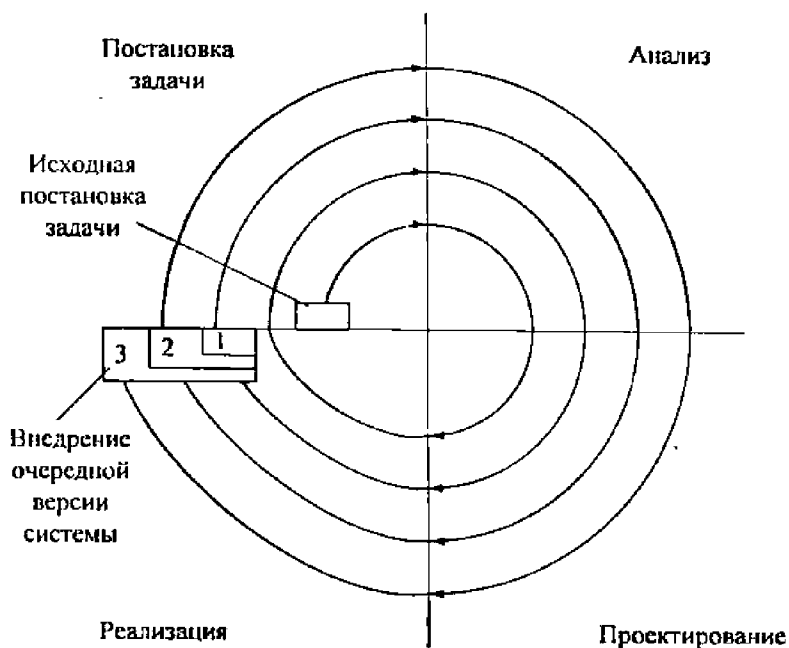


Рис. 1.12. Спиральная или итерационная схема разработки программного обеспечения

- ускорить формирование и уточнение спецификаций за счет появления практики использования продукта;
- уменьшить вероятность морального устаревания системы за время разработки.

Основной проблемой использования спиральной схемы является определение моментов перехода на следующие стадии. Для ее решения обычно ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

Изменение жизненного цикла программного обеспечения при использовании CASE-технологий. CASE-технологии представляют собой совокупность методологий анализа, проектирования, разработки и сопровождения сложных программных систем, основанных как на

структурном, так и на объектном подходах, которые поддерживаются комплексом взаимосвязанных средств автоматизации. В основе любой CASE-технологии лежит парадигма методология/метод/нотация/средство.

Методология строится на базе некоторого подхода и определяет шаги работы, их последовательность, а также правила распределения и назначения методов. Метод определяет способ достижения той или иной цели - выполнение шага работы.

Нотацией называют систему обозначений, используемых для описания некоторого класса моделей. Нотации бывают графические (предоставление моделей в виде графов, диаграмм, таблиц, схем и т. п.) и текстовые (описания моделей на формальных и естественных языках). В CASE-технологиях нотации используют для описания структуры проектируемой системы, элементов данных, этапов обработки и т. п.

Средства - инструментарий для поддержки методов: средства создания и редактирования графического проекта, организации проекта в виде иерархии уровней абстракции, а также проверки соответствия компонентов разных уровней. Различают:

- CASE-средства анализа требований, проектирования спецификаций и структуры, редактирования интерфейсов (первое поколение CASE-I);
- CASE-средства генерации исходных текстов и реализации интегрированного окружения поддержки полного жизненного цикла разработки программного обеспечения (второе поколение CASE-II).

CASE-I в основном включают средства для поддержки графических моделей, проектирования спецификаций, экранных редакторов и словарей данных. CASE-II отличается существенно большими возможностями, обеспечивая: контроль, анализ и связывание системной информации и информации по управлению процессом проектирования, построение прототипов и моделей системы, тестирование, верификацию и анализ сгенерированных программ.

Автоматизируя трудоемкие операции, современные CASE-средства существенно повышают производительность труда программистов и улучшают качество создаваемого программного обеспечения. Они:

- обеспечивают автоматизированный контроль совместимости спецификаций проекта;
- уменьшают время создания прототипа системы;
- ускоряют процесс проектирования и разработки;
- автоматизируют формирование проектной документации для всех этапов жизненного цикла в соответствии с современными стандартами;
- частично генерируют коды программ для различных платформ разработки;
- поддерживают технологии повторного использования компонентов системы;
- обеспечивают возможность восстановления проектной документации по имеющимся исходным кодам.

Появление CASE-технологий изменило все этапы жизненного цикла программного обеспечения, при этом наибольшие изменения касаются анализа и проектирования, которые предполагают строгое и наглядное описание разрабатываемого программного обеспечения.

В табл. 1.1 показано, какие качественные изменения процесса разработки программного обеспечения происходят при переходе к использованию CASE-средств.

Использование CASE-средств позволяет существенно снизить трудозатраты на разработку сложного программного обеспечения {табл. 1.2 [30]} в основном за счет автоматизации процессов документирования и контроля. Однако следует иметь в виду, что современные CASE-средства дороги, а их использование требует *более высокой квалификации разработчиков*. Следовательно, их имеет смысл использовать в сложных проектах, причем, чем сложнее разрабатываемое программное обеспечение, тем больше выигрыш от использования CASE-технологий. На сегодняшний день практически все промышленно производимое сложное программное обеспечение разрабатывается с использованием CASE-средств.

Таблица 1.1

Традиционная разработка	Разработка с использованием CASE - средств
Основные усилия на кодирование и тестирование	Основные усилия на анализ и проектирование
«Бумажные» спецификации	Быстрое итерационное прототипирование
Ручное кодирование	Автоматическая генерация кодов
Ручное документирование	Автоматическая генерация документации
Тестирование кодов	Автоматический контроль проекта
Сопровождение кодов	Сопровождение спецификаций проектирования

Таблица 1.2

Способ разработки	Трудозатраты этапа разработки, %			
	Анализ	Проектирование	Кодирование	Тестирование
Традиционная разработка	20	15	20	45
Структурный подход	30	30	15	25
CASE-технологий	40	40	5	15

1.6. Ускорение разработки программного обеспечения.

Разработка спиральной модели жизненного цикла программного обеспечения и CASE-технологий позволили сформулировать условия, выполнение которых сокращает сроки создания программного обеспечения.

Современная технологии проектирования, разработки и сопровождения программного обеспечения, должна отвечать следующим требованиям:

- поддержка полного жизненного цикла программного обеспечения;
- гарантированное достижение целей разработки с заданным качеством и в установленное время;
- возможность выполнения крупных проектов в виде подсистем, разрабатываемых группами исполнителей ограниченной численности (3-7 человек) с последующей интеграцией составных частей, и координации ведения общего проекта;
- минимальное время получения работоспособной системы;
- возможность управления конфигурацией проекта, ведения версий проекта и автоматического выпуска проектной документации по каждой версии;
- независимость выполняемых проектных решений от средств реализации (СУБД, операционных систем, языков и систем программирования);
- поддержка комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях жизненного цикла.

Этим требованиям отвечает *технология* RAD (Rapid Application Development - Быстрая разработка приложений). Эта технология ориентирована, как следует из названия, на максимально быстрое получение первых версий разрабатываемого программного обеспечения. Она предусматривает выполнение следующих условий:

- ведение разработки небольшими группами разработчиков (3-7 человек), каждая из которых проектирует и реализует отдельные подсистемы проекта - позволяет улучшить управляемость проекта;
- использование итерационного подхода способствует уменьшению времени получения работоспособного прототипа;
- наличие четко проработанного графика цикла, рассчитанного не более чем на три месяца, существенно увеличивает эффективность работы.

Процесс разработки при этом делится на следующие этапы: анализ и планирование требований пользователей, проектирование, реализация, внедрение.

На этапе *анализа и планирования требований* формулируют наиболее приоритетные требования, что ограничивает масштаб проекта.

На этапе *проектирования*, используя имеющиеся CASE-средства, детально описывают процессы системы, устанавливают требования разграничения доступа к данным и определяют состав необходимой документации. При этом для наиболее сложных процессов создают частичный прототип: разрабатывают экранную форму и диалог. По результатам анализа процессов определяют количество так называемых функциональных точек и принимают решение о количестве подсистем и, соответственно, команд, участвующих в разработке.

Под *функциональной точкой* в технологии RAD понимают любой из следующих функциональных элементов разрабатываемой системы:

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

Нормы, рассчитанные исходя из экспертных оценок, для систем со значительной повторяемостью кодов определяются следующим образом:

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

В соответствии с этими нормами разрабатываемую систему делят на подсистемы, слабо связанные по данным и функциям, и точно определяют интерфейсы между различными частями. Использование CASE-средств при этом позволяет избежать неконтролируемого искажения данных при передаче информации о проекте со стадии на стадию.

Далее разработка ведется группами разработчиков, которые продолжают прорабатывать свои части системы. Действия различных групп разработчиков при этом должны быть хорошо скоординированы.

На этапе *реализации* выполняют итеративное построение реальной системы, причем при этом для контроля над выполнением требований к создаваемой системе привлекаются будущие пользователи.

Части постепенно интегрируют в систему, причем при подключении каждой части выполняют тестирование. На завершающих этапах разработки определяют необходимость создания соответствующих баз данных, которые разрабатываются и подключаются к системе. Далее формулируют требования к аппаратным средствам, устанавливают способы увеличения производительности и завершают подготовку документации по проекту.

На этапе *внедрения* проводят обучение пользователей и осуществляют постепенный переход на новую систему, причем эксплуатация старой версии продолжается до полного внедрения новой

системы.

Технология RAD хорошо зарекомендовала себя для относительно небольших проектов, разрабатываемых для конкретного заказчика. Такие системы не требуют высокого уровня планирования и жесткой дисциплины проектирования. Однако эта технология не применима для построения сложных расчетных программ, операционных систем или программ управления сложными объектами в реальном масштабе времени, т. е. программ с большим процентом уникального кода. Не годится она и в случае создания приложений, от которых зависит безопасность людей, например, систем управления самолетами или атомными электростанциями, так как технология RAD предполагает, что первые несколько версий не будут полностью работоспособны, что в данном случае полностью исключается.

1.7. Оценка качества процессов создания программного обеспечения

Как уже упоминалось выше, текущий период на рынке программного обеспечения характеризуется переходом от штучного ремесленного производства программных продуктов к их промышленному}- созданию. Соответственно возросли требования к качеству разрабатываемого программного обеспечения, что требует совершенствования процессов их разработки. На настоящий момент существует несколько стандартов, связанных с оценкой качества этих процессов, которое обеспечивает организация-разработчик. К наиболее известным относят:

- 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);
- рабочая версия международного стандарта ISO/IEC 15504: Information Technology - Software Process Assessment; эта версия более известна под названием SPICE - (Software Process Improvement and Capability dEtermination - определение возможностей и улучшение процесса создания программного обеспечения).

Серия стандартов ISO 9000. В серии ISO 9000 сформулированы необходимые условия для достижения некоторого минимального уровня организации процесса, но не дается никаких рекомендаций по дальнейшему совершенствованию процессов.

CMM. CMM представляет собой совокупность критериев оценки зрелости организации-разработчика и рецептов улучшения существующих процессов.

Примечание. Изначально CMM разрабатывалась и развивалась как методика, позволяющая крупным правительственным организациям США выбирать наилучших поставщиков программного обеспечения. Для этого предполагалось создать исчерпывающее описание способов оценки процессов разработки программного обеспечения и методики их дальнейшего совершенствования. В итоге авторы смогли добиться такой степени подробности и детализации, что стандарт оказался пригодным и для обычных компаний-разработчиков, желающих качественно улучшить существующие процессы разработки, привести их к определенным стандартам.

CMM определяет пять уровней зрелости организаций-разработчиков, причем каждый следующий уровень включает в себя все ключевые характеристики предыдущих.

1. Начальный уровень (initial level)- описан в стандарте в качестве основы для сравнения со следующими уровнями. На предприятии такого уровня организации не существует стабильных условий для создания качественного программного обеспечения. Результат любого проекта целиком и полностью зависит от личных качеств менеджера и опыта программистов, причем успех в одном проекте может быть повторен только в случае назначения тех же менеджеров и программистов на следующий проект. Более того, если эти менеджеры или программисты уходят с предприятия, то резко снижается качество производимых программных продуктов. В стрессовых ситуациях процесс разработки сводится к написанию кода и его минимальному тестированию.

2. *Повторяемый уровень* (repeatable level) - на предприятии внедрены *технологии управления проектами*. При этом планирование и управление проектами основывается на накопленном опыте, существуют стандарты на разрабатываемое программное обеспечение (причем обеспечивается следование этим стандартам) и специальная *группа обеспечения качества*. В случае необходимости организация может взаимодействовать с субподрядчиками. В критических условиях процесс имеет тенденцию скатываться на начальный уровень.

3. *Определенный уровень* (defined level) - характеризуется тем, что стандартный процесс создания и сопровождения программного обеспечения полностью документирован (включая и разработку ПО, и управление проектами). Подразумевается, что в процессе стандартизации происходит переход на наиболее эффективные практики и технологии. Для создания и поддержания подобного стандарта в организации должна быть создана специальная группа. Наконец, обязательным условием для достижения данного уровня является наличие на предприятии программы постоянного *повышения квалификации и обучения сотрудников*. Начиная с этого уровня, организация перестает зависеть от качеств конкретных разработчиков, и процесс не имеет тенденции скатываться на уровень ниже в стрессовых ситуациях.

4. *Управляемый уровень* (managed level) - в организации устанавливаются *количественные показатели качества*, как на программные продукты, так и на процесс в целом. Таким образом, более совершенное управление проектами достигается за счет уменьшения отклонений различных показателей проекта. При этом осмысленные вариации в производительности процесса можно отличить от случайных вариаций (шума), особенно в хорошо освоенных областях.

5. *Оптимизирующий уровень* (optimizing level) - характеризуется тем, что мероприятия по улучшению применяются не только к существующим процессам, но и для оценки эффективности ввода новых технологий. Основной задачей всей организации на этом уровне является *постоянное улучшение* существующих процессов. При этом улучшение процессов в идеале должно помогать предупреждать возможные ошибки или дефекты. Кроме того, должны вестись работы по уменьшению стоимости разработки программного обеспечения, например с помощью создания и повторного использования компонентов.

Сертификационная оценка соответствия всех ключевых областей проводится по 10-балльной шкале. Для успешной квалификации данной ключевой области необходимо набрать не менее 6 баллов. Оценка ключевой области осуществляется по следующим показателям:

- заинтересованность руководства в данной области, например, планируется ли практическое внедрение данной ключевой области, существует ли понимание у руководства необходимости данной области и т. д.;
- насколько широко данная область применяется в организации, например, оценке в 4 балла соответствует фрагментарное применение;
- успешность использования данной области на практике, например, оценке в 0 баллов соответствует полное отсутствие какого-либо эффекта, а оценка в 8 баллов выставляется при наличии систематического и измеримого положительного результата практически во всей организации.

В принципе, можно сертифицировать только один процесс или подразделение организации, например, подразделение разработки программного обеспечения компании IBM сертифицировано на пятый уровень. Кстати, в мире существует совсем немного компаний, которые могут похвастаться наличием у них пятого уровня СММ хотя бы в одном из подразделений - таких всего около 50-ти. С другой стороны, насчитывается несколько тысяч компаний, сертифицированных по третьему или четвертому уровням, т. е. существует колоссальный разрыв между оптимизированным уровнем зрелости и предыдущими уровнями. Однако еще больший разрыв наблюдается между количеством организаций начального уровня и числом их более продвинутых собратьев - по некоторым оценкам, свыше 70 % всех компаний-разработчиков находится на первом уровне СММ [3].

SPICE. Стандарт SPICE унаследовал многие черты более ранних стандартов, в том числе и уже упоминавшихся ISO 9001 и СММ. Больше всего SPICE напоминает СММ. Точно так же, как и в СММ, основной задачей организации является постоянное улучшение процесса разработки

программного обеспечения. Кроме того, в SPICE тоже используется схема с различными уровнями возможностей (в SPICE определено 6 различных уровней), но эти уровни применяются не только к организации в целом, но и к отдельно взятым процессам.

В основе стандарта лежит *оценка процессов*. Эта оценка выполняется путем сравнения процесса разработки программного обеспечения, существующего в данной организации, с описанной в стандарте моделью. Анализ результатов, полученных на этом этапе, помогает определить сильные и слабые стороны процесса, а также внутренние риски, присущие данному процессу. Это помогает оценить эффективность процессов, определить причины ухудшения качества и связанные с этим издержки во времени или стоимости.

Затем выполняется *определение возможностей процесса*, т. е. возможностей его улучшения. В результате в организации может появиться понимание необходимости *улучшения* того или иного *процесса*. К этому моменту цели совершенствования процесса уже четко сформулированы и остается только техническая реализация поставленных задач. После этого весь цикл работ начинается сначала.

Безусловно, совершенствование процессов жизненного цикла программного обеспечения абсолютно необходимо. Однако следует иметь в виду, что построение «более зрелого» процесса разработки не обязательно обеспечивает создание более качественного программного обеспечения. Это хотя и связанные, но *совершенно различные* процессы.

Использование формальных моделей и методов позволяет создавать понятные, непротиворечивые спецификации на разрабатываемое программное обеспечение. Конечно, внедрение таких методов имеет смысл, хотя оно весьма дорого и трудоемко, а возможности их применения весьма ограничены. Основная же проблема - проблема сложности разрабатываемого программного обеспечения с совершенствованием процессов разработки пока не разрешена. Создание программного обеспечения по-прежнему предъявляет повышенные требования к квалификации тех, кто этим занимается: проектировщикам программного обеспечения и непосредственно программистам.

Контрольные вопросы

1. Что понимают под термином «технология программирования»?
2. Что называют подходом и чем подход отличается от метода?
3. Назовите основные периоды истории развития технологии программирования. Чем характеризуются эти периоды? Как изменялись основные подходы и используемые средства?
4. Дайте определение понятию «сложная иерархическая система». Какой подход используют при разработке таких систем? На каких характеристиках этих систем он основан? В чем особенность данного подхода при разработке программного обеспечения?
5. Что понимают под термином «жизненный цикл программного обеспечения»? Какие основные процессы включают в это понятие?
6. Назовите основные этапы разработки программного обеспечения. Какие основные задачи решаются на этих этапах?
7. Назовите основные модели жизненного цикла программного обеспечения. С чем связано появление новых моделей?
8. Какие технологии называют CASE-технологиями? Почему?
9. Назовите основные составляющие любой CASE-технологии.
10. Перечислите основные положения технологии RAD? Какие программные системы нельзя разрабатывать с использованием этой технологии?
11. Что понимают под моделями качества процессов разработки программного обеспечения? Для чего они разработаны? Что гарантирует сертификация качества процессов? Почему?
12. Почему мы говорим, что современный этап развития технологии программирования характеризуется переходом от ремесленного к промышленному производству программного обеспечения?

2. ПРИЕМЫ ОБЕСПЕЧЕНИЯ ТЕХНОЛОГИЧНОСТИ ПРОГРАММНЫХ ПРОДУКТОВ

В условиях индустриального подхода к разработке и сопровождению программного обеспечения особый вес приобретают технологические характеристики разрабатываемых программ. Для обеспечения необходимых технологических свойств применяют специальные технологические приемы и следуют определенным методикам, сформулированным всем предыдущим опытом создания программного обеспечения. К таким приемам и методикам относят правила декомпозиции, методы проектирования, программирования и контроля качества, которые под общим названием "структурный подход к программированию" были сформулированы еще в 60-х годах XX в. В его основу были положены следующие основные концепции:

- нисходящая разработка;
- модульное программирование;
- структурное программирование;
- сквозной структурный контроль.

2.1. Понятие технологичности программного обеспечения

Под *технологичностью* понимают качество проекта программного продукта, от которого зависят трудовые и материальные затраты на его реализацию и последующие модификации. Хороший проект сравнительно быстро и легко кодируется, тестируется, отлаживается и модифицируется.

Из опыта нескольких поколений разработчиков программного обеспечения известно, что *технологичность программного обеспечения определяется проработанностью его моделей, уровнем независимости модулей, стилем программирования и степенью повторного использования кодов.*

Чем лучше проработана модель разрабатываемого программного обеспечения, тем четче определены подзадачи и структуры данных, хранящие входную, промежуточную и выходную информацию, тем проще их проектирование и реализация и меньше вероятность ошибок, для исправления которых потребуется существенно изменять программу.

Чем выше независимость модулей, тем их легче понять, реализовывать, модифицировать, а также находить в них ошибки и исправлять их.

Стиль программирования, под которым понимают стиль оформления программ и их «структурность», также существенно влияет на читаемость программного кода и количество ошибок программирования. Кризис 60-х годов XX в. был вызван в том числе и стилем программирования, при котором программа напоминала клубок спутанных ниток или блюдо спагетти, и отсутствием языковых конструкций поддержки «структурного» стиля.

Увеличение степени повторного использования кодов предполагает как использование ранее разработанных библиотек подпрограмм или классов, так и унификацию кодов текущей разработки. Причем для данного критерия ситуация не так однозначна, как в предыдущих случаях: если степень повторного использования кодов повышается искусственно (например, путем разработки «суперуниверсальных» процедур), то технологичность проекта может существенно снизиться.

Как следует из определения, высокая технологичность проекта особенно важна, если разрабатывается программный продукт, рассчитанный на многолетнее интенсивное использование, или необходимо обеспечить повышенные требования к его качеству.

2.2. Модули и их свойства

При проектировании достаточно сложного программного обеспечения после определения его общей структуры выполняют декомпозицию компонентов в соответствии с выбранным подходом до получения элементов, которые, по мнению проектировщика, в дальнейшей декомпозиции не

нуждаются.

Как уже упоминалось раньше, в настоящее время используют два способа декомпозиции разрабатываемого программного обеспечения, связанные с соответствующим подходом:

- процедурный (или структурный - по названию подхода);
- объектный.

Примечание. Помимо указанных способов декомпозиции, в теории программирования определяют и другие способы декомпозиции: логическую - на факты и правила, продукционную - на правила продукции и т. п. Эти способы декомпозиции используют в языках искусственного интеллекта, поэтому в настоящем учебнике они рассматриваться не будут.

Результатом процедурной декомпозиции является *иерархия подпрограмм* (процедур), в которой функции, связанные с принятием решения, реализуются подпрограммами верхних уровней, а непосредственно обработка - подпрограммами нижних уровней. Это согласуется с *принципом вертикального управления*, который был сформулирован вместе с другими рекомендациями структурного подхода к программированию. Он также ограничивает возможные варианты передачи управления, требуя, чтобы любая подпрограмма возвращала управление той подпрограмме, которая ее вызвала.

Результатом объектной декомпозиции является совокупность объектов, которые затем реализуют как переменные некоторых специально разрабатываемых типов (классов), представляющих собой совокупность полей данных и методов, работающих с этими полями.

Таким образом, при любом способе декомпозиции получают набор связанных с соответствующими данными подпрограмм, которые в процессе реализации организуют в модули.

Модули. Модулем называют *автономно компилируемую* программную единицу. Термин «модуль» традиционно используется в двух смыслах. Первоначально, когда размер программ был сравнительно невелик, и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т. е. *последовательность связанных фрагментов программы, обращение к которой выполняется по имени*. Со временем, когда размер программ значительно вырос, и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин «модуль» стал использоваться и в смысле *автономно компилируемый набор программных ресурсов*.

Данные модуль может получать и/или возвращать через общие области памяти или параметры.

Первоначально к модулям (еще понимаемым как подпрограммы) предъявлялись следующие требования:

- отдельная компиляция;
- одна точка входа;
- одна точка выхода;
- соответствие принципу вертикального управления;
- возможность вызова других модулей;
- небольшой размер (до 50-60 операторов языка);
- независимость от истории вызовов;
- выполнение одной функции.

Требования одной точки входа, одной точки выхода, независимости от истории вызовов и соответствия принципу вертикального управления были вызваны тем, что в то время из-за серьезных ограничений на объем оперативной памяти программисты были вынуждены разрабатывать программы с максимально возможной повторяемостью кодов. В результате подпрограммы, имеющие несколько точек входа и выхода, были не только обычным явлением, но и считались высоким классом программирования. Следствием же было то, что программы было очень сложно не только модифицировать, но и понять, а иногда и просто полностью отладить.

Со временем, когда основные требования структурного подхода стали поддерживаться языками программирования, и под модулем стали понимать отдельно компилируемую библиотеку ресурсов, требование независимости модулей стало основным.

Практика показала, что чем выше степень независимости модулей, тем:

- легче разобраться в отдельном модуле и всей программе и, соответственно, тестировать, отлаживать и модифицировать ее;
- меньше вероятность появления новых ошибок при исправлении старых или внесении изменений в программу, т. е. вероятность появления «волнового» эффекта;
- проще организовать разработку программного обеспечения группой программистов и легче его сопровождать.

Таким образом, уменьшение зависимости модулей улучшает технологичность проекта. Степень независимости модулей (как подпрограмм, так и библиотек) оценивают двумя критериями: сцеплением и связностью.

Сцепление модулей. *Сцепление* является мерой взаимозависимости модулей, которая определяет, насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях хранит модуль, тем больше он с ними сцеплен.

Различают пять типов сцепления модулей:

- по данным;
- по образцу;
- по управлению;
- по общей области данных;
- по содержимому.

Сцепление по данным предполагает, что модули обмениваются данными, представленными скалярными значениями. При небольшом количестве передаваемых параметров этот тип обеспечивает наилучшие технологические характеристики программного обеспечения.

Например, функция *Max* предполагает сцепление по данным через параметры скалярного типа:

```
Function Max(a, b: integer): integer;  
begin  
  if a>b then Max:=a  
    else Max: =b;  
end;
```

Сцепление по образцу предполагает, что модули обмениваются данными, объединенными в структуры. Этот тип также обеспечивает неплохие характеристики, но они хуже, чем у предыдущего типа, так как 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.

Так, функция *MaxEl*, описанная ниже, предполагает сцепление по образцу (параметр *a* - открытый массив).

```
Function MaxEl(a:array of integer): integer;  
Var i:\vord;  
begin  
  MaxEl:=a [0];  
  for i:=1 to High (a) do  
    if a [i]>MaxEl then MaxEl:=a [i];  
end;
```

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 еще худшие характеристики технологичности разрабатываемого программного обеспечения по сравнению с

предыдущими типами связей.

Например, функция MinMax предполагает сцепление по управлению. так как значение параметра flag влияет на логику программы: если функция MinMax получает значение параметра flag, равное true, то возвращает максимальное значение из двух, а если false, то минимальное:

```
Function MinMax (a, b: integer; flag:boolean): integer;  
begin  
    if(a>b) and (flag) then MinMax: =a  
        else MinMax: =b;  
end;
```

Сцепление по общей области данных предполагает, что модули работают с общей областью данных. Этот тип сцепления считается недопустимым, поскольку:

- программы, использующие данный тип сцепления, очень сложны для понимания при сопровождении программного обеспечения;
- ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля, что существенно усложняет локализацию ошибок;
- при ссылке к данным в общей области модули используют конкретные имена, что уменьшает гибкость разрабатываемого программного обеспечения.

Например, функция MaxA, использующая глобальный массив A, сцеплена с основной программой по общей области:

```
Function MaxA: integer;  
Var i:word;  
begin  
    MaxA: =a[Low(a)];  
    for i: = Low (a) + 1 to High(a) do  
        if a [i]>MaxA then MaxA: = a [i];  
end;
```

Следует иметь в виду, что «подпрограммы с памятью», действия которых зависят от истории вызовов, используют сцепление по общей области, что делает их работу в общем случае непредсказуемой. Именно этот вариант используют статические переменные C и C++.

В случае *сцепления по содержимому* один модуль содержит обращения к внутренним компонентам другого (передает управление внутрь, читает и/или изменяет внутренние данные или сами коды), что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком («черным ящиком»): его содержимое должно учитываться в процессе разработки другого модуля. Современные универсальные языки процедурного программирования, например Pascal, данного типа сцепления в явном виде не поддерживают, но для языков низкого уровня, например Ассемблера, такой вид сцепления остается возможным.

В табл. 2.1 приведены характеристики различных типов сцепления по экспертным оценкам [21, 30]. Допустимыми считают первые три типа сцепления, так как использование остальных приводит к резкому ухудшению технологичности программ.

Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество программного обеспечения принято определять по типу сцепления с худшими характеристиками. Так, если использовано сцепление по данным и сцепление по управлению, то определяющим считают сцепление по управлению.

В некоторых случаях сцепление модулей можно уменьшить, удалив необязательные связи и структурировав необходимые связи. Примером может служить объектно-ориентированное программирование, в котором вместо большого количества параметров метод неявно получает адрес области (структуры), в которой расположены поля объекта, и явно-дополнительные параметры. В результате модули оказываются сцепленными по образцу.

Таблица 2.1

Тип сцепления	Сцепление, балл	Устойчивость к ошибкам других модулей	Наглядность (понятность)	Возможность изменения	Вероятность повторного использования
По данным	1	Хорошая *	Хорошая	Хорошая	Большая
По образцу	3	Средняя	Хорошая *	Средняя	Средняя
По управлению	4	Средняя	Плохая	Плохая	Малая
По общей области	6	Плохая	Плохая	Средняя	Малая
По содержимому	10	Плохая	Плохая	Плохая	Малая

Связность модулей. *Связность* - мера прочности соединения функциональных и информационных объектов внутри одного модуля. Если сцепление характеризует качество отделения модулей, то связность характеризует степень взаимосвязи элементов, реализуемых одним модулем. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи и, соответственно, взаимовлияние модулей. В то же время 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, так как такими элементами сложнее мысленно манипулировать.

Различают следующие виды связности (в порядке убывания уровня):

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

При *функциональной связности* все объекты модуля предназначены для выполнения одной функции (рис. 2.1, а): операции, объединяемые для выполнения одной функции, или данные, связанные с одной функцией. Модуль, элементы которого связаны функционально, имеет четко определенную цель, при его вызове выполняется одна задача, например, подпрограмма поиска минимального элемента массива. Такой модуль имеет максимальную связность, следствием которой являются его хорошие технологические качества: простота тестирования, модификации и сопровождения. Именно с этим связано одно из требований структурной декомпозиции «один модуль - одна между модулями - библиотеками ресурсов. Например, если при проектировании текстового редактора предполагается функция редактирования, то лучше организовать модуль - библиотеку функций редактирования, чем поместить часть функций в один модуль, а часть в другой.

При *последовательной связности* функций выход одной функции служит исходными данными для другой функции (рис. 2.1, б). Как правило, такой модуль имеет одну точку входа, т. е. реализует одну подпрограмму, выполняющую две функции. Считают, что данные, используемые последовательными функциями, также связаны последовательно. Модуль с последовательной связностью функций можно разбить на два или более модулей, как с последовательной, так и с функциональной связностью. Такой модуль выполняет несколько функций, и, следовательно, его технологичность хуже: сложнее организовать тестирование, а при выполнении модификации мысленно приходится разделять функции модуля.

Информационно связанными считают функции, обрабатывающие одни и те же желанные (рис. 2.1, в). При использовании структурных языков программирования раздельное выполнение функций можно осуществить только, если каждая функция реализуется своей подпрограммой функции».

Из тех же соображений следует избегать неструктурированного распределения функции

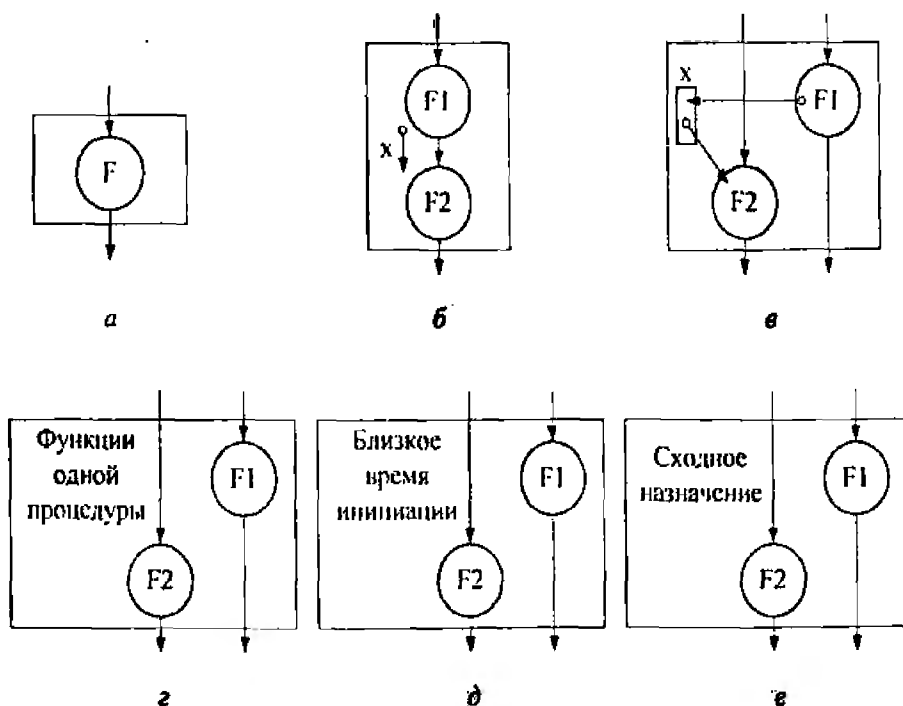


Рис. 2.1. Связность модулей:

а – функциональная; б – последовательная; в – информационная;
г – процедурная; д – временная; е – логическая

Хотя раньше в подобных случаях обычно использовали разные точки входа в модуль, оформленный как одна подпрограмма.

Несмотря на объединение нескольких функций, информационно связанный модуль имеет неплохие показатели технологичности. Это объясняется тем, что все функции, работающие с некоторыми данными, собраны в одно место, что позволяет при изменении формата данных корректировать только один модуль. Информационно связанными также считают данные, которые обрабатываются одной функцией.

Процедурно связаны функции или данные, которые являются частями одного процесса (рис. 2.1, г). Обычно модули с процедурной связностью функций получают, если в модуле объединены функции альтернативных частей программы. При процедурной связности отдельные элементы модуля связаны крайне слабо, так как реализуемые ими действия связаны лишь общим процессом, следовательно, технологичность данного вида связи ниже, чем предыдущего.

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени (рис. 2.1, д). Временная связность данных означает, что они используются в некотором временном интервале. Например, временную связность имеют функции, выполняемые при инициализации некоторого процесса. Отличительной особенностью временной связности является то, что действия, реализуемые такими функциями, обычно могут выполняться в любом порядке. Содержание модуля с временной связностью функций имеет тенденцию меняться: в него могут включаться новые действия и/или исключаться старые. Большая вероятность модификации функции еще больше уменьшает показатели технологичности модулей данного вида по сравнению с предыдущим.

Логическая связь базируется на объединении данных или функций в одну логическую группу

(рис. 2.1, е). В качестве примера можно привести функции обработки текстовой информации или данные одного и того же типа. Модуль с логической связностью функций часто реализует альтернативные варианты одной операции, например, сложение целых чисел и сложение вещественных чисел. Из такого модуля всегда будет вызываться одна какая-либо его часть, при этом вызывающий и вызываемый модули будут связаны по управлению. Понять логику работы модулей, содержащих логически связанные компоненты, как правило, сложнее, чем модулей, использующих временную связность, следовательно, их показатели технологичности еще ниже.

В том случае, если связь между элементами мала или отсутствует, считают, что они имеют *случайную связность*. Модуль, элементы которого связаны случайно, имеет самые низкие показатели технологичности, так как элементы, объединенные в нем, вообще не связаны.

Обратите внимание, что в трех предпоследних случаях связь между несколькими подпрограммами в модуле обусловлена внешними причинами. А в последнем - вообще отсутствует. Это соответствующим образом проецируется на технологические характеристики модулей. В табл. 2.2 представлены характеристики различных видов связности по экспертным оценкам [21, 30].

Анализ табл. 2.2 показывает, что на практике целесообразно использовать функциональную, последовательную и информационную связности.

Таблица 2.2

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

Как правило, при хорошо продуманной декомпозиции модули верхних уровней иерархии имеют функциональную или последовательную связность функций и данных. Для модулей обслуживания данных характерна информационная связность функций. Данные таких модулей могут быть связаны по-разному. Так, модули, содержащие описание классов при объектно-ориентированном подходе, характеризуются информационной связностью методов и функциональной связностью данных. Получение в процессе декомпозиции модулей с другими видами связности, скорее всего, означает недостаточно продуманное проектирование. Исключением являются лишь библиотеки ресурсов.

Библиотеки ресурсов. Различают библиотеки ресурсов двух типов: библиотеки подпрограмм и библиотеки классов.

Библиотеки подпрограмм реализуют функции, близкие по назначению, например, библиотека графического вывода информации. Связность подпрограмм между собой в такой библиотеке - логическая, а связность самих подпрограмм - функциональная, так как каждая из них обычно реализует одну функцию.

Библиотеки классов реализуют близкие по назначению классы. Связность элементов класса - информационная, связность классов между собой может быть функциональной - для родственных или ассоциированных классов и логической - для остальных.

В качестве средства улучшения технологических характеристик библиотек ресурсов в настоящее время широко используют разделение тела модуля на интерфейсную часть и область

реализации (секции Interface и Implementation - в Pascal, h и cpp-файлы в C++ и в Java).

Интерфейсная часть в данном случае содержит совокупность объявлений ресурсов (заголовков подпрограмм, имен переменных, типов, классов и т. п.), которые данная библиотека предоставляет другим модулям. Ресурсы, объявление которых в интерфейсной части отсутствует, извне не доступны. Область *реализации* содержит тела подпрограмм и, возможно, внутренние ресурсы (подпрограммы, переменные, типы), используемые этими подпрограммами. При такой организации любые изменения реализации библиотеки, не затрагивающие ее интерфейс, не требуют пересмотра модулей, связанных с библиотекой, что улучшает технологические характеристики модулей-библиотек. Кроме того, подобные библиотеки, как правило, хорошо отлажены и продуманы, так как часто используются разными программами.

2.3. Нисходящая и восходящая разработка программного обеспечения

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода:

- восходящий;
- нисходящий.

В литературе встречается еще один подход, получивший название «расширение ядра». Он предполагает, что в первую очередь проектируют и разрабатывают некоторую основу-ядро программного обеспечения, например, структуры данных и процедуры, связанные с ними. В дальнейшем ядро наращивают, комбинируя восходящий и нисходящий методы. На практике данный подход в зависимости от уровня ядра практически сводится либо к нисходящему, либо к восходящему подходам.

Восходящий подход. При использовании восходящего подхода сначала проектируют и реализуют компоненты нижнего уровня, затем предыдущего и т. д. По мере завершения тестирования и отладки компонентов осуществляют их сборку, причем компоненты нижнего уровня при таком подходе часто помещают в библиотеки компонентов.

Для тестирования и отладки компонентов проектируют и реализуют специальные тестирующие программы.

Подход имеет следующие недостатки:

- увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;
- наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;
- позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т. д.

Исторически восходящий подход появился раньше, что связано с особенностью мышления программистов, которые в процессе обучения привыкают при написании небольших программ сначала детализировать компоненты нижних уровней (подпрограммы, классы). Это позволяет им лучше осознавать процессы верхних уровней. При промышленном изготовлении программного обеспечения восходящий подход в настоящее время практически не используют.

Нисходящий подход. Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз», т. е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, еще не реализованных уровней заменяют специально разработанными отладочными модулями - «заглушками», что позволяет тестировать и отлаживать уже реализованную часть.

При использовании нисходящего подхода применяют иерархический, операционный и комбинированный методы определения последовательности проектирования и реализации компонентов.

Иерархический метод предполагает выполнение разработки строго по уровням. Исключения

допускаются при наличии зависимости по данным, т. е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля. Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

Операционный метод связывает последовательность выполнения при запуске программы. Применение метода усложняется тем, что порядок выполнения модулей может зависеть от данных. Кроме того, модули вывода результатов, несмотря на то, что они вызываются последними, должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании. С точки зрения распределения человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом *критическом* пути.

Комбинированный метод учитывает следующие факторы, влияющие на последовательность разработки:

- достижимость модуля - наличие всех модулей в цепочке вызова данного модуля;
- зависимость по данным - модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;
- обеспечение возможности выдачи результатов - модули вывода результатов должны создаваться раньше обрабатывающих;
- готовность вспомогательных модулей - вспомогательные модули, например, модули закрытия файлов, завершения программы, должны создаваться раньше обрабатывающих;
- наличие необходимых ресурсов.

Кроме того, при прочих равных условиях сложные модули должны разрабатываться прежде простых, так как при их проектировании могут выявиться неточности в спецификациях, а чем раньше это произойдет, тем лучше.

Нисходящий подход допускает нарушение нисходящей последовательности разработки компонентов в специально оговоренных случаях. Так, если некоторый компонент нижнего уровня используется многими компонентами более высоких уровней, то его рекомендуют проектировать и разрабатывать раньше, чем вызывающие его компоненты. И, наконец, в первую очередь проектируют и реализуют компоненты, обеспечивающие обработку правильных данных, оставляя компоненты обработки неправильных данных напоследок.

Пример определения последовательности реализации модулей будет рассмотрен в § 5.2.

Нисходящий подход обычно используют и при объектно-ориентированном программировании. В соответствии с рекомендациями подхода вначале проектируют и реализуют пользовательский интерфейс программного обеспечения, затем разрабатывают классы некоторых базовых объектов предметной области, а уже потом, используя эти объекты, проектируют и реализуют остальные компоненты.

Нисходящий подход обеспечивает:

- максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;
- раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению;
- возможность нисходящего тестирования и комплексной отладки (см. гл. 9).

2.4. Структурное и «неструктурное» программирование.

Средства описания структурных алгоритмов

Одним из способов обеспечения высокого уровня технологичности разрабатываемого программного обеспечения является *структурное программирование*.

Различают три вида вычислительного процесса, реализуемого программами: линейный,

разветвленный и циклический.

Линейная структура процесса вычислений предполагает, что для получения результата необходимо выполнить некоторые операции в определенной последовательности.

Разветвленная структура процесса вычислений предполагает, что конкретная последовательность операций зависит от значений одной или нескольких переменных.

Циклическая структура процесса вычислений предполагает, что для получения результата некоторые действия необходимо выполнить несколько раз.

Для реализации указанных вычислительных процессов в программах используют соответствующие управляющие операторы. Первые процедурные языки программирования высокого уровня, такие, как FORTRAN, понятием «тип вычислительного процесса» не оперировали. Для изменения линейной последовательности операторов в них, как в языках низкого уровня, использовались команды условной (при выполнении некоторого условия) и безусловной передач управления. Потому и программы, написанные на этих языках, имели запутанную структуру, присущую в настоящее время только низкоуровневым (машинным) языкам.

Именно для изображения схем алгоритмов таких программ в свое время был разработан ГОСТ 19.701-90, согласно которому каждой группе действий ставится в соответствие специальный блок (табл. 2.3). Хотя этот стандарт предусматривает блоки для обозначения циклов, он не запрещает и произвольной передачи управления, т. е. допускает использование команд условной и безусловной передачи управления при реализации алгоритма.

Таблица 2.3

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

В качестве примера, демонстрирующего особенности использования команд передачи управления для организации требуемого типа вычислительного процесса, рассмотрим программу на языке Ассемблера.

Пример 2.1 (вариант 1). Реализовать на языке Ассемблера подпрограмму поиска минимального элемента массива $a(n)$. На рис. 2.2 приведен пример неудачной реализации этой подпрограммы. Стрелками показаны передачи управления. Даже с комментариями и стрелками понять хорошо известный алгоритм достаточно сложно.

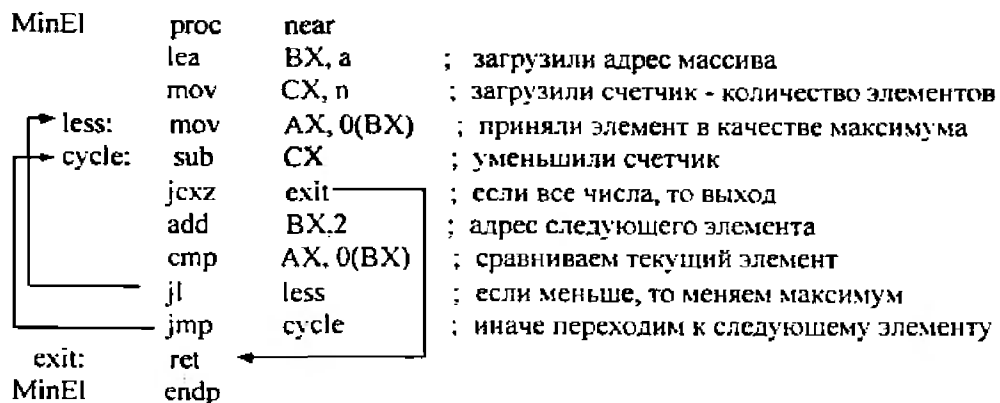


Рис. 2.2. Подпрограмма поиска минимального элемента
(неудачная реализация на Ассемблере)

После того, как в 60-х годах XX в. было доказано, что любой сколь угодно сложный алгоритм можно представить с использованием трех основных управляющих конструкций, в языках программирования высокого уровня появились управляющие операторы для реализации соответствующих конструкций. Эти три конструкции принято считать базовыми. К ним относят конструкции:

- *следование* - обозначает последовательное выполнение действий (рис. 2.3, а);
- *ветвление* - соответствует выбору одного из двух вариантов действий (рис. 2.3, б);

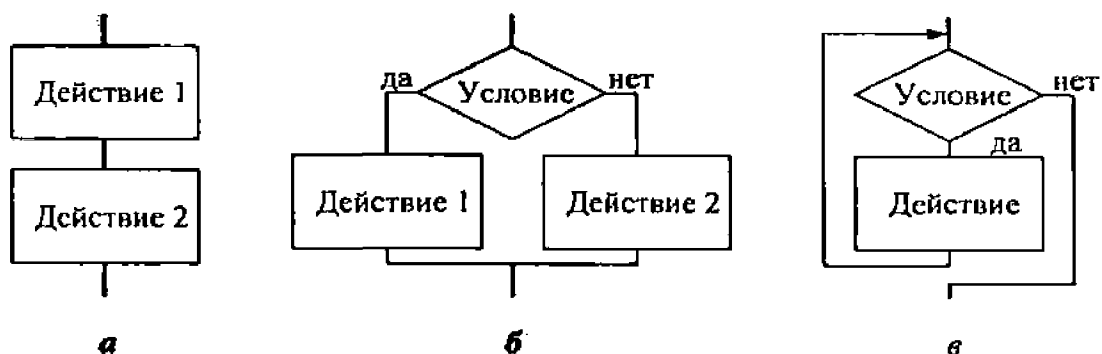


Рис. 2.3. Базовые алгоритмические структуры:

а – следование; б – ветвление; в – цикл-пока

■ *цикл-пока* - определяет повторение действий, пока не будет нарушено некоторое условие, выполнение которого проверяется в начале цикла (рис. 2.3, в).

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. 2.4, а);
- *цикл-do* - обозначает повторение некоторых действий до выполнения заданного условия, проверка которого осуществляется после выполнения действий в цикле (рис. 2.4, б);
- *цикл с заданным числом повторений* (счетный цикл) - обозначает повторение некоторых действий указанное количество раз (рис. 2.4, в).

Любая из дополнительных конструкций легко реализуется через базовые. Перечисленные шесть конструкций были положены в основу структурного программирования.

Перечисленные шесть конструкций были положены в основу структурного программирования.

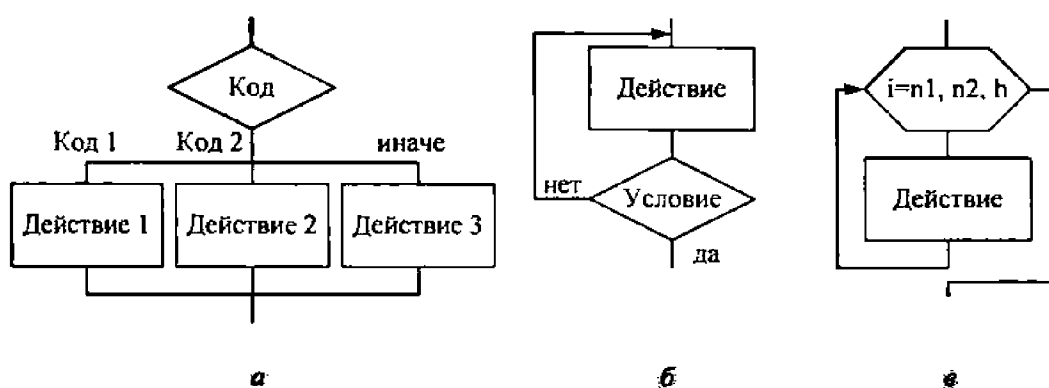


Рис. 2.4. Дополнительные структуры алгоритмов:

а – выбор; б – цикл-до; в – цикл с заданным числом повторений

Примечание. Слово «структурное» в данном названии подчеркиваем тот факт, что при программировании использованы только перечисленные конструкции (структуры). Отсюда и понятие «программирование без go to».

Программы, написанные с использованием только структурных операторов передачи управления, называют *структурными*, чтобы подчеркнуть их отличие от программ, при проектировании или реализации которых использовались низкоуровневые способы передачи управления.

Несмотря на то, что Ассемблер не предусматривает соответствующих конструкций, «структурно» можно программировать и на нем. Вернемся к примеру 2.1.

Пример 2.1 (вариант 2). Поскольку реализуемый цикл по типу «счетный» с количеством повторений $n-1$, используем соответствующую команду Ассемблера. Уберем и усложняющий понимание возврат на метку less, заменив его дубликатом команды сохранения текущего максимального элемента.

Полученный в результате «структурированный» вариант программы поиска максимального элемента массива приведен на рис. 2.5. Единственный возврат реализует циклический процесс, а передача управления на следующие команды - ветвление.

Представление алгоритма программы в виде схемы с точки зрения структурного программирования имеет два недостатка:

- предполагает слишком низкий уровень детализации, что часто скрывает суть сложных алгоритмов;
- позволяет использовать неструктурные способы передачи управления, причем часто на схеме алгоритма они выглядят проще, чем эквивалентные структурные.

```

MinEl      proc      near
            lea       BX, a      ; загрузили адрес массива
            mov       CX, n      ; загрузили счетчик - количество элементов
            mov       AX, 0(BX)  ; приняли элемент в качестве максимума
            sub       CX        ; уменьшили счетчик
            add       BX, 2      ; занесли адрес следующего элемента
cycle:      cmp       AX, 0(BX)   ; сравниваем текущий элемент
            jge       next      ; если не меньше, то пропускаем
            mov       AX, 0(BX)  ; приняли элемент в качестве максимума
next:      add       BX, 2      ; адрес следующего элемента
            loop      cycle      ; если не все, то повторяем
            ret
MinEl      endp

```

Рис. 2.5. Подпрограмма поиска максимального элемента массива («структурный» вариант)

Классическим примером последнего является организация поискового цикла с использованием неструктурной передачи управления (рис. 2.6, а) и эквивалентный структурный вариант (рис. 2.6, б).

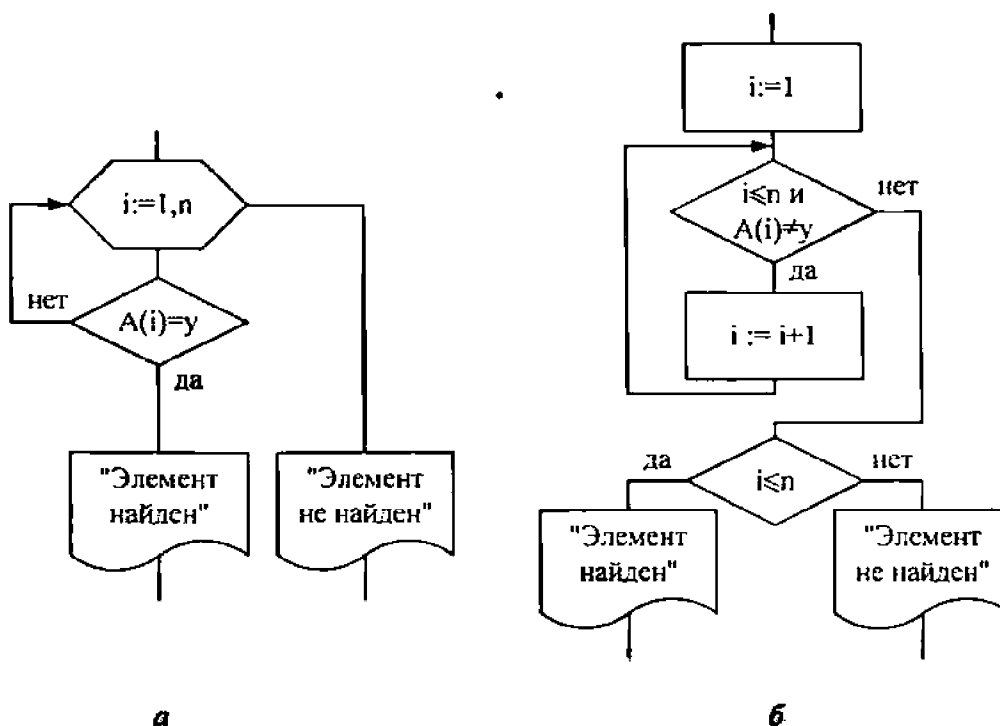


Рис. 2.6. Схема алгоритма реализации поискового цикла:
а – неструктурный вариант; б – структурный вариант

Кроме схем, для описания алгоритмов можно использовать псевдокоды, Flow-формы и диаграммы Насси-Шнейдермана. Все перечисленные нотации с одной стороны базируются на тех же основных структурах, что и структурное программирование, а с другой - допускают разные уровни детализации.

Псевдокоды. *Псевдокод* - формализованное текстовое описание алгоритма (текстовая нотация). В литературе были предложены несколько вариантов псевдокодов. Один из них приведен в табл. 2.4.

Описать с помощью псевдокодов неструктурный алгоритм невозможно. Использование псевдокодов изначально ориентирует проектировщика только на структурные способы передачи управления, а потому требует более тщательного анализа разрабатываемого алгоритма. В отличие от схем алгоритмов, псевдокоды не ограничивают степень детализации проектируемых операций. Они позволяют соизмерять степень детализации действия с уровнем абстракции, на котором это действие рассматривают, и хорошо согласуются с основным методом структурного программирования - методом пошаговой детализации.

Таблица 2.4

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> = <п>, <к>, <н> <действие> Все-цикл
Цикл-пока	Цикл-пока <условие> <действие> Все-цикл	Цикл-до	Выполнять <действие> До <условие>

В качестве примера посмотрим, как будет выглядеть на псевдокоде описание алгоритма поискового цикла, представленного на рис. 2.6:

```

i: =1
Цикл-пока i ≤ n и A[i] ≠ y
i: =i+1
Все-цикл
Если i ≥ n
    то Вывести «Элемент найден»
    иначе Вывести «Элемент не найден»
Все-если

```

Flow-формы. *Flow-формы* представляют собой графическую нотацию описания структурных алгоритмов, которая иллюстрирует вложенность структур. Каждый символ Flow-формы соответствует управляющей структуре и изображается в виде прямоугольника. Для демонстрации вложенности структур символ Flow-формы может быть вписан в соответствующую область прямоугольника любого другого символа. В прямоугольниках символов содержится текст на естественном языке или в математической нотации. Размер прямоугольника определяется длиной вписанного в него текста и размерами вложенных прямоугольников. Символы Flow-форм,

соответствующие основным и дополнительным управляющим конструкциям, приведены на рис. 2.7.

На рис. 2.8 представлено описание рассмотренного ранее поискового цикла с использованием Flow-формы. Хорошо видны вложенность и следование конструкций, изображенных прямоугольниками.

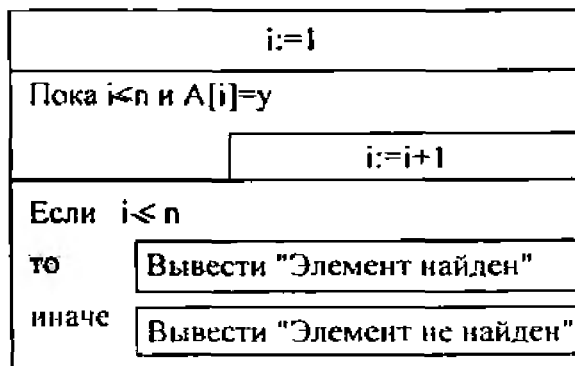


Рис. 2.8. Алгоритм поискового цикла

Диаграммы Насси-Шнейдермана. *Диаграммы Насси-Шнейдермана* являются развитием Flow-форм. Основное их отличие от Flow-форм заключается в том, что область обозначения условий и вариантов ветвления изображают в виде треугольников (рис. 2.9). Такое обозначение обеспечивает большую наглядность представления алгоритма.

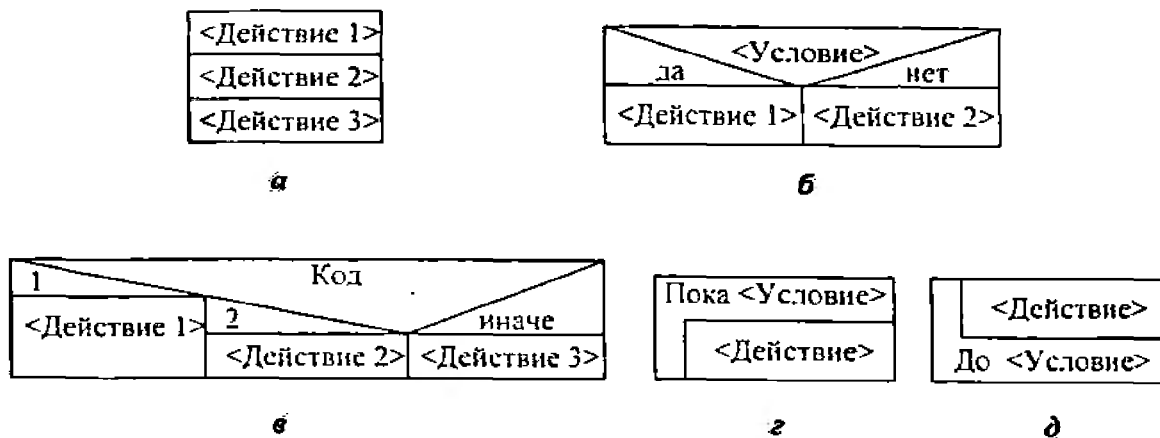


Рис. 2.9. Условные обозначения диаграмм Насси-Шнейдермана для основных конструкций:

а – следование; б – ветвление; в – выбор; г – цикл-пока; д – цикл-до

Также, как при использовании псевдокодов, описать неструктурный алгоритм, применяя Flow-формы или диаграммы Насси-Шнейдермана, невозможно (для неструктурных передач управления в этих нотациях просто отсутствуют условные обозначения). В то же время, являясь графическими, эти нотации лучше отображают вложенность конструкций, чем псевдокоды.

Общим недостатком Flow-форм и диаграмм Насси-Шнейдермана является сложность построения изображений символов, что усложняет практическое применение этих нотаций для описания больших алгоритмов.

2.5. Стиль оформления программы

С точки зрения технологичности хорошим считают стиль оформления программы, облегчающий ее восприятие как самим автором, так и другими программистами, которым, возможно, придется ее проверять или модифицировать. «Помните, программы читаются людьми», призывал Д. Ван Тассел, автор одной из известных монографий, посвященной проблемам программирования [60].

Именно исходя из того, что любую программу неоднократно придется просматривать, следует придерживаться хорошего стиля написания программ.

Стиль оформления программы включает:

- правила именования объектов программы (переменных, функций, типов, данных и т. п.);
- правила оформления модулей;
- стиль оформления текстов модулей.

Правила именования объектов программы. При выборе имен программных объектов следует придерживаться следующих правил:

- имя объекта должно соответствовать его содержанию, например:

MaxItem - максимальный элемент;

NextItem - следующий элемент;

- если позволяет язык программирования, можно использовать символ «_» для визуального разделения имен, состоящих из нескольких слов, например:

Max_Item, Next_Item;

- необходимо избегать близких по написанию имен, например:

Index и InDec.

Правила оформления модулей. Каждый модуль должен предваряться заголовком, который, как минимум, содержит:

- название модуля;
- краткое описание его назначения;
- краткое описание входных и выходных параметров с указанием единиц измерения;
- список используемых (вызываемых) модулей;
- краткое описание алгоритма (метода) и/или ограничений;
- ФИО автора программы;
- идентифицирующую информацию (номер версии и/или дату последней корректировки).

Например:

```
{
*****
*   Функция: Length_Path(n:word; L: array of real):real   *
*   Цель: определение суммарной длины отрезков             *
*   Исходные данные:                                       *
*       n – количество отрезков.                           *
*       L – массив длин отрезков (в метрах)                 *
*   Результат:  длина (в метрах)                           *
*   Вызываемые модули:  нет                                 *
*   Описание алгоритма:                                     *
*       отрезки суммируются методом накопления,  $n \geq 0$    *
*   Дата: 25.12.2001   Версия 1.01.                         *
*   Автор: Иванов И.И.                                       *
*   Исправления: нет                                         *
*****
}
```

Стиль оформления текстов модулей. Стиль оформления текстов модулей определяет использование отступов, пропусков строк и комментариев, облегчающих понимание программы. Как правило, пропуски строк и комментарии используют для визуального разделения частей модуля, например:

```
{проверка количества отрезков и выход, если отрезки не заданы}
ifn<0 then
  begin
    WriteLn (' Количество отрезков отрицательно');
    exit;
  end;
{цикл суммирования длин отрезков}
S:= 0;
for i:= 0 to n-1 do S:= S + Len [i];
```

Для таких языков, как Pascal, C++ и Java, использование отступов позволяет прояснить структуру программы: обычно дополнительный отступ обозначает вложение операторов языка, например:

```
amax:= a[1,1];
for i:= 1 to n do
  for j:= 1 to τ do
    ifa[i,j]>amax then amax:= a [i,j];
```

Несколько сложнее дело обстоит с комментариями. Опыт показывает, что переводить с английского языка каждый оператор программы не нужно: любой программист, знающий язык программирования, на котором написана программа, без труда прочитает тот или иной оператор. Комментировать следует цели выполнения тех или иных действий, а также группы операторов, связанные общим действием, т. е. комментарии должны содержать некоторую дополнительную (неочевидную) информацию, например:

```
{проверка условия и выход, если условие не выполняется}
ifn<0 then
  begin
    WriteLn('Количество отрезков отрицательно');
    exit;
  end;
```

Для языков низкого уровня, например, Ассемблера, стиль, облегчающий понимание, предложить труднее. В этом случае может оказаться целесообразным комментировать и блоки операторов, и каждый оператор, например:

```
; цикл суммирования элементов массива
;      установки цикла
      mov  AX, 0          ; обнуляем сумму
      mov  CX, n          ; загружаем счетчик цикла
      mov  BX, 0          ; смещение первого элемента массива
;      тело цикла
cycle: add  AX, a [BX]     ; добавляем элемент
      add  BX, 2          ; определяем адрес следующего
      loop cycle          ; цикл на n повторений
;      выход из цикла при обнулении счетчика
```

2.6. Эффективность и технологичность

Традиционно эффективными считают 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.

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 вмешиваться 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 удастся, если передавать данные «по ссылке», но как неизменяемые (описанные const). В последнем случае в стеке размещается только адрес данных, например:

```
Type Mas.4iv = array [1.. 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 2.2. 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 10000 раз. Оптимизируем цикл, используя, что $320 = 2^8 + 2^6$:

```
skl:=S [k,l];      {выносим обращение к элементу массива из цикла}
for x: = 0 to 99 do {меняем циклы местами}
  begin
    i:= 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 10000 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 один раз.

Пример 2.3. 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 2.2 понять, что делает программа, стало сложнее, а в примере 2.3 - практически нет. Следовательно, оптимизация, выполненная в первом случае, может ухудшить технологичность программы, а потому не очень желательна.

2.7. Программирование «с защитой от ошибок»

Любая из ошибок программирования, которая не обнаруживается на этапах компиляции и компоновки программы, в конечном счете может проявиться тремя способами: привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.



Рис. 2.10. Способы проявления ошибок

Однако до того, как результат работы программы становится фатальным, ошибки обычно много раз проявляются в виде неверных промежуточных результатов, неверных управляющих переменных, неверных типов данных, индексах структур данных и т. п. (рис. 2.10). А это значит, что часть ошибок можно попытаться обнаружить и нейтрализовать, пока они еще не привели к тяжелым последствиям.

Программирование, при котором применяют специальные приемы раннего обнаружения и нейтрализации ошибок, было названо *защитным* или *программированием с защитой от ошибок*. При его использовании существенно уменьшается вероятность получения неверных результатов.

Детальный анализ ошибок и их возможных ранних проявлений показывает, что целесообразно проверять:

- правильность выполнения операций ввода-вывода;
- допустимость промежуточных результатов (значений управляющих переменных, значений индексов, типов данных, значений числовых аргументов и т. д.).

Проверки правильности выполнения операций ввода-вывода. Причиной неверного определения исходных данных могут являться, как внутренние ошибки-ошибки устройств ввода-вывода или программного обеспечения, так и внешние ошибки - ошибки пользователя. При этом принято различать:

- *ошибки передачи* - аппаратные средства, например, вследствие неисправности, искажают данные;
- *ошибки преобразования* - программа неверно преобразует исходные данные из входного формата во внутренний;
- *ошибки перезаписи* - пользователь ошибается при вводе данных, например, вводит лишний или другой символ;
- *ошибки данных* - пользователь вводит неверные данные. Ошибки передачи обычно контролируются аппаратно.

Для защиты от ошибок преобразования данные после ввода обычно сразу демонстрируют пользователю («эхо»). При этом выполняют сначала преобразование во внутренний формат, а затем обратно. Однако предотвратить все ошибки преобразования на данном этапе обычно крайне сложно, поэтому соответствующие фрагменты программы тщательно тестируют [31], используя

методы эквивалентного разбиения и граничных значений (см. § 9.4).

Обнаружить и устранить ошибки перезаписи можно только, если пользователь вводит избыточные данные, например контрольные суммы. Если ввод избыточных данных по каким-либо причинам нежелателен, то следует по возможности проверять вводимые данные, хотя бы контролировать интервалы возможных значений, которые обычно определены в техническом задании, и выводить введенные данные для проверки пользователю.

Неверные данные обычно может обнаружить только пользователь.

Проверка допустимости промежуточных результатов. Проверки промежуточных результатов позволяют снизить вероятность позднего проявления не только ошибок неверного определения данных, но и некоторых ошибок кодирования и проектирования. Для того чтобы такая проверка была возможной, необходимо, чтобы в программе использовались переменные, для которых существуют ограничения любого происхождения, например, связанные с сущностью моделируемых процессов.

Однако следует также иметь в виду, что любые дополнительные операции в программе требуют использования дополнительных ресурсов (времени, памяти и т. п.) и могут также содержать ошибки. Поэтому имеет смысл проверять не все промежуточные результаты, а только те, проверка которых целесообразна, т. е. возможно позволит обнаружить ошибку, и не сложна. Например:

- если каким-либо образом вычисляется индекс элемента массива, то следует проверить, что этот индекс является допустимым;
- если строится цикл, количество повторений которого определяется значением переменной, то целесообразно убедиться, что значение этой переменной не отрицательно;
- если определяется вероятность какого-либо события, то целесообразно проверить, что полученное значение не более 1. а сумма вероятностей всех возможных независимых событий равна 1 и т. д.

Предотвращение накопления погрешностей. Чтобы снизить погрешности результатов вычислений, необходимо соблюдать следующие рекомендации:

- избегать вычитания близких чисел (машинный ноль);
- избегать деления больших чисел на малые;
- сложение длинной последовательности чисел начинать с меньших по абсолютной величине;
- стремиться по возможности уменьшать количество операций;
- использовать методы с известными оценками погрешностей;
- не использовать условие равенства вещественных чисел;
- вычисления производить с двойной точностью, а результат выдавать с одинарной.

Обработка исключений. Поскольку полный контроль данных на входе и в процессе вычислений, как правило, невозможен, следует предусматривать перехват обработки аварийных ситуаций.

Для перехвата и обработки аппаратно и программно фиксируемых ошибок в некоторых языках программирования, например, Delphi Pascal, C++ Java, предусмотрены средства обработки *исключений*. Использование этих средств позволяет не допустить выдачи пользователю сообщения об аварийном завершении программы, ничего ему не говорящего. Вместо этого программист получает возможность предусмотреть действия, которые позволяют исправить эту ошибку или, если это невозможно, выдать пользователю сообщение с точным описанием ситуации и продолжить работу.

2.8. Сквозной структурный контроль

Сквозной структурный контроль представляет собой совокупность технологических операций контроля, позволяющих обеспечить как можно более раннее обнаружение ошибок в процессе разработки. Термин «сквозной» в названии отражает выполнение контроля на всех этапах разработки. Термин «структурный» означает наличие четких рекомендаций по выполнению контролируемых операций на каждом этапе.

Сквозной структурный контроль должен выполняться на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовать специально приглашенные эксперты. Время между сессиями определяет объем материала, который выносится на сессию: при частых сессиях материалы рассматривают небольшими порциями, при редких - существенными фрагментами. Материалы для очередной сессии должны выдаваться участникам заранее, чтобы они могли их обдумать.

Одна из первых сессий должна быть организована на этапе определения спецификаций. На этой сессии проверяют полноту и точность спецификаций, при этом целесообразно присутствие заказчика или специалиста по предметной области, которые смогут определить, насколько правильно полно определены спецификации программного обеспечения.

На этапе проектирования вручную по частям проверяют алгоритмы разрабатываемого программного обеспечения на конкретных наборах данных и сверяют полученные результаты с соответствующими спецификациями. Основная задача - убедиться в правильности понимания спецификаций и проанализировать достоинства и недостатки концептуальных решений, закладываемых в проект.

На этапе реализации проверяют план (последовательность) реализации модулей, набор тестов, а также тексты отдельных модулей.

Для всех этапов целесообразно иметь списки наиболее часто встречающихся ошибок, которые формируют по литературным источникам и исходя из опыта предыдущих разработок. Такие списки позволяют сконцентрировать усилия на конкретных моментах, а не проверять все подряд. При этом все найденные ошибки фиксируют в специальном документе, но не исправляют их (более подробно см. § 9.2).

Помимо раннего обнаружения ошибок, сквозной структурный контроль обеспечивает своевременную подготовку качественной документации по проекту.

Контрольные вопросы и задания

1. Что понимают под технологичностью программного обеспечения? Почему?
2. Дайте определение модуля. Чем вызвано изменение этого понятия? Как изменились требования к модулям в настоящее время и почему?
3. Что понимают под связностью и сцеплением модулей? Какие типы связности и сцепления считаются допустимыми и почему? В чем особенность библиотек ресурсов?
4. Чем нисходящий подход к разработке отличается от восходящего? Перечислите достоинства и недостатки этих подходов?
5. Что называют структурным программированием и почему? Назовите основные и дополнительные структуры. Объясните, в чем сложность использования схем алгоритмов при проектировании структурных программ? Какие способы описания структурных алгоритмов существуют?
6. Предложите структурный алгоритм перевода чисел в 16-ричную систему счисления. Опишите его с использованием схемы алгоритма, псевдокода, диаграмм Насси-Шнейдермана и flow-форм. В чем, по вашему, основной недостаток двух последних нотаций, который препятствует их широкому применению?
7. Что называют «хорошим стилем» оформления программ и почему? Реализуйте решение предыдущего задания на любом языке программирования. Подумайте, как следует назвать переменные, и какие комментарии необходимы.
8. От каких ошибок защищает «программирование с защитой от ошибок» и почему? Что понимают под термином «исключение»? В каких случаях «исключения» используют?
9. Почему «сквозной структурный контроль» так назван? Что значит «сквозной» контроль? В чем заключается его «структурность»?

3. ОПРЕДЕЛЕНИЕ ТРЕБОВАНИЙ К ПРОГРАММНОМУ ОБЕСПЕЧЕНИЮ И ИСХОДНЫХ ДАННЫХ ДЛЯ ЕГО ПРОЕКТИРОВАНИЯ

Этап постановки задачи - один из наиболее ответственных этапов создания программного продукта. На этом этапе формулируют основные требования к разрабатываемому программному обеспечению. Оттого, насколько полно определены функции и эксплуатационные требования, насколько правильно приняты принципиальные решения, определяющие процесс проектирования, во многом зависит стоимость разработки и ее качество.

3.1. Классификация программных продуктов по функциональному признаку

Каждый программный продукт предназначен для выполнения определенных функций. По назначению все программные продукты можно разделить на три группы: системные, прикладные и гибридные (рис. 3.1).



Рис. 3.1. Классификация программных продуктов по их назначению

К с и с т е м н ы м обычно относят программные продукты, обеспечивающие функционирование вычислительных систем (как отдельных компьютеров, так и сетей). Это - операционные системы, оболочки и другие служебные программы (утилиты).

Операционные системы, как правило, управляют ресурсами (процессором и памятью), процессами (задачами и потоками) и устройствами. Сложность организации операционных систем обуславливается степенью автоматизации и достигаемой эффективностью процессов управления. Так мультипрограммные операционные системы существенно сложнее однопрограммных, что хорошо видно на примере MS DOS и WINDOWS.

Оболочки (например, NORTON COMMANDER) в свое время появились для организации более удобного интерфейса пользователя с файловой системой MS DOS. Современные оболочки,

такие, как FAR, используют для обеспечения пользователю привычной среды при работе с файловой системой.

К *утилитам* принято относить программы и системы, непосредственно не входящие в состав операционной системы, но обеспечивающие выполнение определенных функций, таких как архивация файлов, проверка компьютера на заражение вирусами, осуществление удаленного доступа к информации и др.

П р и к л а д н ы е программы и системы ориентированы на решение конкретных пользовательских задач.

Различают пользователей:

- разработчиков программ;
- непрограммистов, использующих компьютерные системы для достижения своих целей.

Разработчики программ используют специальные инструментальные средства, такие как компиляторы, компоновщики, отладчики, которые последнее время обычно интегрируют в *системы программирования* и *среды разработки*. Современные среды программирования, например, Delphi, Visual C++, реализуют визуальную технологию разработки программных продуктов и предоставляют программистам огромные библиотеки компонентов, которые можно включать в свою разработку. К этой же группе относят *инструментальные комплексы создания баз данных*, такие как Access, FoxPro, Oracle, средства создания интеллектуальных систем, например, экспертных, обучающих, систем контроля знаний и т. д. Последнее достижение в этом направлении - CASE-средства разработки программного обеспечения, такие как ERwin, BPwin, Paradigm Plus, Rational Rose и др.

Пользователи-непрограммисты в соответствии с современными требованиями не должны быть профессионалами в проблемах создания программных продуктов и специфике их взаимодействия с операционной системой. Для них разрабатывают специальные программные продукты, ориентированные на определенную предметную область. Такие продукты условно можно разделить на продукты общего назначения, профессиональные среды или пакеты, обучающие системы, развлекающие программы и т. д.

Продукты *общего назначения* используют разные группы пользователей. К ним можно отнести текстовые редакторы, например, WinWord, электронные таблицы типа Excel, графические редакторы, информационные системы общего назначения, например, карта Москвы, программы-переводчики, и т. п.

Профессиональные продукты предназначены для специалистов в различных областях, например, к ним можно отнести:

- системы автоматизации проектирования, ориентированные на различные технические области;
- системы-тренажеры, например, тренажер для отработки действий пилотов в аварийной ситуации;
- бухгалтерские системы, например, 1С;
- издательские системы, например, PageMaker, QuarkXpress;
- профессиональные графические системы, например, Adobe Illustrator, PhotoShop, CorelDraw и т. п.;
- экспертные системы и т. д.

Системы автоматизации производственных процессов отличаются от профессиональных тем, что они ориентированы на пользователей разных профессий, связанных единым производственным процессом.

Обучающие программы и *системы* в соответствии со своим названием предназначены для обучения, например, иностранному языку, правилам дорожного движения и т. п.

К *развлекающим* относят игровые программы, музыкальные программы, опять же информационные системы, но с тестами развлекательного характера, например гороскопы и т. п.

Г и б р и д н ы е системы сочетают в себе признаки системного и прикладного программного обеспечения. Как правило, это большие, но узкоспециализированные системы, предназначенные для управления технологическими процессами различных типов в режиме реального времени. Для

повышения надежности и снижения времени обработки в такие системы обычно включают программы, обеспечивающие выполнение функций операционных систем.

К каждому из перечисленных выше типов программного обеспечения при разработке, помимо функциональных, обычно предъявляют еще и определенные эксплуатационные требования.

3.2. Основные эксплуатационные требования к программным продуктам

Как уже упоминалось в § 1.4, эксплуатационные требования определяют некоторые характеристики разрабатываемого программного обеспечения, проявляемые в процессе его функционирования. К таким характеристикам относят:

- **правильность** - функционирование в соответствии с техническим заданием;
- **универсальность** - обеспечение правильной работы при любых допустимых данных и защиты от неправильных данных;
- **надежность (помехозащищенность)** - обеспечение полной повторяемости результатов, т. е. обеспечение их правильности при наличии различного рода сбоев;
- **проверяемость** - возможность проверки получаемых результатов;
- **точность результатов** - обеспечение погрешности результатов не выше заданной;
- **защищенность** - обеспечение конфиденциальности информации;
- **программная совместимость** - возможность совместного функционирования с другим программным обеспечением;
- **аппаратная совместимость** - возможность совместного функционирования с некоторым оборудованием;
- **эффективность** - использование минимально возможного количества ресурсов технических средств, например, времени микропроцессора или объема оперативной памяти;
- **адаптируемость** - возможность быстрой модификации с целью приспособления к изменяющимся условиям функционирования;
- **повторная входимость** - возможность повторного выполнения без перезагрузки с диска;
- **реентерабельность** - возможность «параллельного» использования несколькими процессами.

Правильность является обязательным требованием для любого программного обеспечения: все, что указано в техническом задании, непременно должно быть реализовано. Однако следует понимать, что ни тестирование (см. гл. 9), ни верификация не доказывают правильности созданного программного продукта. В этой связи обычно говорят об определенной *вероятности наличия ошибок*. Естественно, чем большая ответственность перекладывается на компьютерную систему, тем меньше должна быть вероятность как программного, так и аппаратного сбоя. Например, очевидно, что вероятность неправильной работы для системы управления атомной электростанцией должна быть близка к нулю.

Требование *универсальности* также обычно входит в группу обязательных. Ничего хорошего нет в том, что разработанная система выдает результат для некорректных данных или аварийно завершает свою работу на некоторых наборах данных. Однако, как уже упоминалось выше, доказать универсальность сравнительно сложной программы, так же, как ее правильность, невозможно, поэтому имеет смысл говорить о степени универсальности программы.

Практически, чем выше требования к правильности и универсальности программного обеспечения, тем выше и требования к его *надежности*. Источниками помех могут являться все участники вычислительного процесса: технические средства, программные средства и люди. Технические средства подвержены сбоям, например, из-за резких скачков напряжения питания или помех при передаче информации по сетям. Программное обеспечение может содержать ошибки. А люди могут ошибаться при вводе исходных данных.

Современные вычислительные устройства уже достаточно надежны. Сбои технических средств, как правило, регистрируются аппаратно, соответственно результаты вычислений в этом случае восстанавливаются. В случае длительных вычислений, как правило, промежуточные результаты сохраняют (прием получил название «создание контрольных точек»), что позволяет

при возникновении сбоя продолжить вычисления с данными, записанными в последней контрольной точке.

Передача информации по сетям также аппаратно контролируется, кроме того, обычно применяется специальное помехозащитное кодирование, которое позволяет находить и исправлять ошибки передачи данных. Однако полностью исключить ошибки технических средств невозможно, поэтому в тех случаях, когда требования к надежности высоки, обычно используют дублирование систем, при котором две системы решают одну и ту же задачу параллельно, периодически сверяя полученные результаты.

Часто самым «ненадежным элементом» современных систем являются люди. Уже хорошо известно, что в условиях монотонной работы за пультом вычислительной установки операторы допускают большое количество ошибок. Известны и средства, позволяющие снизить количество ошибок в конкретных ситуациях. Так, там, где это возможно, используют ввод избыточной информации, позволяющий выполнять проверки правильности вводимых данных (ввод контрольных сумм и т. п., см. § 2.7). Кроме этого, широко используют всякого рода подсказки, когда информацию необходимо не вводить, а выбирать из некоторого списка и т. п.

Повышенные требования к надежности предъявляют при разработке систем управления, функционирующих в режиме реального времени, когда вычисления выполняются параллельно с технологическими процессами. Существенно это требование и для научно-технических систем и баз данных.

Для обеспечения *проверяемости* следует документально фиксировать исходные данные, установленные режимы и прочую информацию, которая влияет на получаемые результаты. Особенно это существенно в случаях, когда данные поступают непосредственно от датчиков. Если такие данные не выводить вместе с результатами, то последние нельзя будет проверить.

Точность или величина погрешности результатов зависит от точности исходных данных, степени адекватности используемой модели, точности выбранного метода и погрешности выполнения операций в компьютере. Требования к точности результатов обычно наиболее жесткие для систем управления технологическими процессами (например, химическими) и систем навигации (например, система управления стыковкой космических аппаратов).

Обеспечение *защищенности* (конфиденциальности) информации, используемой проектируемой системой, отдельная и в условиях наличия сетей достаточно сложная задача. Помимо чисто программных средств защиты, таких как кодирование информации и идентификация пользователя, для обеспечения защищенности используют также специальные организационные приемы. Наиболее жесткие требования предъявляются к системам, в которых хранится информация, связанная с государственной и коммерческой тайной.

Требование *программной совместимости* может варьироваться от возможности совместной установки с указанным программным обеспечением до обеспечения взаимодействия с ним, например обмена данными и т. п. Чаще всего приходится обеспечивать функционирование программного обеспечения под управлением заданной операционной системы. Однако может потребоваться предусмотреть получение данных из какой-то программы или передачу некоторых данных ей. В этом случае необходимо точно оговорить форматы передаваемых данных.

Требование *аппаратной совместимости* в основном формулируют в виде минимально возможной конфигурации оборудования, на котором будет работать программное обеспечение. Если предполагается использование нестандартного оборудования, то для него должны быть указаны интерфейсы или протоколы обмена информацией. При этом для операционных систем класса Windows нестандартными считают устройства, для которых в системе отсутствуют драйверы - программы, обеспечивающие взаимодействие устройства с операционной системой.

Эффективность системы обычно оценивается отдельно по каждому ресурсу вычислительной установки. Часто используют следующие критерии:

- время ответа системы (обычно отнесенное к быстродействию используемого оборудования) - для систем, взаимодействующих с пользователем в интерактивном режиме, и систем реального времени;
- объем оперативной памяти - для продуктов, работающих в системах с ограниченным

объемом оперативной памяти, например MS DOS;

- объем внешней памяти - для продуктов, интенсивно использующих внешнюю память, например баз данных;

- количество обслуживаемых внешних устройств - для продуктов, осуществляющих интенсивное взаимодействие с внешними устройствами, например датчиками.

Требования эффективности могут противоречить друг другу. Например, чтобы уменьшить время выполнения некоторого фрагмента программы, может потребоваться дополнительный объем оперативной памяти.

Адаптируемость, по сути дела, оценивает технологическое качество программного обеспечения, поэтому оценить эту характеристику количественно практически невозможно. Можно только констатировать, что при создании продукта использованы технологии и специальные приемы, облегчающие его модернизацию.

Требование *повторной входимости* обычно предъявляется к программному обеспечению, резидентно загруженному в оперативную память, например драйверам. Для обеспечения данного требования необходимо так организовать программу, чтобы никакие ее исходные данные не затирались в процессе выполнения или восстанавливались в начале или при завершении каждого вызова.

Требование *реентерабельности* является более жестким, чем повторная входимость, так как в этом случае все данные, изменяемые программой в процессе выполнения, должны быть выделены в специальный блок, копия которого создается для каждого процесса при вызове программы.

Сложность многих программных систем не позволяет сразу сформулировать четкие требования к ним. Обычно для перехода от идеи создания некоторого программного обеспечения к четкой формулировке требований, которые могут быть занесены в техническое задание, необходимо выполнить предпроектные исследования в области разработки.

3.3. Предпроектные исследования предметной области

Целью предпроектных исследований является преобразование общих нечетких знаний о предназначении будущего программного обеспечения в сравнительно точные требования к нему. Существуют два варианта неопределенности:

- неизвестны методы решения формулируемой задачи - такого типа неопределенности обычно возникают при решении научно-технических задач;

- неизвестна структура автоматизируемых информационных процессов - обычно встречается при построении автоматизированных систем управления предприятиями.

В первом случае во время предпроектных исследований определяют возможность решения поставленной задачи и методы, позволяющие получить требуемый результат, что может потребовать соответствующих *научных исследований* как фундаментального, так и прикладного характера, разработки и исследования новых моделей объектов реального мира.

Во втором случае определяют:

- структуру и взаимосвязи автоматизируемых информационных процессов;
- распределение функций между человеком и системой, а также между аппаратурой и программным обеспечением;

- функции программного обеспечения; внешние условия его функционирования и особенности его интерфейсов, как с пользователями, так и при необходимости - с аппаратной частью;

- требования к программным и информационным компонентам, необходимые аппаратные ресурсы, требования к базам данных и физические характеристики программных компонент.

Результаты предпроектных исследований предметной области используют в процессе разработки технического задания.

3.4. 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 приемно-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.

На рис. 3.2 схематически показаны основные факторы, определяющие характеристики разрабатываемого программного обеспечения. Такими факторами являются:

- исходные данные и требуемые результаты, которые определяют функции программы или системы;
- среда функционирования (программная и аппаратная) - может быть задана, а может выбираться для обеспечения параметров, указанных в техническом задании;
- возможное взаимодействие с другим программным обеспечением/или специальными техническими средствами - также может быть определено, а может выбираться исходя из набора выполняемых функций.

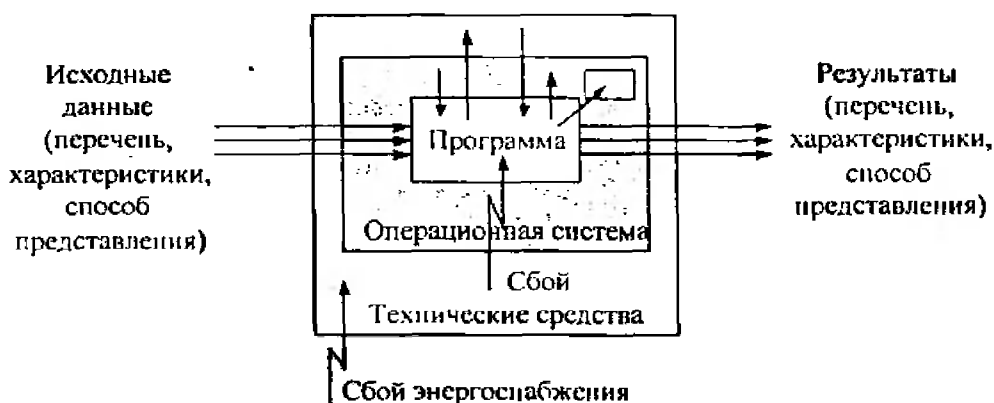


Рис. 3.2. Факторы, определяющие параметры разрабатываемого программного обеспечения

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.

На техническое задание существует стандарт ГОСТ 19.201-78 «Техническое задание. Требования к содержанию и оформлению». В соответствии с этим стандартом техническое задание должно содержать следующие разделы:

- введение;
- основания для разработки;
- назначение разработки;
- требования к программе или программному изделию;
- требования к программной документации;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки.

При необходимости допускается в техническое задание включать приложения (см. правила оформления программной документации в § 11.5).

Рассмотрим более подробно содержание каждого раздела.

Введение должно включать наименование и 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.

Раздел *Основания для разработки* должен содержать наименование документа, на основании которого ведется разработка, организации, утвердившей данный документ, и наименование или условное обозначение темы разработки. Таким документом может служить план, приказ, договор и т. п.

Раздел *Назначение разработки* должен содержать описание функционального и эксплуатационного назначения программного продукта с указанием категорий пользователей.

Раздел *Требования к программе* или программному изделию должен включать 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.

Наиболее важным из перечисленных выше является подраздел *Требования к функциональным характеристикам*. В этом разделе должны быть перечислены выполняемые функции и описаны состав, характеристики и формы представления исходных данных и результатов. В этом же разделе при необходимости указывают критерии эффективности: максимально допустимое время ответа системы, максимальный объем используемой оперативной и/или внешней памяти и др.

Примечание. Если разработанное программное обеспечение не будет выполнять указанных в техническом задании функций, то оно считается не соответствующим техническому заданию, т. е. неправильным с точки зрения критериев качества (см. § 2.2). Универсальность будущего продукта также обычно специально не оговаривается, но подразумевается.

В подразделе *Требования к надежности* указывают уровень надежности, который должен быть обеспечен разрабатываемой системой (см. § 3.2) и время восстановления системы после сбоя. Для систем с обычными требованиями к надежности в этом разделе иногда регламентируют действия разрабатываемого продукта по увеличению надежности результатов (контроль входной и выходной информации, создание резервных копий промежуточных результатов и т. п.).

В подразделе *Условия эксплуатации*, указывают особые требования к условиям эксплуатации: температуре окружающей среды, относительной влажности воздуха и т. п. Как правило, подобные требования формулируют, если разрабатываемая система будет эксплуатироваться в нестандартных условиях или использует специальные внешние устройства, например для хранения информации. Здесь же указывают вид обслуживания, необходимое количество и квалификация персонала. В противном случае допускается указывать, что требования не предъявляются.

В подразделе *Требования к составу и параметрам технических средств* указывают необходимый состав технических средств с указанием их основных технических характеристик: тип микропроцессора, объем памяти, наличие внешних устройств и т. п. При этом часто указывают два варианта конфигурации: минимальный и рекомендуемый.

В подразделе *Требования к информационной и программной совместимости* при необходимости можно задать методы решения, определить язык или среду программирования для

разработки, а также используемую операционную систему и другие системные и пользовательские программные средства, с которым должно взаимодействовать разрабатываемое программное обеспечение. В этом же разделе при необходимости указывают, какую степень защиты информации необходимо предусмотреть.

В разделе *Требования к программной документации* указывают необходимость наличия руководства программиста, руководства пользователя, руководства системного программиста, пояснительной записки и т. п. На все эти типы документов также существуют ГОСТы. Правила их составления рассмотрены в гл. 11.

В разделе *Технико-экономические показатели* рекомендуется указывать ориентировочную экономическую эффективность, предполагаемую годовую потребность и экономические преимущества по сравнению с существующими аналогами.

В разделе *Стадии и этапы разработки* указывают стадии разработки, этапы и содержание работ с указанием сроков разработки и исполнителей.

В разделе *Порядок контроля и приемки* указывают виды испытаний и общие требования к приемке работы.

В приложениях при необходимости приводят: перечень научно-исследовательских работ, обосновывающих разработку; схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые следует использовать при разработке.

В зависимости от особенностей разрабатываемого продукта разрешается уточнять содержание разделов, т. е. использовать подразделы, вводить новые разделы или объединять их.

В случаях, если какие-либо требования, предусмотренные техническим заданием, заказчик не предъявляет, следует в соответствующем месте указать «Требования не предъявляются».

Разработка технического задания - процесс трудоемкий, требующий определенных навыков. Наиболее сложным, как правило, является четкое формулирование основных разделов: введения, назначения и требований к программному продукту. В качестве примеров рассмотрим два технических задания на выполнение курсового проектирования, составленных по сокращенной схеме, и сравнительно полное техническое задание на выполнение госбюджетной научно-исследовательской работы.

Пример 3.1. 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.

На рис. 3.3 представлен пример титульного листа технического задания на учебный программный продукт. Ниже приведен его текст.

1. ВВЕДЕНИЕ

Настоящее техническое задание распространяется на разработку программы построения графиков и таблиц значений функций одной переменной, предназначенной для использования школьниками старших классов.

В школьном курсе элементарной алгебры тема анализа функций является одной из самых сложных. При изучении данной темы школьники должны научиться исследовать и строить графики функций одной переменной, используя все известные характеристические точки функции, включая корни, точки разрыва первого и второго рода и т. д.

Существующее программное обеспечение, которое может решать подобные задачи, является универсальным, например Eurica или MathCad. Оно имеет сравнительно сложный пользовательский интерфейс, ориентированный на пользователя, прослушавшего, как минимум, институтский курс высшей математики, что делает использование подобных средств школьниками невозможным.

Разрабатываемая программа позволит школьникам проверить свои знания при изучении указанной темы.

Министерство образования Российской Федерации
Московский государственный технический университет имени Н.Э. Баумана

Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети

УТВЕРЖДАЮ

Зав. кафедрой ИУ6,

д-р техн. наук, проф. _____ Сюзев В.В.

« __ » _____ 2002 г.

ПРОГРАММА ПОСТРОЕНИЯ ТАБЛИЦ И ГРАФИКОВ ФУНКЦИЙ

Техническое задание на курсовую работу

Листов 5

Руководитель,

канд. техн. наук, доцент _____ Петров П.П.

Исполнитель,

студент гр. ИУ6-31 _____ Иванов И.И.

2002

Рис. 3.3. Пример оформления титульного листа технического задания
на учебный программный продукт

2. ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ

Программа разрабатывается на основе учебного плана кафедры «Компьютерные системы и сети» и в соответствии с договором кафедры со школой № ... от 5.09.2001.

3. НАЗНАЧЕНИЕ

Основным назначением программы является помощь школьникам при изучении раздела «Исследование функций одного аргумента» школьного курса элементарной алгебры.

4.ТРЕБОВАНИЯ К ПРОГРАММЕ ИЛИ ПРОГРАММНОМУ ИЗДЕЛИЮ

4.1.Требования к функциональным характеристикам

4.1.1. Программа должна обеспечивать возможность выполнения следующих функций:

- ввод аналитического представления функции одной переменной и длительное хранение его в системе;
- ввод и изменение интервала определения функции;
- ввод и корректировку шага аргумента;
- построение таблицы значений функции на заданном интервале или изображение графика функции на заданном интервале при условии, что на указанном интервале она не имеет точек разрыва.

4.1.2. Исходные данные:

- аналитическое задание функции;
- интервал определения функции;
- шаг изменения аргумента, определяющий количество точек на интервале.

4.2. Требования к надежности

4.2.1.Предусмотреть контроль вводимой информации.

4.2.2.Предусмотреть блокировку некорректных действий пользователя при работе с системой.

4.3. Требования к составу и параметрам технических средств

4.3.1.Система должна работать на IBM совместимых персональных компьютерах.

4.3.2.Минимальная конфигурация:

- тип процессораPentium и выше;
- объем оперативного запоминающего устройств32 Мб и более.

4.4. Требования к информационной и программной совместимости

Система должна работать под управлением семейства операционных систем Win 32 (Windows 95, Windows 98, Windows 2000, Windows NT 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 об основных терминах соответствующего раздела математики и подсказки учащимся.

5.3.В состав сопровождающей документации должны входить:

5.3.1. Пояснительная записка на 25-30 листах, содержащая описание разработки.

5.3.2. Руководство пользователя.

Пример 3.2. Разработать техническое задание на создание системы «Учет успеваемости студентов». Система предназначена для оперативного учета успеваемости студентов в сессию деканом, заместителями декана по курсам 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. Текст технического задания приведен ниже.

1. ВВЕДЕНИЕ

Настоящее техническое задание распространяется на разработку системы учета успеваемости студентов, предназначенной для сбора и хранения информации о ходе сдачи экзаменационной сессии. Предполагается, что использовать данную систему будут сотрудники деканата, декан и его заместители.

Во время сессии необходимо получение оперативной информации о ходе ее сдачи студентами, однако выполнение такого контроля вручную требует значительного времени.

Автоматизированная система учета успеваемости позволит улучшить качество контроля сдачи сессии со стороны куратора и деканата и обеспечит получение сведений о динамике работы каждого студента, группы и курса в целом.

Кроме того, хранение информации о сдаче сессий в течение всего времени обучения позволит осуществлять автоматическую генерацию справок о прослушанных курсах и приложений к диплому выпускника.

2. ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ

Система разрабатывается на основании приказа декана факультета Js'a ... от ... и в соответствии с планом мероприятий по совершенствованию учебного процесса на 2001-2002 учебный год.

3. НАЗНАЧЕНИЕ

Система предназначена для хранения и обработки сведений об успеваемости студентов учебных групп факультета в течение всего срока обучения. Обработанные сведения об успеваемости студентов могут быть использованы для оценки успеваемости каждого студента, группы, курса и факультета в целом.

4. ТРЕБОВАНИЯ К ПРОГРАММЕ ИЛИ ПРОГРАММНОМУ ИЗДЕЛИЮ

4.1. Требования к функциональным характеристикам

4.1.1. Система должна обеспечивать возможность выполнения следующих функций:

- инициализацию системы (ввод списков групп, перечней изучаемых дисциплин в соответствии с учебными планами и т. п.);
- ввод и коррекцию текущей информации о ходе сдачи сессии конкретными студентами;
- хранение информации об успеваемости в течение времени обучения студента;
- получение сведений о текущем состоянии сдачи сессии студентами.

4.1.2. Исходные данные:

- списки студентов учебных групп;
- учебные планы кафедр - перечень предметов и контрольных мероприятий по каждому предмету;
- расписания сессий;
- текущие сведения о сдаче сессии каждым студентом.

4.1.3. Результаты:

- итоги сдачи сессии конкретным студентом;
- итоги сдачи сессии студентами конкретной группы;
- процент успеваемости по всем студентам группы при сдаче конкретного предмета в целом на текущий момент;
- проценты успеваемости по всем группам специальности на текущий момент;
- проценты успеваемости по всем группам курса на текущий момент;
- проценты успеваемости по всем курсам и в целом по факультету на текущий момент;

- список задолжников группы на текущий момент;
- список задолжников курса на текущий момент.

4.2. Требования к надежности

- 4.2.1.Предусмотреть контроль вводимой информации.
- 4.2.2.Предусмотреть блокировку некорректных действий пользователя при работе с системой.
- 4.2.3.Обеспечить целостность хранимой информации.

4.3. Требования к составу и параметрам технических средств

- 4.3.1.Система должна работать на IBM совместимых персональных компьютерах.
- 4.3.2.Минимальная конфигурация:
 - тип процессора Pentium и выше;
 - объем оперативного запоминающего устройства32 Мб и более.

4.4. Требования к информационной и программной совместимости

Система должна работать под управлением семейства операционных систем Win 32 (Windows 95, Windows 98, Windows 2000, Windows NT 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.Программная система должна включать справочную информацию о работе и подсказки пользователю.

5.3.В состав сопровождающей документации должны входить:

- 5.3.1.Пояснительная записка на 25-30 листах, содержащая описание разработки.
- 5.3.2.Руководство системного программиста.
- 5.3.3.Руководство пользователя.
- 5.3.4.Графическая часть на трех листах формата А1:
 - 5.3.4.1.Схема структурная программной системы.
 - 5.3.4.2.Диаграмма компонентов данных.
 - 5.3.4.3.Формы интерфейса пользователя.

Пример 3.3. Разработать техническое задание на создание системы решения комбинаторно-оптимизационных задач. Первая версия системы должна включать алгоритмы решения задач: поиска цикла минимальной длины (задача коммивояжера), поиска кратчайшего пути и поиска минимального связывающего дерева.

Комбинаторными называют задачи, решение которых сводится к выбору варианта из конечного множества решений. В комбинаторно-оптимизационных задачах в конечном множестве допустимых решений отыскивается такое, для которого целевая функция достигает оптимального (минимального или максимального) значения.

Задача коммивояжера или поиска цикла минимальной длины в простейшем варианте формулируется следующим образом. Задан список городов и дорог, соединяющих данные города. Известны расстояния между городами. Необходимо объехать все города, не заезжая ни в какой город дважды, и вернуться в исходный город так, чтобы суммарная длина пути была минимальной.

Задача поиска кратчайшего пути при тех же исходных данных предполагает другую цель: необходимо проехать из одного города в другой так, чтобы суммарная длина пути была

минимальной.

Задача поиска минимального связывающего дерева ставится на тех же исходных данных, но теперь мы прокладываем телефонные линии вдоль дороги и хотим, чтобы длина кабеля была минимальной.

Текст технического задания приведен ниже.

1. ВВЕДЕНИЕ

Настоящее техническое задание распространяется на разработку системы решения комбинаторно-оптимизационных задач, предназначенной для ввода и хранения данных указанных задач, а также для их решения и хранения полученных результатов, и использования разработчиками программных и аппаратных средств вычислительной техники.

Широкий круг задач проектирования различного рода технических объектов, в том числе и компьютеров, относится к классу комбинаторно-оптимизационных задач, точные методы решения которых, как правило, имеют экспоненциальную вычислительную сложность и нереализуемы на современных компьютерах. В настоящее время для решения таких задач широко используются приближенные методы и алгоритмы, которые требуют различных вычислительных ресурсов и обеспечивают разную точность решения.

В то же время эти методы и алгоритмы не систематизированы, оценки их вычислительной и емкостной сложности и сведения о возможной точности получаемых решений неполны и разбросаны по многим источникам. В рамках единой системы не существует программной реализации даже для ограниченного круга алгоритмов решения основных комбинаторно-оптимизационных задач проектирования.

Создание системы, в рамках которой были бы реализованы наиболее часто упоминаемые методы и алгоритмы решения комбинаторно-оптимизационных задач, позволит как оценивать и исследовать отдельные методы и алгоритмы, так и сравнивать их с точки зрения затрат вычислительных ресурсов и точности получаемых решений.

2. ОСНОВАНИЕ ДЛЯ РАЗРАБОТКИ

Система разрабатывается на основании приказа проректора по научной работе МГТУ им. Баумана № ... от и в соответствии с планом госбюджетных научно-исследовательских работ факультета ... на 2001—2002 гг.

3. НАЗНАЧЕНИЕ

Первая версия системы предназначена для решения небольшого круга комбинаторно-оптимизационных задач на графах (поиск кратчайшего пути, минимального покрывающего дерева и покрывающего цикла минимальной длины). В следующих версиях предполагается увеличение количества решаемых задач.

Пользователями могут выступать научные работники и инженеры, занимающиеся проектированием компьютеров, и студенты соответствующих специальностей. Пользователями могут также быть и специалисты других предметных областей, которым приходится решать подобные задачи.

4. ТРЕБОВАНИЯ К ПРОГРАММЕ ИЛИ ПРОГРАММНОМУ ИЗДЕЛИЮ

4.1. Требования к функциональным характеристикам

4.1.1. Система должна представлять совокупность методических и программных средств решения следующих задач:

- задачи построения минимального покрывающего дерева;

- поиск покрывающего цикла минимальной длины (задача коммивояжера);
- задачи поиска кратчайшего пути.

4.1.2. Для этих задач должны быть реализованы:

- алгоритм, обеспечивающий получение точного решения;
- в том случае, если точное решение получается алгоритмом, имеющим неполиномиальную вычислительную сложность, то необходимо дополнительно разработать алгоритм, обеспечивающий получение приближенных решений с полиномиальной вычислительной сложностью,

4.1.3. Методическое обеспечение должно быть реализовано в пользовательском интерфейсе системы, который должен предполагать выбор задачи, метода и алгоритма ее решения; ввод данных; решение проектной задачи и сохранение исходных данных, промежуточных и окончательных результатов во встроенной базе данных для последующего анализа.

4.2. Требования к надежности

4.2.1. Предусмотреть контроль вводимой информации и блокировку некорректных действий пользователя при работе с системой.

4.2.2. Обеспечить корректное завершение вычислений с соответствующей диагностикой при превышении имеющихся вычислительных ресурсов.

4.2.3. Обеспечить целостность информации, хранящейся в базе данных.

4.3. Требования к составу и параметрам технических средств

4.3.1. Система должна работать на IBM совместимых персональных компьютерах.

4.3.2. Минимальная конфигурация:

- тип процессора.....Pentium-100;
- объем оперативного запоминающего устройства 16 Мб;
- тип монитора.....SVGA (15").

4.4. Требования к информационной и программной совместимости

Система должна работать под управлением операционной системы Windows'95 и выше.

5. ТРЕБОВАНИЯ К ПРОГРАММНОЙ ДОКУМЕНТАЦИИ

5.1. Разрабатываемая система должна включать справочную информацию о работе системы и подсказки пользователю.

5.2. В состав сопровождающей документации должны входить:

- пояснительная записка;
- руководство пользователя.

6. ЭТАПЫ РАЗРАБОТКИ

После утверждения технического задания организация-разработчик непосредственно приступает к созданию программного обеспечения. Однако переход к следующему этапу разработки - этапу уточнения спецификаций требует принятия еще некоторых принципиальных решений, от которых во многом зависят как характеристики и возможности разрабатываемого программного обеспечения, так и особенности его разработки, начиная с выбора моделей этапа уточнения спецификаций.

№	Название этапа	Срок	Отчетность
1	Разработка ядра системы	1.1.2000 – 31.3.2000	Описание внутренних форматов, интерфейса и форматов данных базы. Реализация системы на уровне интерфейса
2	Разработка методов и алгоритмов и их реализация для задачи коммивояжера	1.4.2000 – 30.6.2000	Описание методов и алгоритмов. Программные модули, реализующие методы
3	Разработка методов и алгоритмов и их реализация для задачи построения минимального связывающего дерева и задачи поиска кратчайшего пути в графе	1.7.2000 – 30.9.2000	Описание методов и алгоритмов. Программные модули, реализующие методы
4	Тестирование программного продукта и составление программной документации	1.10.2000 – 31.12.2000	Тесты. Документация. Программный продукт

3.5. Принципиальные решения начальных этапов проектирования

На начальных этапах процесса проектирования должны быть приняты принципиальные решения, во многом определяющие этот процесс, а также качество и трудоемкость разработки. К таким решениям относят:

- выбор архитектуры программного обеспечения;
- выбор типа пользовательского интерфейса и технологии работы с документами;
- выбор подхода к разработке (структурного или объектного);
- выбор языка и среды программирования.

Другими словами, эти решения определяют, что проектируется, с какими потребительскими характеристиками, как и какими средствами.

Часть решений может быть определена в техническом задании, образовав группу *технологических требований* остальные должны быть приняты как можно раньше, так как представляют собой исходные данные для процесса проектирования.

Выбор архитектуры программного обеспечения. *Архитектурой программного обеспечения* называют совокупность базовых концепций (принципов) его построения. Архитектура программного обеспечения определяется сложностью решаемых задач, степенью универсальности разрабатываемого программного обеспечения и числом пользователей, одновременно работающих с одной его копией. Различают:

- однопользовательскую архитектуру, при которой программное обеспечение рассчитано на одного пользователя, работающего за персональным компьютером;
- многопользовательскую архитектуру, которая рассчитана на работу в локальной или глобальной сети.

Кроме того, в рамках однопользовательской архитектуры различают:

- программы;
- пакеты программ;
- программные комплексы;
- программные системы.

Многопользовательскую архитектуру реализуют системы, построенные по принципу «клиент-сервер» (см. § 1.1).

Программой называют адресованный компьютеру набор инструкций, точно описывающий последовательность действий, которые необходимо выполнить *для решения конкретной задачи*. При структурном подходе программы представляют собой иерархию подпрограмм, вызывающих друг друга в процессе решения поставленной задачи, при объектном подходе - совокупность обменивающихся сообщениями объектов, для реализации которых разработаны специальные классы. Программа в этом случае представляет собой отдельно компилируемую программную единицу, которая может использовать стандартные библиотеки подпрограмм, но, как правило, не организует свои. Это самый простой вид архитектуры, который обычно используется при решении небольших задач.

Пакеты программ представляют собой *совокупность программ, решающих задачи некоторой прикладной области*. Например, пакет графических программ, пакет математических программ. Программы такого пакета связаны между собой только принадлежностью к определенной прикладной области. Пакет программ реализуют как набор отдельных программ, каждая из которых сама вводит необходимые данные и выводит результаты. По сути дела пакет программ - это некоторая *библиотека программ*.

Программные комплексы представляют собой совокупность программ, *совместно обеспечивающих решение небольшого класса сложных задач одной прикладной области*. Для решения такой задачи может потребоваться решить несколько подзадач, последовательно вызывая программы комплекса. Вызов программ в программном комплексе осуществляется специальной программой - *диспетчером*, который обеспечивает несложный интерфейс с пользователем и, возможно, выдачу некоторой справочной информации. От пакета программ программный комплекс отличается еще и тем, что несколько программ могут последовательно или циклически вызываться для решения одной задачи, и, следовательно, желательно хранить исходные данные и результаты вызовов в пределах одного пользовательского проекта. Программы в этом случае могут реализовываться как отдельно, так и как совместно компилируемые программные единицы, а исходные данные храниться в оперативной памяти или в файлах.

Программные системы представляют собой *организованную совокупность программ (подсистем)*, позволяющую решать *широкий класс задач из некоторой прикладной области*. В отличие от программных комплексов программы, входящие в программную систему, *взаимодействуют через общие данные*. Программные системы обычно имеют развитые пользовательский и внутренние интерфейсы, что требует их тщательного проектирования.

Многопользовательские программные системы в отличие от обычных программных систем должны организовывать *сетевое взаимодействие* отдельных компонентов программного обеспечения, что еще усложняет процесс его разработки. Для разработки подобного программного обеспечения используют специальные технологии или платформы, например, технологии CORBA, COM, Java и т. п.

Выбор типа пользовательского интерфейса. Различают четыре типа пользовательских интерфейсов:

- *примитивные* - реализуют единственный сценарий работы, например, ввод данных - обработка - вывод результатов;

- *меню* - реализуют множество сценариев работы, операции которых организованы в иерархические структуры, например, «вставка»: «вставка файла», «вставка символа» и т. д.;

- *со свободной навигацией* - реализуют множество сценариев, операции которых не привязаны к уровням иерархии, и предполагают определение множества возможных операций на конкретном шаге работы; интерфейсы данной формы в основном используют Windows-приложения;

- *прямого манипулирования* - реализуют множество сценариев, представленных в операциях над объектами, основные операции инициируются перемещением пиктограмм объектов мышью, данная форма реализована в интерфейсе самой операционной системы Windows альтернативно интерфейсу со свободной навигацией.

Тип пользовательского интерфейса во многом определяет сложность и трудоемкость

разработки, которые существенно возрастают в порядке перечисления типов. По последним данным до 80 % программного кода может реализовывать именно пользовательский интерфейс [49]. Поэтому понятно, что на ранних стадиях обучения программированию реализуют в основном примитивные интерфейсы и меню, хотя они и не удобны для пользователей.

Появление объектно-ориентированных визуальных сред разработки программного обеспечения, использующих событийный подход к программированию и в основном рассчитанных на создание интерфейсов со свободной навигацией, существенно снизило трудоемкость разработки подобных интерфейсов и упростило реализацию интерфейсов прямого манипулирования. Таким образом, выбор двух последних типов интерфейсов предполагает использование одной из визуальных сред разработки программного обеспечения. Если соответствующие среды разработчику не доступны, то следует учитывать большую трудоемкость создания подобных интерфейсов.

Кроме того, выбор типа интерфейса включает выбор *технологии работы с документами*. Различают две технологии:

- однок документная, которая предполагает однок документный интерфейс (SDI - Single Document Interface);
- много документная, которая предполагает много документный интерфейс (MDI - Multiple Document Interface).

Много документную технологию используют, если программное обеспечение должно работать с несколькими документами одновременно, например, с несколькими текстами или несколькими изображениями. Однок документную - если одновременная работа с несколькими документами не обязательна.

Трудоемкость реализации много документных интерфейсов с использованием современных библиотек примерно на 3...5 % выше, чем первого. Выбор типа влияет на трудоемкость более существенно (более подробно типы интерфейсов будут рассмотрены в § 8.1).

Выбор подхода к разработке. Если выбран интерфейс со свободной навигацией или прямого манипулирования, то, как указывалось выше, это практически однозначно предполагает использование событийного программирования и объектного подхода, так как современные среды визуального программирования, такие как Visual C++, Delphi, Builder C++ и им подобные, предоставляют интерфейсные компоненты именно в виде объектов библиотечных классов. При этом в зависимости от сложности предметной области программное обеспечение может реализовываться как с использованием объектов и, соответственно, классов, так и чисто процедурно. Исключение составляют случаи использования специализированных языков разработки Интернет-приложений, таких как Perl, построенных по совершенно другому принципу.

Примитивный интерфейс и интерфейс типа меню совместимы как со структурным, так и с объектным подходами к разработке. Поэтому выбор подхода осуществляют с использованием дополнительной информации.

Практика показывает, что объектный подход эффективен для разработки очень больших программных систем (более 100000 операторов универсального языка программирования) и в тех случаях, когда объектная структура предметной области ярко выражена.

Следует также учитывать, что необходимо осторожно использовать объектный подход при жестких ограничениях на эффективность разрабатываемого программного обеспечения, например, при разработке систем реального времени.

Во всех прочих случаях выбор подхода остается за разработчиком.

Выбор языка программирования. В большинстве случаев никакой проблемы выбора языка программирования реально не существует. Язык может быть определен:

- организацией, ведущей разработку; например, если фирма владеет лицензионным вариантом C++ Builder, то она будет вести разработки преимущественно в данной среде;
- программистом, который по возможности всегда будет использовать хорошо знакомый язык;
- устоявшимся мнением («все разработки подобного рода должны выполняться на C++ или на Java или на ...») и т. п.

Если же все-таки выбор языка реально возможен, то нужно иметь в виду, что все

существующие языки программирования можно разделить на следующие группы:

- 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 (вместе с C++). Действительно различные версии C и C++ имеют целый ряд очень существенных достоинств:

- многоплатформенность - для всех используемых в настоящее время платформ существуют компиляторы с языка C и C++;
- наличие операторов, реализующих основные структурные алгоритмические конструкции (условную обработку, все виды циклов);
- возможность программирования на низком (системном) уровне с использованием адресов оперативной памяти;
- огромные библиотеки подпрограмм и классов.

Все это сделало C и C++ основными языками, используемыми для создания операционных систем, и, в свою очередь, служит для них дополнительной рекламой. Однако C и C++ имеют и серьезные недостатки:

- отсутствие полноценных встроенных структурных типов данных (имеющиеся псевдоструктурные типы, использующие адресную арифметику, недостаточно жестко определены, чтобы контролировать многие операции над этими данными, что приводит к большому количеству ошибок, выявляемых только в процессе отладки программы);
- наличие синтаксических неоднозначностей, которые также не позволяют компилятору контролировать правильность программы;
- ограниченный контроль параметров, передаваемых в подпрограмму, что также обнаруживается только в процессе отладки программы, и т. п.

Альтернативой C и C++ среди универсальных языков программирования, используемых для создания прикладного программного обеспечения, на сегодня является Pascal, компиляторы которого в силу четкого синтаксиса обнаруживают помимо синтаксических и большое количество семантических ошибок. Версия Object Pascal, использованная в среде Delphi, сопровождается профессиональными библиотеками классов, упрощающими ведение больших разработок, в том числе и требующих использования баз данных, что делает Delphi достаточно эффективной средой для создания приложений Windows.

Кроме этих языков к группе универсальных принадлежат также Basic, Modula, Ada и некоторые другие. Каждый из указанных языков, так же, как C++ и Pascal, имеет свои особенности и, соответственно, свою область применения.

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.

Эти языки изучаются в специальных курсах и в настоящем учебнике рассматриваться не будут.

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. Но эти языки совершенно не годятся для создания больших программ и, тем более, программных систем. Основная причина - низкий уровень абстракций, которыми должен оперировать разработчик, откуда недопустимо большое время разработки. Существенно и то, что сами языки низкого уровня

не поддерживают принципов структурного программирования, что значительно ухудшает технологичность разрабатываемых программ.

В настоящее время языки типа Ассемблера обычно используют:

- при написании сравнительно простых программ, взаимодействующих непосредственно с техническими средствами, например драйверов, поскольку в этом случае приходится кропотливо настраивать соответствующее оборудование, преимущества языков программирования высокого уровня становятся несущественными;

- в виде вставок в программы на языках высокого уровня, например, для ускорения преобразования данных в циклах с большим количеством повторений.

Выбор среды программирования. *Средой программирования* называют программный комплекс, который включает специализированный текстовый редактор, встроенные компилятор, компоновщик, отладчик, справочную систему и другие программы, использование которых упрощает процесс написания и отладки программ.

Последнее время широкое распространение получили упоминавшиеся выше среды визуального программирования, в которых программист получает возможность визуального подключения к программе некоторых кодов из специальных библиотек компонентов, что стало возможным с развитием объектно-ориентированного программирования.

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.

Между основными визуальными средами этих фирм Delphi, C++ Builder и Visual C++ имеется существенное различие: визуальные среды фирмы Microsoft обеспечивают более низкий уровень программирования «под Windows». Это является их достоинством и недостатком. Достоинством - так как уменьшается вероятность возникновения «нестандартной» ситуации, т. е. ситуации, не предусмотренной разработчиками библиотеки компонентов, а недостатком - так как это существенно загружает программиста «рутинной» работой, от которой избавлен программист, работающий с Delphi или C++ Builder. Много нареканий вызывает также интерфейс Visual C++, также ориентированный на «низкоуровневое» программирование.

В общем случае, если речь идет о выборе между этими средами, то он в значительной степени должен определяться характером проекта.

Выбор или формирование стандартов разработки. Реальное применение любой технологии проектирования требует формирования или выбора ряда стандартов, которые должны соблюдаться всеми участниками проекта:

- стандарт проектирования;
- стандарт оформления проектной документации;
- стандарт интерфейса пользователя.

Стандарт проектирования должен определять:

- набор необходимых моделей (схем, диаграмм) на каждой стадии проектирования и степень их детализации;

- правила фиксации проектных решений на диаграммах, в том числе правила именования объектов и соглашения по терминологии, набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм, включая требования к форме и размерам объектов;

- требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы и используемых CASE-средств;

- механизм обеспечения совместной работы над проектом, в том числе и правила интеграции подсистем проекта и анализа проектных решений на непротиворечивость.

Стандарт оформления проектной документации должен регламентировать:

- комплектность, состав и структуру документации на каждой стадии;
- требования к ее содержанию и оформлению;
- правила подготовки, рассмотрения, согласования и утверждения документов.

Стандарт интерфейса пользователя должен определять:

- правила оформления экранов (шрифты и цветовую палитру), состав и расположение окон и элементов управления;
- правила пользования клавиатурой и мышью;
- правила оформления текстов помощи;
- перечень стандартных сообщений;
- правила обработки реакции пользователя.

Все описанные выше проектные решения существенно влияют на трудоемкость и сложность разработки. Только после их принятия следует переходить к анализу требований и разработке спецификаций проектируемого программного обеспечения.

Контрольные вопросы и задания

1. Что понимают под технологичностью программного обеспечения? Почему?

2. Какие типы программных продуктов можно выделить? Чем они различаются?

3. Назовите основные эксплуатационные требования к программным продуктам. Какими средствами и приемами обеспечивается каждый из них? Для каких типов программных систем целесообразно указывать каждый из них?

4. В каких ситуациях необходимы предпроектные исследования? Какие вопросы при этом решают? Что получают в результате таких исследований?

5. Назовите, какой раздел технического задания можно считать основным и почему? Какую информацию должны содержать остальные разделы? В чем основная сложность разработки технического задания?

6. Составьте техническое задание на разработку «калькулятора» по типу, предлагаемого Windows. Проанализируйте, какие программы или программные системы могли бы отвечать указанным вами требованиям. Попробуйте ограничить их количество, уточнив техническое задание.

7. Какие решения ранних этапов проектирования считают основными и почему?

4. АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ СТРУКТУРНОМ ПОДХОДЕ

Собственно разработка любого программного обеспечения начинается с анализа требований к будущему программному продукту. В результате анализа получают спецификации разрабатываемого программного обеспечения: выполняют декомпозицию и содержательную постановку решаемых задач, уточняют их взаимодействие и эксплуатационные ограничения. В целом в процессе определения спецификаций строят общую модель предметной области, как некоторой части реального мира, с которой будет тем или иным способом взаимодействовать разрабатываемое программное обеспечение, и конкретизируют его основные функции.

4.1. Спецификации программного обеспечения при структурном подходе

Как уже упоминалось в § 1.4, спецификации представляют собой *полное* и *точное* описание функций и ограничений разрабатываемого программного обеспечения. При этом одна часть спецификаций (*функциональные*) описывает функции разрабатываемого программного обеспечения, а другая часть (*эксплуатационные*) определяет требования к техническим средствам, надежности, информационной безопасности и т. д.

Определение отражает главные требования к спецификациям. Применительно к функциональным спецификациям подразумевается, что:

- требование *полноты* означает, что спецификации должны содержать всю существенную информацию, где ничего важного не было бы упущено, и отсутствует несущественная информация, например детали реализации, чтобы не препятствовать разработчику в выборе наиболее эффективных решений;
- требование *точности* означает, что спецификации должны однозначно восприниматься как заказчиком, так и разработчиком.

Последнее требование выполнить достаточно сложно в силу того, что естественный язык для описания спецификаций не подходит: даже подробные спецификации на естественном языке не обеспечивают необходимой точности. Точные спецификации можно определить, только разработав некоторую *формальную модель* разрабатываемого программного обеспечения.

Формальные модели, используемые на этапе определения спецификаций можно разделить на две группы: модели, *зависящие от подхода к разработке* (структурного или объектно-ориентированного), и модели, *не зависящие* от него. Так диаграммы переходов состояний, которые демонстрируют особенности поведения разрабатываемого программного обеспечения при получении тех или иных сигналов извне (см. § 4.2), и математические модели предметной области (см. § 4.6) используют при любом подходе к разработке.

В рамках структурного подхода на этапе анализа и определения спецификаций используют три типа моделей: ориентированные на функции, ориентированные на данные и ориентированные на потоки данных. Каждую модель целесообразно использовать для своего специфического класса программных разработок.

На рис. 4.1 показана классификация моделей разрабатываемого программного обеспечения, используемых на этапе определения спецификаций.

Следует иметь в виду, что все функциональные спецификации описывают одни и те же характеристики разрабатываемого программного обеспечения: перечень функций и состав обрабатываемых данных. Они различаются только системой приоритетов (акцентов), которая используется разработчиком в процессе анализа требований и определения спецификаций. Диаграммы переходов состояний определяют основные аспекты поведения программного обеспечения во времени, диаграммы потоков данных - направление и структуру потоков данных, а

концептуальные диаграммы классов - отношение между основными понятиями предметной области.



Рис. 4.1. Классификация моделей разрабатываемого программного обеспечения, используемых на этапе определения спецификаций

Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, рекомендуется использовать сразу несколько моделей и сопровождать их текстами: словарями, описаниями и т. п., которые поясняют соответствующие диаграммы.

Так методологии *структурного анализа* и *проектирования*, основанные на моделировании потоков данных, обычно используют комплексное представление проектируемого программного обеспечения в виде совокупности моделей:

- диаграмм потоков данных (DFD - Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе (см. § 4.4);
- диаграмм «сущность-связь» (ERD - Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы (см. § 4.5);
- диаграмм переходов состояний (STD - State Transition Diagrams), характеризующих поведение системы во времени (см. § 4.2);
- спецификаций процессов;
- словаря терминов.

Взаимосвязь элементов такой обобщенной модели показана на рис. 4.2.

Спецификации процессов. Спецификации процессов обычно представляют в виде краткого текстового описания, схем алгоритмов, псевдокодов, Flow-форм или диаграмм Насси-Шнейдермана. Поскольку описание процесса должно быть кратким и понятным как разработчику, так и заказчику, для их спецификации чаще всего используют псевдокоды.

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. Он предназначен для повышения степени понимания предметной области и исключения риска возникновения разногласий при обсуждении моделей между заказчиками и разработчиками.

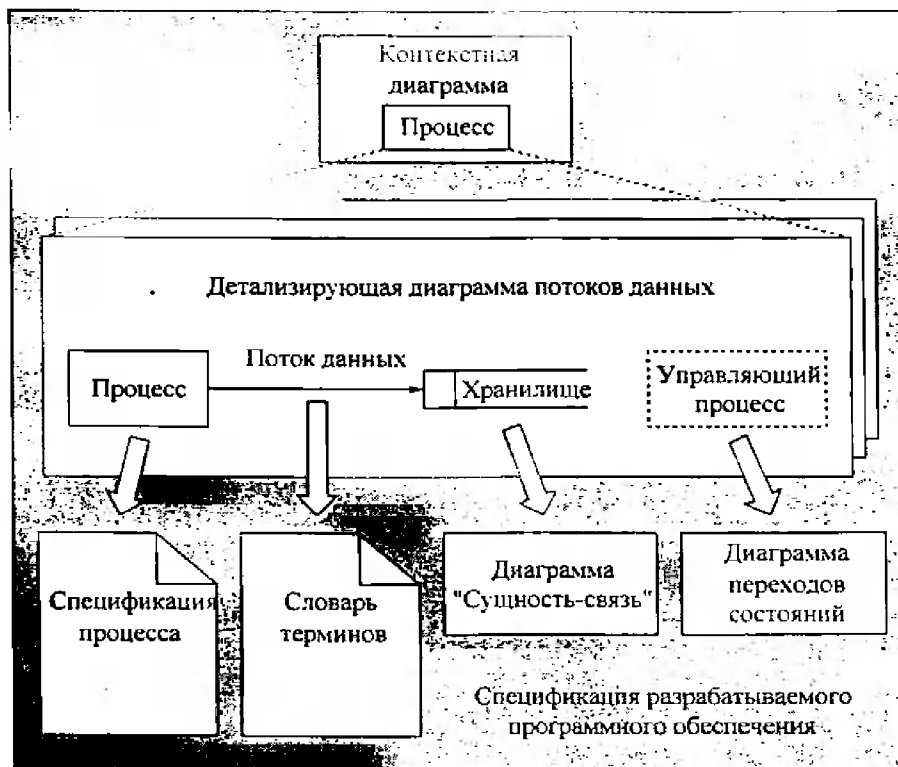


Рис. 4.2. Элементы полной спецификации методологий структурного анализа и проектирования программного обеспечения, основанных на потоках данных

Обычно описание термина в словаре выполняют по следующей схеме:

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

В качестве примера приведем описание одного из терминов системы решения комбинаторно-оптимизационных задач:

ТерминАлгоритм

КатегорияПонятие предметной области

ОписаниеВ настоящем проекте используется для обозначения обобщенного понятия «реализация процедуры решения конкретной задачи выбранным методом»

Кроме указанных моделей в состав полной спецификации при любом подходе могут входить математические модели описания объектов предметной области, которые позволяют уточнить основные соотношения анализируемых величин и накладываемые на них ограничения. Перейдем к более подробному рассмотрению перечисленных моделей.

4.2. Диаграммы переходов состояний

Диаграмма переходов состояний является графической формой предоставления конечного автомата - математической абстракции, используемой для моделирования детерминированного поведения технических объектов или объектов реального мира.

На этапе анализа требований и определения спецификаций диаграмма переходов состояний демонстрирует *поведение* разрабатываемой программной системы при получении управляющих воздействий. Под *управляющими воздействиями* или сигналами в данном случае понимают

управляющую информацию, получаемую системой извне. Например, управляющими воздействиями считают команды пользователя и сигналы датчиков, подключенных к компьютерной системе. Получив такое управляющее воздействие, разрабатываемая система должна выполнить определенные действия и или остаться в том же состоянии, или перейти в другое состояние взаимодействия с внешней средой.

Для построения диаграммы переходов состояний необходимо в соответствии с теорией конечных автоматов определить: основные состояния, управляющие воздействия (или условия перехода), выполняемые действия и возможные варианты переходов из одного состояния в другое. Условные обозначения, используемые при построении диаграмм переходов состояний, показаны на рис. 4.3.

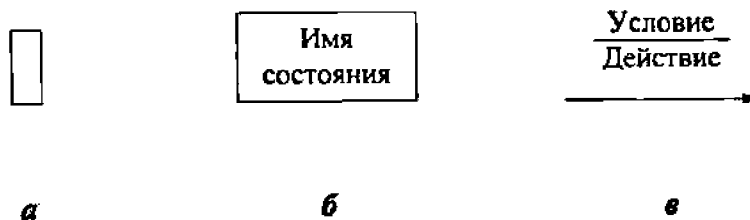


Рис. 4.3. Условные обозначения диаграмм переходов состояний:

a – терминальное состояние; *б* – промежуточное состояние;
в – переход

Если программная система в процессе функционирования активно не взаимодействует с окружающей средой (пользователем или датчиками), например, использует примитивный интерфейс и выполняет некоторые вычисления по заданным исходным данным, то диаграмма переходов состояний обычно интереса не представляет. В этом случае она демонстрирует только последовательно выполняемые переходы: из исходного состояния в состояние ввода данных, затем после выполнения вычислений - в состояние вывода и, наконец, в состояние завершения работы (рис. 4.4).

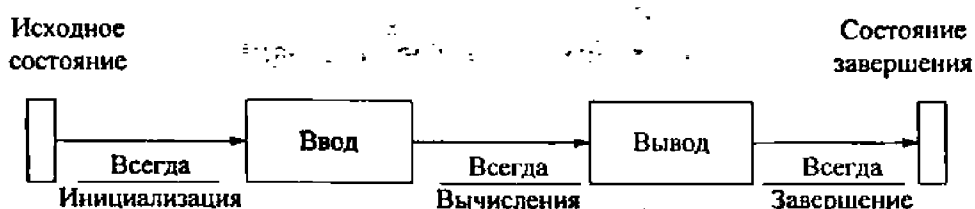


Рис. 4.4. Диаграмма переходов состояний программного обеспечения, активно не взаимодействующего с окружающей средой

Для интерактивного программного обеспечения с развитым пользовательским интерфейсом основные управляющие воздействия - команды пользователя, для программного обеспечения реального времени — сигналы от датчиков и/или оператора производственного процесса. Общим для этих типов программного обеспечения является наличие состояния ожидания, когда программное обеспечение приостанавливает работу до получения очередного управляющего воздействия. Для интерактивного программного обеспечения наиболее характерно получение команд различных типов (рис. 4.5), а если это еще и программное обеспечение реального времени - однотипных сигналов (либо от многих датчиков, либо требующих продолжительной обработки).

В отличие от интерактивных систем для систем реального времени обычно установлено более жесткое ограничение на время обработки полученного сигнала программного обеспечения. Такое ограничение часто требует выполнения дополнительных исследований поведения системы во времени, например, с использованием сетей Петри или марковских процессов.

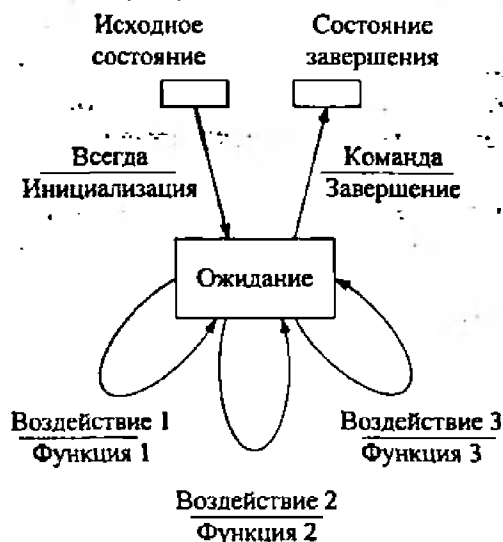


Рис. 4.5. Пример диаграммы переходов состояний программного обеспечения, активно взаимодействующего с окружающей средой

К программному обеспечению, требующему уточнения особенностей поведения посредством построения диаграммы переходов состояний, относится и программное обеспечение, ориентированное на работу в сети (см. § 1.1). При этом отдельно строят модели поведения сервера и клиента, представляя сообщения, передаваемые между ними, в виде управляющих воздействий.

Пример 4.1. Рассмотрим диаграмму переходов состояний для программы построения графиков функций одной переменной, техническое задание на которую представлено в § 3.4.

Программа относится к классу интерактивных, соответственно на этапе анализа и определения спецификаций целесообразно уточнить поведение программы на уровне интерфейса с пользователем, тем более, что наличие простого интерфейса оговорено в техническом задании. Один из возможных вариантов диаграммы переходов состояний программы представлен на рис. 4.6.

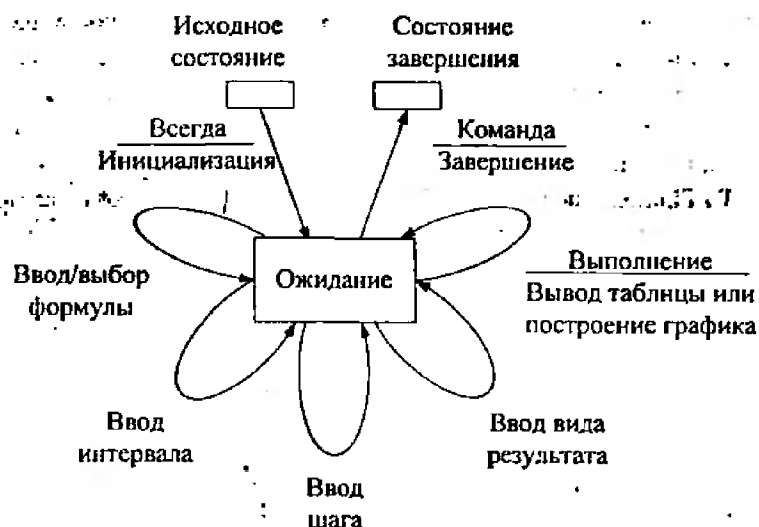


Рис. 4.6. Диаграмма переходов состояний программы построения графиков/таблиц функций

Полученную диаграмму переходов состояний следует согласовать с заказчиком программного обеспечения.

диаграмму

4.3. Функциональные диаграммы

Функциональными называют диаграммы, в первую очередь отражающие *взаимосвязи функций* разрабатываемого программного обеспечения. В качестве примера функциональной модели рассмотрим активностную модель, предложенную Д. Россом в составе методологии функционального моделирования SADT (Structured Analysis and Design Technique - технология структурного анализа и проектирования) в 1973 г. [58].

Примечание. Методология SADT предполагает, что модель может основываться либо на функциях системы, либо на ее предметах (данных, оборудовании, информации и т. п.). В обоих случаях используют схожие графические нотации, но в первом случае блок соответствует функции, а во втором — элементу данных. Соответствующие модели принято называть активностными моделями и моделями данных. Полная модель включает построение обеих моделей, обеспечивающих более полное описание программного обеспечения, однако широкое распространение получили только активностные (функциональные) модели. На основе методологии SADT в дальнейшем была построена известная методология описания сложных систем IDEFO (Icam DEFinition - нотация ICAM), которая является основной частью программы ICAM (Integrated Computer-Aided Manufacturing - интегрированная компьютеризация производства), проводимой по инициативе ВВС США.

Отображение взаимосвязи функций активностной модели осуществляется посредством построения *иерархии функциональных диаграмм*, схематически представляющих взаимосвязи нескольких функций. Каждый блок такой диаграммы соответствует некоторой функции, для которой должны быть определены: исходные данные, результаты, управляющая информация и механизмы ее осуществления — человек или технические средства.

Все перечисленные выше связи функции представляются дугами, причем тип связи и ее направление строго регламентированы. Дуги, изображающие каждый тип связей, должны подходить к блоку с определенной стороны (рис. 4.7), а направление связи должно указываться стрелкой в конце дуги.

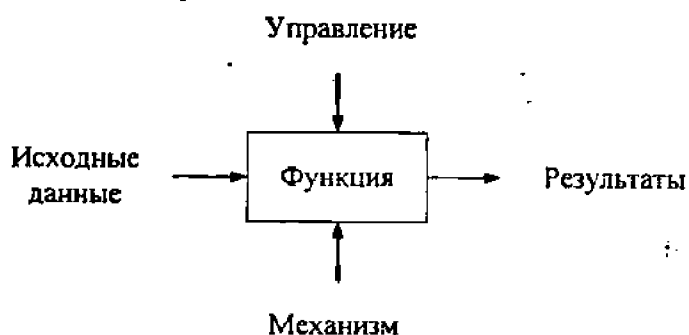


Рис. 4.7. Функциональный блок и интерфейсные дуги

Физически дуги исходных данных, результатов и управления представляют собой наборы данных, передаваемые между функциями. Дуги, определяющие механизм выполнения функции, в основном используются при описании спецификаций сложных информационных систем, которые включают как автоматизированные, так и ручные операции. Блоки и дуги маркируются текстами на естественном языке.

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:

- вход - выход блока подается на вход блока с меньшим доминированием, т. е. Следующего (рис. 4.8, а);
- управление - выход блока используется как управление для блока с меньшим доминированием (следующего) (рис. 4.8, б);
- обратная связь по входу - выход блока подается на вход блока с большим доминированием (предыдущего) (рис. 4.8, в);
- обратная связь по управлению — выход блока используется как управляющая информация для блока с большим доминированием (предыдущего) (рис. 4.8, г);
- выход-исполнитель - выход блока используется как механизм для другого блока (рис. 4.8, д).

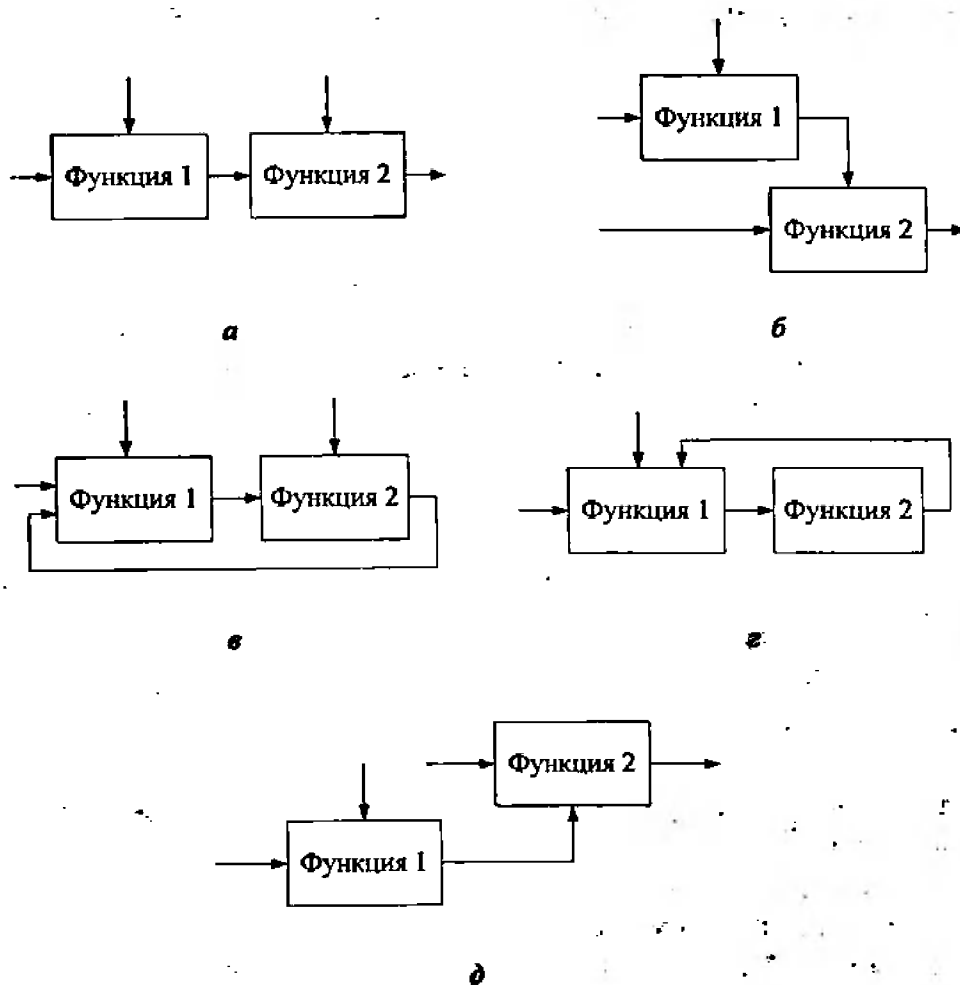


Рис. 4.8. Типы влияний блоков:

а – вход; б – управление; в – обратная связь по входу;
г – обратная связь по управлению; д – выход-исполнитель

Дуги могут разветвляться и соединяться вместе различными способами. Разветвление означает, что часть или вся информация может использоваться в каждом ответвлении дуги. Дуга всегда помечается до ветвления, чтобы идентифицировать передаваемый набор данных. Если ветвь дуги после ветвления непомечена, то непомеченная ветвь содержит весь набор данных. Каждая метка ветви уточняет, что именно содержит данная ветвь (рис. 4.9).

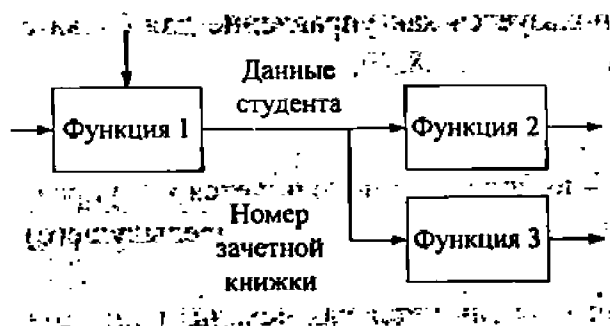


Рис. 4.9. Пример обозначения дуг при ветвлении

Построение иерархии функциональных диаграмм ведется поэтапно с увеличением уровня детализации: диаграммы каждого следующего уровня уточняют структуру родительского блока. Построение модели начинают с единственного блока, для которого определяют исходные данные, результаты, управление и механизмы реализации. Затем он последовательно детализируется с использованием метода пошаговой детализации (см. § 1.3). При этом рекомендуется каждую функцию представлять не более чем 3—7-ю блоками. Во всех случаях *каждая подфункция может использовать или продуцировать только те элементы данных, которые использованы или продуцируются родительской функцией*, причем никакие элементы не могут быть опущены, что обеспечивает непротиворечивость построенной

Стрелки, приходящие с родительской диаграммы или уходящие на нее, нумеруют, используя символы и числа. Символ обозначает тип связи: I - входная, С - управляющая, М - механизм, R - результат. Число - номер связи по соответствующей стороне родительского блока, считая сверху вниз и слева направо.

Все диаграммы связывают друг с другом иерархической нумерацией блоков: первый уровень - АО, второй - А1, А2 и т. п., третий - А11, А12, А13 и т. п., где первые цифры — номер родительского блока, а последняя — номер конкретного субблока родительского блока.

Детализацию завершают после получения функций, назначение которых хорошо понятно как заказчику, так и разработчику. Эти функции описывают, используя естественный язык или псевдокоды.

В процессе построения иерархии диаграмм фиксируют всю уточняющую информацию и строят словарь данных, в котором определяют структуры и элементы данных, показанных на диаграммах.

Таким образом, в результате получают спецификацию, которая состоит из иерархии функциональных диаграмм, спецификаций функций нижнего уровня и словаря, имеющих ссылки друг на друга.

Пример 4.2. Разработку функциональных диаграмм продемонстрируем на примере уточнения спецификаций программы построения таблиц/графиков функций одной переменной.

Диаграмма, показанная на рис. 4.10, а, является диаграммой верхнего уровня. На ней хорошо видно, что является исходными данными для программы, и каких результатов работы от нее ожидают.

Диаграмма, представленная на рис. 4.10, б, уточняет функции программы. На ней показаны четыре блока: Ввод/выбор функций и ее разбор, Добавление функции в список, Построение таблицы значений и Построение графика функции. Для каждого блока определены исходные данные, управляющие воздействия и результаты. Согласно правилам наименования входов/выходов, имеющих продолжение на родительской диаграмме, на диаграмме использованы следующие обозначения:

- I1 - функция,
- I2 - отрезок,
- I3 - шаг,

C1 - вид график/таблица,
 R1 - график функции на отрезке,
 R2 - таблица значений функции на отрезке.

Словарь в этом случае должен содержать описание всех данных, используемых в системе.

Функциональную модель целесообразно применять для определения спецификаций программного обеспечения, не предусматривающего работу со сложными структурами данных, так как она ориентирована на декомпозицию функций.

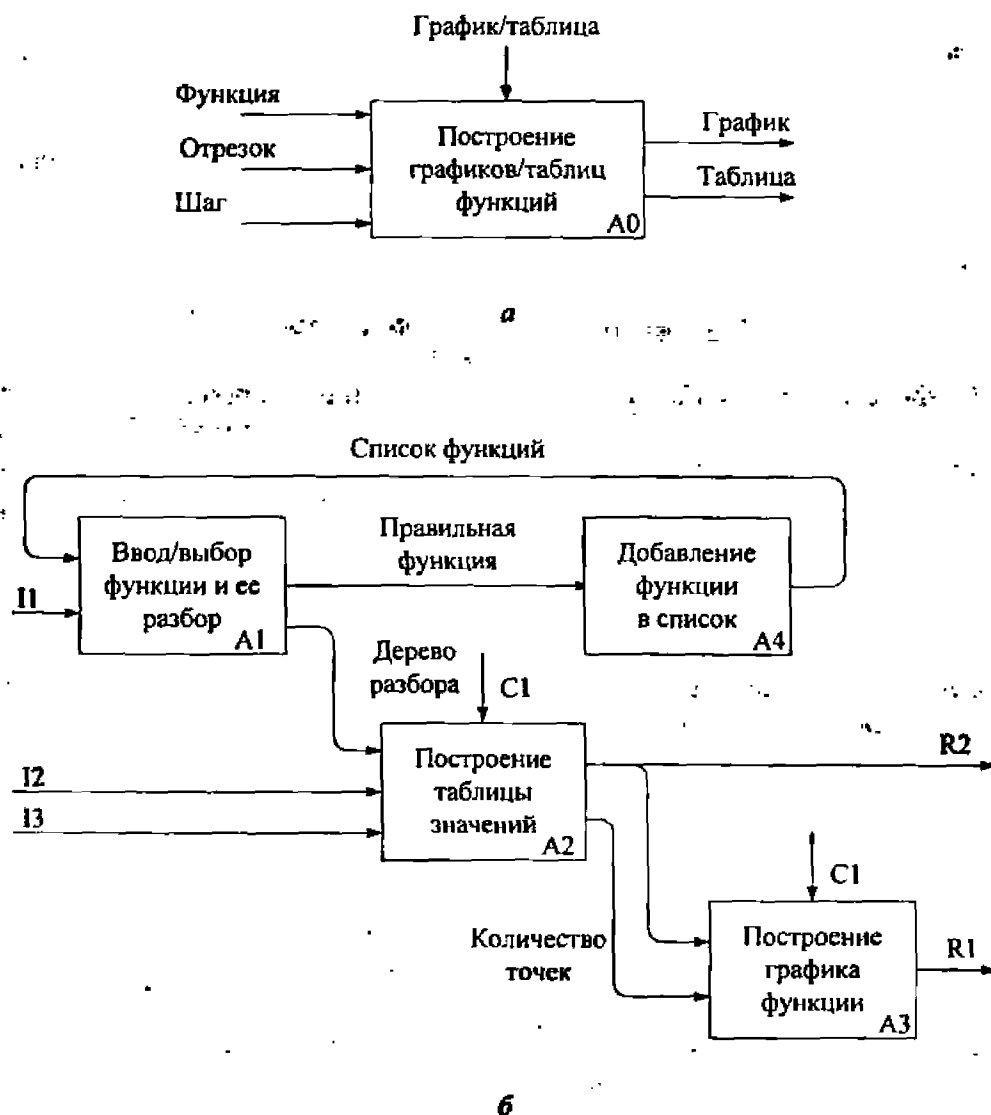


Рис. 4.10. Функциональные диаграммы для системы исследования функций:
 а – диаграмма верхнего уровня; б – уточняющая диаграмма

4.4. Диаграммы потоков данных

Диаграммы потоков данных позволяют специфицировать как функции разрабатываемого программного обеспечения, так и обрабатываемые им данные. При использовании этой модели систему представляют в виде иерархии диаграмм потоков данных, описывающих асинхронный процесс преобразования информации с момента ввода в систему до выдачи пользователю. На каждом следующем уровне иерархий происходит уточнение процессов, пока очередной процесс не будет признан элементарным.

Примечание. Модели потоков данных были независимо предложены сначала Е. Йорданом (1975), затем Ч. Гейном и Т. Сарсоном (1979). На этих моделях основаны классические методологии структурного анализа и проектирования программного обеспечения соответственно Йордана-Де Марка и Гейна-Сарсона. Та же модель используется в методологии структурного анализа и проектирования SSADM (Structured Systems Analysis and Design Method) принятой в Великобритании в качестве национального стандарта разработки информационных систем.

В основе модели лежат понятия внешней сущности, процесса, хранилища (накопителя) данных и потока данных.

Внешняя сущность - материальный объект или физическое лицо, выступающие в качестве источников или приемников информации, например, заказчики, персонал, поставщики, клиенты, банк и т. п.

Процесс - преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом. Каждый процесс в системе имеет свой номер и связан с исполнителем, который осуществляет данное преобразование. Как в случае функциональных диаграмм, физически преобразование может осуществляться компьютерами, вручную или специальными устройствами. На верхних уровнях иерархии, когда процессы еще не определены, вместо понятия «процесс» используют понятия «система» и «подсистема», которые обозначают соответственно систему в целом или ее функционально законченную часть.





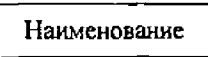

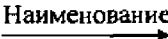
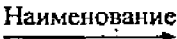
Хранилище данных - абстрактное устройство для хранения информации. Тип устройства и способы помещения, извлечения и хранения для такого устройства не детализируют. Физически это может быть база данных, файл, таблица в оперативной памяти, картотека на бумаге и т. п.

Поток данных — процесс передачи некоторой информации от источника к приемнику. Физически процесс передачи информации может происходить по кабелям под управлением программы или программной системы или вручную при участии устройств или людей вне проектируемой системы.

Таким образом, диаграмма иллюстрирует как потоки данных, порожденные некоторыми внешними сущностями, трансформируются соответствующими процессами (или подсистемами), сохраняются накопителями данных и передаются другим внешним сущностям — приемникам информации. В результате мы получаем сетевую модель хранения/обработки информации.

Для изображения диаграмм потоков данных традиционно используют два вида нотаций: нотации Йордана и Гейна-Сарсона (табл. 4.1).

Таблица 4.1

Понятие	Нотация Йордана	Нотация Гейна-Сарсона
Внешняя сущность		
Система, подсистема или процесс		
Накопитель данных		
Поток		

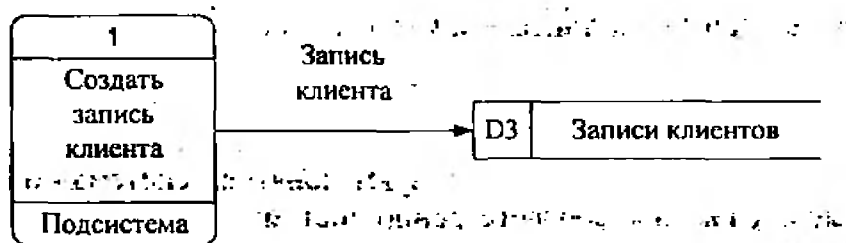


Рис. 4.11. Пример потока данных (нотация Гейна-Сарсона)

Над линией потока, направление которого обозначают стрелкой, указывают, какая конкретно информация в данном случае передается (рис. 4.11).

Построение иерархии диаграмм потоков данных начинают с диаграммы особого вида - *контекстной диаграммы*, которая определяет наиболее общий вид системы. На такой диаграмме показывают, как разрабатываемая система будет взаимодействовать с приемниками и источниками информации без указания исполнителей, т. е. описывают интерфейс между системой и внешним миром. Обычно начальная контекстная диаграмма имеет форму звезды.

Если проектируемая система содержит большое количество внешних сущностей (более 10-ти), имеет распределенную природу или включает уже существующие подсистемы, то строят *иерархии* контекстных диаграмм.

При разработке контекстных диаграмм происходит детализация функциональной структуры будущей системы, что особенно важно, если разработка ведется несколькими коллективами разработчиков.

Полученную таким образом модель системы проверяют на полноту исходных данных об объектах системы и изолированность объектов (отсутствие информационных связей с другими объектами).

На следующем этапе каждую подсистему контекстной диаграммы детализируют при помощи диаграмм потоков данных. В процессе детализации соблюдают правило **б а л а н с и р о в к и** - *при детализации подсистемы можно использовать компоненты только тех подсистем, с которыми у разрабатываемой подсистемы существует информационная связь* (т. е. с которыми она связана потоками данных).

Решение о завершении детализации процесса принимают в следующих случаях:

- процесс взаимодействует с 2-3-мя потоками данных;
- возможно описание процесса последовательным алгоритмом;
- процесс выполняет единственную логическую функцию преобразования входной информации в выходную.

На недетализируемые процессы составляют спецификации, которые должны содержать описание логики (функций) данного процесса. Такое описание может, выполняться: на естественном языке, с применением структурированного естественного языка (псевдокодов), с применением таблиц и деревьев решений, в виде схем алгоритмов, в том числе flow-форм и диаграмм Насси-Шнейдермана (см. § 2.4).

Для облегчения восприятия процессы детализируемой подсистемы нумеруют, соблюдая иерархию номеров: так процессы, полученные при детализации процесса или подсистемы «1», должны нумероваться «1.1», «1.2» и т. д. Кроме этого желательно размещать на каждой диаграмме от 3-х до 6-7-ми процессов и не загромождать диаграммы деталями, не существенными на данном уровне.

Декомпозицию потоков данных необходимо осуществлять параллельно с декомпозицией процессов.

Окончательно разработку модели выполняют в два этапа.

1 этап - построение контекстной диаграммы - включает выполнение следующих действий:

- классификацию множества требований и организацию их в основные функциональные группы — процессы;

- идентификацию внешних объектов - внешних сущностей, с которыми система должна быть связана;

- идентификацию основных видов информации - потоков данных, циркулирующей между системой и внешними объектами;

- предварительную разработку контекстной диаграммы;

- изучение предварительной контекстной диаграммы и внесение в нее изменений по результатам ответов на возникающие при изучении вопросы по всем ее частям;

- построение контекстной диаграммы путем объединения всех процессов предварительной диаграммы в один процесс, а также группирования потоков.

2 этап - формирование иерархии диаграмм потоков данных – включает для каждого уровня:

- проверку и изучение основных требований по диаграмме соответствующего уровня (для первого уровня - по контекстной диаграмме);

- декомпозицию каждого процесса текущей диаграммы потоков данных с помощью детализирующей диаграммы или — если некоторую функцию сложно или невозможно выразить комбинацией процессов, построение спецификации процесса;

- добавление определений новых потоков в словарь данных при каждом появлении их на диаграмме;

- проведение ревизии с целью проверки корректности и улучшения наглядности модели после построения двух-трех уровней.

Полная спецификация процессов включает также описание *структур данных*, используемых как при передаче информации & потоке, так и при хранении в накопителе. Описываемые структуры данных могут содержать альтернативы, условные вхождения и итерации. Условное вхождение означает, что соответствующие элементы данных в структуре могут отсутствовать. Альтернатива означает, что в структуру может входить один из перечисленных элементов. Итерация означает, что элемент может повторяться некоторое количество раз (см. § 4.5).

Кроме того, для данных должен быть указан тип: непрерывное или дискретное значение. Для непрерывных данных могут определяться единицы измерений, диапазон значений, точность представления и форма физического кодирования. Для дискретных - может указываться таблица допустимых значений.

Полученную законченную модель необходимо проверить на полноту и согласованность. Под *согласованностью* модели в данном случае понимают выполнение для всех потоков данных *правил сохранения информации*: все поступающие куда-либо данные должны быть считаны и записаны.

Пример 4.3. Разработаем иерархию диаграмм потоков данных программы построения графиков/таблиц функций.

Разработку начнем с построения контекстной диаграммы. Для чего определим внешние сущности и потоки данных между программой и внешними сущностями. У данной системы единственная внешняя сущность Учащийся. Он вводит или выбирает из списка функцию, задает интервал и количество точек, а затем получает таблицу значений функции и ее график. На рис. 4.12 представлена контекстная диаграмма системы.

Детализируя эту диаграмму, получаем три процесса: Ввод/выбор функции и ее разбор, Построение таблицы значений функции и Построение графика функции. Для хранения функций добавляем хранилище функций. Затем определяем потоки данных.

Если сравнить полученную детализирующую диаграмму потоков данных (рис. 4.13) и функциональные диаграммы для той же системы (см. рис. 4.10), то можно отменить некоторые различия в представлении одной и той же информации. Например, на диаграмме потоков данных можно показать хранилище данных, что очень существенно для систем, включающих базы данных. Кроме того, диаграммы потоков данных позволяют точно адресовать функции системы при наличии нескольких категорий пользователей, что демонстрирует следующий пример.

Пример 4.4. Разработать иерархию диаграмм потоков данных системы учета успеваемости студентов (см. Техническое задание в примере 3.2).

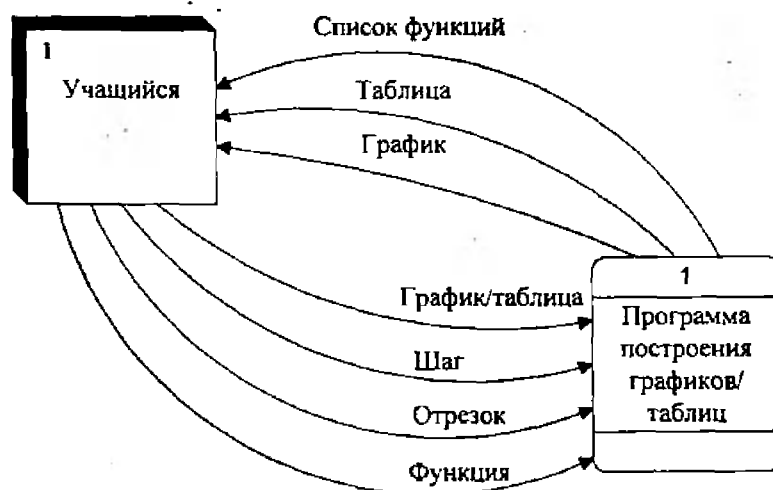


Рис. 4.12. Контекстная диаграмма программы построения графиков функций (нотация Гейна-Сарсона)

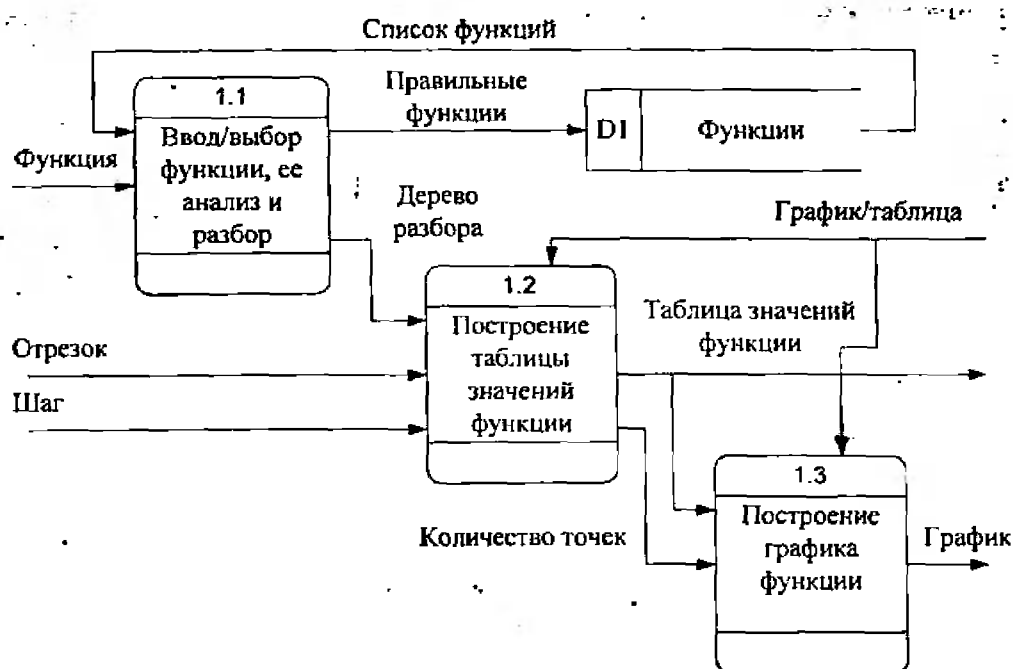


Рис. 4.13. Детализирующая диаграмма потоков данных системы исследования функций (нотация Гейна-Сарсона)

В качестве внешних сущностей для системы выступают Декан, Заместитель декана по курсу и Сотрудник деканата. Определим потоки данных между этими сущностями и системой.

Декан должен получать (рис. 4.14):

- сводку успеваемости по факультету (процент успеваемости групп, курсов и в целом по факультету) на текущий или указанный момент времени;
- полные сведения об учебе конкретного студента (успеваемость по всем изученным предметам всех завершенных семестров обучения с учетом пересдач).

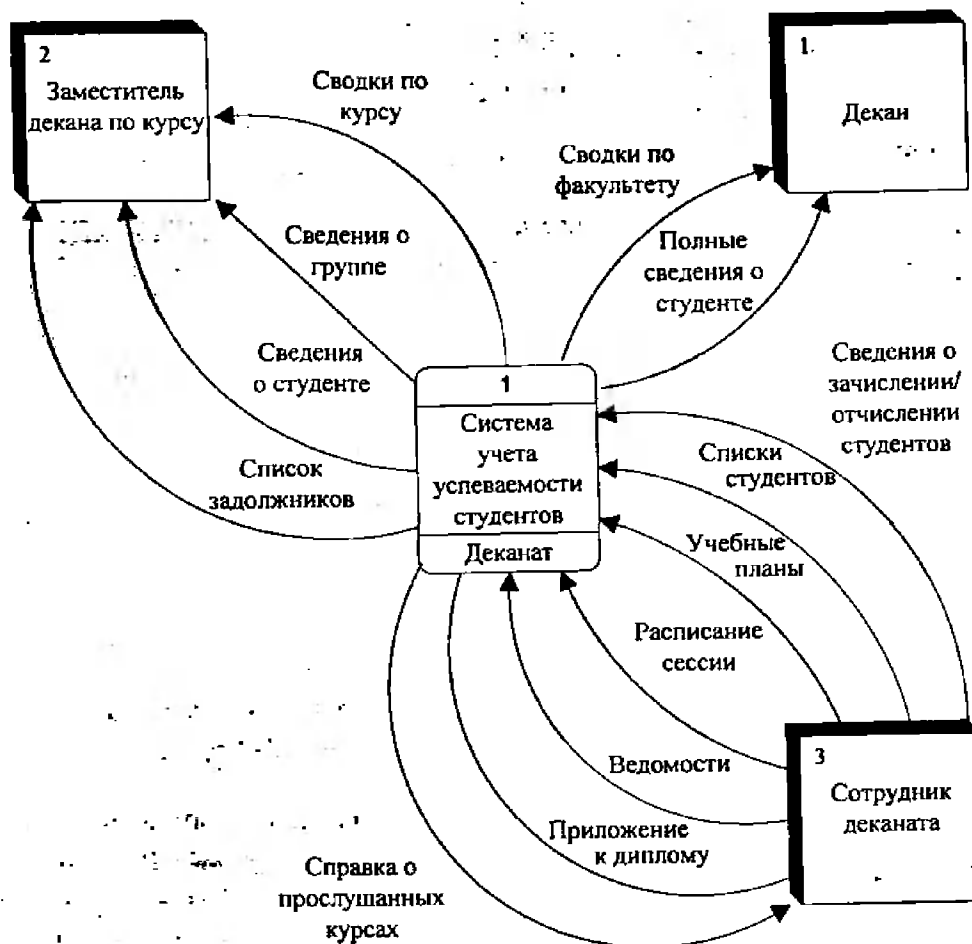


Рис. 4.14. Контекстная диаграмма системы учета успеваемости студентов (нотация Гейна-Сарсона)

Заместитель декана по курсу должен получать:

- сводку успеваемости по курсу (процент успеваемости по группам) на текущий или указанный момент;
- сведения о сдаче экзаменов и зачетов указанной группой;
- текущие сведения об успеваемости конкретного студента;
- полные сведения об учебе конкретного студента (успеваемость по всем изученным предметам всех завершенных семестров обучения с учетом пересдач);
- список задолжников по факультету с указанием групп и несданных предметов.

Сотрудник деканата должен обеспечивать:

- ввод списков студентов, зачисленных на первый курс;
- корректировку списков студентов в соответствии с приказами о зачислении, отчислении, переводе и т. п.;
- ввод учебных планов кафедр;
- ввод расписания сессии;
- ввод результатов сдачи зачетов и экзаменов на основании ведомостей и направлений.

Кроме того, сотрудник декана должен иметь возможность получать:

- справку о прослушанных студентом предметах с указанием часов и итоговых оценок;
- приложение к диплому выпускника также с указанием часов и итоговых оценок.

Далее детализируем процессы в системе. На рис. 4.15 представлена детализирующая диаграмма потоков данных, где выделены две подсистемы: Подсистема наполнения базы данных и Подсистема формирования отчетов, а также хранилище данных, которое может быть реализовано как с помощью средств СУБД, так и без них. Решение о целесообразности использования средств СУБД может быть принято позднее, после анализа структур хранимых данных.

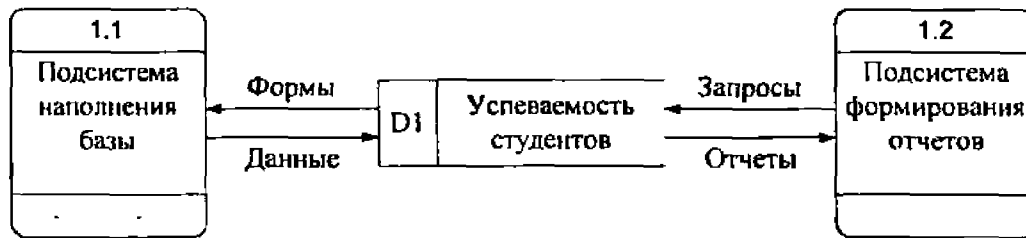


Рис. 4.15. Детализирующая диаграмма потоков данных второго уровня (нотация Гейна-Сарсона)

Дальнейшую детализацию процессов можно не выполнять, так как их сущность для разработчика очевидна. Однако становится ясно, что полная спецификация данной разработки должна включать описание базы данных. Такое описание в виде диаграммы «сущность-связь» будет рассмотрено в §4.5.

Кроме этого, как уже упоминалось в § 4.1, для данной системы целесообразно выполнить моделирование управляющих процессов, что позволит уточнить организацию процесса обработки данных.

Моделирование управляющих процессов с помощью диаграмм потоков данных. Для представления управляющих процессов в проектируемых системах можно применить диаграммы переходов состояний, рассмотренные в § 4.2, или диаграммы управляющих потоков данных, которые используют понятия: управляющий процесс, управляющий поток данных и, возможно, хранилище управляющих данных.

Управляющий процесс получает с помощью управляющих потоков некоторую информацию о ситуации в системе и инициирует посредством управляющего потока соответствующие процессы.

На диаграммах управляющих потоков данных используют те же обозначения, что и для обычных потоков, но изображают их пунктирной линией. Дополнительно может быть указан тип управляющего потока:

- Т-поток (Trigger Flow - триггерный поток) - поток управления, который может только «включать» процесс - следующий управляющий сигнал опять «включит» процесс, даже если процесс уже активен;

- А-поток (Activator Flow - активирующий поток) - поток управления, который может как «включать», так и «выключать» управляемый процесс - если процесс включен, то следующий сигнал его выключит;

- Е/Д-поток (Enable/Disable Flow - переключающий поток) - поток управления, который может включать процесс сигналом по одной (Е) линии и выключать - сигналом по другой (D) линии.

При необходимости тип потока данных (управляющий или обычный) можно изменять. Для этого используют специальное обозначение - узел изменения типа потока данных (рис. 4.16). К этому узлу поток подходит как поток данных, а выходит из него как управляющий поток.

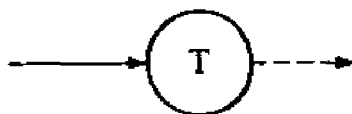


Рис. 4.16. Узел изменения типа потока данных

Пример 4.5. Построим диаграмму потоков управляющих данных для программы построения таблиц/графиков функций и наложим ее на диаграмму потоков данных для этой программы, представленную на рис. 4.13.

Для управления процессом исследования функции добавляем процесс Управление программой. Этот процесс получает четыре потока управляющих данных (команды Функция, Отрезок, Шаг и График/Таблица) и генерирует два управляющих потока Т-типа: Изменить функцию и Заменить отрезок или шаг, а также управляющий поток А-типа: Изменить вид результата.

Управляющий поток Изменить функцию активизирует процесс Ввода/выбора и разбора функции. Сначала функция проверяется с точки зрения корректности записи. Если функция введена правильно, то она заносится в список и обработка продолжается, если нет, то процесс прекращается с выдачей соответствующего сообщения. Нормальное завершение выполнения первого блока инициирует выполнение второго блока и т. д.

При получении команд Изменить отрезок или Изменить шаг генерируется управляющий поток Изменить отрезок или шаг, который отвечает за пересчет таблицы значений функции. При выбранном виде результата График генерируется управляющий поток Построение графика функции.

Полученная комбинированная диаграмма потоков обычных и управляющих данных представлена на рис. 4.17.

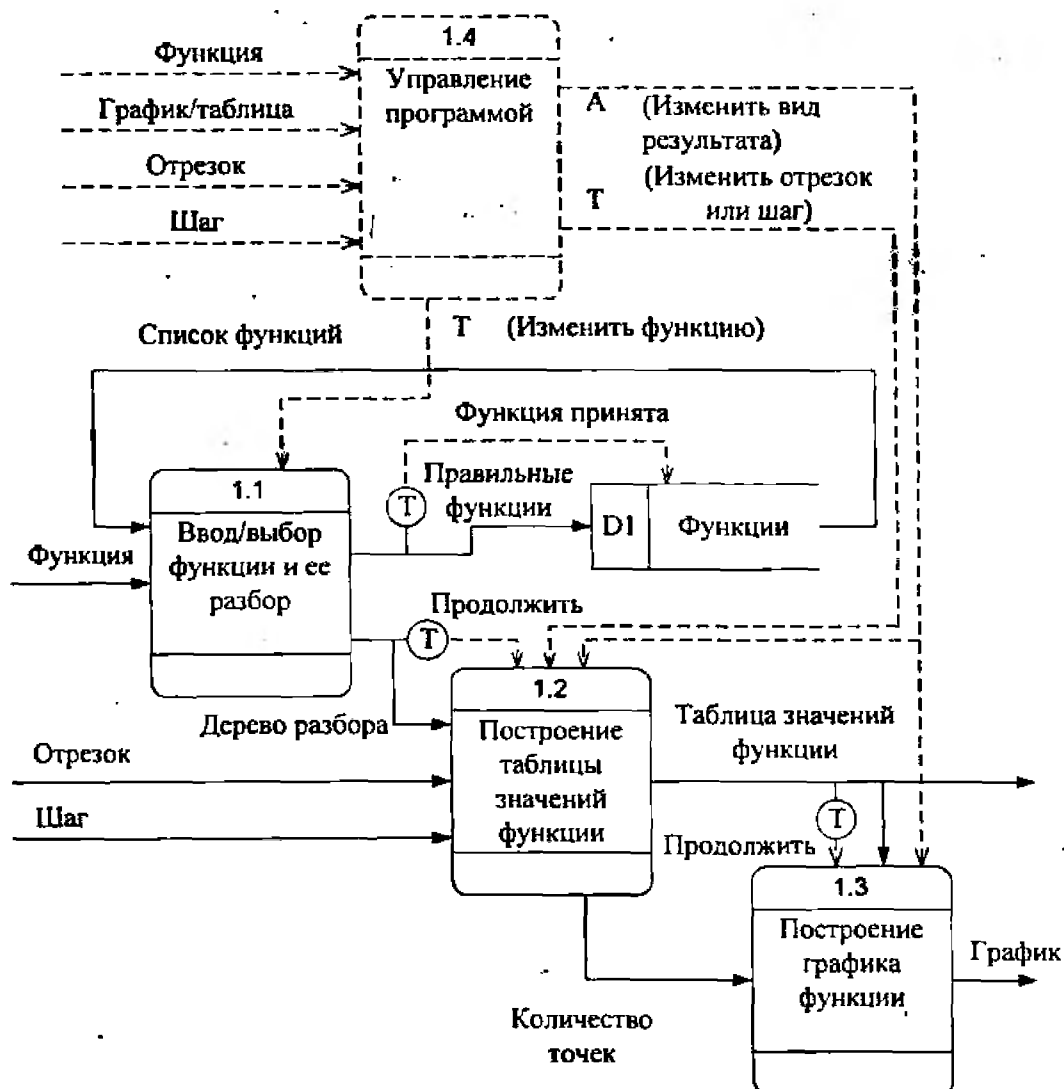


Рис. 4.17. Диаграмма потоков данных, дополненная диаграммой потоков управляющих данных

4.5. Структуры данных и диаграммы отношений компонентов данных

Структурой данных называют совокупность правил и ограничений, которые отражают связи, существующие между отдельными частями (элементами) данных.

Различают *абстрактные структуры* данных, используемые для уточнения связей между элементами, и *конкретные структуры*, используемые для представления данных в программах.

Все абстрактные структуры данных можно разделить на три группы: структуры, элементы которых не связаны между собой, структуры с неявными связями элементов — таблицы и структуры, связь элементов которых указывается явно — графы (рис. 4.18).



Рис. 4.18. Классификация абстрактных структур данных

В первую группу входят *множества* (рис. 4.19, а) и *кортежи* (рис. 4.19, б). Наиболее существенная характеристика элемента данных в этих структурах — его принадлежность некоторому набору, т. е. *отношение вхождения*. Данные абстрактные структуры используют, если никакие другие отношения элементов не являются существенными для описываемых объектов.

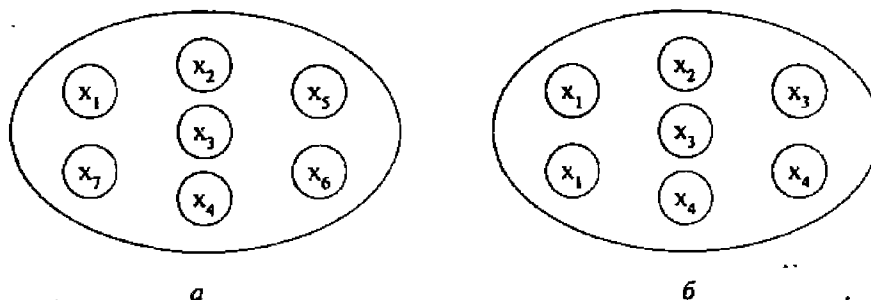


Рис. 4.19. Множество (а) и кортеж (б)

Ко второй группе относят *векторы*, *матрицы*, *массивы* (многомерные), *записи*, *строки*, а также *таблицы*, включающие перечисленные структуры в качестве частей. Использование этих абстрактных типов может означать, что существенным является не только вхождение элемента данных в некоторую структуру, но и их порядок, а также отношения *иерархии структур*, т. е. вхождение структуры в структуру более высокой степени общности (рис. 4.20).

а

1	2	3	4	5
---	---	---	---	---

б

1	2	-3	4	5
-7	4	0	-2	8
3	9	3	-4	45

в

1	2	-3	4	2
-7	4	0	-2	8
3	9	3	-4	45

г

а	б	в	г	д	
---	---	---	---	---	--

д

Иванов	Иван	1376034
--------	------	---------

е

ИУ6-12		Оценки					Ср. балл
ИУ6-11		Оценки					
№	ФИО	АЯ и П	Мат.ан.	Ан.геом.	История	Инф-ка	Ср. балл
1	Иванов И.И.	4	5	3	4	5	4.2
2	Петров П.П.	4	4	4	4	4	4.0
3	Сидоров С.С.	3	3	3	4	3	3.2
В среднем		3.7	4	3.3	4	4	3.4

Рис. 4.20. Таблицы:

а – числовой вектор; б – матрица; в – трехмерный массив; г – строка;
д – запись; е – массив однотипных таблиц с вложенными структурами

В тех случаях, когда существенны связи элементов данных между собой, в качестве модели структур данных используют *графы* [55]. На рис. 4.21 показаны различные варианты графовых моделей.

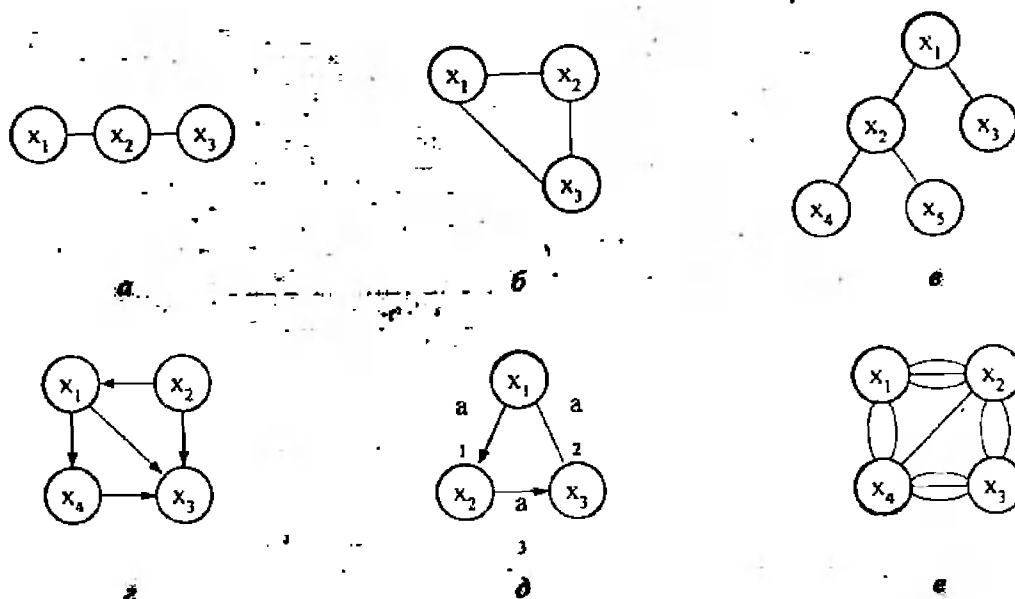


Рис. 4.21. Графы:

а – цепь; б – цикл; в – дерево; г – ориентированный граф;
д – взвешенный смешанный граф; е – мультиграф

Очень существенно, что в реальности возможно вложение структур данных, в том числе и разных типов, а потому для их описания могут потребоваться специальные модели. В зависимости от описываемых типов отношений модели структур данных принято делить на иерархические и сетевые.

Иерархические модели позволяют описывать упорядоченные или неупорядоченные отношения вхождения элементов данных в компонент более высокого уровня, т. е. множества, таблицы и их комбинации. К иерархическим моделям относят модель Джексона-Орра, для графического представления которой можно использовать:

- диаграммы Джексона, предложенные в составе методики проектирования программного обеспечения того же автора в 1975 г.;

- скобочные диаграммы Орра, предложенные в составе методики проектирования программного обеспечения Варнье-Орра (1974).

Сетевые модели основаны на графах, а потому позволяют описывать связность элементов данных независимо от вида отношения, в том числе комбинации множеств, таблиц и графов. К сетевым моделям, например, относят модель «сущность-связь» (ER - Entity-Relationship), обычно используемую при разработке баз данных.

Диаграммы Джексона. В основе *диаграмм Джексона* лежит предположение о том, что структуры данных, так же, как и программ, можно строить из элементов с использованием всего трех основных конструкций: последовательности, выбора и повторения.

Каждая конструкция представляется в виде двухуровневой иерархии, на верхнем уровне которой расположен блок конструкции, а на нижнем - блоки элементов. Нотации конструкций различаются специальными символами в правом верхнем углу блоков элементов. В изображении последовательности дополнительный символ отсутствует. В изображении выбора ставится символ «о» (латинское) - сокращение английского «или» (or). Конструкции последовательности и выбора должны содержать по два или более элементов второго уровня. В изображении повторения в блоке единственного (повторяющегося) элемента ставится символ «*».

Так схема, показанная на рис. 4.22, а, означает, что конструкция А состоит из элементов В, С и D, следующих в указанном порядке. Схема на рис. 4.22, б означает, что конструкция S состоит либо из элемента Р, либо из элемента Q, либо из элемента R. Схема, изображенная на рис. 4.22, в, показывает, что конструкция I может не содержать элементов или содержать один или более элементов X.

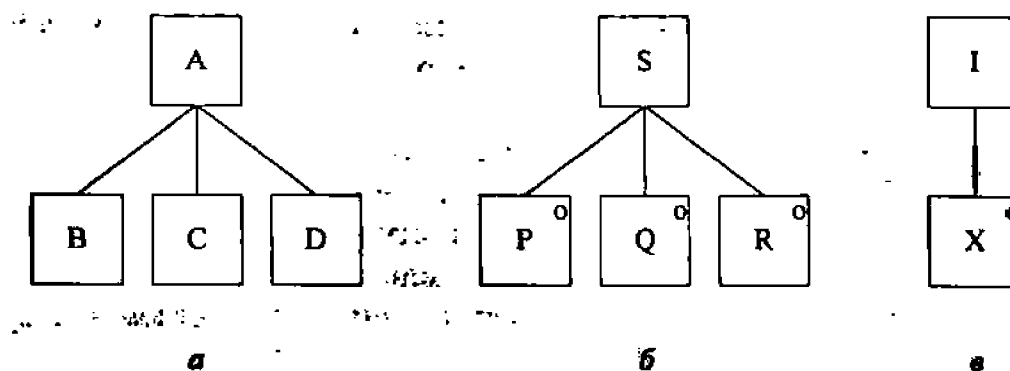


Рис. 4.22. Нотация Джексона для представления конструкций:

а – последовательность; б – выбор; в – повторение

В случае если необходимо показать, что конструкция повторения должна включать один или более элементов, используют комбинацию из двух структур последовательности и повторения (рис. 4.23).

Скобочные диаграммы Орра. *Диаграмма Орра* базируется на том же предположении о сходстве структур программ и данных, что и диаграмма Джексона. Отличие состоит лишь в нотации. Автор предлагает для представления конструкций данных использовать фигурные скобки (рис. 4.24).



Рис. 4.23. Пример описания конструкции, в которой повторение встречается один или более раз

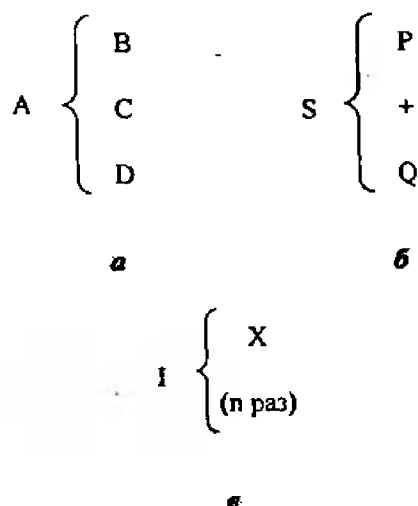


Рис. 4.24. Скобочная нотация для представления структур данных Орра:
 а – последовательность; б – выбор;
 в – повторение

Пример 4.6. Рассмотрим описание структуры данных файла «Электронная ведомость», содержащего сведения о сдаче экзаменов студентами. Файл состоит из записей о результатах сдачи сессии студентами одной группы. Он имеет следующую структуру: номер группы, записи об успеваемости студентов (ФИО студента, название предмета и оценка, полученная студентом, в завершении записи специальный символ «конец записи») и специальный символ «конец файла». На рис. 4.25 показано, как выглядит описание данной структуры с использованием диаграммы Джексона, а на рис. 4.26 - с использованием скобок Орра.

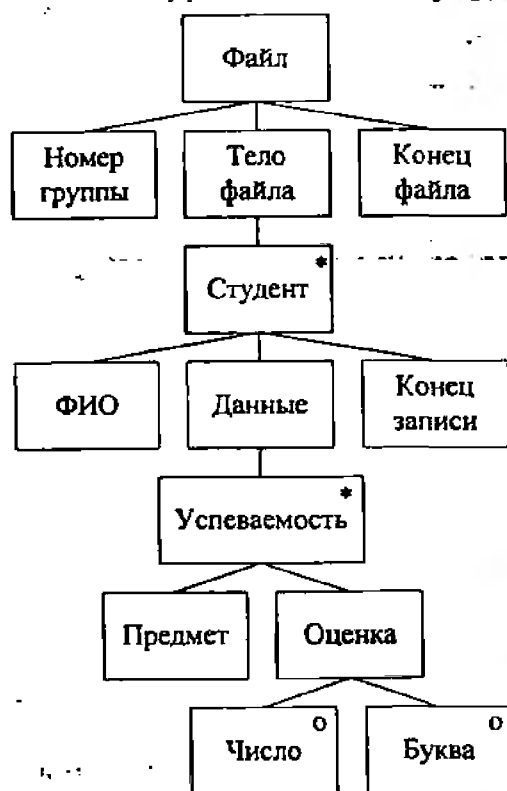


Рис. 4.25. Описание структуры файла «Электронная ведомость» в виде диаграммы Джексона

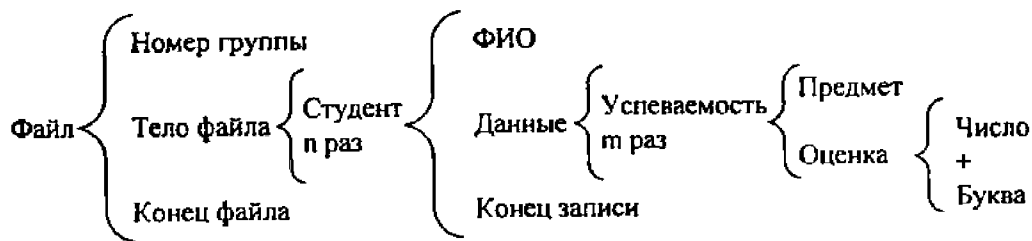


Рис. 4.26. Описание структуры файла «Электронная ведомость» в виде скобочной диаграммы Орра

Сетевая модель данных. Сетевые модели данных используют в тех случаях, если отношение между компонентами данных не исчерпывается включением. Для графического представления разновидностей этой модели используют несколько нотаций. Наиболее известны из них следующие:

- нотация П. Чена;
- нотация Р. Баркера;
- нотация IDEF1 (более современный вариант этой нотации - IDEF1X используется в CASE-системах, например в системе ERWin).

Нотация Баркера является наиболее распространенной. Далее в настоящем разделе будем придерживаться именно этой нотации.

Базовыми понятиями сетевой модели данных являются: сущность, атрибут и связь.

Сущность — реальный или воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Каждая сущность должна:

- иметь уникальное имя;
- обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;
- обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Сущность представляет собой множество экземпляров реальных или абстрактных объектов (людей, событий, состояний, предметов и т. п.). Имя сущности должно отражать тип или класс объекта, а не его конкретный экземпляр (Аэропорт, а не Внуково).

На диаграмме в нотации Баркера сущность изображается прямоугольником, иногда с закругленными углами (рис. 4.27, а).

Каждая сущность обладает одним или несколькими атрибутами. *Атрибут* - любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности (рис. 4.27, б).

В сетевой модели атрибуты ассоциируются с конкретными сущностями, и, соответственно, экземпляр сущности должен обладать единственным определенным значением для ассоциированного атрибута. Атрибут, таким образом, представляет собой некоторый тип характеристик или свойств, ассоциированных с множеством реальных или абстрактных объектов. Экземпляр атрибута - определенная характеристика конкретного экземпляра сущности. Он определяется типом характеристики и ее значением, называемым значением атрибута.

Атрибуты делятся на *ключевые*, т. е. входящие в состав уникального идентификатора, который называют первичным ключом, и *описательные* - прочие.

Первичный ключ - это атрибут или совокупность атрибутов и/или связей, предназначенная для уникальной идентификации каждого экземпляра сущности (совокупность признаков, позволяющих идентифицировать объект). Ключевые атрибуты помещают в начало списка и помечают символом «#» (рис. 4.27, в).

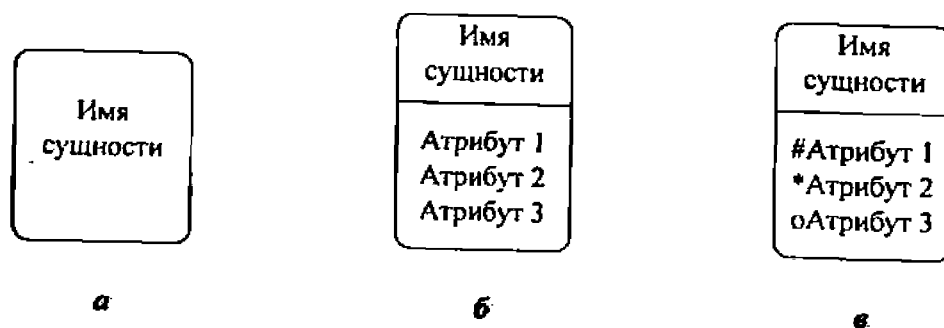


Рис. 4.27. Обозначения сущности в нотации Баркера:

а – без атрибутов; *б* – с указанием атрибутов; *в* – с уточнением атрибутов и их типов
(# – ключевой, * – обязательный, о – необязательный)

Описательные атрибуты бывают обязательными или необязательными. Обязательные атрибуты для каждой сущности всегда имеют конкретное значение, необязательные - могут быть не определены. Обязательные и необязательные описательные атрибуты помечают символами «*» и «о» соответственно.

Для сущностей определено понятие супертип и подтип. *Супертип* - сущность обобщающая некую группу сущностей (*подтипов*). Супертип характеризуется общими для подтипов атрибутами и отношениями. Например, для некоторых задач супертип «учащийся» обобщает подтипы «школьник» и «студент» (рис. 4.28).

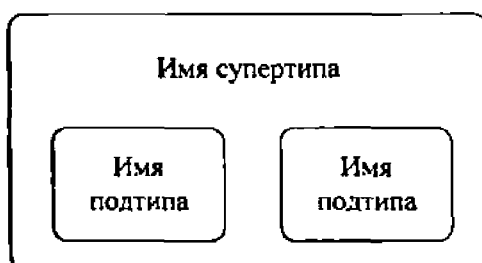


Рис. 4.28. Обозначение супертипов и подтипов в нотации Баркера

Связь - поименованная ассоциация между двумя или более сущностями, значимая для рассматриваемой предметной области. Связь, таким образом, означает, что каждый экземпляр одной сущности ассоциирован с произвольным (в том числе и нулевым) количеством экземпляров второй сущности и наоборот. Если любой экземпляр одной сущности связан хотя бы с одним экземпляром другой сущности, то связь является обязательной (рис. 4.29, а). Необязательная связь представляет собой условное отношение между сущностями (рис. 4.29, б).

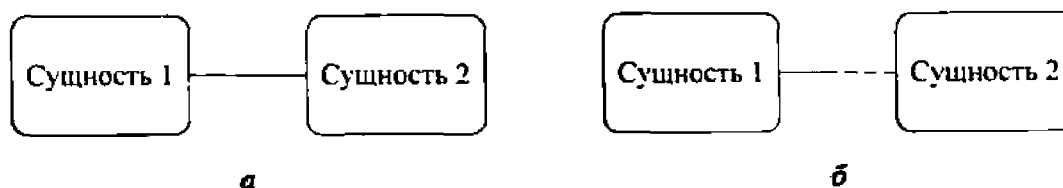


Рис. 4.29. Обозначение связи в нотации Баркера:

а – обязательная; *б* – необязательная (пунктир показывают до середины линии связи)

Каждая сущность может быть связана любым количеством связей с другими сущностями модели. Связь предполагает некоторое отношение сущностей, которое характеризуется количеством экземпляров сущности, участвующих в связи с каждой стороны.

Различают три типа отношений (рис. 4.30):

1*1 - «один-к-одному» - одному экземпляру первой сущности соответствует один экземпляр второй;

1*n - «один-ко-многим» - одному экземпляру первой сущности соответствуют несколько экземпляров второй;

n*m - «многие-ко-многим» -> каждому экземпляру первой сущности может соответствовать несколько экземпляров второй и, наоборот, каждому экземпляру второй сущности может соответствовать несколько экземпляров первой.

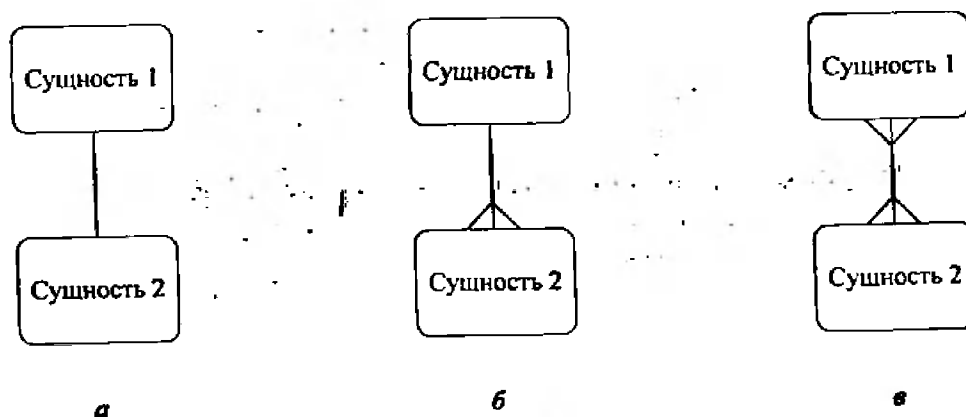


Рис. 4.30. Обозначение отношений в нотации Баркера:
а - «один-к-одному»; б - «один-ко-многим»; в - «многие-ко-многим»

Кроме того, сущности бывают независимыми, зависимыми и ассоциированными. *Независимая* сущность представляет независимые данные, которые всегда присутствуют в системе. Они могут быть связаны или не связаны с другими сущностями той же системы.

Зависимая сущность представляет данные, зависящие от других сущностей системы, поэтому она всегда должна быть связана с другими сущностями.

Ассоциированная сущность представляет данные, которые ассоциируются с отношениями между двумя и более сущностями. Обычно данный вид сущностей используется в модели для разрешения отношения «многие-ко-многим» (рис. 4.31).



Рис. 4.31. Обозначение ассоциированной сущности в нотации Баркера

Если экземпляр сущности полностью идентифицируется своими ключевыми атрибутами, то говорят о *полной идентификации сущности*. В противном случае идентификация сущности осуществляется с использованием атрибутов связанной сущности, что указывается черточкой на линии связи (рис. 4.32).

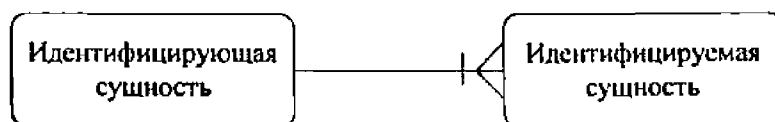


Рис. 4.32. Обозначение идентификации посредством другой сущности в нотации Баркера

Кроме этого, модель включает понятия взаимно исключающих, рекурсивных и непеременяемых связей. При наличии взаимно исключающей связи экземпляр сущности участвует только в одной связи из некоторой группы связей (рис. 4.33, а). Рекурсивная связь предполагает, что сущность может быть связана сама с собой (рис. 4.33, б). Непеременяемая связь означает, что экземпляр сущности не может быть перенесен из одного экземпляра связи в другой (рис. 4.33, в).

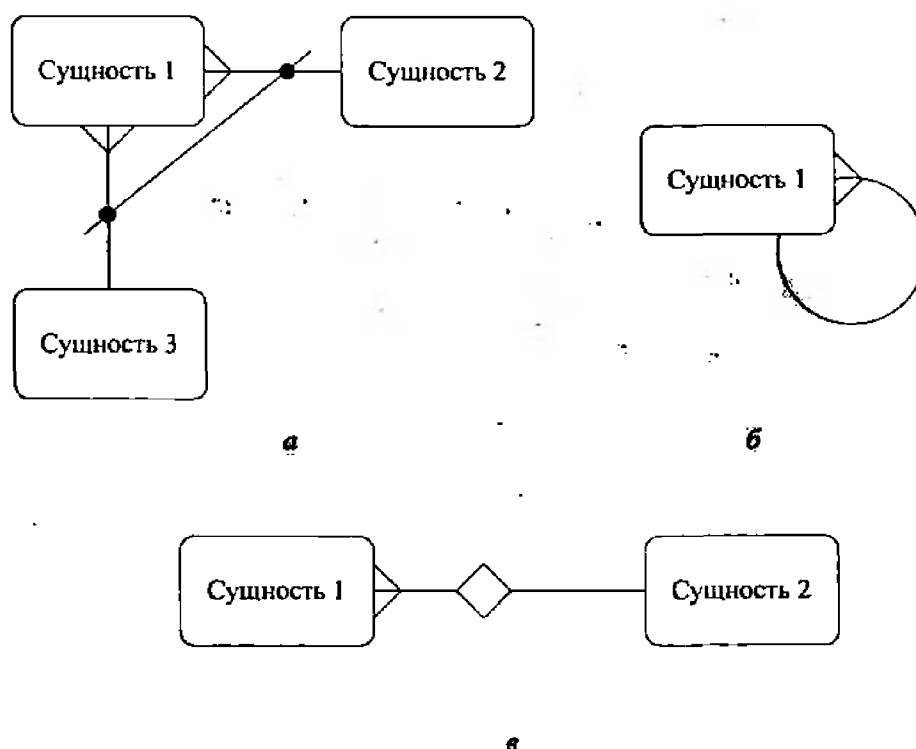


Рис. 4.33. Обозначение особых видов связи в нотации Баркера:
а – взаимно исключающая; б – рекурсивная; в – непеременяемая

Пример 4.7. Рассмотрим структуру базы данных для системы учета успеваемости студентов. Основными сущностями для решения указанной задачи являются: Студент и Предмет (изучаемый учебный курс).

Отношение между ними относится к типу «многие-ко-многим». Для разрешения этого отношения введем ассоциированную сущность Экзамен/Зачет, которая отражает текущее выполнение предметов учебного плана студентом.

Предметы, которые изучает и по которым отчитывается студент, запланированы кафедрой в учебном плане. Учебный план включает список предметов каждого семестра (сущность Семестр).

Для получения справок различного рода потребуются сущности, определяющие структуру организации:

- Факультет;
- Курс - совокупность студентов, поступивших в институт в одном году;
- Кафедра;

- Группа.

Для определения момента времени, начиная с которого отсутствие положительных результатов сдачи экзамена следует считать задолженностью, необходимо хранить даты экзаменов для каждой группы (сущность Дата экзамена). На рис. 4.34 показаны основные отношения между указанными сущностями.

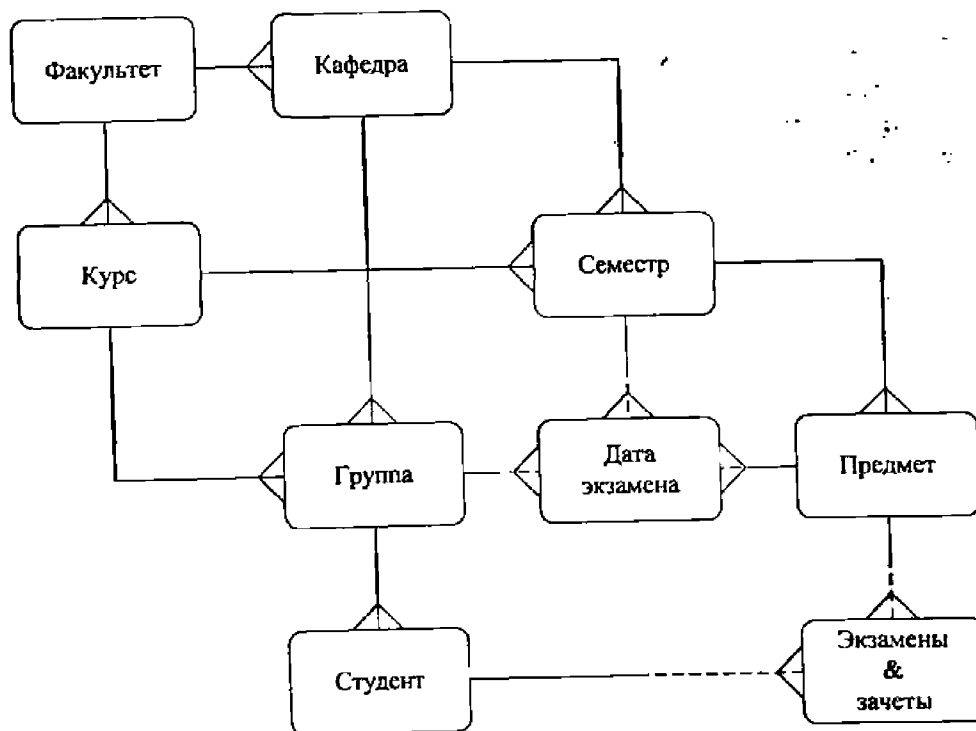


Рис. 4.34. Диаграмма «сущность-связь» для описания базы данных системы учета успеваемости студентов

На следующем шаге определяем атрибуты каждой сущности и уточняем их типы (атрибуты, используемые для дополнительной идентификации сущности другой сущностью, не указаны, так как они описаны в соответствующей сущности).

Факультет:

- DepID - уникальное имя факультета (ключевое поле);
- DepName - название факультета.

Курс:

- CursID - уникальное имя кафедры (ключевое поле);
- EnterYear - год начала обучения для большинства студентов курса.

Кафедра:

- SpecID - уникальное имя кафедры (ключевое поле);
- SpecName - название кафедры.

Семестр:

- SemestrID - уникальное имя семестра обучения на конкретной кафедре (ключевое поле);
- SemName - название семестра обучения на кафедре.

Группа:

- GroupID - уникальное имя группы (ключевое поле);
- GroupName - название группы.

Предмет:

- SubjectID - уникальное имя предмета (ключевой атрибут);
- SubjectName - название предмета;
- ExamKind - вид оценки знаний (необязательный атрибут):
экзамен/зачет/экзамен+зачет.

Дата экзамена:

- Date - дата экзамена;
- AudNumber - номер аудитории.

Студент:

- StudentID - уникальное имя студента (ключевое поле);
- Name - фамилия;
- FirstName - имя;
- SecondName - отчество;
- StEnterYear - год поступления в институт.

Экзамен/Зачет:

- Date - дата сдачи экзамена или зачета;
- ExamType - тип (экзамен или зачет);
- Mark - оценка.

Полученная диаграмма «сущность-связь» приведена на рис. 4.35.

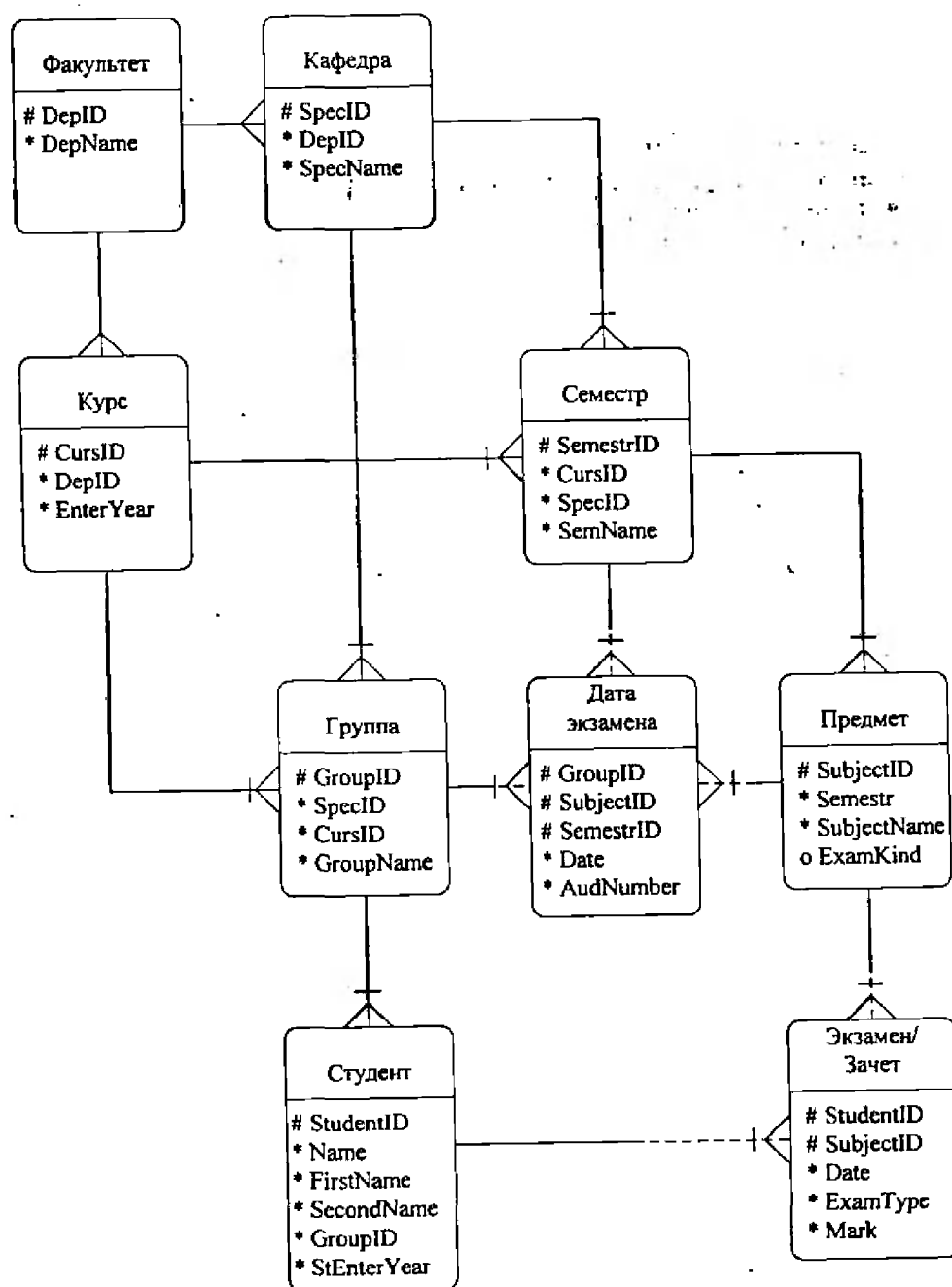


Рис. 4.35. Диаграмма «сущность-связь» для описания базы данных системы учета успеваемости студентов

Данная диаграмма должна быть проверена с точки зрения возможности получения всех справок, указанных в техническом задании или показанных на диаграмме потоков данных разрабатываемой системы (см. рис. 4.14).

4.6. Математические модели задач, разработка или выбор методов решения

Для задач, алгоритм решения которых не очевиден, используют разного рода математические модели. Процесс построения такой модели включает:

- анализ условия задачи;
- выбор математических абстракций, *адекватно*, т. е. с требуемой *точностью* и *полнотой* представляющих исходные данные и результаты;
- формальную постановку задачи;
- определение метода преобразования исходных данных в результат, т. е. *метода решения задачи*.

Для многих задач, которые часто встречаются на практике, в математике определены как модели, так и методы решения. К таким задачам, например, относится большинство задач аналитической геометрии на плоскости и в пространстве, задачи моделирования дискретных систем и т. д.

Основная проблема в подобных случаях - *обоснование применимости* той или иной математической модели для решения конкретной задачи.

В ряде случаев формальная постановка задачи однозначно определяет метод ее решения, но, как правило, методов решения существует несколько, и тогда для выбора метода решения может потребоваться специальное исследование. При выборе метода учитывают:

- особенности данных конкретной задачи, связанные с предметной областью (погрешность, возможные особые случаи и т. п.);
- требования к результатам (допустимую погрешность);
- характеристики метода (точный или приближенный, погрешности результатов, вычислительную и емкостную сложности, сложность реализации и т. п.).

Пример 4.8. Выполнить формальную постановку задачи поиска цикла минимальной длины (задачи коммивояжера).

Вспомним, что задача коммивояжера или поиска цикла минимальной длины в простейшем варианте формулируется следующим образом. Задан список городов и дорог, соединяющих данные города. Известны расстояния между городами. Необходимо объехать все города, не заезжая ни в какой город дважды, и вернуться в исходный город так, чтобы суммарная длина пути была минимальной.

Анализ условия задачи показывает, что математической моделью объектов системы и существующих или возможных связей между ними может являться взвешенный ориентированный или неориентированный граф $G(X, <U, L>)$, где X , U , L - множества вершин, ребер и весов ребер соответственно.

Для перехода от объектов задачи к их математическим моделям необходимо [55]:

- сформулировать правила соответствия компонентов объекта компонентам модели;
- определить вид этих соответствий (взаимно однозначные, однозначные, многозначные);
- определить способ отображения свойств и характеристик компонентов объекта в характеристики выбранной математической абстракции.

Все это определяется, исходя из отношений, существующих между компонентами объекта, а также свойств объекта и характеристик его компонентов.

В графе G множество \mathcal{O} объектов системы (городов) поставлено во взаимно однозначное соответствие множеству X , а множеству связей C между парами объектов (дорог) поставлено в такое же соответствие множество ребер U . Расстояние между городами $(\varepsilon_i, \varepsilon_j)$ интерпретируется

как вес соответствующего ребра $w(x_i, x_j)$

Таким образом, в терминах теории графов рассматриваемая задача - это поиск в ориентированном или неориентированном графе гамильтонова цикла минимальной длины.

Формальная постановка задачи имеет вид - выполнить преобразование исходного графа G в граф результата G_c :

$$G(X, \langle U, W \rangle) \xrightarrow{D} G_c(X, U_c)$$

так что

$$U_c \subset U, |U_c| = |X_c| : W(G_c) = \sum_{u_r \in U_c} w(u_r) \rightarrow \min$$

причем

$$(\forall x_i \in X) p(x_i) = 2, (\forall x_i, x_j \in X) \exists S(x_i, x_j),$$

где $p(x_i)$ - количество ребер, подходящих к вершине; а $S(x_i, x_j)$ - маршрут между вершинами x_i, x_j , т. е. последовательность смежных ребер, связывающих эти вершины.

Следует иметь в виду, что граф имеет гамильтонов цикл, если сумма локальных степеней любой пары вершин больше или равна числу его вершин:

$$(\forall x_i, x_j \in X) [p(x_i) + p(x_j)] \geq n.$$

В противном случае граф может не иметь *гамильтонова цикла*, что для нас означает, что в некоторых случаях *задача может не иметь решения*.

Задача относится к классу NP-сложных - задач, вычислительная сложность которых не выражается в виде полинома от размерности ее входа (количества городов), а носит экспоненциальный характер, т. е. очень быстро возрастает при увеличении размерности задачи. Известно, что точное решение задачи коммивояжера может быть получено алгоритмами, реализующими полный перебор или метод ветвей и границ. Приближенное решение дают методы поиска в глубину и двоичной свертки.

Поскольку по техническому заданию необходимо обеспечить получение точного решения, следует реализовать хотя бы по одному методу из указанных групп. Для выбора методов нужно провести дополнительные исследования.

Определив методы решения, целесообразно для некоторых вариантов исходных данных вручную, на калькуляторе или с использованием других средств подсчитать ожидаемые результаты. Эти данные в дальнейшем будут использованы при тестировании программного обеспечения. Кроме того, выполнение операций вручную позволяет точно уяснить последовательность действий, что упростит разработку алгоритмов.

Кроме того, имеет смысл продумать, для каких сочетаний исходных данных результат не существует или не может быть получен данным методом, что тоже необходимо учесть при разработке программного обеспечения.

Контрольные вопросы и задания

1. В чем сущность структурного подхода к программированию? Какие этапы охватывает данный подход?

2. Что понимают под термином «спецификации»? В чем сложность их уточнения? Назовите модели, используемые в качестве функциональных спецификаций при структурном подходе. Какие характеристики проектируемого программного обеспечения описывает каждая из них?

3. В каких случаях целесообразно использовать диаграммы переходов состояний? Разработайте диаграмму переходов для калькулятора, техническое задание на который составлялось вами в соответствии заданием к предыдущей главе.

4. В чем заключается основное различие между функциональными диаграммами и диаграммами потоков данных? Постройте оба вида диаграмм для выполнения вычислений с использованием внутренней памяти калькулятора. Проанализируйте сходство и различие. В каких случаях использование диаграмм потоков данных является предпочтительным?

5. Что называют «структурами данных»? Какие данные имеются в виду? В каких случаях структуры данных необходимо описывать? Какие модели используют для описания структур данных?

6. Опишите стек и очередь с использованием предлагаемых моделей описания данных. Какие аспекты этих структур остались не описанными и почему?

7. В каких случаях используют математические модели? Что понимают под адекватностью модели? Зачем необходимо выполнять доказательство адекватности и как строятся подобные доказательства?

5. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ СТРУКТУРНОМ ПОДХОДЕ

Как уже упоминалось ранее, сущность структурного подхода заключается в декомпозиции программы или программной системы по функциональному принципу. Все предлагаемые методы декомпозиции используют интерфейсы простейшего типа: примитивные интерфейсы и традиционные меню, и рассчитаны на анализ и проектирование как структур данных, так и обрабатывающих их программ.

Причем в большинстве случаев первичным считают проектирование обрабатывающих компонентов, проектирование же структур данных выполняют параллельно. Существует и альтернативный подход, при котором первичным считают проектирование данных, а обрабатывающие программы получают, анализируя полученные структуры данных.

В любом случае проектирование программного обеспечения начинают с определения его структуры.

5.1. Разработка структурной и функциональной схем

Процесс проектирования сложного программного обеспечения начинают с уточнения его структуры, т. е. определения структурных компонентов и связей между ними. Результат уточнения структуры может быть представлен в виде структурной и/или функциональной схем и описания (спецификаций) компонентов.

Структурная схема разрабатываемого программного обеспечения. Структурной называют схему, отражающую *состав и взаимодействие по управлению* частей разрабатываемого программного обеспечения.

Структурные схемы пакетов программ не информативны, поскольку организация программ в пакеты не предусматривает передачи управления между ними. Поэтому структурные схемы разрабатывают для каждой программы пакета, а список программ пакета определяют, анализируя функции, указанные в техническом задании.

Самый простой вид программного обеспечения - программа, которая в качестве структурных компонентов может включать только подпрограммы и библиотеки ресурсов. Разработку структурной схемы программы обычно выполняют методом пошаговой детализации (см. § 5.2).

Структурными компонентами программной системы или программного комплекса могут служить программы, подсистемы, базы данных, библиотеки ресурсов и т. п.

Структурная схема программного комплекса демонстрирует передачу управления от программы-диспетчера соответствующей программе (рис. 5.1).

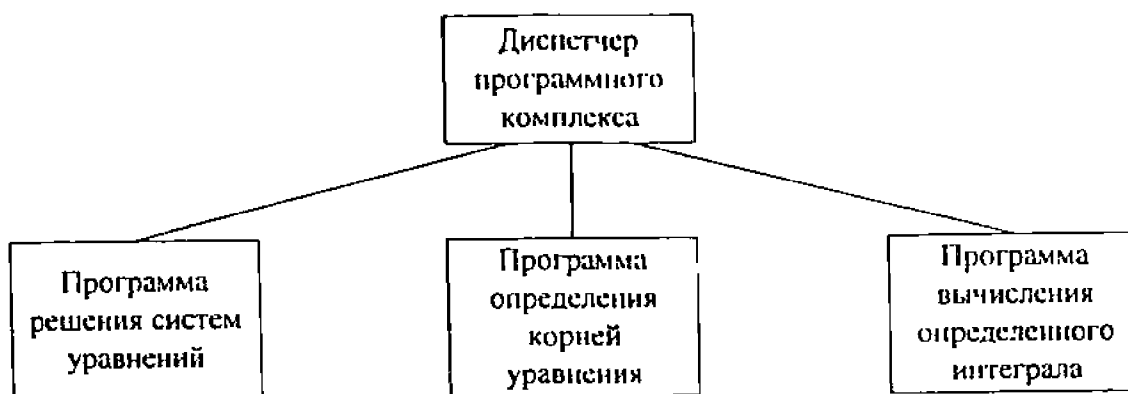


Рис. 5.1. Пример структурной схемы программного комплекса

Структурная схема программной системы, как правило, показывает наличие подсистем или других структурных компонентов. В отличие от программного комплекса отдельные части (подсистемы) программной системы интенсивно обмениваются данными между собой и, возможно, с основной программой. Структурная же схема программной системы этого обычно не показывает (рис. 5.2).

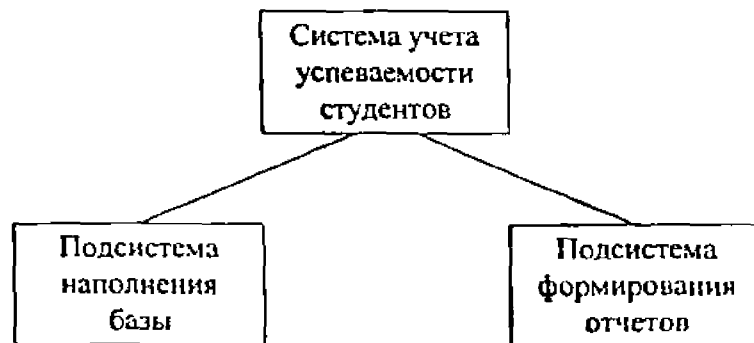


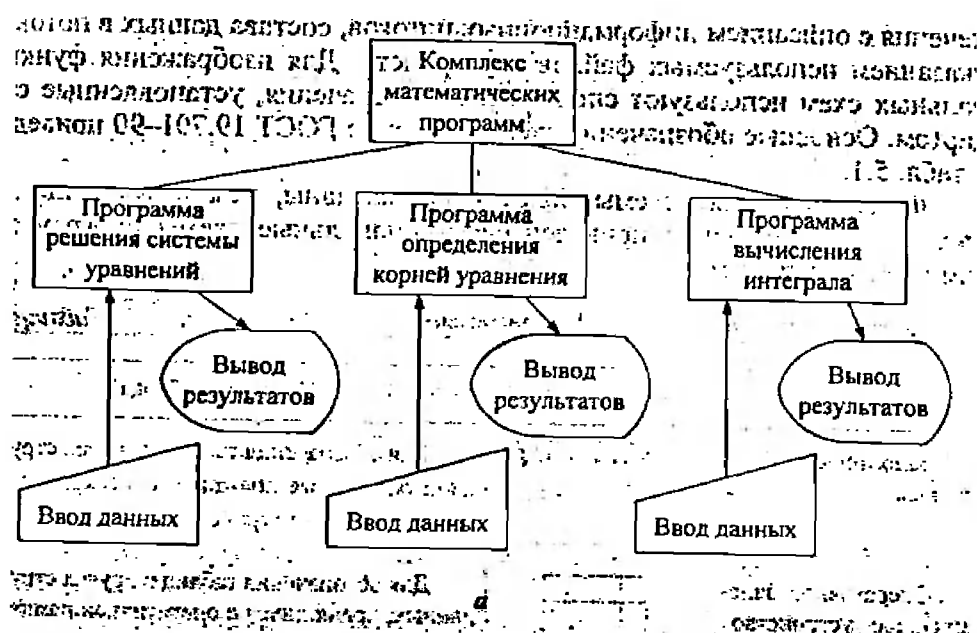
Рис. 5.2. Пример структурной схемы программной системы

Более полное представление о проектируемом программном обеспечении с точки зрения взаимодействия его компонентов между собой и с внешней средой дает функциональная схема.

Функциональная схема. Функциональная схема или схема данных (ГОСТ 19.701-90) - схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств. Для изображения функциональных схем используют специальные обозначения, установленные стандартом. Основные обозначения схем данных по ГОСТ 19.701-90 приведены в табл. 5.1.

Функциональные схемы, более информативны, чем структурные. На рис. 5.3 для сравнения приведены функциональные схемы программных комплексов и систем.

Все компоненты структурных и функциональных схем должны быть описаны. При структурном подходе особенно тщательно необходимо прорабатывать спецификации межпрограммных интерфейсов, так как от качества их описания зависит количество самых дорогостоящих ошибок. К самым дорогим относятся ошибки, обнаруживаемые при комплексном тестировании, так как для их устранения могут потребоваться серьезные изменения уже отлаженных текстов.





б

Рис. 5.3. Примеры функциональных схем:

а – комплекс программ; б – программная система

Таблица 5.1

Название блока	Обозначение	Назначение блока
Запоминаемые 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 последовательной выборкой		Для обозначения таблиц и других структур данных, хранящихся на устройствах с последовательной выборкой (магнитной ленте и т.п.)
Запоминающее устройство с прямым доступом		Для обозначения таблиц и других структур данных, хранящихся на устройствах с прямым доступом (дисках)
Документ		Для обозначения таблиц и других структур данных, выводимых на печатающее устройство
Ручной ввод		Для обозначения ручного ввода данных с клавиатуры
Карта		Для обозначения данных на магнитных или перфорированных картах
Дисплей		Для обозначения данных, выводимых на дисплей компьютера

5.2. Использование метода пошаговой детализации для проектирования структуры программного обеспечения

Структурный подход к программированию в том виде, в котором он был сформулирован в 70-х годах XX в., предлагал осуществлять декомпозицию программ методом пошаговой детализации. Результатом декомпозиции является структурная схема программы, которая представляет собой многоуровневую иерархическую схему взаимодействия подпрограмм по управлению. Минимально такая схема отображает два уровня иерархии, т. е. показывает общую структуру программы. Однако тот же метод позволяет получить структурные схемы с большим количеством уровней.

Метод пошаговой детализации (см. § 1.3) реализует нисходящий подход (см. § 2.3) и базируется на основных конструкциях структурного программирования (см. § 2.4). Он предполагает пошаговую разработку алгоритма. Каждый шаг при этом включает разложение функции на подфункции. Так на первом этапе описывают решение поставленной задачи, выделяя общие подзадачи, на следующем аналогично описывают решение подзадач, формулируя при этом подзадачи следующего уровня. Таким образом, на каждом шаге происходит уточнение функций проектируемого программного обеспечения. Процесс продолжают, пока не доходят до подзадач, алгоритмы решения которых очевидны.

Декомпозируя программу методом пошаговой детализации, следует придерживаться основного правила структурной декомпозиции, следующего из принципа вертикального управления: в первую очередь детализировать управляющие процессы декомпозируемого компонента, оставляя уточнение операций с данными напоследок. Это связано с тем, что приоритетная детализация управляющих процессов существенно упрощает структуру компонентов всех уровней иерархии и позволяет не отделять процесс принятия решения от его выполнения: так, определив условие выбора некоторой альтернативы, - сразу же вызывают модуль, ее реализующий.

Детализация операций со структурами в последнюю очередь позволит отложить уточнение их спецификаций и обеспечит возможность относительно безболезненной модификации этих структур за счет сокращения количества модулей, зависящих от этих данных.

Кроме этого, целесообразно. Придерживаться следующих рекомендаций:

- не отделять операции инициализации и завершения от соответствующей обработки, так как модули инициализации и завершения имеют плохую связность (временную) и сильное сцепление (по управлению);
- не проектировать слишком специализированных или слишком универсальных модулей, так как проектирование излишне специальных модулей увеличивает их количество, а проектирование излишне универсальных модулей повышает их сложность;
- избегать дублирования действий в различных модулях, так как при их изменении исправления придется вносить во все фрагменты программы, где они выполняются - в этом случае целесообразно просто реализовать эти действия в отдельном модуле;
- группировать сообщения об ошибках в один модуль по типу библиотеки ресурсов, тогда будет легче согласовать формулировки, избежать дублирования сообщений, а также перевести сообщения на другой язык.

При этом, описывая решение каждой задачи, желательно использовать не более 1—2-х структурных управляющих конструкций, таких, как цикл-пока или ветвление, что позволяет четче представить себе структуру организуемого вычислительного процесса.

Пример 5.1. Разработать алгоритм программы построения графиков функций одной переменной на заданном интервале изменения аргумента $[x_1, x_2]$ при условии непрерывности функции на всем интервале определения.

Примечание. Для того чтобы программировать построение графиков функций с точками разрыва первого и второго рода, необходимо аналитически исследовать заданные функции, что

представляет собой отдельную и достаточно сложную задачу. Численными методами данная задача не решается.

В общем виде задача построения графика функции ставится как задача отображения реального графика (рис. 5.4, а), выполненного в некотором масштабе, в соответствующее изображение в окне на экране (рис. 5.4, б).

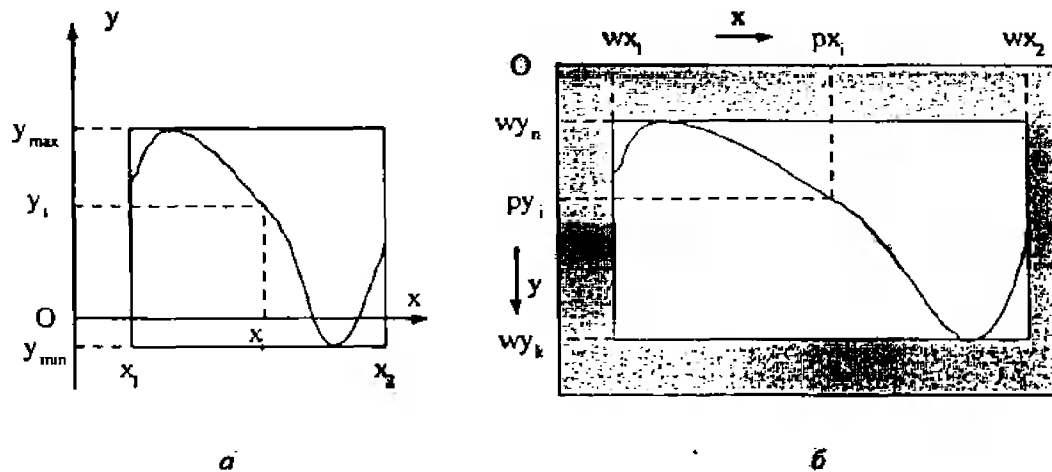


Рис. 5.4. Задача построения графика функции:
а – график функции; б – его отображение в окне на экране

Для построения графика необходимо определить масштабы по осям координат:

$$m_x = \frac{wx_2 - wx_1}{x_2 - x_1}, \quad m_y = \frac{wy_2 - wy_1}{y_{\max} - y_{\min}}$$

и координаты точек графика:

$$px_i = \lceil (x_i - x_1) * m_x \rceil + wx_1,$$

$$py_i = \lceil (y_{\max} - y_i) * m_y \rceil + wy_1.$$

Шаг координатной сетки по вертикали и горизонтали при этом можно определить по формулам:

$$Ipm_x = \frac{wx_2 - wx_1}{nI_x}, \quad Ipm_y = \frac{wy_2 - wy_1}{nI_y},$$

где nI_x, nI_y - соответственно количество вертикальных и горизонтальных линий.

Для разметки сетки необходимо определить шаги разметки по горизонтали и вертикали:

$$Im_x = \frac{x_2 - x_1}{nI_x}, \quad Im_y = \frac{y_{\max} - y_{\min}}{nI_y}.$$

Таким образом, для того чтобы построить график, необходимо задать функцию, интервал изменения аргумента $[x_1, x_2]$, на котором функция непрерывна, количество точек графика n , размер и положение окна экрана, в котором необходимо построить график: $w_{x_1}, w_{y_1}, w_{x_2}, w_{y_2}$ и количество линий сетки по горизонтали и вертикали n_{lx}, n_{ly} . Значения $w_{x_1}, w_{y_1}, w_{x_2}, w_{y_2}, n_{lx}, n_{ly}$ можно задать, исходя из размера экрана, а интервал и число точек графика надо вводить.

Разработку алгоритма выполняем методом пошаговой детализации, используя для записи псевдокод.

Примем, что программа будет взаимодействовать с пользователем через традиционное иерархическое меню, которое содержит пункты: Функция, Отрезок, Шаг, Вид результата, Выполнить и Выход (см. рис. 8.5). Для каждого пункта этого меню необходимо реализовать сценарий, предусмотренный в техническом задании.

Шаг 1. Определяем структуру управляющей программы, которая для нашего случая реализует работу с меню через клавиатуру:

Программа.

Инициализировать глобальные значения

Вывести заголовок и меню

Выполнять

Если выбрана Команда

то Выполнить Команду

иначе Обработать нажатие клавиши управления

Все-если

до Команда=Выход

Конец.

Очистка экрана, вывод заголовка и меню, а также выбор Команды - операции сравнительно простые, следовательно, их можно не детализировать.

Шаг 2. Детализируем операцию Выполнить команду:

Выполнить Команду:

Выбор Команда

Функция:

Ввести или выбрать формулу Fun

Выполнить разбор формулы

Отрезок:

Ввести значения x_1, x_2

Шаг:

Ввести значения h

Вид результата:

Ввести вид_результата

Выполнить:

Рассчитать значения функции

Если Вид_результата=График

то Построить график

иначе Вывести таблицу

Все-если

Все-выбор

Определим, какие фрагменты имеет смысл реализовать в виде подпрограмм. Во-первых, фрагмент Вывод заголовка и меню, так как это достаточно длинная линейная последовательность

операторов и ее выделение в отдельную процедуру позволит сократить управляющую программу. Во-вторых, фрагменты Разбор формулы, Расчет значений функции, Построение графика и Вывод таблицы, так как это достаточно сложные операции. Это - подпрограммы первого уровня, которые определяют структуру программы (рис. 5.5). Определим для этих подпрограмм интерфейсы по данным с основной программой, т. е. списки параметров.

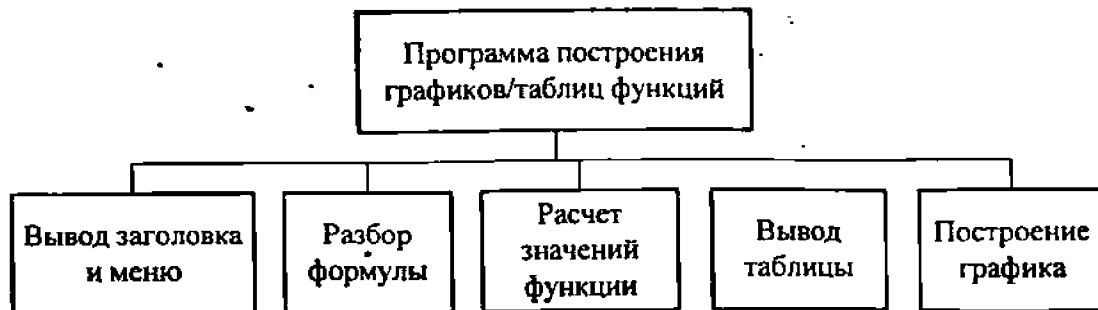


Рис. 5.5. Структурная схема программы построения графиков/таблиц функций.

Подпрограмма Вывод заголовка и меню параметров не предполагает.

Подпрограмма Разбор формулы должна иметь два параметра: Fun - аналитическое задание функции, Tree - возвращаемый параметр - адрес дерева разбора.

Подпрограмма Расчет значений функции должна получать адрес дерева разбора Tree, отрезок: значения x_1 и x_2 , а также шаг h . Обратно в программу она должна возвращать таблицу значений функции $X(n)$ и $Y(n)$, где n - количество точек функции.

Подпрограммы Вывода таблицы и Построения графика должны получать таблицу значений функции и количество точек.

После уточнения имен переменных алгоритм основной программы будет выглядеть следующим образом:

Программа.

Вывод заголовка и меню

Выполнять

Если выбрана Команда

то

Выбор Команда

Функция:

Ввести или выбрать формулу Fun

Разбор формулы (Fat; Var Tree)

Отрезок:

Ввести значения x_1, x_2

Шаг:

Ввести значения h

Вид результата:

Ввести вид результата

Выполнить:

Расчет значений функции ($x_1, x_2, h, Tree$; Var X,Y,n)

Если Вид_результата=График
 то Построение графика(X, Y, n)
 иначе Вывод таблицы(X, Y, n)
 Все-если
 Все-выбор
 иначе Обработать нажатие клавиш управления
 Все-если
 до Команда=Выход
 Конец.

На следующих шагах необходимо выполнить детализацию алгоритмов подпрограмм. Детализацию выполняют, пока алгоритм программы не станет полностью понятен. Один из возможных вариантов полной структурной схемы данной программы показан на рис. 5.6.



Рис. 5.6. Полная многоуровневая структурная схема программы построения графиков/таблиц

Как уже упоминалось в § 2.4, использование метода пошаговой детализации обеспечивает высокий уровень технологичности разрабатываемого программного обеспечения, так как он позволяет использовать только структурные способы передачи управления.

Разбиение на модули при данном виде проектирования выполняется эвристически, исходя из рекомендуемых размеров модулей (20-60 строк) и сложности структуры (две-три вложенных управляющих конструкции). В принципе в качестве модуля (подпрограммы) можно реализовать решение подзадач, сформулированных на любом шаге процесса детализации, однако определяющую роль при разбиении программы на модули играют принципы обеспечения технологичности модулей, рассмотренные в § 2.2.

Для анализа технологичности полученной иерархии модулей целесообразно использовать структурные карты Консантайна или Джексона.

5.3. Структурные карты Константайна

На структурной карте отношения между модулями представляют в виде графа, вершинам которого соответствуют модули и общие области данных, а дугам - межмодульные вызовы и обращения к общим областям данных.

Различают четыре типа вершин (рис. 5.7):

- модуль - подпрограмма,
- подсистема - программа,
- библиотека - совокупность подпрограмм, размещенных в отдельном модуле,
- область данных - специальным образом оформленная совокупность данных,, к которой возможно обращение извне.

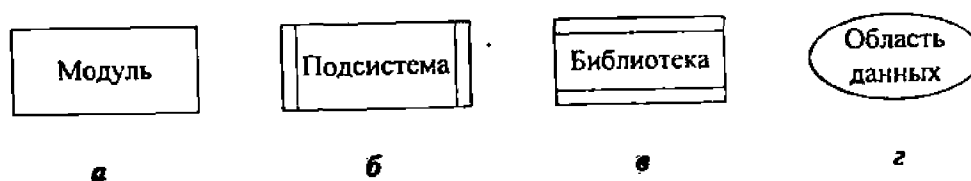


Рис. 5.7. Обозначения вершин по стандартам IBM, ISO и ANSI:

а - модуль; б - подсистема; в - библиотека, г - область данных

При этом отдельные части программной системы (программы, подпрограммы) могут вызываться последовательно, параллельно или как сопрограммы (рис. 5.8).

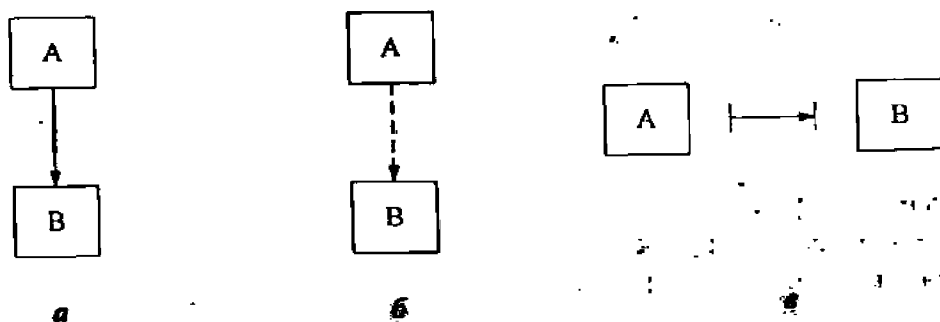


Рис. 5.8. Обозначения типа вызова:

а - последовательный вызов; б - параллельный вызов; в - вызов сопрограммы

Чаще всего используют *последовательный* вызов, при котором модули, передав управление, ожидают завершения выполнения вызванной программы или подпрограммы, чтобы продолжить прерванную обработку.

Под *параллельным* вызовом понимают распараллеливание вычислений на нескольких вычислителях, когда при активизации другого процесса данный процесс продолжает работу (рис. 5.9, а). На однопроцессорных компьютерах в мультипрограммных средах в этом случае начинается попеременное выполнение соответствующих программ. Параллельные процессы бывают синхронные и асинхронные. Для синхронных процессов определяют точки синхронизации - моменты времени, когда производится обмен информацией между процессами. Асинхронные процессы обмениваются информацией только в момент активизации параллельного процесса.

Под *вызовом сопрограммы* понимают возможность поочередного выполнения двух одновременно запущенных программ, например, если одна программа подготовила пакет данных для вывода, то вторая может ее вывести, а затем перейти в состояние ожидания следующего

пакета. Причем в мультипрограммных системах основная программа, передав данные, продолжает работать, а не переходит в состояние ожидания, как изображено на рис. 5.9, б.

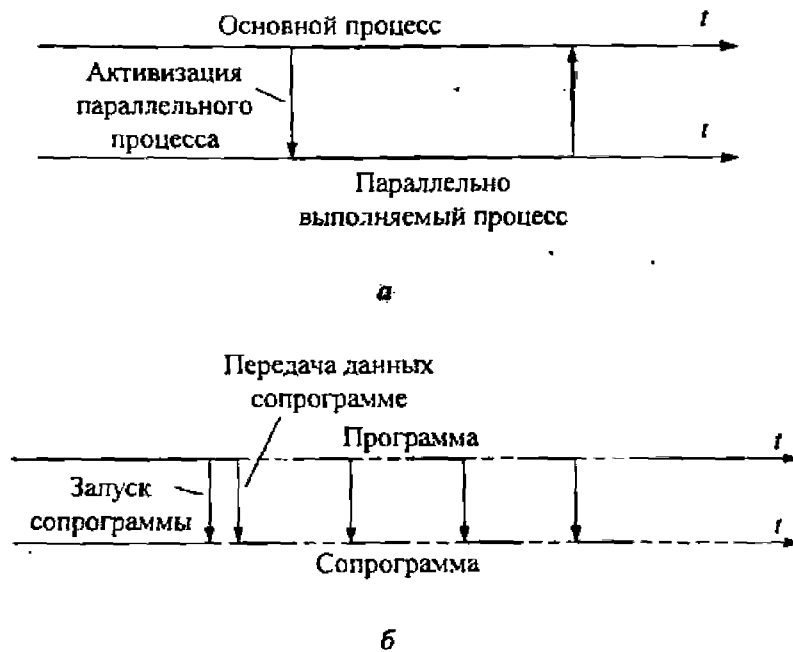


Рис. 5.9. Диаграммы реализации параллельного вызова (а) и вызова сопрограммы (б):

— — выполнение; - - - - ожидание

Если стрелка, изображающая вызов, касается блока, то обращение происходит к модулю целиком, а если входит в блок, то - к элементу внутри модуля.

При необходимости на структурной карте можно уточнить особые условия вызова (рис. 5.10): циклический вызов, условный вызов и однократный вызов - при повторном вызове основного модуля однократно вызываемый модуль не активизируется!

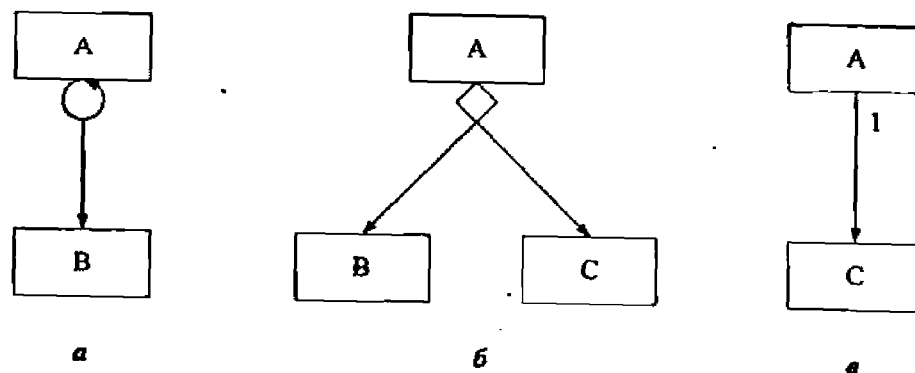


Рис. 5.10. Обозначения особых условий вызова:

а - циклический; б - условный; в - однократный

Связи по данным и управлению обозначают стрелками, параллельными дуге вызова, направление стрелки указывает направление связи (рис. 5.11).

Структурные карты Константайна позволяют наглядно представить результат декомпозиции программы на модули и оценить ее качество, т. е. соответствие рекомендациям структурного программирования (сцепление и связность).

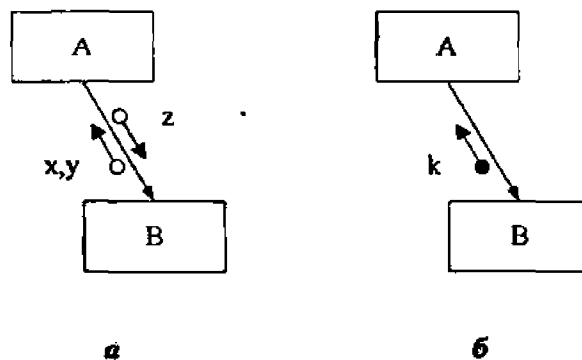


Рис. 5.11. Обозначение типа связи:
a – по данным; *б* – по управлению

Пример 5.3. Представим в виде структурной карты Константайна полную структурную схему, полученную в предыдущем примере (см. рис. 5.6).

Подпрограммы Очистка окна, Вывод прямоугольника, Вывод строки текста, Вывод отрезка прямой, Задание цвета рисования и Задания цвета фона являются частью библиотеки графических примитивов практически в любой среде программирования универсального языка, поэтому их включать в структурную карту не будем.

Для остальных подпрограмм покажем особые условия вызова и типы связей (рис. 5.12).

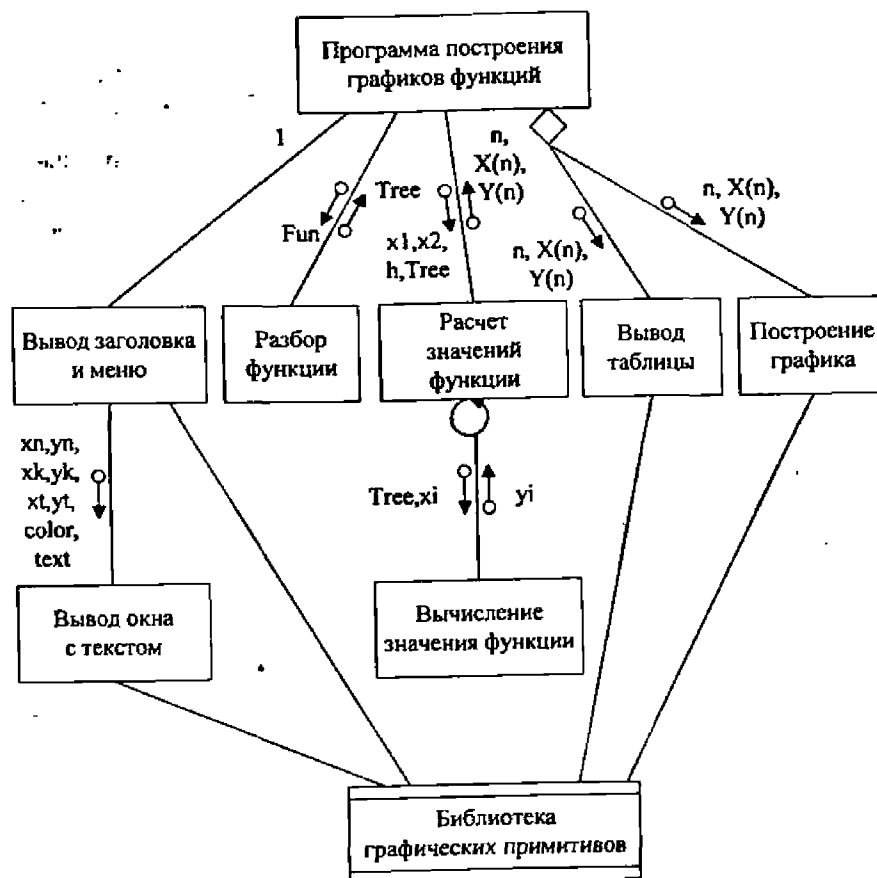


Рис. 5.12. Структурная карта Константайна для программы построения графиков/таблиц функций (вариант 1)

Модули Расчет значений функции, Вывод таблицы и Построение графика связаны с основной программой по образцу, так как параметры X и Y структурные (массивы), следовательно программа считается сцепленной по образцу.

Анализ показывает, что количество сцеплений по образцу в программе можно уменьшить, если подпрограмму Расчет значений функции перенести на следующий уровень (рис. 5.13). Однако в этом случае при смене вида результата таблица значений будет рассчитываться заново.

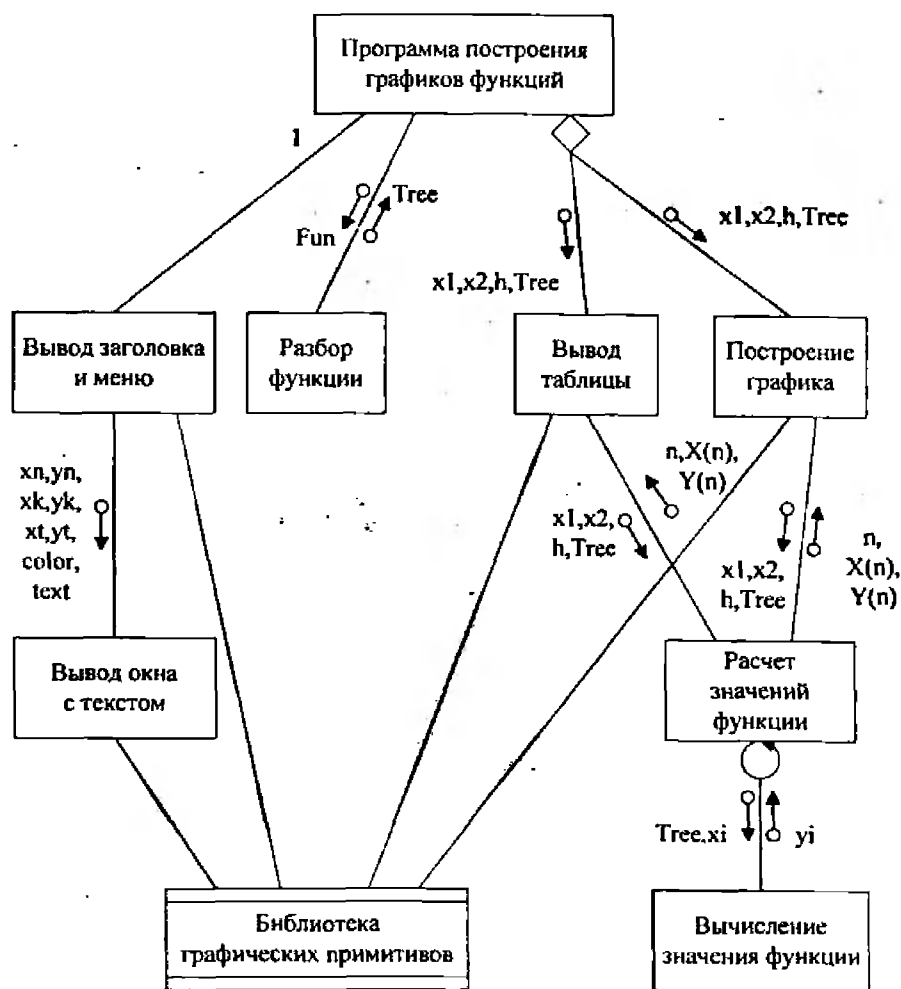


Рис. 5.13. Структурная карта Константайна для программы построения таблицы/графика функции (вариант 2)

Аналогично можно перенести подпрограмму Разбор функции на более низкий уровень и вызывать ее, например, из подпрограммы Расчет значений функции, но поскольку велика вероятность многократного вычисления значений одной функции на разных интервалах, вряд ли это целесообразно.

После внесения соответствующих изменений в алгоритм следует определить полную спецификацию модулей. Спецификация должна включать: имя, краткое описание назначения, перечень входных и выходных параметров с указанием типа и области допустимых входных и выходных значений. Затем можно приступить к реализации модулей.

В соответствии с требованиями нисходящей разработки (комбинированный подход) можно предложить следующий порядок реализации модулей:

- Основная программа,
- Вывод окна с текстом,
- Вывод заголовка и меню,

- Разбор функции,
- Вычисление значений функции,
- Вывод таблицы,
- Расчет значений функции,
- Построение графика.

Структурные карты Джексона будут рассмотрены вместе с предложенной им методикой проектирования программ, основанной на декомпозиции данных в §5.5.

5.4. Проектирование структур данных

Под проектированием структур данных понимают разработку их представлений в памяти. Основными параметрами, которые необходимо учитывать при проектировании структур данных, являются:

- вид хранимой информации каждого элемента данных;
- связи элементов данных и вложенных структур;
- время хранения данных структуры («время жизни»);
- совокупность операции над элементами данных, вложенными структурами и структурами в целом.

Вид хранимой информации определяет тип соответствующего поля памяти. В качестве элементов данных в зависимости от используемого языка программирования могут рассматриваться:

- целые и вещественные числа различных форматов;
- символы;
- булевские значения: true и false;

а также некоторые структурные типы данных, например:

- строки;
- записи;
- специально объявленные классы.

При этом для числовых полей очень важно правильно определить диапазон возможных значений, а для строковых данных - максимально возможную длину строки.

Связи элементов и вложенных структур, а также их *устойчивость* и *совокупность операций* над элементами и вложенными структурами определяют структуры памяти, используемые для представления данных. *Время жизни* учитывают при размещении данных в статической или динамической памяти, а также во внешней памяти.

Рассмотрим существующие варианты внутреннего представления данных, их элементов и связей между ними более подробно.

Представление данных в оперативной памяти. Различают две базовые структуры организации данных в оперативной памяти: векторную и списковую.

Векторная структура представляет собой последовательность байт памяти, которые используются для размещения полей данных (рис. 5.14). Последовательное размещение организованных структур данных позволяет осуществлять прямой доступ к элементам: по индексу (совокупности индексов) - в массивах или строках или по имени поля - в записях или объектах.

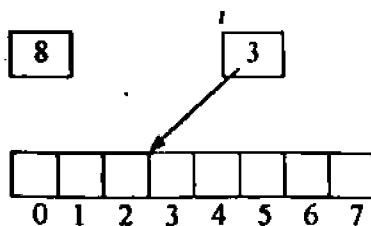


Рис. 5.14. Векторная структура памяти

Однако выполнение операций добавления и удаления элементов при использовании векторных структур для размещения элементов массивов может потребовать осуществления многократных сдвигов элементов.

Структуры данных в векторном представлении можно размещать как в *статической*, так и в *динамической* памяти. Расположение векторных представлений в динамической памяти иногда позволяет существенно увеличить эффективность использования оперативной памяти. Желательно размещать в динамической памяти временные структуры, хранящие промежуточные результаты, и структуры, размер которых сильно зависит от вводимых исходных данных.

Списковые структуры строят из специальных элементов, включающих помимо информационной части еще и один или несколько указателей-адресов элементов или вложенных структур, связанных с данным элементом. Размещая подобные элементы в динамической памяти можно организовывать различные внутренние структуры (рис. 5.15).

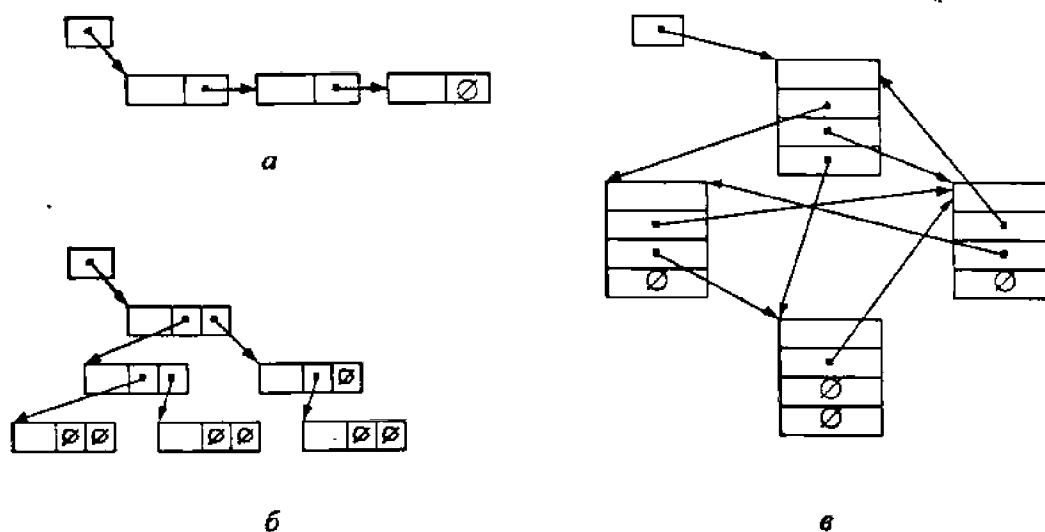


Рис. 5.15. Примеры списковых структур памяти:

а – линейный односвязный список; б – древовидный список; в – n-связный список

Однако при использовании списковых структур следует помнить, что:

- для хранения указателей необходима дополнительная память;
- поиск информации в линейных списках осуществляется *последовательно*, а потому требует больше времени;
- построение списков и выполнение операций над элементами данных, хранящимися в списках, требует более высокой квалификации программистов, более трудоемко, а соответствующие подпрограммы содержат больше ошибок и, следовательно, требуют более тщательного тестирования.

Обычно векторное представление используют для хранения статических множеств, таблиц (одномерных и многомерных), например, матриц, строк, записей, а также графов, представленных матрицей смежности, матрицей инцидентности или аналитически [55]. Списковое представление удобно для хранения динамических (изменяемых) структур и структур со сложными связями.

В наиболее ответственных случаях при выборе внутреннего представления целесообразно определять вычислительную сложность [24,55] выполнения наиболее часто встречающихся операций со структурой данных или ее элементами для различных вариантов. А также оценивать их емкостную сложность.

Пример 5.3. Разработать внутреннее представление неориентированного графа, над которым в основном выполняют операции определения смежности вершин, определения вершин, смежных данной, и удаления вершины.

В [27,55] предложены 10 вариантов внутреннего представления неориентированного графа, приведённого на рис. 5.16, а. Причем представление в виде матрицы смежности (рис. 5.16, б) использует табличный способ описания связности вершин. Комбинации векторов и односвязных списков (рис. 5.16, в-и) реализуют аналитическое задание графа, а вектор и список n-связных списков напрямую отображают связи вершин. Интересно также, что структуры, изображенные на рис. 5.16, б, в и д, могут быть размещены в статической памяти.

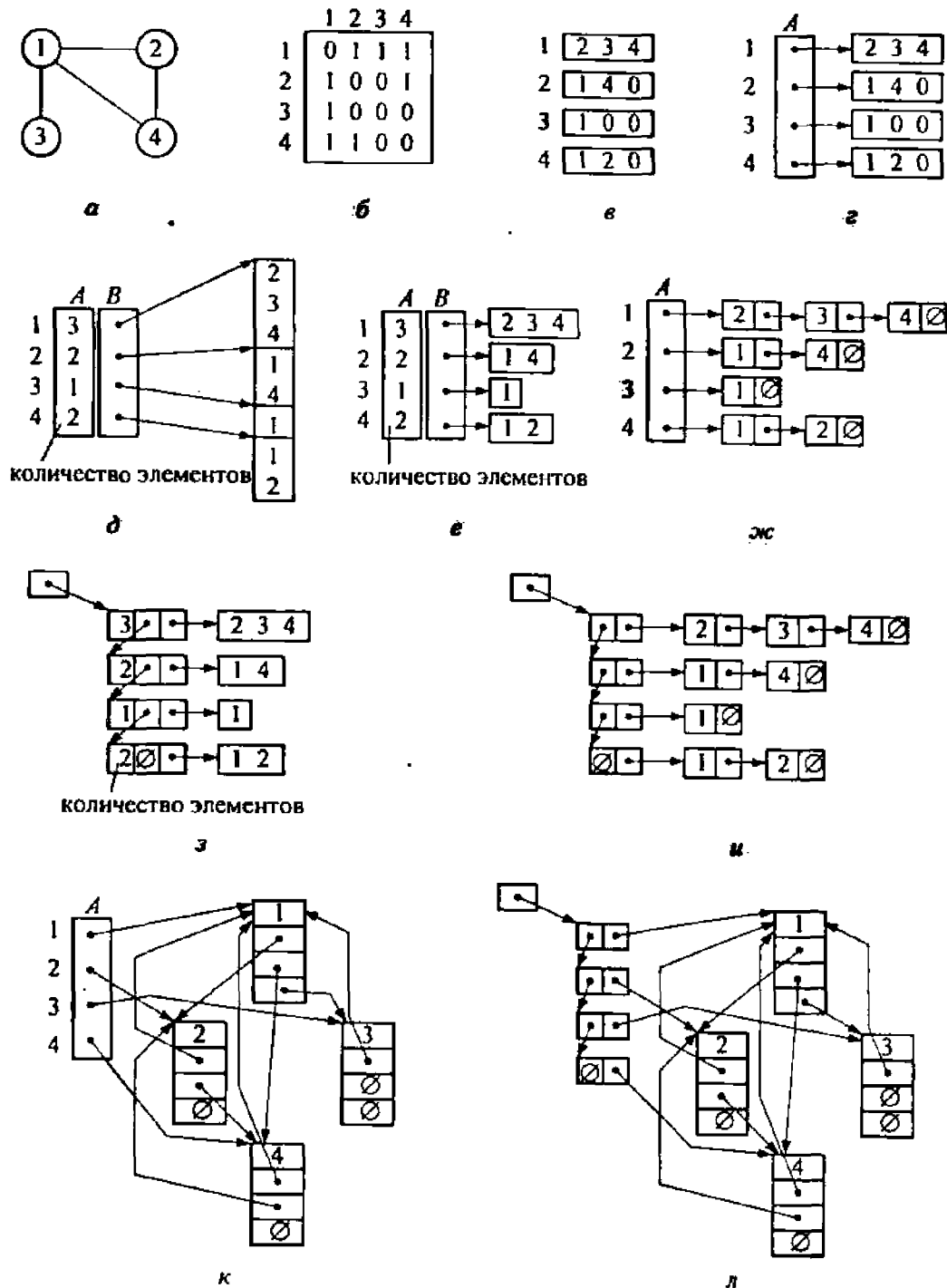


Рис. 5.16. Внутренние представления неориентированного графа:

а – граф; б – матрица смежности; в – массив статических векторов; г – массив динамических векторов фиксированного размера; д – массив статических векторов с граничными указателями; е – массив динамических векторов; ж – массив списков; з – список массивов; и – список списков; к – массив n-связных списков; л – список n-связных списков

Для выбора структуры необходимы исследования. В табл. 5.2 приведены результаты расчета временной сложности указанных операций на уровне машинных команд в тактах микропроцессора для каждого представления и емкостной сложности этих представлений. (Оценка временной сложности выполнялась по методике, предложенной в [27, 55].)

Анализ результатов показывает, что, если число вершин $n \approx 100$, то с точки зрения уменьшения времени выполнения наиболее эффективное представление - массив списков. Если же существенно экономное использование оперативной памяти, то наиболее эффективное представление - массив динамических векторов.

Представление данных во внешней памяти. Современные операционные системы поддерживают два способа организации данных во внешней памяти: последовательный и с прямым доступом.

Таблица 5.2

Структуры данных	Временная сложность, такт			Емкостная сложность, байт
	Определение смежности вершин	Определение вершин, смежных данной	Операция удаления вершины	
Матрица смежности	27 (27)	$32n+4$ (3204)	$59n^2-82n+33$ (598233)	$2n^2$ (20000)
Массив статических векторов (1)	$16,5\rho_{\max}-12,5$ (152,5)	$32\rho_{\max}+26\rho+4$ (454)	$29,5n\rho_{\max}+4n+32\rho+32\rho_{\max}\rho+29,5\rho_{\max}+5$ (31460)	$2n\rho_{\max}$ (2000)
Массив статических векторов (2)	$4,5\rho_{\max}+3,5$ (48,5)	$8\rho_{\max}+2\rho+8$ (98)	$5,5n+8\rho_{\max}\rho+11\rho_{\max}+10\rho+7,5$ (1117,5)	$2n\rho_{\max}+2n$ (2200)
Массив динамических векторов	$9\rho+12$ (57)	$6\rho+12$ (42)	$9n+15\rho^2+19,5\rho+8$ (1393)	$2n(\rho+2)$ (1400)
Массив списков	$9\rho+7$ (52)	$6\rho+7$ (37)	$5,5n+11\rho^2+22\rho+9,5$ (944,5)	$2n(2\rho+1)$ (2200)
Список векторов	$3,5n+9\rho+8,5$ (403,5)	$3,5n+6\rho+10$ (390)	$4,5n+3,5n\rho+13,5\rho^2+20\rho+20,5$ (2658)	$2n\rho+6n+2$ (1602)
Список списков	$3,5n+9\rho+12,5$ (407,5)	$3,5n+6\rho+8,5$ (388,5)	$4,5n+3,5n\rho+11\rho^2+25,5\rho+16,5$ (2619)	$4n\rho+4n+2$ (2602)
Массив п-связных списков	$9\rho_{\max}+6\rho+8$ (128)	$9\rho_{\max}+4\rho+8$ (118)	$5,5n+11\rho_{\max}\rho+13\rho_{\max}+10\rho+8,5$ (1288,5)	$2n\rho_{\max}+4n$ (2400)
Список п-связных списков	$3,5n+9\rho_{\max}+6\rho+9,5$ (479,5)	$3,5n+9\rho_{\max}+4\rho+6$ (466)	$4,5n+11\rho_{\max}\rho+13\rho_{\max}+10\rho+17,5$ (1197,5)	$2n\rho_{\max}+6n+2$ (2602)

Примечание. В таблице использованы следующие обозначения:

n – размерность задачи (количество вершин графа);

p и p_{\max} - среднее и максимальное количество вершин, смежных данной.

В круглых скобках под выражениями приведены результаты расчета по ним – для $n=100$,

$p = 5$, и $p_{\max} = 10$.

При последовательном доступе к данным возможно выполнение только последовательного чтения элементов данных или последовательная их запись. Такой вариант предполагается при работе с логическими устройствами типа клавиатуры или дисплея, при обработке текстовых файлов или файлов, формат записей которых меняется в процессе работы.

Прямой доступ возможен только для дисковых файлов, обмен информацией с которыми осуществляется записями фиксированной длины (двоичные файлы C или типизированные файлы Pascal). Адрес записи такого файла можно определить по ее *номеру*, что и позволяет напрямую обращаться к нужной записи.

При выборе типа памяти для размещения структур данных следует иметь в виду, что:

- в оперативной памяти размещают данные, к которым необходим быстрый доступ как для чтения, так и для их изменения;

- во внешней - данные, которые должны сохраняться после завершения программы.

Возможно, что во время работы данные целесообразно хранить в оперативной памяти для ускорения доступа к ним, а при ее завершении - переписывать во внешнюю память для длительного хранения. Именно этот способ используют большинство текстовых редакторов: во время работы с текстом он весь или его часть размещается в оперативной памяти, откуда по мере надобности переписывается во внешнюю память. В подобных случаях разрабатывают два представления данных: в оперативной и во внешней памяти.

Правильный выбор структур во многом определяет эффективность разрабатываемого программного обеспечения и его технологические качества, поэтому данному вопросу должно уделяться достаточное внимание независимо от используемого подхода.

5.5. Проектирование программного обеспечения, основанное на декомпозиции данных

В § 4.5 уже упоминалось, что практически одновременно были предложены методики проектирования программного обеспечения Джексона и Варнье-Орра, основанные на декомпозиции данных. Обе методики предназначены для создания «простых» программ, работающих со сложными, но иерархически организованными структурами данных. При необходимости разработки программных систем в обоих случаях предлагается вначале разбить систему на отдельные программы, а затем использовать данные методики.

Методика Джексона. При создании своей методики М. Джексон исходил из того, что структуры исходных данных и результатов определяют структуру программы.

Методика основана на поиске соответствий структур исходных данных и результатов. Однако при ее применении возможны ситуации, когда на каких-то уровнях соответствия отсутствуют. Например, записи исходного файла сортированы не в том порядке, в котором соответствующие строки должны появляться в отчете. Такие ситуации были названы «столкновениями». Выделяют несколько типов столкновений, которые разрешают по-разному. При различной последовательности записей их просто сортируют до обработки. Более подробно способы разрешения столкновений изложены в [33].

Разработка структуры программы в соответствии с методикой выполняется следующим образом:

- строят изображение структур входных и выходных данных;
- выполняют идентификацию связей обработки (соответствия) между этими данными;
- формируют структуру программы на основании структур данных и обнаруженных соответствий;

- добавляют блоки обработки элементов, для которых не обнаружены соответствия;
- анализируют и обрабатывают несоответствия, т.е. разрешают «столкновения»;
- добавляют необходимые операции (ввод, вывод, открытие/закрытие файлов и т. п.);
- записывают программу в структурной нотации (псевдокоде).

Пример 5.4. Разработать структуру программы, которая читает записи об успеваемости студентов и формирует список неуспевающих студентов группы.

На рис. 5.17 представлены структуры входных и выходных данных программы. Анализ этих структур показывает, что между ними есть соответствия (на рис. 5.17 эти соответствия показаны полужирными дугами). Помимо полного соответствия, имеет место еще частичное соответствие — соответствие, отмечаемое только, если студент имеет задолженность (на рис. 5.17 оно отмечено полужирным пунктиром).

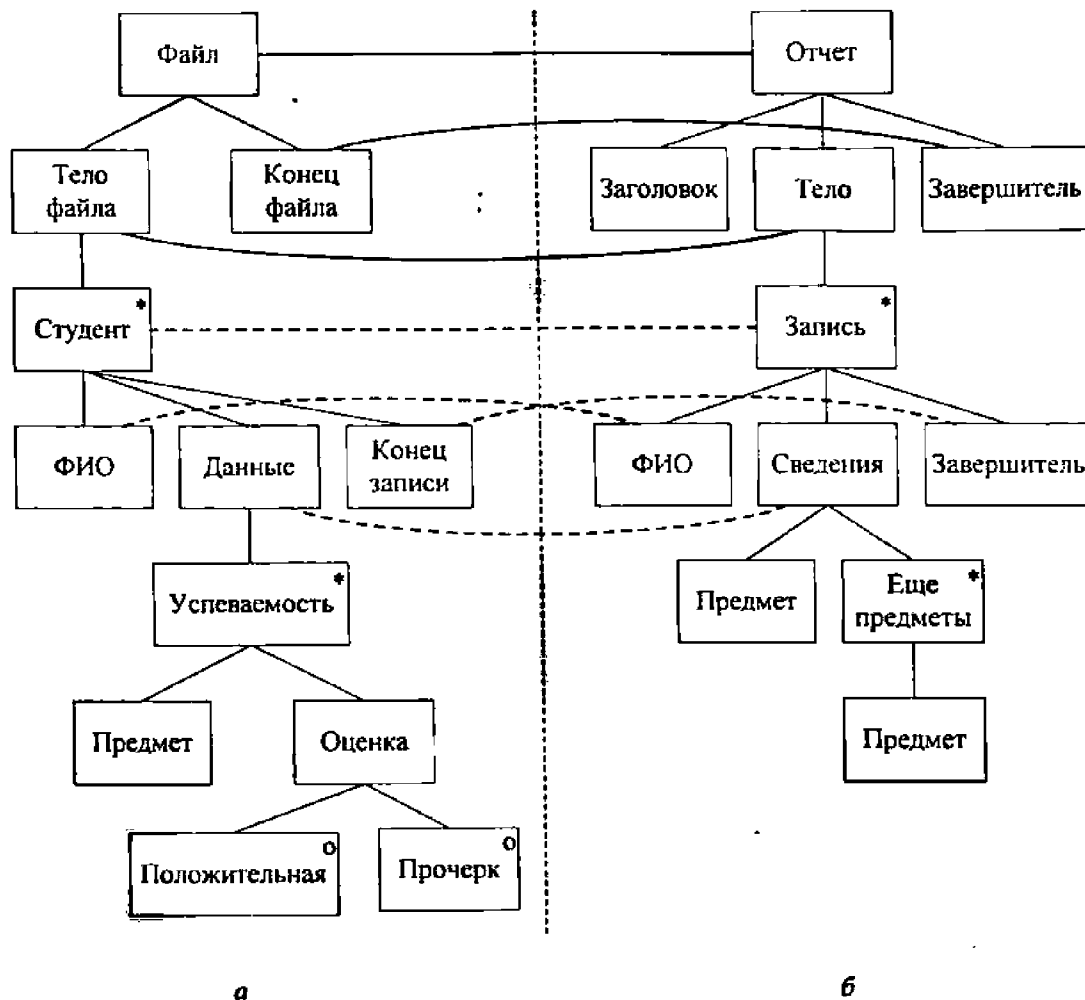


Рис. 5.17. Структуры входных (а) и выходных (б) данных программы

Используя найденные полные и неполные соответствия, строим «каркас» программы (затемненные блоки на рис. 5.18). Согласно методике добавляем блоки, которые позволят разрешить «столкновения» (светлые блоки на рис. 5.18).

Далее строим полный список операций, которые должна выполнять программа, учитывая, что не каждой записи исходного файла соответствует строка отчета (признак «формировать запись вывода» установлен), и выводить надо названия только тех предметов, по которым у студента есть задолженности (признак «задолженность» установлен):

- 1 - завершить работу;
- 2 - открыть входной файл;

- 3 - открыть выходной файл;
- 4 - закрыть входной файл;
- 5 - закрыть выходной файл;
- 6 - вывести заголовок;
- 7 - вывести завершитель;
- 8 - ввести запись входного файла;
- 9 - вывести строку отчета (при включенном состоянии признака «формировать запись вывода»);
- 10 - очистить буфер вывода;
- 11 - установить признак «формировать запись вывода»;
- 12 - сбросить признак «формировать запись вывода»;
- 13 - поместить в строку вывода ФИО;
- 14 - установить признак «задолженность»;
- 15 - сбросить признак «задолженность»;
- 16 - занести название предмета в строку вывода;
- 17 - стереть название предмета из буфера.

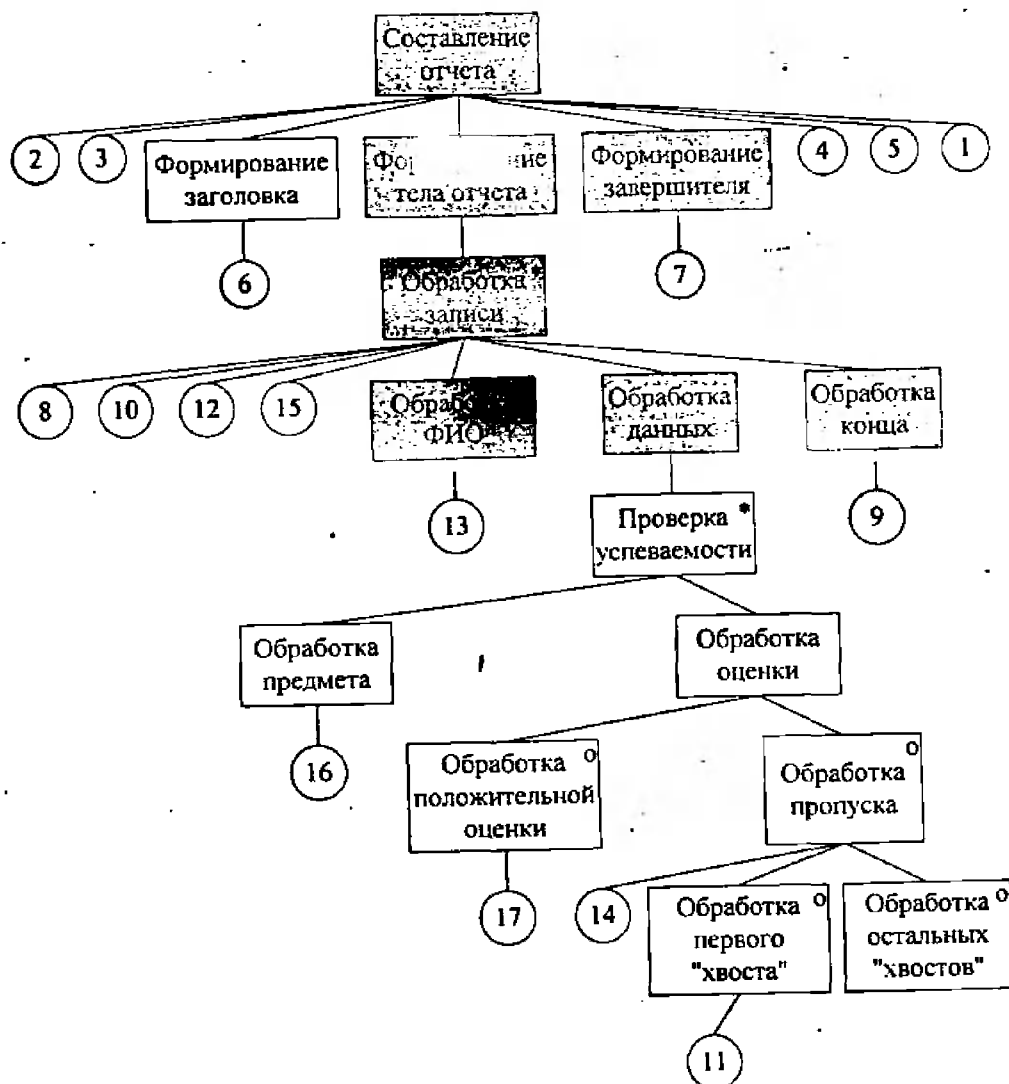


Рис. 5.18. Структура программы формирования списка студентов с задолженностями:

- – блоки, полученные при анализе соответствий;
- – блоки, полученные при анализе несоответствий;
- – блоки, добавленные при анализе операций

Затем определяем местоположение операций обработки (на рис. 5.18 они показаны кружочками с номерами). Результатом является полная структура разрабатываемой программы в нотации Джексона. Далее в соответствии с методикой следует записать алгоритм программы на псевдокоде.

В методике Джексона предлагается псевдокод, точно соответствующий графической нотации. Он использует следующие конструкции.

Последовательность:

```
<Имя> Посл.  
    Выполнить <действие 1>  
    Выполнить <действие 2>  
<Имя> конец
```

Выбор:

```
<Имя> Выбор <условие действия 1>  
    Выполнить <действие 1>  
<Имя> или <условие действия 2>  
    Выполнить <действие 2>  
<Имя> конец;
```

Повторение:

```
<Имя> Повт. пока не <условие действия>  
    Выполнить <действие 1>  
<Имя> конец
```

С применением этого псевдокода запись алгоритма программы выглядит следующим образом:

Составление отчета **посл.**

```
Открыть входной файл  
Открыть выходной файл  
Вывести заголовок отчета  
Формирование тела отчета повт. пока не конец входного файла  
    Ввести запись  
    Очистить буфер вывода  
    Сбросить признак «формировать запись вывода»  
    Сбросить признак «задолженность»  
    Вывести в буфер ФИО  
    Обработка данных повт. пока не конец записи  
        Обработка предмета посл.  
            Занести название предмета в строку вывода  
        Обработка предмета конец  
        Обработка оценки выбор если оценка положительна  
            Стереть название предмета из буфера  
        Обработка оценки или если оценка пропуск  
            Установить признак «задолженность»  
        Обработка пропуска выбор если не установлен  
            признак «формировать запись вывода»  
            Установить признак «формировать запись вывода»  
        Обработка пропуска конец  
        Обработка оценки конец
```

Обработка конца записи **выбор** если установлен признак
«формировать запись вывода»

Вывести строку отчета

Обработка конца записи **конец**

Обработка данных **конец**

Формирование тела отчета **конец**

Вывести завершитель

Закрыть входной файл

Закрыть выходной файл

Завершить работу

Составление отчета **конец**

Методика Варнье-Орра. Методика Варнье-Орра базируется на том же положении, что и методика Джексона, но основными при построении программы считаются структуры выходных данных и, если структуры входных данных не соответствуют структурам выходных, то их допускается менять. Таким образом, ликвидируется основная причина столкновений. В примере 5 4 целесообразно поменять местами оценки и названия предметов, чтобы упростить обработку.

Однако на практике не всегда существует возможность пересмотра структур входных данных: эти структуры уже могут быть строго заданы, например, если используются данные, полученные при выполнении других программ, поэтому данную методику применяют реже.

Как следует из вышеизложенного, методики Джексона и Варнье-Орра могут использоваться только в том случае, если данные разрабатываемых программ могут быть представлены в виде иерархии или совокупности иерархий.

5.6. Case-технологии, основанные на структурных методологиях анализа и проектирования

К нашему времени накоплен опыт успешного использования большинства известных методологий структурного анализа и проектирования в соответствующих CASE-средствах. Наибольшее распространение получили методологии [30]: SADT (3,3%), структурного системного анализа Гейна-Сар-сона (20,2%), структурного анализа и проектирования Йордана-Де Марко (36,5%), развития систем Джексона (7,7%), развития структурных схем DSSD (Data Structured System Development) Варнье-Орра (5,8%), анализа и проектирования систем реального времени Уорда-Меллора и Хатли, информационного моделирования Мартина (22,1%).

Как видно из приведенных статистических данных, наибольшее применение нашли структурные методологии, использующие диаграммы потоков данных. Это вызвано двумя причинами:

- диаграммы потоков данных более детально по сравнению с функциональными диаграммами отображают специфику многочисленных в настоящее время *информационных* систем: не требуют строгой типизации обрабатываемой информации, предусматривают возможность хранения данных, конкретизируют взаимодействие с внешним миром, предусматривают получение комплексной модели программного обеспечения и т. п.;

- разработан метод построения проектных спецификаций (структурных карт Джексона или Костантайна) по диаграммам потоков данных, что позволяет автоматически создавать такие спецификации.

В табл. 5.3 представлены данные о моделях, поддерживающих соответствующий пакет, а в табл. 5.4 - нотации представления соответствующей информации.

Несмотря на то, что последнее время все большее распространение получают объектно-ориентированные средства разработки программного обеспечения, структурные методологии продолжают совершенствоваться. Их успешно применяют при разработке многих программных продуктов, например, для уточнения требований к системам, основной частью которых являются базы данных, очень часто используют диаграммы потоков данных.

Таблица 5.3

Название	Фирма	Функции	Данные	События
BPWin	Logic Works	+	—	—
CASE Аналитик	Эйтекс	+	+	+
CASE/4/0	MicroTOOL	+	+	+
DatabaseDesigner	Oracle	—	+	—
Design/IDEF	Meta Software	+	+	—
Designer/2000	Oracle	+	+	—
EasyCASE	Evergreen CASE Tools	+	+	+
ERWin	Logic Works	—	+	—
I-CASE Yourdan	CAYENNE	+	+	+
Prokit*WORK	BENCHMDIS	+	+	—
S-Designer	Sybase/Powersoft	+	+	—
SILVERRun	CSA	+	+	+
Visible Analyst Workbench	Visible Systems	+	+	—

Таблица 5.4

Название	Нотация DFD	Спецификации	Поведение	Структурные карты
CASE Аналитик	Гейн-Сарсон	Структурный язык	Управляющие потоки	—
CASE/4/0	Иордан (расширенная)	—	Уорд-Меллор (с STD)	Джексон
Designer/2000	Гейн-Сарсон	—	—	Джексон
I-CASE Yourdan	Гейн-Сарсон, Иордан	Структурный язык	Уорд-Меллор (с STD)	Константайн
EasyCASE	Иордан	3GL	STD	Константайн
Prokit*WORKBENCH	Гейн-Сарсон	—	—	Константайн
S-Designer	Гейн-Сарсон, Иордан	—	—	Константайн
SILVERRun	произвольная	—	Управляющие потоки	—
Visible Analyst WORKBENCH	Гейн-Сарсон, Иордан	—	—	Константайн

Контрольные вопросы в задания

1. Что понимают под структурной и функциональной схемами программного обеспечения? В каких случаях их применяют? Чем отличаются структурные и функциональные схемы программного обеспечения с различной архитектурой?

2. На каких свойствах программных систем основан метод пошаговой детализации? Почему с его применением получают только структурные алгоритмы? В чем, по-вашему, заключается основная сложность данного метода?

3. Как используется метод пошаговой детализации при разработке алгоритмов и структуры программного обеспечения?

4. Используя метод пошаговой детализации, разработайте алгоритм сложения чисел ($n, m \leq 1000$), записанных римскими цифрами: I - 1; II - 2; III - 3; IV - 4; V - 5; VI - 6; VII - 7; VIII - 8; IX - 9; X - 10; L - 50; C - 100; D - 500; M - 1000.

5. Для чего строят структурные карты Константайна? Постройте структурные карты Константайна для задания 4. Чем структурные карты Джексона отличаются от структурных карт Константайна?

7. Что положено в основу методик Джексона и Варнье-Орра? Чем различаются данные методики?

8. Какие вопросы решают при проектировании структур данных? Какие характеристики проектируемых структур при этом учитывают? Предложите несколько вариантов структур данных для программы задания 3. Какая из них является лучшей и почему?

9. Для каких разработок целесообразно использовать структурные методологии?

6. АНАЛИЗ ТРЕБОВАНИЙ И ОПРЕДЕЛЕНИЕ СПЕЦИФИКАЦИЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ ОБЪЕКТНОМ ПОДХОДЕ

Модели разрабатываемого программного обеспечения при объектном подходе основаны на предметах и явлениях реального мира. В основе этих моделей также лежит описание требуемого поведения разрабатываемого программного обеспечения, т. е. его функциональности, но это поведение связывается с состояниями элементов (объектов) конкретной предметной области.

Таким образом, на этапе анализа ставятся две задачи:

- уточнить требуемое поведение разрабатываемого программного обеспечения;
- разработать концептуальную модель его предметной области с точки зрения поставленных задач.

6.1. UML - стандартный язык описания разработки программных продуктов с использованием объектного подхода

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 и происходит выполнение требуемых функций (рис. 6.1).

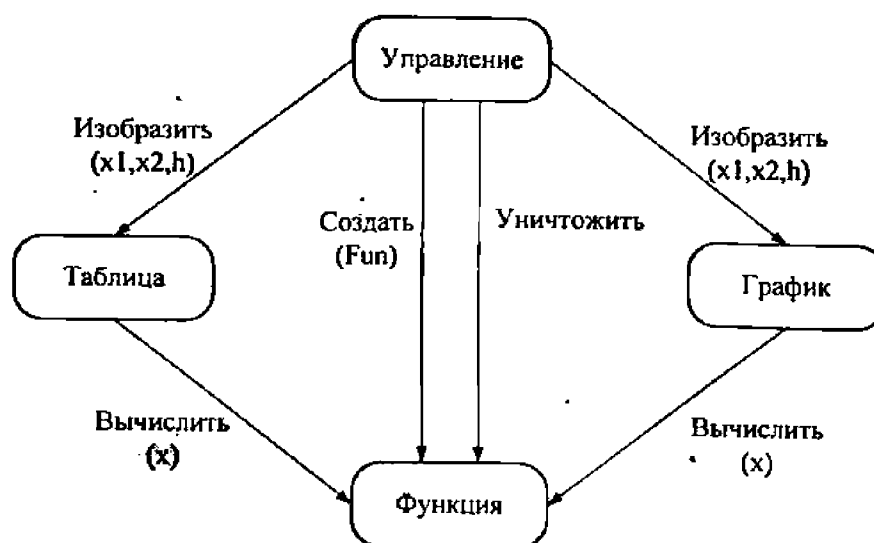


Рис. 6.1. Объектная декомпозиция программы построения
таблиц и графиков

Однако при объектном подходе так же, как при структурном подходе, сразу можно выполнить декомпозицию только очень простого программного обеспечения. Поэтому на заре эпохи объектно-ориентированного программирования были предложены различные методы анализа и проектирования программного обеспечения в рамках объектного подхода, использующие различные модели и нотации. Спорить о достоинствах и недостатках этих методов и моделей можно было бесконечно. Эта ситуация получила название «войны методов».

Конец «войне методов» положило появление в 1995 г. первой версии языка UML (Unified Modeling Language - унифицированный язык моделирования - см. приложение), который в настоящее время фактически признан стандартным средством описания проектов, создаваемых с

использованием объектно-ориентированного подхода. Его создателями являются ведущие специалисты в этой области: Град Буч, Ивар Якобсон и Джеймс Рамбо, которые использовали в своем языке все лучшее, что появилось в подходах этих авторов во время «войны методов».

Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей: использования, логическую, реализации, процессов, развертывания (рис. 6.2).

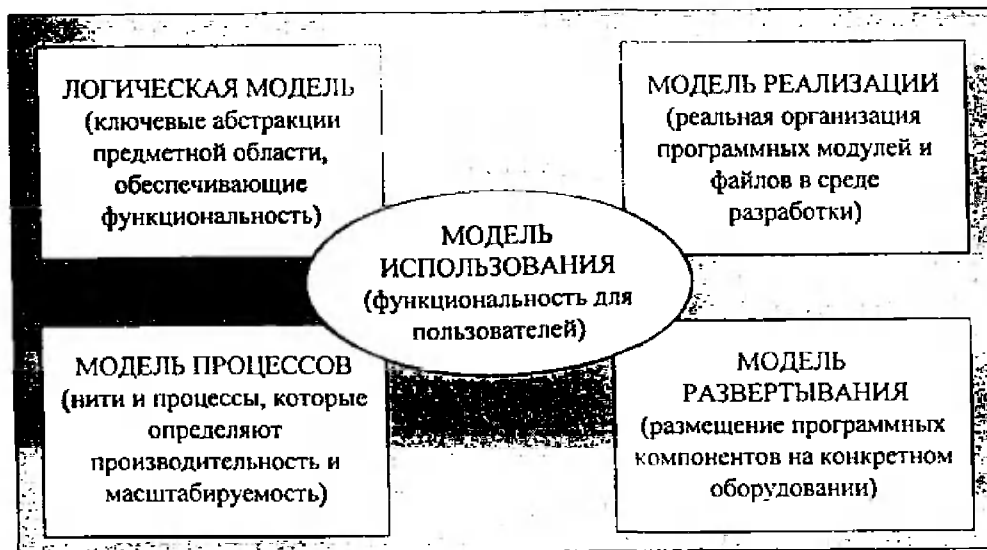


Рис. 6.2. Полная спецификация разрабатываемого программного обеспечения при объектном подходе (UML)

Модель использования представляет собой описание функциональности программного обеспечения с точки зрения пользователя.

Логическая модель описывает ключевые абстракции программного обеспечения (классы, интерфейсы и т. п.), т. е. средства, обеспечивающие требуемую функциональность.

Модель реализации определяет реальную организацию программных модулей в среде разработки.

Модель процессов отображает организацию вычислений и оперирует понятиями «процессы» и «нити». Она позволяет оценить производительность, масштабируемость и надежность программного обеспечения.

И, наконец, *модель развертывания* показывает особенности размещения программных компонентов на конкретном оборудовании.

Таким образом, каждая из указанных моделей характеризует определенный аспект проектируемой системы, а все они вместе составляют относительно полную модель разрабатываемого программного обеспечения.

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

- диаграммы вариантов использования (см. § 6.2);
- диаграммы классов (см. § 6.3, 7.3 и 7.4);
- диаграммы пакетов (см. § 7.1);
- диаграммы последовательностей действий (см. § 6.4 и 7.4);
- диаграммы кооперации (см. § 7.3);
- диаграммы деятельностей (см. § 6.5 и 7.4);
- диаграммы состояний объектов (см. § 7.4);
- диаграммы компонентов (см. § 7.5);
- диаграммы размещения (см. § 7.6).

Все указанные диаграммы по возможности используют единую графическую нотацию, что облегчает их понимание.

Помимо указанных диаграмм, как и при структурном подходе, спецификация обязательно включает словарь терминов, а также различного рода описания и текстовые спецификации. Конкретный набор документации определяется разработчиком.

UML и предлагаемая теми же авторами методика Rational Unified Process поддерживаются пакетом Rational Rose фирмы Rational Software Corporation. Ряд диаграмм UML можно построить также средствами программы Microsoft Visual Modeler и других CASE-средств. По данным «USA Today» в настоящее время 49 из 50-ти ведущих компьютерных компаний используют UML при разработке программного обеспечения с использованием объектного подхода, что и позволяет говорить о том, что сегодня UML фактически стал стандартом описания подобных разработок.

6.2. Определение «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. Аспекты поведения программного обеспечения были названы «вариантами использования» или «прецедентами» (use cases).

Примечание. Варианты использования основаны на неформальном описании сценариев функционирования проектируемых программных систем, применяемом многими разработчиками программного обеспечения в 1980-1990 годах.

Вариант использования представляет собой характерную процедуру применения разрабатываемой системы конкретным действующим лицом, в качестве которого могут выступать не только люди, но и другие системы или устройства.

Не следует путать вариант использования с конкретными операциями будущей системы. Каждый вариант использования связан с некоторой целью, имеющей самостоятельное значение, например для текстового редактора Формирование оглавления - это вариант использования, а Связывание заголовков со специальными стилями - операция, которую необходимо выполнить, чтобы стало возможно автоматическое построение оглавления,

В зависимости от цели выполнения конкретной процедуры различают следующие варианты использования:

- основные - обеспечивают требуемую функциональность разрабатываемого программного обеспечения;
- вспомогательные - обеспечивают выполнение необходимых настроек системы и ее обслуживание (например, архивирование информации и т. п.);
- дополнительные - обеспечивают дополнительные удобства для пользователя (как правило, реализуются в том случае, если не требуют серьезных затрат каких-либо ресурсов ни при разработке, ни при эксплуатации).

Вариант использования можно описать кратко или подробно. Краткая форма описания содержит: название варианта использования, его цель, действующих лиц, тип варианта использования (основная, второстепенная или дополнительная) и его краткое описание. Краткое описание варианта использования Выполнение задания системы решения комбинаторно-оптимизационных задач можно представить в следующем виде:

Название варианта	Выполнение задания
Цель	<i>Получение результатов решения задачи</i>
Действующие лица	<i>Пользователь</i>
Краткое описание	<i>Решение задачи предполагает выбор задачи, выбор алгоритма, задание данных и получение результатов решения.</i>
Тип варианта	<i>Основной</i>

Основные варианты использования обычно описывают подробно, стараясь отразить особенности предметной области разрабатываемого программного обеспечения. Подробная форма, кроме указанной выше информации, включает описание типичного хода событий и возможных альтернатив. Типичный ход событий представляют в виде диалога между пользователями и системой, последовательно нумеруя события. Если пользователь может выбирать варианты, то их описывают в отдельных таблицах. Также отдельно приводят альтернативы, связанные с нарушением типичного хода событий. Ниже представлено подробное описание варианта использования **Выполнение задания**:

Вариант использования **Выполнение задания**
Типичный ход событий

Действие исполнителя	Отклик системы
<p><i>1. Пользователь инициирует новое задание</i></p> <p><i>3. Пользователь выбирает тип задачи</i></p> <p><i>5. Пользователь выбирает способ задания данных:</i> <i>а) Если выбран ввод с клавиатуры, см. раздел Ввод данных</i> <i>б) Если выбран ввод из базы данных, см. раздел Выбор данных из базы</i></p> <p><i>7. Пользователь выбирает алгоритм</i></p> <p><i>9. Пользователь инициирует процесс решения</i></p> <p><i>11. Пользователь ожидает</i></p> <p><i>13. Пользователь анализирует результаты и выбирает, сохранять их в базе или нет</i></p>	<p><i>2. Система регистрирует новое задание и предлагает список типов задач</i></p> <p><i>4. Система регистрирует тип задачи и предлагает список способов задания данных</i></p> <p><i>6. Система регистрирует данные и предлагает список алгоритмов решения</i></p> <p><i>8. Система регистрирует алгоритм и предлагает начать решение</i></p> <p><i>10. Система проверяет полноту определения задания и запускает подпрограмму решения задачи</i></p> <p><i>12. Система демонстрирует пользователю результаты и предлагает сохранить их в базе данных</i></p> <p><i>14. Если выбрано сохранение данных, то система выполняет запись данных задания в базу</i></p> <p><i>15. Система переходит в состояние ожидания</i></p>

Альтернатива

11. Если время выполнения программы с точки зрения пользователя велико, то он прерывает процесс выполнения.

12. Система прерывает расчеты, предлагает список алгоритмов решения и возвращается на шаг 7.

Дополнительная информация

1. Необходимо обеспечить произвольную последовательность выбора типа задачи, данных и алгоритма.

2. Необходимо обеспечить возможность выхода из варианта на любом этапе.

Раздел **Ввод данных** Типичный ход событий

Действие исполнителя	Отклик системы
1. Пользователь выбрал Ввод данных 3. Пользователь вводит данные 5. Пользователь отвечает на запрос	2. Система последовательно запрашивает ввод данных 4. Система проверяет данные и запрашивает, сохранять ли данные в базе 6. Если выбран вариант сохранения данных, то система выполняет запись данных в базу и регистрирует их в текущем задании

Альтернатива

4. Если обнаружены некорректные данные, то система выдает сообщение об ошибке и предлагает их исправить, возвращаясь на предыдущий шаг.

Раздел **Выбор данных из базы**

Типичный ход событий

Действия исполнителя	Отклик системы
1. Пользователь выбрав Выбор данных из базы 2. Пользователь выбирает данные	3. Система демонстрирует список данных в базе 4. Система читает данные и регистрирует их в текущем задании

Диаграммы вариантов использования. Диаграммы вариантов использования позволяют наглядно представить ожидаемое поведение системы. Основными понятиями диаграмм вариантов использования являются: действующее лицо, вариант использования, связь.

Действующее лицо - внешняя по отношению к разрабатываемому программному обеспечению сущность, которая взаимодействует с ним с целью получения или предоставления какой-либо

информации. Как уже упоминалось выше, действующими лицами могут быть пользователи, другое программное обеспечение или какие-либо технические средства, взаимодействующие с разрабатываемым программным обеспечением.

Вариант использования - некоторая очевидная для действующего лица процедура, решающая его конкретную задачу. Все варианты использования, так или иначе, связаны с требованиями к функциональности разрабатываемой системы и могут сильно отличаться по объему выполняемой работы.

Связь - взаимодействие действующих лиц и соответствующих вариантов использования.

Варианты использования также могут быть связаны между собой. При этом фиксируют связи использования и расширения.

Использование подразумевает, что существует некоторый фрагмент поведения разрабатываемого программного обеспечения, который повторяется в нескольких вариантах использования. Этот фрагмент оформляют, как отдельный вариант использования и указывают связь с ним типа «использование».

Расширение применяют, если имеется два подобных варианта использования, различающиеся наличием в одном из них некоторых дополнительных действий. В этом случае дополнительные действия определяют как отдельный вариант использования, который связан с основным вариантом связью типа «расширение».

На рис. 6.3 приведены условные обозначения, которые применяют при изображении диаграмм вариантов использования.



Рис. 6.3. Основные условные обозначения диаграмм вариантов использования:

а – действующее лицо; *б* – вариант использования; *в* – связь

Пример 6.1. Построить диаграмму вариантов использования для системы решения комбинаторно-оптимизационных задач.

Действующее лицо у данной системы одно - Пользователь, который, по сути дела, обращается к системе либо для решения новой задачи, либо для просмотра результатов ранее решенной задачи, которые должны сохраняться в базе данных. Представим эти варианты использования на диаграмме (рис. 6.4).

Вариант Выполнение задания на самом деле включает несколько вариантов, различающихся способом определения данных (ввод с клавиатуры или чтение из базы) и сохранением введенных данных в базе. Изобразим эти варианты на схеме, указав соответствующие расширения данного варианта (см. рис. 6.4).

Помимо двух основных вариантов использования, система должна также предусматривать вспомогательные прецеденты для удаления лишних данных и результатов из базы.

Пример 6.2. Построить диаграмму вариантов использования для системы учета успеваемости студентов.

Действующими лицами системы являются Декан, Заместитель декана по курсу и Сотрудник деканата. Варианты использования выявляем, анализируя техническое задание, и изображаем на диаграмме, связывая с соответствующими действующими лицами (рис. 6.5).

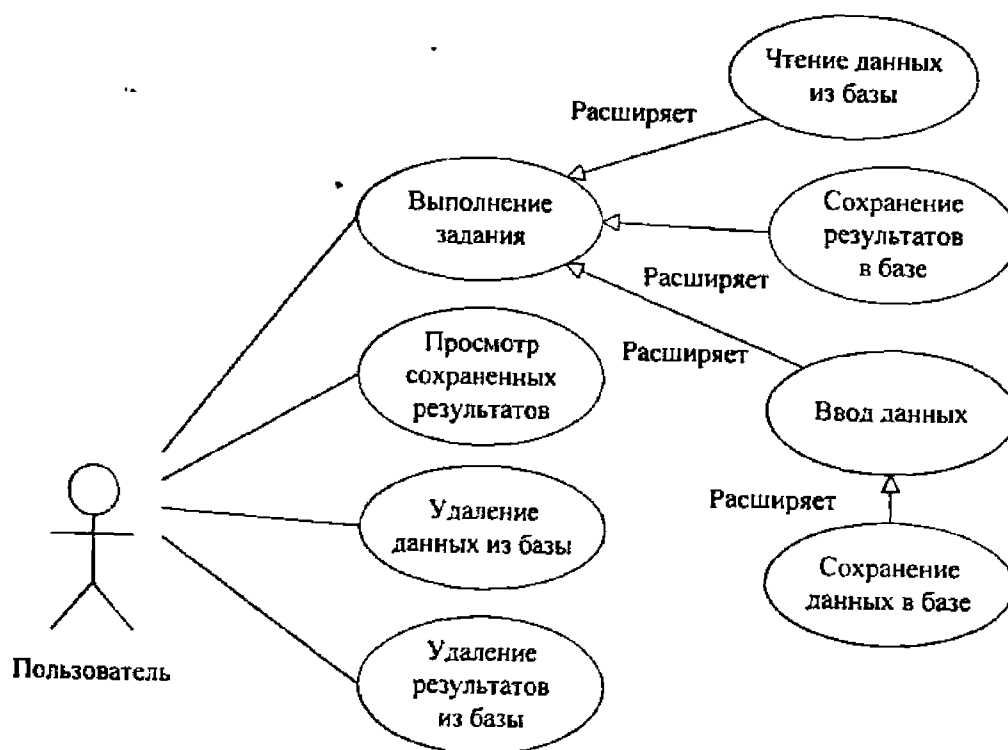


Рис. 6.4. Диаграмма вариантов использования для системы решения комбинаторно-оптимизационных задач

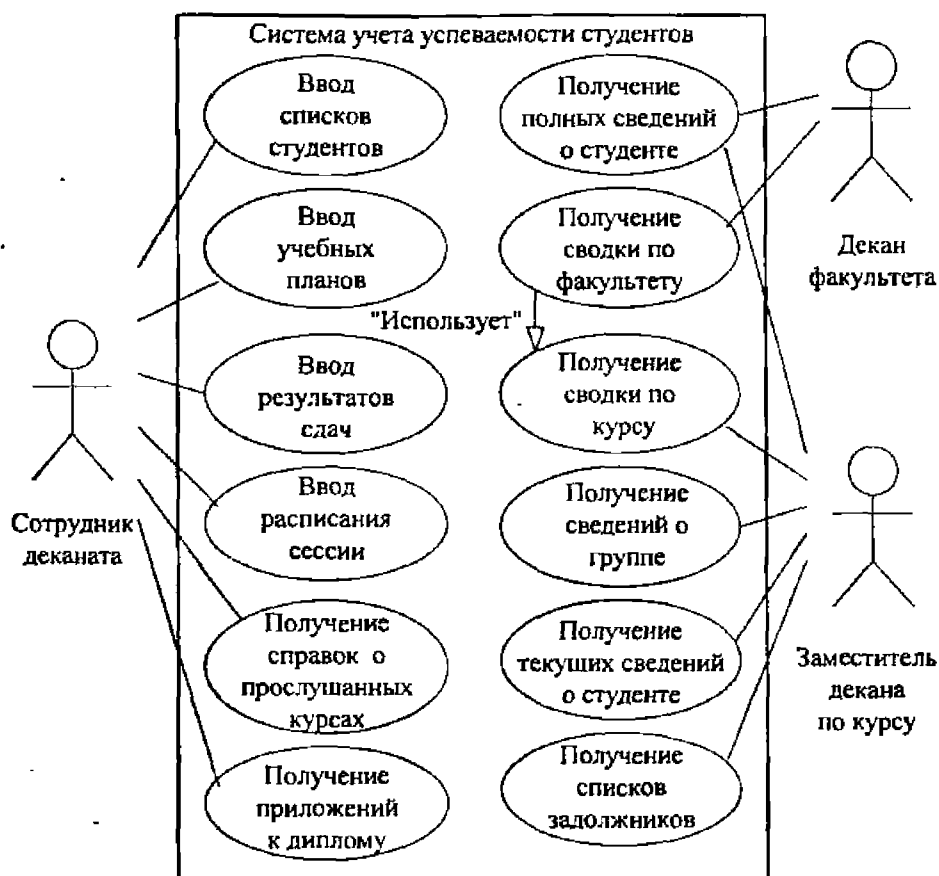


Рис. 6.5. Диаграмма вариантов использования системы учета успеваемости студентов

Анализ вариантов использования показывает, что вариант получения сводки успеваемости по факультету «использует» вариант получения сводки по курсу, что и представлено на диаграмме.

Полученная диаграмма вариантов использования отражает типичное взаимодействие пользователя с разрабатываемым программным обеспечением. Ее необходимо обсудить с заказчиком для определения как можно большего числа основных вариантов использования и проанализировать на полноту обслуживания системы.

Естественно, все варианты использования определить, как правило, не удастся: 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.

6.3. Построение концептуальной модели предметной области

Диаграммы классов - центральное звено объектно-ориентированных методов разработки программного обеспечения, поэтому все существующие методы используют диаграммы классов в одной из известных нотаций. Однако в основном диаграммы классов в этих методах применяют на этапе проектирования, для того чтобы показать особенности построения конкретных классов. В отличие от ранее существовавших нотаций, UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

- концептуальный уровень, на котором диаграммы классов, называемые в этом случае контекстными, демонстрируют связи между основными понятиями предметной области;
- уровень спецификаций, на котором диаграммы классов отображают интерфейсы классов предметной области, т. е. связи объектов этих классов;
- уровень реализации, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Практически это три разных модели, связь между которыми неоднозначна. Так, если концептуальная модель определяет некоторое понятие предметной области как класс, то это не означает, что для реализации этого понятия будет использован отдельный класс. Однако во всех трех моделях нас интересуют типы объектов (классы) и их статические отношения, что позволяет использовать единую нотацию.

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.

Концептуальные модели в соответствии с определением оперируют понятиями предметной области, атрибутами этих понятий и отношениями между ними. Понятию в предметной области разрабатываемого программного обеспечения могут соответствовать как материальные предметы, так и абстракции, которые применяют специалисты предметной области.

Основным понятиям в модели ставятся в соответствие классы. Класс при этом традиционно понимают как совокупность общих признаков заданной группы объектов предметной области. В соответствии с этим определением на диаграмме классов каждому классу соответствует группа объектов, общие признаки которых и фиксирует класс. Так класс Студент объединяет общие признаки группы людей, обучающихся в высших учебных заведениях. Экземпляр класса или объект (например, Иванов И.И.) обязательно обладает всей совокупностью признаков своего класса и может иметь собственные признаки, не фиксированные в классе. Так, например, помимо того, что Иванов И.И. является студентом, он еще может быть спортсменом, музыкантом и т. д. Строго говоря, таким собственным признаком является и идентифицирующее студента имя.

На диаграммах класс изображается в виде прямоугольника, внутри которого указано имя класса (рис. 6.6, а). При необходимости допускается указывать характеристики класса, например атрибуты, используя специальные секции условного обозначения (рис. 6.6, б).

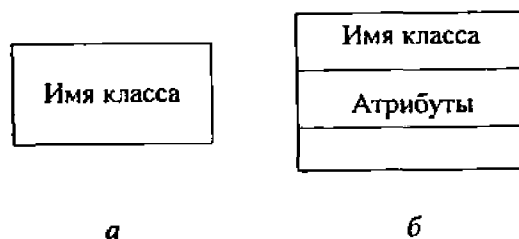


Рис. 6.6. Обозначение класса на концептуальной диаграмме классов:

a – без уточнения характеристик;
б – с уточнением атрибутов

В качестве *атрибутов* представляют некоторые, существенные с точки зрения решаемой задачи характеристики объектов, например идентифицирующие значения (имя, номер). Для конкретного объекта атрибут всегда имеет определенное значение. На диаграмме классов атрибуты обычно показывают в секции атрибутов.

Под *отношением классов* понимают статическую, т. е. не зависящую от времени, связь между классами. Различают два основных вида отношений: ассоциация и обобщение.

Отношение ассоциации означает наличие связи между экземплярами классов или объектами, например, класс Студент ассоциирован с классом Институт. Ассоциация может иметь имя, например Обучается. Рядом с именем ассоциации обычно ставят стрелку, указывающую направление чтения имени («Студент обучается в институте», а не наоборот).

Связь между экземплярами классов подразумевает некоторые *роли*, которые соответствующие объекты играют по отношению друг к другу. Роль связана с направлением ассоциации. Так по отношению к студентам институт – организация, осуществляющая их обучение, т. е. роль института можно назвать Место учебы. Студент для института – объект обучающей деятельности института, т. е. Обучаемый. Если роль собственного имени не имеет, то можно считать, что ее имя совпадает с именем класса, по отношению к которому определяется эта роль. Для рассматриваемого примера это соответственно роли Студент и Институт (рис. 6.7, а), но роль можно указать и явно (рис. 6.7, б).

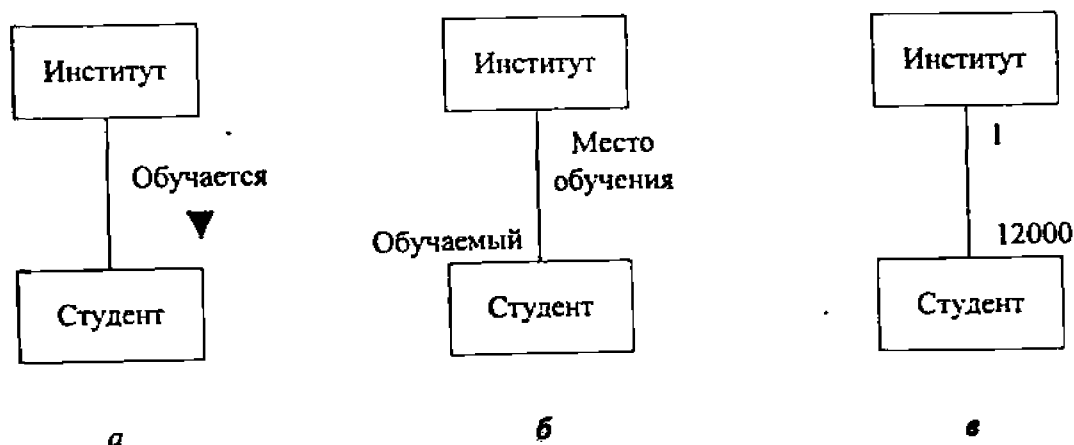


Рис. 6.7. Обозначение ассоциации:

a – с указанием имени ассоциации и ее направления; *б* – с указанием имен ролей;
в – с указанием множественности

Роль также обладает характеристикой множественности, которая показывает, сколько объектов может участвовать в одной связи с каждой стороны. Допускается указывать множественность:

* - от 0 до бесконечности;

<целое>..* - от заданного числа до бесконечности;

<целое> — точно определенное количество объектов;

<целое!>, <целое2> - несколько вариантов точного количества объектов;

<целое1>..<целое2> - диапазон объектов.

С теоретической точки зрения атрибут тоже класс, экземпляры которого жестко ассоциированы с рассматриваемым классом. В концептуальной модели для отображения соответствующих отношений могут использоваться как ассоциации, так и атрибуты. Например, отношение двух понятий Студент и Имя можно представить, как в виде ассоциации соответствующих классов, так и в варианте, когда классу Студент ставится в соответствие атрибут Имя.

Чтобы избежать излишних нагромождений рекомендуется следовать простому правилу [37]: *если некоторый объект X в реальном мире не является числом или текстом, то это скорее всего понятие*. В противном случае - это атрибут.

Обобщением называют такое отношение между классами, при котором любой объект одного класса (*подтипа*) обязательно является также и объектом другого класса, называемого в данном контексте *супертипом*. Так, если некоторый конкретный студент Иванов И.И. является объектом подтипа Студент первого курса супертипа Студент, то тот же самый Иванов И.И. является объектом указанного супертипа. Следовательно, все, что известно об объектах супертипа (ассоциации, атрибуты, операции), касается и объектов подтипа. На диаграмме классов обобщение обозначают линией с треугольной стрелкой на конце, подходящей к супертипу (рис. 6.8).

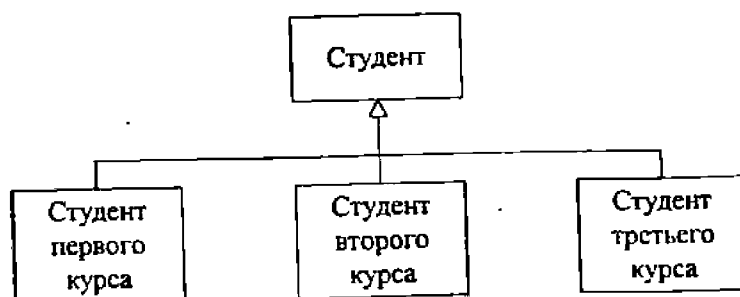


Рис. 6.8. Обозначение обобщения

На практике определение основных понятий предметной области, которые должны представляться на контекстной диаграмме в виде классов, является не тривиальной задачей. Обычно используют следующий способ:

- формируют множество понятий-кандидатов из существительных, характеризующих предметную область в описании вариантов использования;
- исключают понятия, не существенные для данного варианта использования, например, в предыдущем примере, «информация», «ввод» и т. п.

Для определения множества понятий-кандидатов полезно использовать перечень возможных категорий понятий-кандидатов, приведенный в табл. 6.1.

Пример 6.3. Построить концептуальную модель для системы решения комбинаторно-оптимизационных задач. Множество понятий-кандидатов для данной разработки включает следующие словосочетания:

Таблица 6.1

Категория понятий-кандидатов	Примеры
Физические или материальные объекты	Самолет – как целое
Спецификации, элементы дизайна или описания объекта – сохраняются в системе даже при отсутствии объектов	Цвет, спецификация товара
Место	Аэропорт, город
Роль человека	Продавец, покупатель, преподаватель
Контейнеры других объектов	Самолет – как совокупность частей, каталог – как совокупность описаний
Содержимое контейнеров	Часть, элемент
Другие компьютеры или внешние системы по отношению к разрабатываемой	Система бронирования билетов
Абстрактные понятия	Летательный аппарат
Организации	Фирма, предприятие, НИИ
События	Встреча, покупка билета
Процессы и их части*	Покупка билета, оплата стоимости
Правила и политика	Правила аннулирования заказа билета
Записи финансовой, трудовой, юридической и другой деятельности, руководства, книги	Чек, книга учета, должностная инструкция

* Представляется в виде класса, если не анализируются элементы процесса.

Задание, тип задачи, список типов задач, способ задания данных, ввод данных, выбор данных из базы, алгоритм решения задачи, список конкретных алгоритмов решения задачи, полнота описания задания, результаты, данные, база данных.

Попробуем выделить основные понятия и связать их между собой.

Цель основного варианта использования системы - выполнение задания. Полное описание задания включает: тип задачи, данные и указание на алгоритм. С ним же будут связаны и полученные результаты. Данные могут сохраняться в базе и вводиться. Описание задания и все, что с ним связано, может сохраняться в базе.

Определим возможные обобщения:

1) способ задания данных: *ввод данных, выбор данных из базы;*

2) алгоритм: алгоритм решения задачи: *конкретный алгоритмы решения задачи.*

Переходим к построению концептуальной модели.

Основной класс-понятие, исходя из описания, Задание. Связываем с ним классы-понятия Данные, Алгоритм и Результаты.

В разрабатываемой системе планируется реализовать алгоритмы решения задач трех типов: поиск цикла минимальной длины, проходящего через все вершины; поиск кратчайшего пути и поиск минимального покрывающего дерева. Следовательно, класс-понятие Алгоритм является супертипом для классов Алгоритм поиска цикла минимальной длины, Алгоритм поиска кратчайшего пути и Алгоритм поиска минимального покрывающего дерева (рис. 6.9). От которых, в

свою очередь, будут наследоваться Алгоритмы, реализующие конкретные методы. Алгоритм также связан с Данными и Результатами.

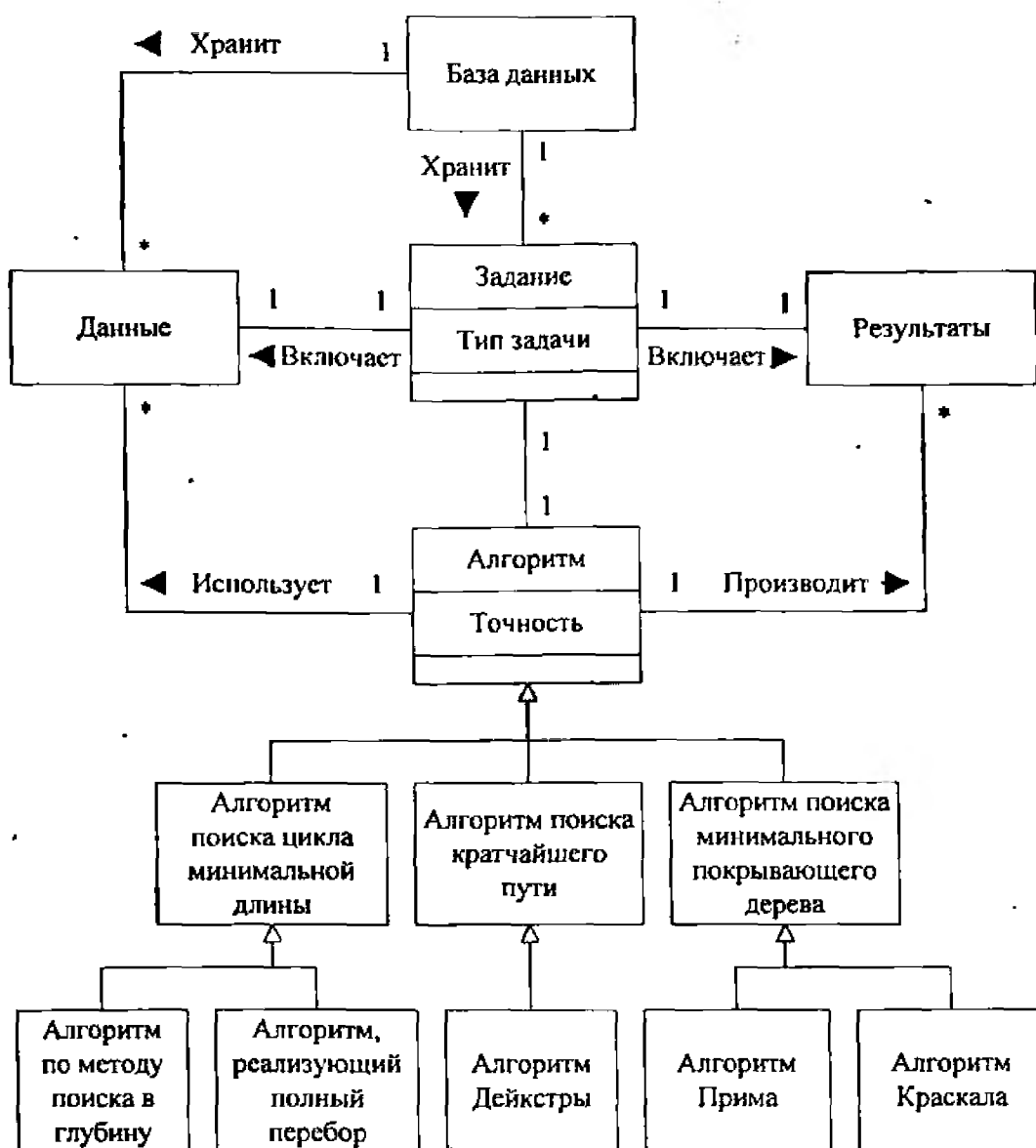


Рис. 6.9. Контекстная диаграмма классов для системы решения комбинаторно-оптимизационных задач

Данные и Задания должны храниться в Базе данных, что показывают ассоциациями соответствующих классов. Способ задания данных для понимания основной концепции проектируемой системы пока не очень существен.

Вид задачи в нашем случае, скорее, атрибут класса Задание, чем самостоятельный класс, так как в реальном мире - это имя, которое позволяет уточнить группу возможных алгоритмов решения, а также структуры исходных данных и получаемых результатов. Для алгоритма очень существенной характеристикой является его точность, соответственно добавим атрибут Точность. Другие атрибуты пока не проявились.

6.4. Описание поведения. Системные события и операции

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 (см. § 7.4).

Диаграмма последовательностей системы. Системные события и операции. *Диаграмма последовательностей системы* — графическая модель, которая для определенного сценария варианта использования показывает генерируемые действующими лицами события и их порядок. При этом система рассматривается как единое целое.

Для построения диаграммы последовательностей системы необходимо:

- представить систему как «черный ящик» и изобразить для нее *линию жизни* - вертикальную пунктирную линию, подходящую к блоку снизу;
- идентифицировать каждое действующее лицо и изобразить для него *линию жизни* (много действующих лиц бывает в вариантах совместного использования программного обеспечения);
- 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.

В отличие от внутренних событий, события, которые генерируются для системы действующими лицами, называют *системными*. Системные события инициируют выполнение соответствующего множества операций, также называемых *системными*. Каждую системную операцию называют по имени соответствующего сообщения.

Множество всех системных операций определяют, идентифицируя системные события всех вариантов использования. Для наглядности системные операции изображают в виде операций абстрактного класса (типа) *System*. Если необходимо разделить множество операций на подмножества, инициируемые разными пользователями, то используют несколько абстрактных классов: *System1*, *System2* и т. д.

Каждую системную операцию необходимо описать. Обычно описание системной операции содержит:

- имя операции и ее параметры;
- описание обязанности;
- указание типа;
- названия вариантов использования, в которых она используется;
- примечания для разработчиков алгоритмов и т. д.;
- описание обработки возможных исключений;
- описание вывода неинтерфейсных сообщений;
- предположение о состоянии системы до выполнения операции (предусловие);
- описание изменения состояния системы после выполнения операции (постусловие).

Пример 6.4. Разработать диаграмму последовательностей системы для варианта использования Выполнение задания решения комбинаторно-оптимизационных задач.

Анализируем описание варианта использования и определяем, что действующее лицо должно инициировать девять системных событий, включая загрузку задания из базы, которая логически следует из операции сохранения. Покажем эти события на диаграмме последовательностей (рис. 6.10). В скобках укажем параметры, которые должны формировать эти события.

Следовательно, система должна обеспечивать выполнение соответствующих операций. Полученное множество операций приписывается классу *System* (рис. 6.11).

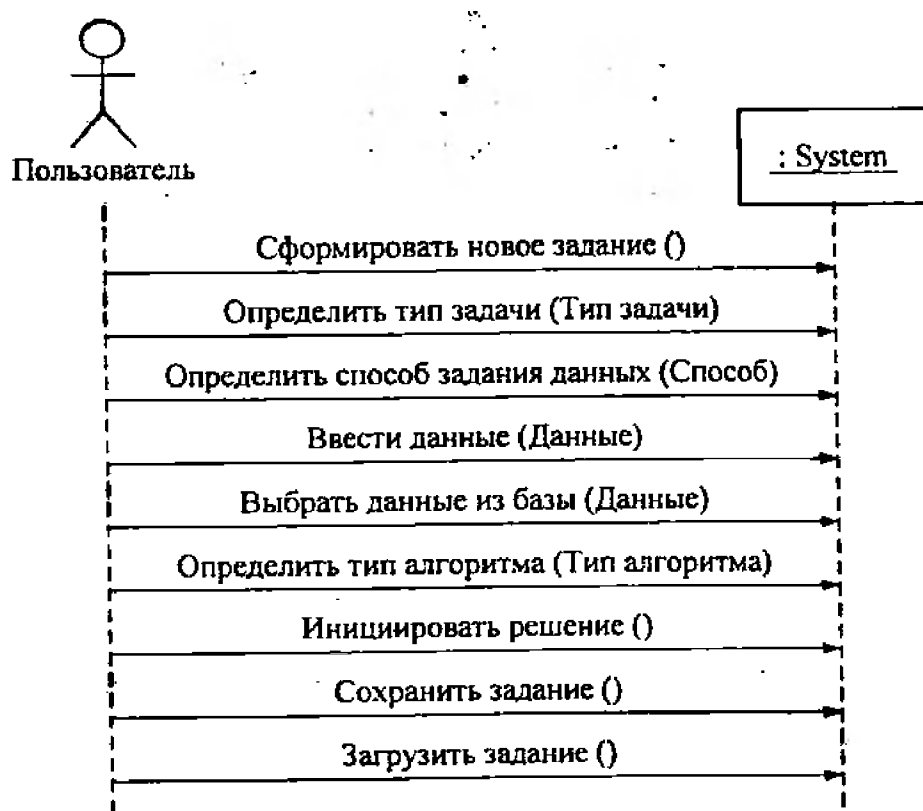


Рис. 6.10. Диаграмма последовательностей системы

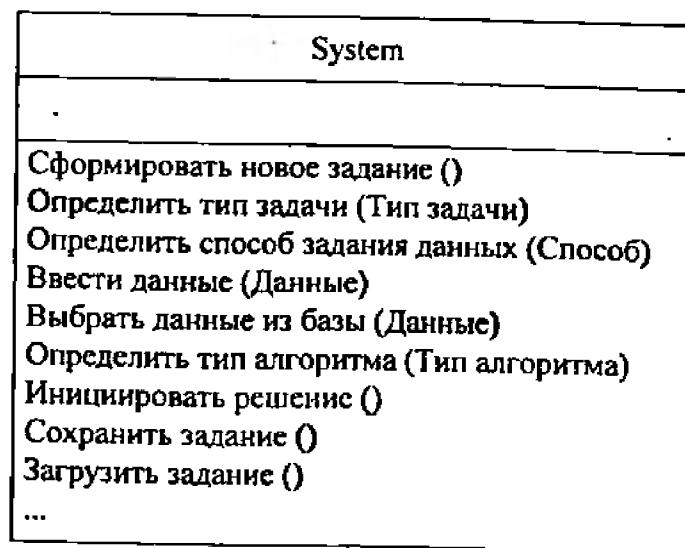


Рис. 6.11. Системные операции

Далее каждую операцию необходимо описать. Для примера опишем операцию Инициировать решение ():

Раздел	Описание
Имя	<i>Инициировать решение ()</i>
Обязанности	<i>Выполнить задание и вывести результаты пользователю</i>
Тип	<i>Системная</i>
Ссылки	<i>Вариант использования Выполнить задание</i>
Примечания	<i>Предусмотреть возможность прерывания процесса решения пользователем</i>
Исключения	<i>1.Если в задании указаны не все исходные данные, то вывести сообщение об ошибке 2.Если при указанных исходных данных решение задачи указанным методом невозможно, то вывести сообщение об ошибке</i>
Вывод	-
Предусловия	<i>Предполагает наличие всех исходных данных задания</i>
Постусловие	<i>Получен результат</i>

Диаграммы деятельности. В зависимости от степени детализации диаграммы деятельности так же, как диаграммы классов, используют на разных этапах разработки. На этапе анализа требований и уточнения спецификаций диаграммы деятельности позволяют конкретизировать основные функции разрабатываемого программного обеспечения.

Под *деятельностью* в данном случае понимают задачу (операцию), которую необходимо выполнить вручную или с помощью средств автоматизации. Каждому варианту использования соответствует своя последовательность задач. В теоретическом плане диаграммы деятельности являются обобщенным представлением алгоритма, реализующего анализируемый вариант использования. На диаграмме деятельность обозначается прямоугольником с закругленными углами (рис. 6.12, а).

Диаграммы деятельности позволяют описывать альтернативные и параллельные процессы. Для обозначения альтернативных процессов используют ромб (рис. 6.12, б), условие указывают над ним слева или справа, а альтернативы «да», «нет» - рядом с соответствующими выходами. С помощью этого же блока можно построить циклический процесс. Множественность активации деятельности обозначают символом «*», помещенным рядом со стрелкой активации деятельности, и при необходимости уточняют надписью вида «для каждой строки».

Для обозначения параллельных процессов используют линейки синхронизации (рис. 6.12, в), причем условие синхронизации можно уточнить, указав его на диаграмме.

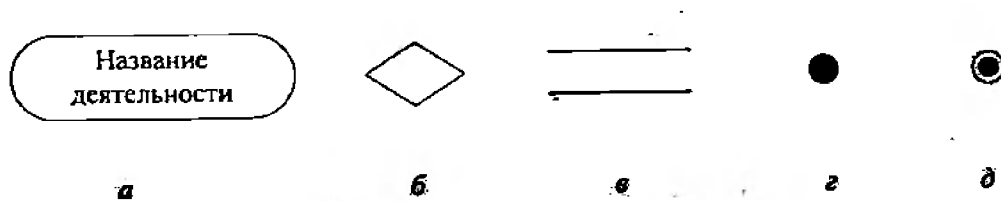


Рис. 6.12. Условные обозначения диаграммы деятельности:

а – деятельность; б – выбор; в – линейки синхронизации; г – начало; д – конец

На рис. 6.13 показано, что «Деятельность 1» и «Деятельность 2» могут выполняться параллельно.

На этапе определения спецификаций имеет смысл уточнять только варианты использования, краткое описание которых недостаточно для понимания сущности решаемых проблем. Диаграммы деятельности, таким образом, можно использовать вместо описания вариантов использования или как дополнение к ним.

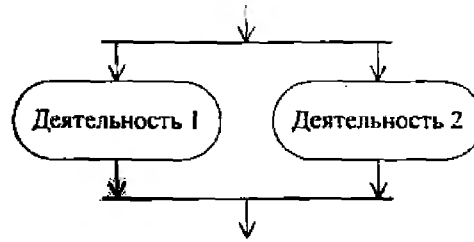


Рис. 6.13. Пример диаграммы деятельности с указанием параллельности процессов

Пример 6.5. Построить диаграмму деятельности, уточняющую вариант использования Выполнение задания системы решения комбинаторно-оптимизационных задач.

Учитывая описание предметной области в виде контекстной диаграммы классов, анализируем описание варианта использования. Разбиваем процесс на отдельные операции. Полученные операции показываем на диаграмме деятельности (рис. 6.14).

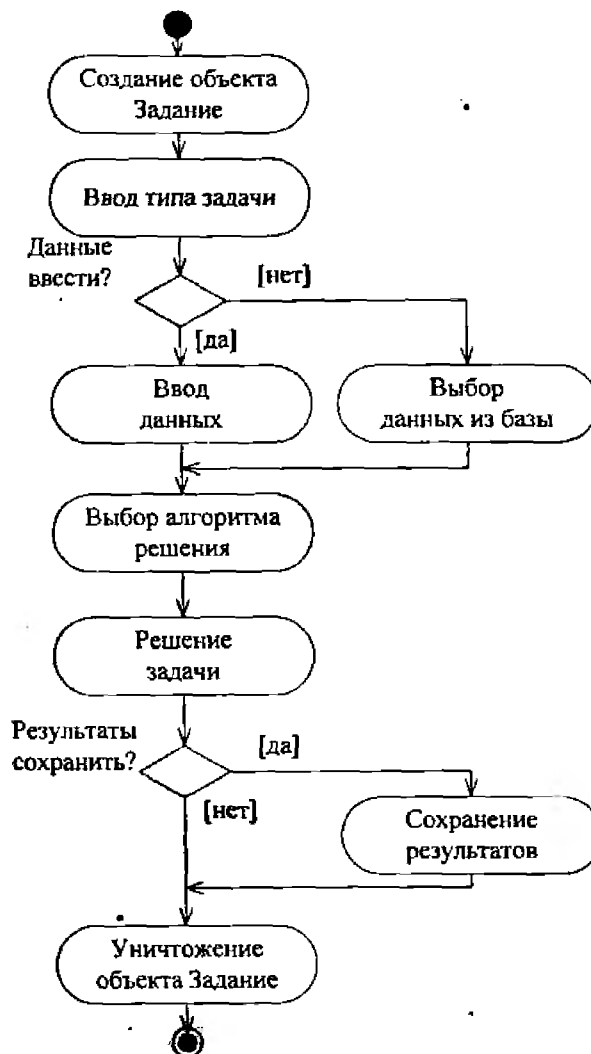


Рис. 6.14. Диаграмма деятельности, уточняющая вариант использования Решение задачи системы решения комбинаторно-оптимизационных задач

Контрольные вопросы и задания

1. В чем сущность объектной декомпозиции?
2. Для чего используют язык UML? Почему его называют языком моделирования? Чем обусловлен выбор именно этого языка в качестве стандарта описания объектных разработок?
3. Какие диаграммы используют в качестве спецификаций программного обеспечения при объектном подходе?
4. Что такое «вариант использования»? Как строится диаграмма вариантов использования, и какую информацию она содержит?
5. Для чего нужны концептуальные модели предметной области? Поясните методику их построения.
6. Какие отношения между основными понятиями предметной области отображают концептуальные модели?
7. Какие диаграммы UML применяют для описания поведения разрабатываемого программного обеспечения?
8. Что понимают под системными событиями и операциями?
9. Разработайте спецификации простейшего графического редактора, использующего векторную графику. Какие диаграммы целесообразно строить в данном случае?

7. ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ПРИ ОБЪЕКТНОМ ПОДХОДЕ

Основной задачей *логического* проектирования при объектном подходе является разработка классов для реализации объектов, полученных при объектной декомпозиции, что предполагает полное описание полей и методов каждого класса.

Физическое проектирование при объектном подходе включает объединение классов и других программных ресурсов в программные компоненты, а также размещение этих компонентов на конкретных вычислительных устройствах.

7.1. Разработка структуры программного обеспечения при объектном подходе

Большинство классов можно отнести к определенному типу, который применительно к данному подходу называют *стереотипам*, например:

- классы-сущности (классы предметной области);
- граничные (интерфейсные) классы;
- управляющие классы;
- исключения и т. д. (рис. 7.1).

Классы-сущности используют для представления сущностей реального мира или внутренних элементов системы, например структур данных. Как правило, они не зависят от окружения, и их используют в различных приложениях. Для выявления классов-сущностей изучают описания вариантов использования, концептуальную модель и диаграммы деятельности. Полученный таким образом список классов-кандидатов фильтруют, удаляя слова, не относящиеся к предметной области, языковые выражения и т. п. Среди оставшихся отбирают классы-кандидаты, объекты которых обладают как состоянием, так и поведением.



Рис. 7.1. Условные обозначения стереотипов классов:

а – класс-сущность; б – граничный класс;
в – управляющий класс; г – явное указание стереотипа

Граничные классы обеспечивают взаимодействие между действующими лицами и внутренними элементами системы. К этому типу относят как классы, реализующие пользовательские интерфейсы, так и классы, обеспечивающие интерфейс с аппаратными средствами или программными системами. Для обнаружения граничных классов изучают пары «действующее лицо - вариант использования».

Управляющие классы служат для моделирования последовательного поведения, заложенного в один или несколько вариантов использования.

Если количество классов-кандидатов и других ресурсов велико, то их целесообразно объединить в группы - пакеты. *Пакетом* при объектном подходе называют совокупность описаний классов и других программных ресурсов, в том числе и самих пакетов. Объединение в пакеты используют *только* для удобства создания больших проектов, количество классов в

которых велико. При этом в один пакет обычно собирают классы и другие ресурсы *единого назначения*.

Диаграмма пакетов показывает, из каких частей состоит проектируемая программная система, и как эти части связаны друг с другом.

Связь между пакетами фиксируют, если изменения в одном пакете могут повлечь за собой изменения в другом. Она определяется внешними связями классов и других ресурсов, объединенных в пакет. Возможны различные виды зависимости классов, например:

- объекты одного класса посылают сообщения объектам другого класса;
- объекты одного класса обращаются к компонентам объектов другого;
- объекты одного класса используют объекты другого в списке параметров методов и т. п.

Самыми хорошими технологическими характеристиками отличается вариант, при котором каждый пакет включает *интерфейс*, содержащий описание всех ресурсов данного пакета, и взаимодействие пакетов осуществляется только через этот интерфейс. Изменения реализации ресурсов пакета в этом случае не затрагивают других пакетов. И только изменения в интерфейсе могут потребовать изменения пакетов, использующих ресурсы данного пакета.

Пакеты, с которыми связаны все пакеты программной системы, называют *глобальными*. Интерфейсы таких пакетов необходимо проектировать особенно тщательно, так как изменения в них потребуют проверки всех пакетов разрабатываемой системы.

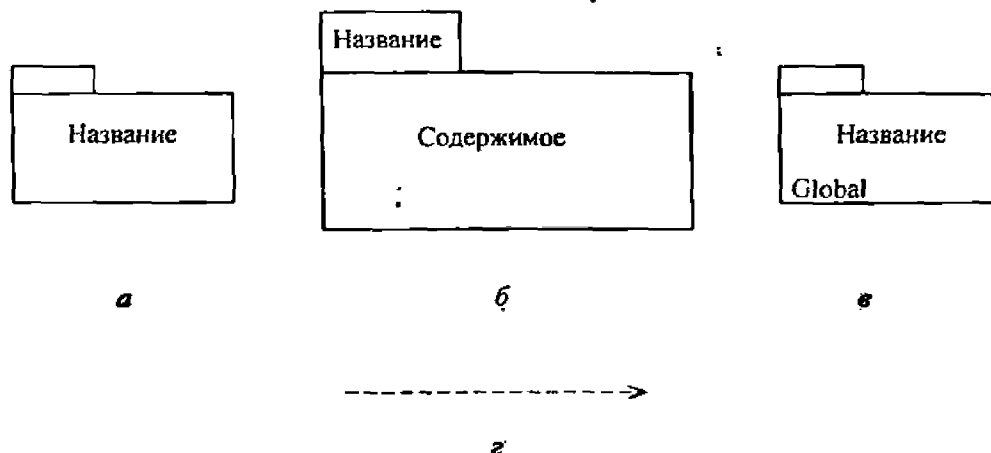


Рис. 7.2. Условные обозначения, применяемые на диаграммах пакетов:

а – пакет, *б* – пакет с обозначением содержимого; *в* – глобальный пакет,
г – зависимость классов (стрелка указывает направление вызовов)

На рис. 7.2 приведены обозначения нотации UML; которые допустимо использовать на диаграммах пакетов. Кроме указанных обозначений на диаграммах пакетов допустимо показывать обобщения (рис. 7.3), что, как правило, подразумевает наличие единого интерфейса нескольких пакетов. В этом случае фиксируется связь от пакета-подтипа к пакету-супертипу.

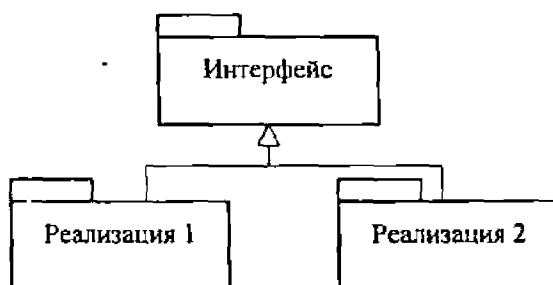


Рис. 7.3. Пример обозначения обобщения пакетов

Пример 7.1. Разработать диаграмму пакетов системы решения комбинаторно-оптимизационных задач.

Анализ концептуальной модели (см. рис. 6.9) и вариантов использования (см. рис. 6.4) позволяют выделить следующие группы классов или пакеты:

- Пользовательский интерфейс - классы, реализующие объекты интерфейса с пользователем;
- Библиотека интерфейсных компонентов — классы, реализующие интерфейсные компоненты: окна, кнопки, метки и т. п.;
- Объекты управления - классы, реализующие сценарии вариантов использования;
- Объекты задачи - классы, реализующие объекты предметной области системы;
- Интерфейс базы данных - классы, реализующие интерфейс с базой данных;
- База данных;
- Базовые структуры данных - классы, реализующие внутренние структуры данных, такие, как деревья, n-связные списки и т. п.;
- Обработка ошибок - классы исключений, реализующие обработку нештатных ситуаций.

Последние два пакета объявим глобальными, так как их элементы могут использовать классы всех пакетов.

Определим зависимости классов и построим диаграмму пакетов (рис. 7.4).

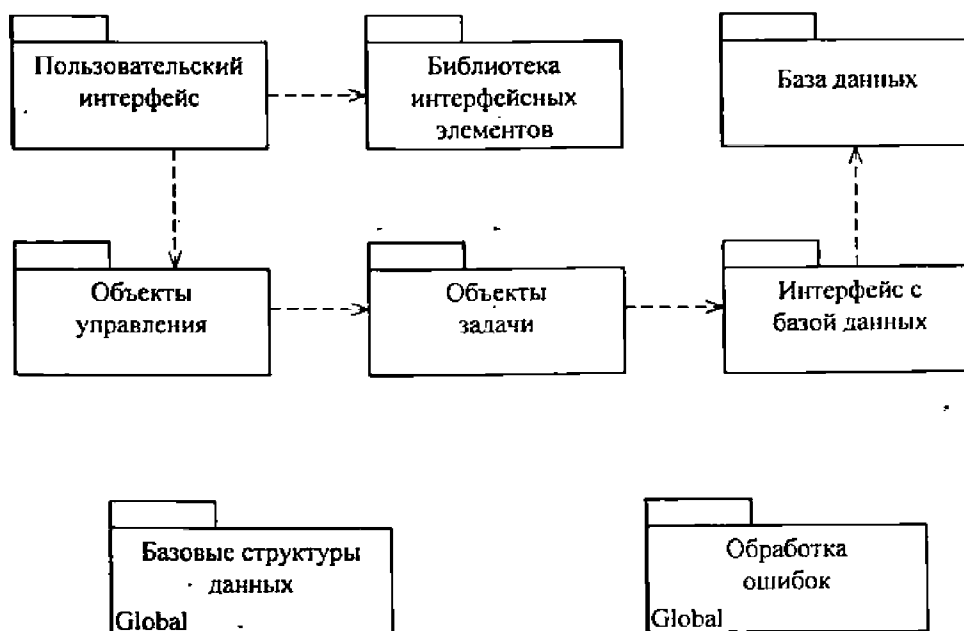


Рис. 7.4. Диаграмма пакетов системы решения комбинаторно-оптимизационных задач

7.2. Определение отношений между объектами

После определения основных пакетов разрабатываемого программного обеспечения переходят к детальному проектированию классов, входящих в каждый пакет. Классы-кандидаты, которые предположительно должны войти в конкретный пакет; показывают на диаграмме классов этапа проектирования и уточняют отношения между объектами указанных классов.

Пример 7.2. Определить классы-кандидаты пакета Объекты задачи.

Используя рекомендации, приведенные в § 7.1, выполним анализ концептуальной модели предметной области (рис. 6.9), описания основного варианта использования Решение задачи (см. § 6.2) и его диаграммы деятельности (см. рис. 6.4).

Список классов-кандидатов, полученный на основе данного анализа, выглядит следующим образом:

- класс **Задание** - объекты данного класса должны создаваться каждый раз, когда пользователь инициирует новое задание;
- семейство классов с базовым классом **Алгоритм** — объекты данного класса должны создаваться, когда определен алгоритм решения задачи;
- класс **Данные** — объекты данного класса должны создаваться при определении (вводе или выборе из базы) данных;
- класс **Результаты** — объекты данного класса должны создаваться при решении конкретной задачи конкретным алгоритмом с использованием конкретных данных.

Исходный вариант диаграммы классов пакета **Объекты задачи** показан на рис. 7.5.

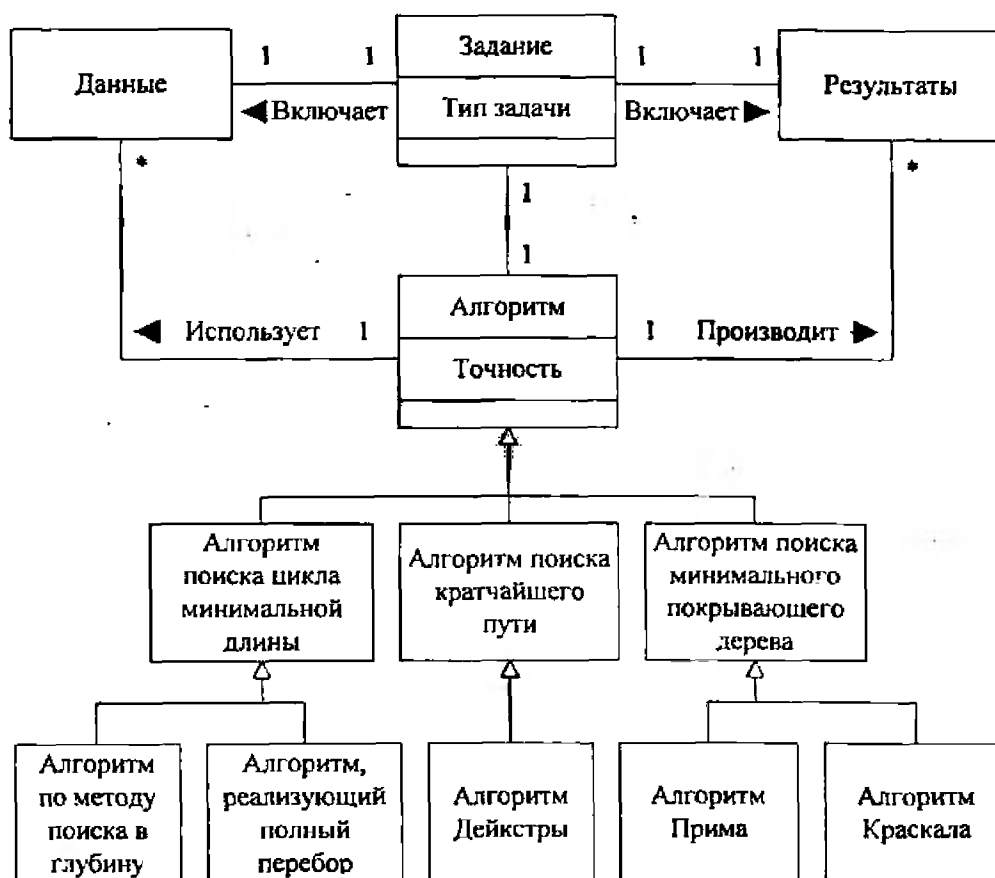


Рис. 7.5. Исходная диаграмма классов пакета **Объекты задачи**

Основой для проектирования классов является уточнение взаимодействия объектов этих классов в процессе реализации вариантов использования. При этом применяют диаграммы последовательностей и диаграммы кооперации. Если же необходимо описать взаимодействие объектов при обработке конкретного сообщения, удобны именно диаграммы последовательностей.

Диаграммы последовательностей этапа проектирования. Диаграммы последовательностей этапа проектирования отображают взаимодействие объектов, упорядоченное по времени. В отличие от диаграмм последовательности этапа анализа на ней показывают внутренние объекты, а также последовательность сообщений, которыми обмениваются объекты в процессе реализации фрагмента варианта использования, называемого сценарием.

Объекты изображают в виде прямоугольников, внутри которого указана информация, идентифицирующая объект: имя, имя объекта и имя класса или только имя класса (рис. 7.6).

Каждое сообщение представляют в виде линии со стрелкой, соединяющей линии жизни двух объектов. Эти линии помещают на диаграмму в порядке генерации сообщений (сверху вниз и

слева направо). Сообщению присваивают имя, но можно указать и аргументы, и управляющую информацию, например, условие формирования или маркер итерации (*). Возврат при передаче *синхронных* сообщений подразумевают по умолчанию.

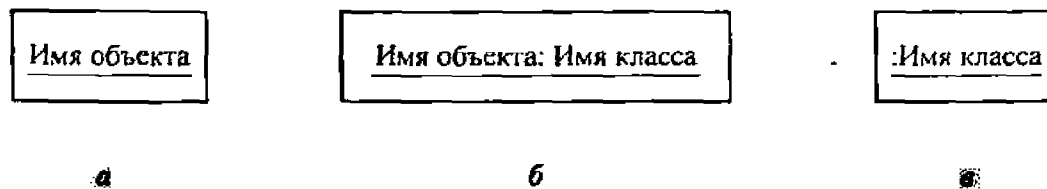


Рис. 7.6. Условные обозначения объектов в UML:

а – объект, *б* – объект с уточнением класса, *в* – неименованный объект указанного класса

Если объект создается сообщением, то его рисуют справа от стрелки сообщения так, чтобы стрелка сообщения входила в него слева.

Диаграммы последовательностей также позволяют изображать параллельные процессы. *Асинхронные* сообщения, которые не блокируют работу вызывающего объекта, показывают половинкой стрелки (рис. 7.7, а). Такие сообщения могут:

- создавать новую ветвь процесса;
- создавать новый объект (рис. 7.7, б);
- устанавливать связь с уже выполняющейся ветвью процесса.

На линии жизни в этом случае дополнительно показывают активации, которые обозначаются прямоугольником, наложенным поверх линии жизни (рис. 7.7, в).

Уничтожение объекта показывают большим знаком «Х» (рис. 7.7, г).

При необходимости линию жизни можно прервать, чтобы не уточнять обработку, не связанную с анализируемыми объектами (рис. 7.7, д).

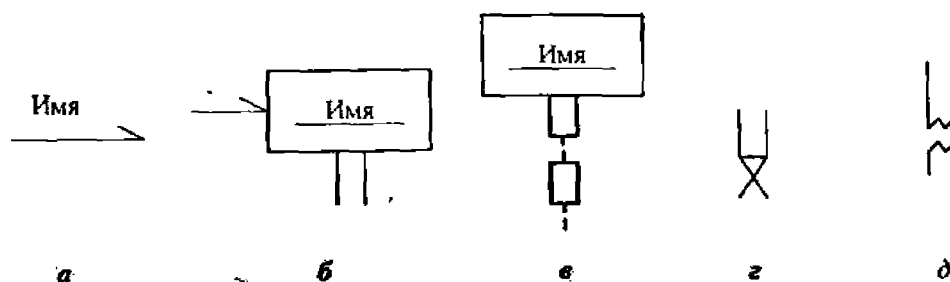


Рис. 7.7. Условные обозначения асинхронных передач управления:

а – асинхронное сообщение; *б* – создание объекта (не обязательно асинхронное);

в – активации объекта; *г* – уничтожение объекта;

д – разрыв (выполнение прочей обработки)

Пример 7.3. Разработать диаграмму последовательностей для сценария Решение задачи (фрагмент варианта использования Выполнение задания от момента инициализации пользователем процесса решения до его завершения).

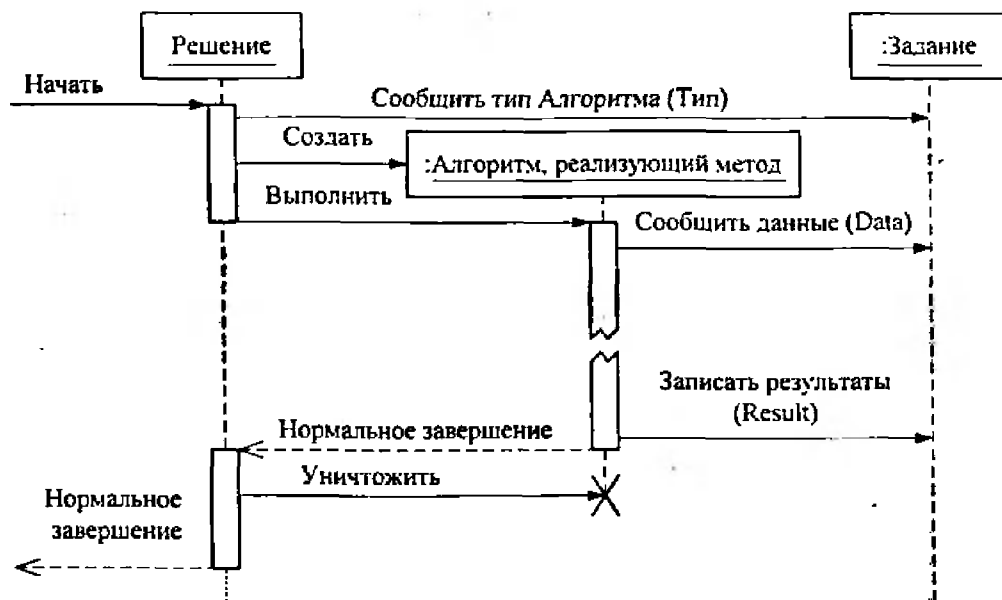
Анализ описания варианта использования показывает, что необходимо рассмотреть три варианта последовательности действий:

- нормальный процесс;
- прерывание процесса пользователем;
- возникновение исключения при выполнении алгоритма.

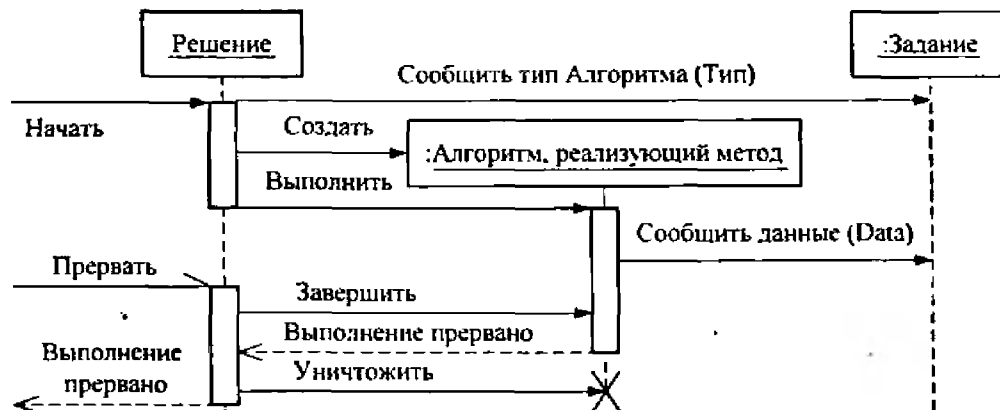
Нормальный процесс предполагает, что при выдаче команды Создать создается объект Решение, управляющий данным сценарием. Следующее сообщение Начать активизирует этот

объект. Объект Решение запрашивает у объекта класса Задание тип объекта Алгоритм, создает объект требуемого класса и активизирует его, сохраняя способность получать и обрабатывать сообщения (параллельный процесс).

Объект класса Алгоритм, реализующий метод, запрашивает у объекта класса Задание данные и начинает обработку, используя вспомогательные объекты. Нормально завершив обработку, объект класса Алгоритм, реализующий метод, передает объекту класса Задание результаты и возвращает объекту Решение признак нормального завершения. Объект Решение уничтожает объект класса Алгоритм, реализующий метод, и возвращает вызвавшему его объекту признак нормального завершения решения (рис. 7.8, а).



а



б

Рис. 7.8. Диаграмма последовательности действий сценария Процесс решения:

а – нормальный процесс; б – прерывание процесса пользователем

В случае прерывания процесса объект Решение прерывает процесс решения, уничтожает объект Алгоритм и возвращает признак прерванного выполнения (рис. 7.8, б). В этом случае при выполнении обработки возникает аварийная ситуация, результатом которой является генерация исключения. Обработывая исключение, объект класса Решение, генерирует соответствующее

сообщение пользователю, уничтожает объект класса Алгоритм, реализующий метод, и возвращает признак завершения выполнения с ошибкой (рис. 7.9).

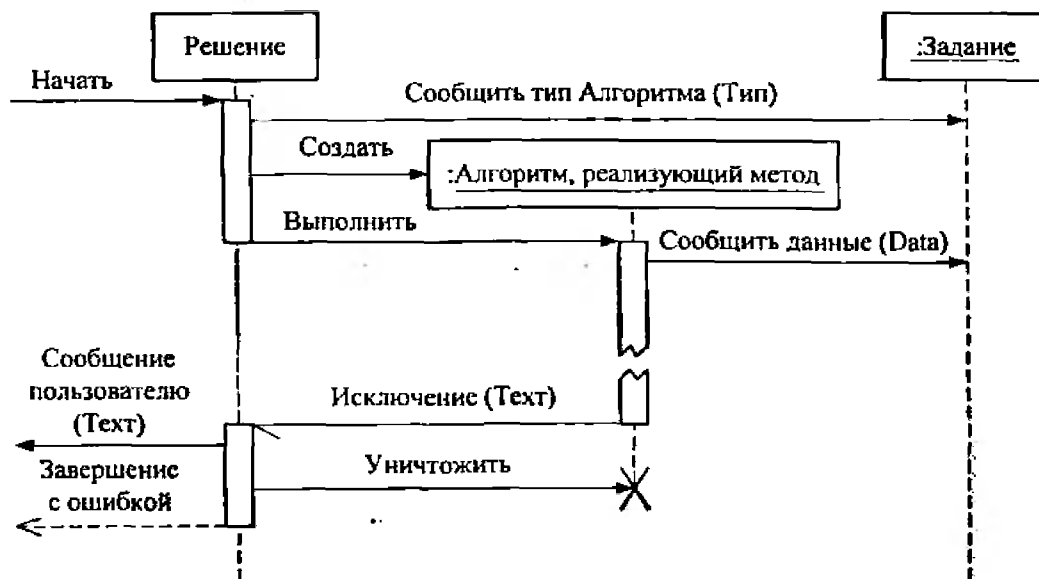


Рис. 7.9. Диаграмма последовательностей сценария Процесс решения для случая возникновения исключения

Диаграмма кооперация. *Диаграмма кооперации* - это альтернативный способ представления взаимодействия объектов в процессе реализации сценария, который позволяет по-другому взглянуть на ту же информацию. В отличие от диаграмм последовательностей диаграммы кооперации показывают потоки данных между объектами классов, что позволяет уточнить связи между ними.

Пример 7.4. Разработать диаграмму кооперации для сценария Процесс решения. Изобразим на Одной диаграмме три возможных случая реализации сценария, нумеруя сообщения в порядке их возможной генерации (рис. 7.10).

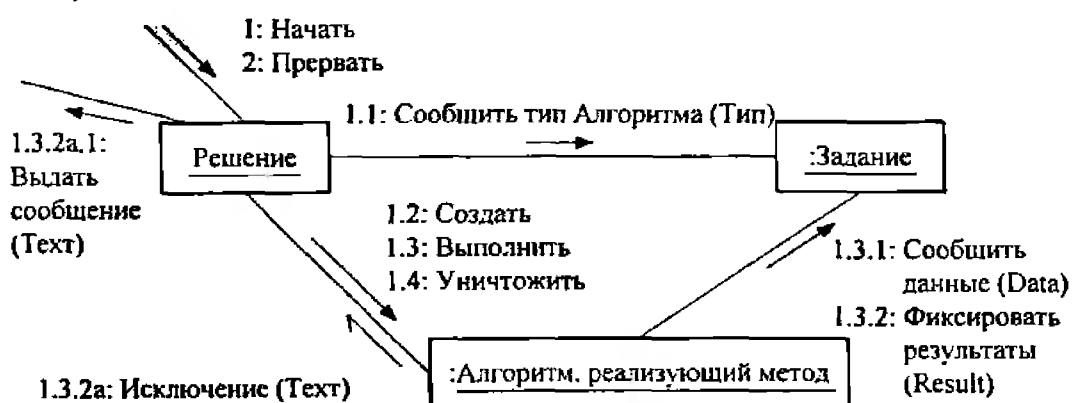


Рис. 7.10. Диаграмма кооперации сценария Процесс решения

Такое представление позволяет описать потоки данных, передаваемых между объектами классов Решение, Задание и Алгоритм, реализующий метод, для сценария Процесс решения.

7.3. Уточнение отношений классов

Процесс проектирования классов начинают с уточнения отношений между ними. На этапе проектирования помимо ассоциации и обобщения различают еще два типа отношения между классами: агрегацию и композицию.

К сожалению, до настоящего времени не существует единой устоявшейся терминологии объектно-ориентированного проектирования. В табл. 7.1 приведены соответствия между основными терминами, используемыми наиболее известными авторами в этой области.

Агрегацией называют ассоциацию между целым и его частью или частями. Агрегацию вместо ассоциации указывают, если отношение «целое-часть» в конкретном случае существенно. Например, если колесо нас интересует только как часть автомобиля, то между соответствующими классами целесообразно указать отношение агрегации, а если колесо - товар, также как и автомобиль, то связь целое-часть не существенна.

Композиция - более сильная разновидность агрегации, которая подразумевает, что объект-часть может принадлежать только единственному целому. Объект-часть при этом создается и уничтожается только вместе со своим целым.

Уточненные отношения между классами фиксируют на диаграмме классов. Для этого используют специальные условные обозначения (рис. 7.11).

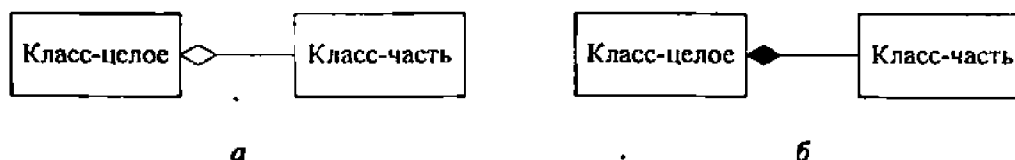


Рис. 7.11. Условные обозначения специальных видов ассоциации:

a – композиция; *б* – агрегация

Поскольку отношение ассоциации и его подвиды (агрегация и композиция) означают наличие обмена сообщениями между объектами классов целесообразно уточнить направление передачи сообщений. *Навигацию* (направление ассоциации) показывают стрелкой на конце линии ассоциации. Если стрелки указаны с обеих сторон, то это означает *двунаправленную* ассоциацию.

Таблица 7.1

Нотация	Термины			
UML	Класс	Ассоциация	Обобщение	Агрегация
Буч	Класс	Использование	Наследование	Включение
Кoad	Класс, объект	Связь экземпляров	Обобщение-специализация	Часть-целое
Якобсон	Объект родства	Ассоциация	Наследование	Состоит из
Оделл	Тип объекта	Связь	Подтип	Композиция
Рамбо	Класс	Ассоциация	Обобщение	Агрегация
Шлеер/Меллор	Объект	Связь	Подтип	Не определена

Специальное обозначение на диаграмме классов этапа проектирования используют для указания абстрактных классов и методов: на диаграмме классов их имена выделяют курсивом, либо перед именем класса указывают стереотип «abstract».

UML также включает специальную нотацию для обозначения *параметризованных классов* или шаблонов (рис. 7.12, а). Получение из такого класса, класса с конкретными типами элементов называют *связыванием*. Связывание можно обозначить двумя способами: явно указав тип параметра (рис. 7.12, б) и используя условное обозначение уточнения (рис. 7.12, в).

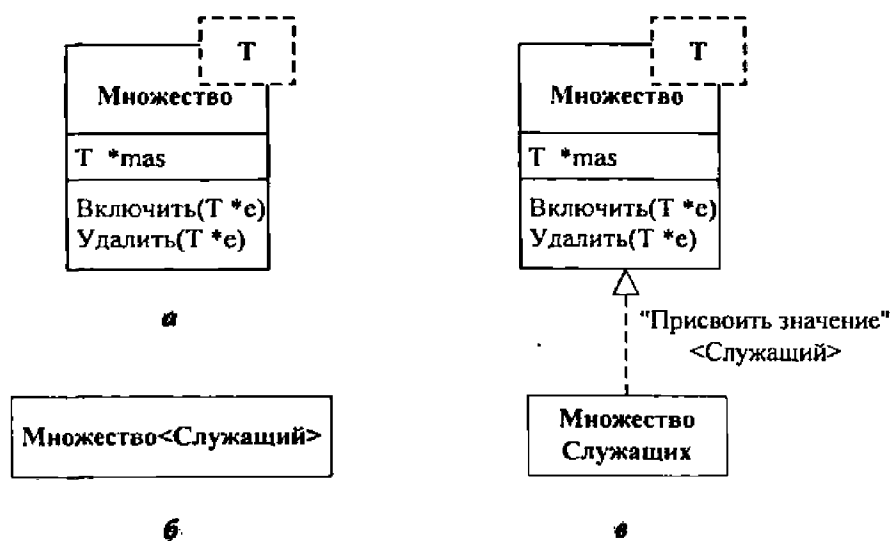


Рис. 7.12. Условное обозначение параметризованного класса:
а – параметризованный класс; б – явное указания типа параметра при связывании;
в – использование уточнения

Диаграммы классов позволяют также отобразить *ограничения*, которые невозможно показать, используя только понятия, рассмотренные выше (ассоциации, обобщения, атрибуты, операции). Например, показать, что средний балл студентов должен определяться по соответствующей формуле. Подобную, информацию на диаграмме классов можно представить в виде записи на естественном языке или в виде математической формулы, поместив их в фигурные скобки.

Особое место в процессе проектирования классов занимает проектирование интерфейсов.

Интерфейсы. *Интерфейсам* в UML называют класс, содержащий только объявление операций. Отдельное описание интерфейсов улучшает технологические качества проектируемого программного обеспечения. Интерфейсы широко применяют при разработке сетевого программного обеспечения, которое должно идентично функционировать в гетерогенных средах, а также для организации взаимодействия с системами управления базами данных и т. п., так как механизм полиморфного наследования позволяет создавать различные реализации одного и того же интерфейса.

С точки зрения теории объектно-ориентированного программирования интерфейс представляет собой особый вид абстрактного класса, отличающийся тем, что он не содержит методов, реализующих указанные операции, и объявлений полей. Другими словами, абстрактные классы позволяют определить реализацию некоторых методов, а интерфейсы требуют отложить определение всех методов.

На диаграмме классов интерфейс можно показать двумя способами: с помощью специального условного обозначения (рис. 7.13, а) или, объявив для класса стереотип «Interface» (рис. 7.13, б).

Реализацию интерфейса также можно показать двумя способами: сокращенно (рис. 7.14, а) или, используя отношение реализации (рис. 7.14, б).

Для остальных классов, ассоциированных с интерфейсом, следует уточнить ассоциацию, показав отношение зависимости. Это отношение в данном случае означает, что класс использует указанный интерфейс (рис. 7.15), т. е. обращается к описанным в интерфейсе функциям.

Одновременно с уточнением отношений классов в пакете следует продумать и отношения классов, включенных в различные пакеты, между собой.

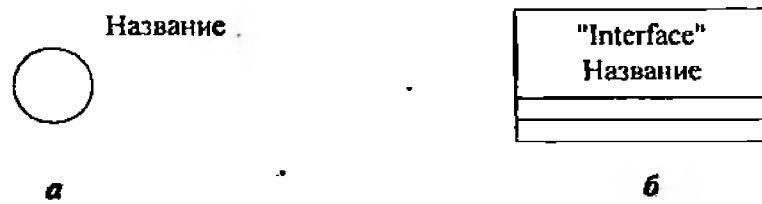


Рис. 7.13. Условные обозначения интерфейса в UML:
а – специальное обозначение; *б* – с указанием стереотипа

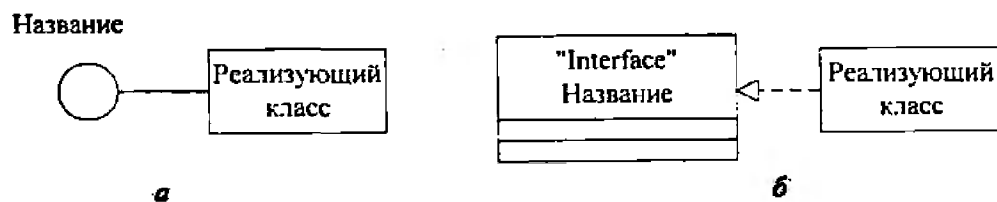


Рис. 7.14. Условные обозначения реализации интерфейсов:
а – сжатая форма; *б* – с указанием отношения реализации

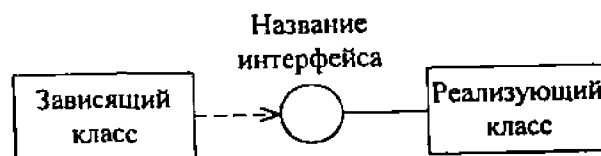


Рис. 7.15. Обозначение зависимости класса от интерфейса

Пример 7.5. Уточнить отношения классов пакета Объекты задачи между собой и с классом Решение из пакета Объекты управления, используя результаты детализации отношений между объектами рассматриваемых классов.

Анализ диаграммы кооперации, представленной на рис. 7.10, показывает, что:

- класс Задание по сути дела представляет собой таблицу, в которой фиксируется вся информация о конкретной задаче: вид задачи, алгоритм решения, данные и результат, причем результат связан с заданием неразрывно, так как теряет смысл вне контекста задания (отношение композиции), а данные имеют смысл сами по себе (отношение агрегации);
- класс Алгоритм целесообразно разрабатывать как абстрактный; этот класс будет описывать интерфейс между объектом класса Решение и конкретным алгоритмом, а также между объектом класса Задание и опять же конкретным алгоритмом;
- отношение между классами Задание и Алгоритм, Решение и Алгоритм, а также Задание и Решение - ассоциации, направленные к классу Задание (рис. 7.16).

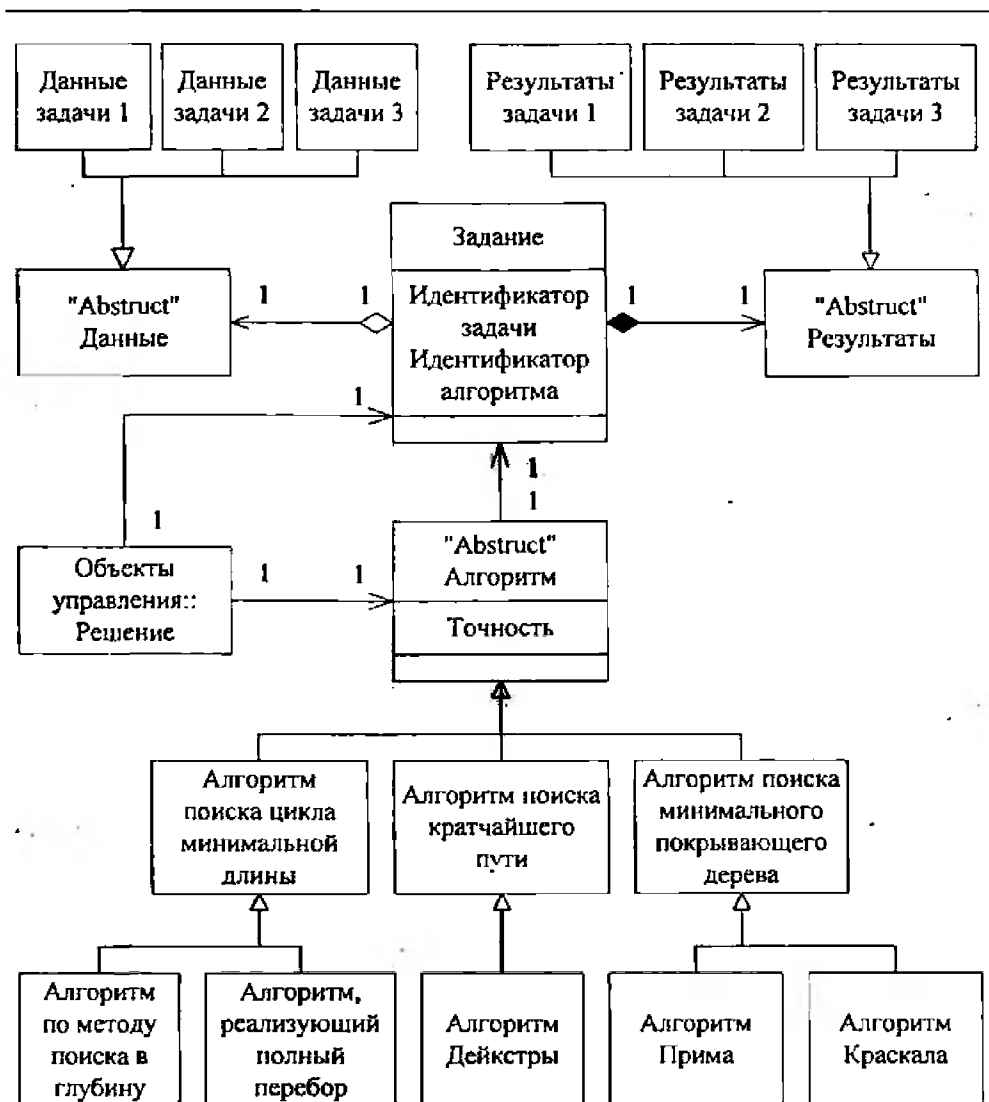


Рис. 7.16. Уточненная диаграмма классов пакета Объекты задачи и класса Решение из пакета Управляющие объекты

Кроме того, анализ структур исходных данных и результатов решаемых задач показывает их существенное различие, следовательно, классы Данные и Результаты также необходимо реализовать как абстрактные и наследовать от них классы, уточняющие структуры данных и результатов для каждого случая. При дальнейшем анализе следует выяснить, будут ли классы Данные и Результаты описывать какие-либо поля или они только определяют интерфейсы, через которые будет осуществляться доступ к данным и результатам конкретных заданий.

На диаграмме классов целесообразно также указать множественность объектов. Поскольку каждый раз решается одна задача с единственными данными, используя конкретный алгоритм, и в результате получают единственное решение, все перечисленные выше ассоциации связывают объекты «один к одному».

7.4. Проектирование классов

Собственно проектирование классов предполагает окончательное определение структуры и поведения его объектов. Структура объектов определяется совокупностью атрибутов и операций

класса. Каждый атрибут - это поле или совокупность полей данных, содержащихся в объекте класса.

Поведение объектов класса определяется реализуемыми *обязанностями*. Обязанности выполняются посредством *операций* класса.

Таким образом, при проектировании класса, помимо имени и максимально полного списка атрибутов, необходимо уточнить его ответственность и операции. Причем как атрибуты, так и операции в процессе проектирования целесообразно дополнительно специфицировать. В зависимости от степени детализации диаграммы классов обозначение атрибута может, помимо имени, включать: тип, описание видимости и значение по умолчанию. Для этого используют следующий формат:

<признак видимости> <имя>:<тип>=<значение по умолчанию>,

где признак видимости может принимать одно из трех значений: «+» - общий; «#» - защищенный; «-» - скрытый.

Как упоминалось выше, операциями называют основные действия, реализуемые классом. В отличие от методов, операции не всегда реализуются классом непосредственно. Например, операция Ввод числа может быть реализована агрегированным интерфейсным элементом «окно ввода».

Полное описание операции на диаграмме класса в UML может выглядеть следующим образом:

**<признак видимости><имя>(<список параметров>):
<тип возвращаемого значения>.**

Ответственностью класса называют кратное неформальное перечисление основных функций объектов класса. Ответственность класса обычно определяют на начальных этапах проектирования, когда атрибуты и операции класса еще не определены. Эту информацию отображают на диаграмме классов в специальных секциях условного изображения класса (рис. 7.17).

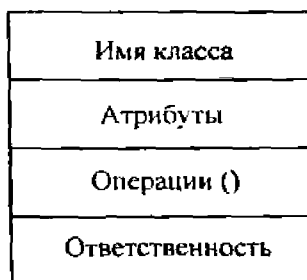


Рис. 7.17. Полное условное обозначение класса в UML

Исходный список операций класса формируют, анализируя диаграммы деятельности, диаграммы взаимодействия и диаграммы последовательностей действий, построенные для различных сценариев с участием объектов проектируемого класса. На начальных этапах проектирования в секции операций класса обычно указывают лишь имена основных операций, определяющих наиболее общее поведение объектов соответствующих классов. По мере уточнения добавляют новые операции, а информацию об уже имеющихся операциях детализируют.

Большинство атрибутов выявляется при анализе предметной области, требований технического задания и описаний потоков событий.

Кроме того, как указывалось выше, отношение ассоциации и его подвиды - агрегация и композиция - означают наличие обмена сообщениями между объектами классов. Для организации

передачи сообщений необходимо, чтобы генерирующий сообщения объект содержал информацию о вызываемом объекте, что означает наличие у этого объекта соответствующего указателя. Причем при отношении композиции объекты-части могут быть организованы как объектные поля объекта-целого.

В том случае, если объекты проектируемого класса должны реализовывать сложное поведение, для них целесообразно разрабатывать диаграммы состояний.

Диаграммы состояний объекта. Под *состоянием* объекта применительно к диаграмме состояний понимают ситуацию в жизненном цикле объекта, во время которой он: удовлетворяет некоторому условию, осуществляет определенную деятельность или ожидает некоторого события. Изменение состояния, связанное с нарушением условия или, соответственно, завершением деятельности или наступлением события называют переходом.

Диаграммы состояний показывают состояния объекта, возможные переходы, а также события или сообщения, вызывающие каждый переход.

Условные обозначения состояний приведены на рис. 7.18. Действие, указанное после слова **Вход**, выполняется при входе в состояние, а действие, указанное после слова **Выход** - при выходе из него. Деятельность связывается с нахождением в состоянии.

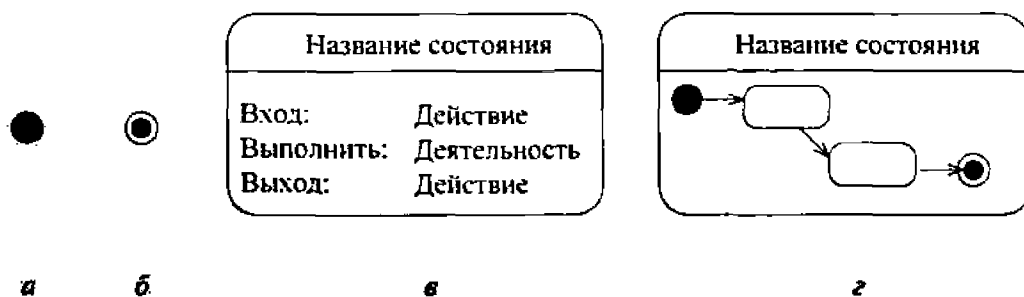


Рис. 7.18. Основные условные обозначения диаграммы состояний объекта:

а – начальное состояние; *б* – конечное состояние; *в* – промежуточное состояние;

г – суперсостояние

Переход обозначается линией со стрелкой и может быть помечен меткой, состоящей из трех частей, каждую из которых можно опустить:

<Событие> [<Условие>**]/<Действие>.**

Если *событие* не указано, то это означает, что переход выполняется по завершению деятельности, связанной с данным состоянием. Если же оно указано - то при наступлении события.

Условие записывается в виде логического выражения. Переход происходит, если результат выражения — «истина». Объект не может одновременно перейти в два разных состояния, поэтому условия перехода для любого события должны быть взаимоисключающими.

В отличие от деятельностей, *действия*, указанные для перехода, связывают с последним и рассматривают как мгновенные и непрерываемые.

При необходимости можно определять суперсостояния (рис. 7.18, г), которые объединяют несколько состояний в одно. Этот механизм обычно используют, чтобы показать переход из нескольких состояний в одно и то же состояние, например, при отмене каких-либо действий.

Пример 7.6. Разработать диаграмму состояний для объекта класса Решение.

Диаграмму состояний объекта строим, анализируя соответствующие диаграммы последовательности действий (см. рис. 7.8 и 7.9). При этом необходимо уточнить, в какой момент разрешить прерывание процесса извне. Чтобы показать, что прерывание процесса возможно еще во время его инициализации, вводим суперсостояние Процесс. При реализации следует учесть возможность прерывания процесса до активации Алгоритма (рис. 7.19).

Результаты уточнения структуры и поведения объектов классов отразим на диаграмме классов.

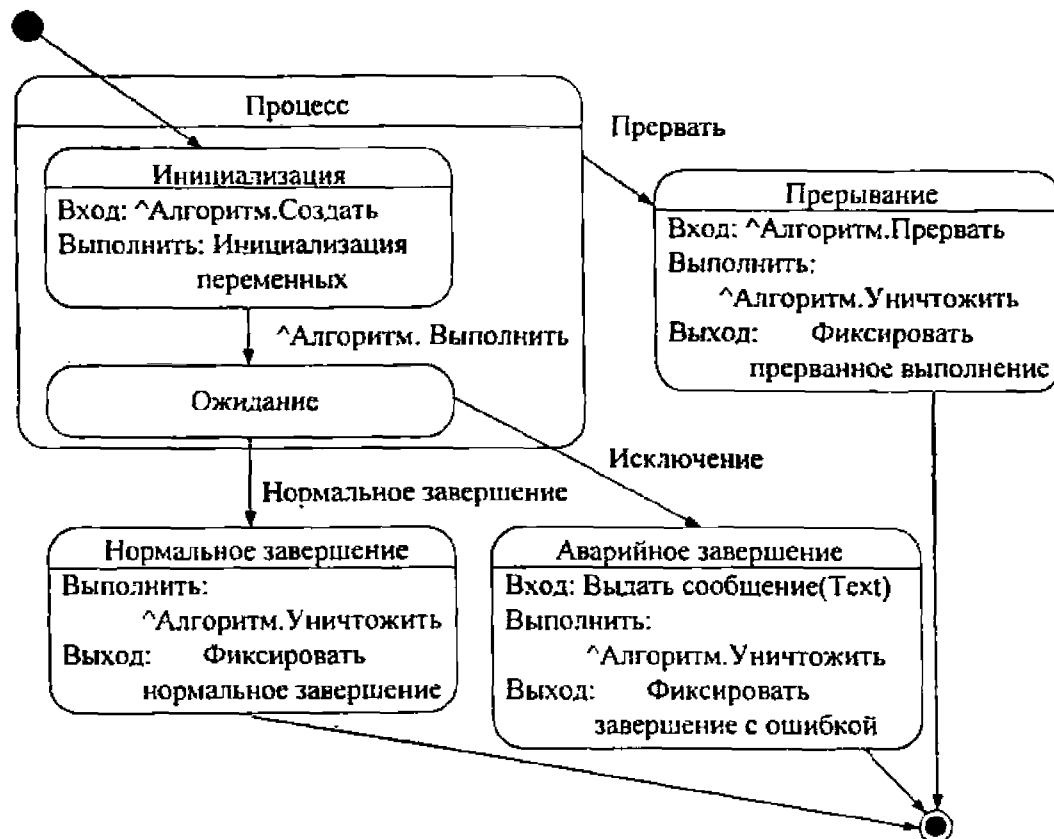


Рис. 7.19. Диаграмма состояний объекта класса Решение

Пример 7.7. Уточнить атрибуты и операции классов Решение, Задание, Алгоритм, Данные и Результаты, используя полученные в данном параграфе сведения.

К л а с с З а д а н и е. Поскольку объект класса Задание должен хранить идентификатор задачи и тип алгоритма, то он должен иметь соответствующие поля и включать операции Определить задач (), Определить алгоритм (), Сообщить тип задачи (), Сообщить тип алгоритма ().

Кроме того, объект класса Задание отвечает за объекты классов Данные и Результаты, связанные с ним, следовательно, он должен хранить их адреса и выполнять операции: Определить данные (), Сообщить данные (), Фиксировать результаты (), Сообщить результаты ().

К л а с с ы Д а н н ы е и Р е з у л ь т а т ы. Данные задач и их результаты должны храниться в базе данных, но они имеют различные структуры. Эту проблему можно решить, если хранить и данные, и результаты в упакованном виде, распаковывая их по мере надобности. Значит, соответствующие классы должны объявлять абстрактные операции Упаковать () и Распаковать (), которые будут реализовываться классами-подтипами в зависимости от реальной структуры данных, определяемой типом задачи. Следовательно, указанные классы должны также хранить тип задачи, с которой они связаны, и содержать операции Определить тип задачи () и Сообщить тип Задачи ().

К л а с с А л г о р и т м. Объекты класса Алгоритм отвечают за реализацию метода решения задачи. Поскольку они посылают сообщение объектам класса Задания, то, естественно, должны хранить его адрес. Кроме того, класс Алгоритм должен объявлять абстрактную операцию Выполнить (). Эта операция должна переопределяться классами Алгоритм, реализующий метод.

Примечание. Возможно более удачное решение: в классе Алгоритм определить операцию Выполнить () и внутреннюю абстрактную операцию Реализовать метод (), которая вызывается из первой. Такое решение позволит не дублировать общее поведение всех алгоритмов, например, запрос и распаковку данных, а также упаковку и запись результата. Это решение не приведено, чтобы не усложнять и так достаточно сложный пример.

К л а с с Р е ш е н и е. Объект класса Решение обращается к объектам классов Задание и Алгоритм, следовательно, необходимо хранить их адреса. Операции Начать () и Прервать () получены из диаграмм последовательностей действий. Операция Обработать исключение () получена отсюда же, но она должна реализовываться особым образом, так как будет получать управление через механизм исключений. Результаты уточнения приведены на рис. 7.20.



Рис. 7.20. Результат уточнения атрибутов и операций классов Задание, Решение, Алгоритм, Данные и Результаты

Проектирование методов класса. Достаточно существенную информацию о действиях, которые должны, выполняться методами класса, можно получить, анализируя диаграммы последовательности действий. Однако алгоритмы всех сколько-нибудь сложных методов необходимо проработать детально. При этом можно использовать как уже известные нотации (схемы алгоритмов и псевдокоды), так и диаграммы деятельности.

В § 6.5 были описаны диаграммы деятельности, которые предлагалось использовать в процессе уточнения спецификаций для описания вариантов использования. Эти же диаграммы могут использоваться и при проектировании методов обработки сообщений, в том числе и затрагивающих несколько объектов. В последнем случае целесообразно указать вертикальными пунктирными линиями ответственности объектов соответствующих классов, что позволит проследить вызовы других объектов.

Следует помнить, что в соответствии с общими правилами процедурной декомпозиции *любую деятельность можно декомпозировать и изобразить в виде диаграммы деятельности более низкого уровня.*

Пример 7.8. Построить диаграмму деятельности для операции Начать () класса Решение. Анализ рис. 6.4, 7.7 - 7.8 показывает, что данная деятельность затрагивает три объекта уже детализированных классов Решение, Алгоритм и Задание. Определим зоны ответственности объектов этих классов (рис.7.21):



Рис. 7.21. Диаграмма деятельности для операции Решение^. Начать()

- объект класса Решения организует обработку, т. е. инициализирует переменные (в том числе определяет тип Алгоритма), создает объект класса Алгоритм требуемого типа, активизируют обработку, а затем уничтожает объект класса Алгоритм;
- объект класса Задание должен в ответ на запрос сообщить тип Алгоритма, предоставить данные и запомнить результаты;
- объект класса Алгоритм отвечает за решение задачи.

Полностью спроектированные классы реализуют на конкретном языке программирования.

7.5. Компоновка программных компонентов

Диаграммы компонентов применяют при проектировании физической структуры разрабатываемого программного обеспечения. Эти диаграммы показывают, как выглядит программное обеспечение на физическом уровне, т. е. из каких частей оно состоит и как эти части связаны между собой.

Диаграммы компонентов оперируют понятиями компонент и зависимость. Под *компонентами* при этом понимают физические заменяемые части программного обеспечения, которые соответствуют некоторому набору интерфейсов и обеспечивают их реализацию. По сути дела, это отдельные файлы различных типов: исполняемые (.exe), текстовые, графические, таблицы баз данных и т. п., составляющие разрабатываемое программное обеспечение. Условные графические обозначения компонентов различных типов приведены на рис. 7.22.

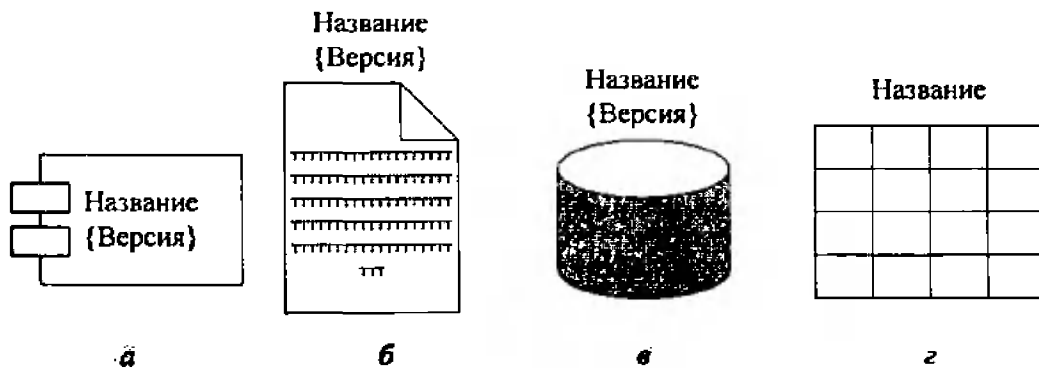


Рис. 7.22. Условные обозначения компонентов в UML:

a – программный компонент; б – файл; в – база данных; z – таблица базы данных

Зависимость между компонентами фиксируют, если один компонент содержит некоторый ресурс (модуль, объект, класс и т. д.), а другой - его использует. Качество компоновки оценивают по количеству и типу связей между компонентами, т. е. по степени независимости компонентов. На диаграмме компонентов зависимость обозначают пунктиром со стрелкой на конце.

На рис. 7.23 в качестве примера приведена диаграмма компонентов системы решения комбинаторно-оптимизационных задач.

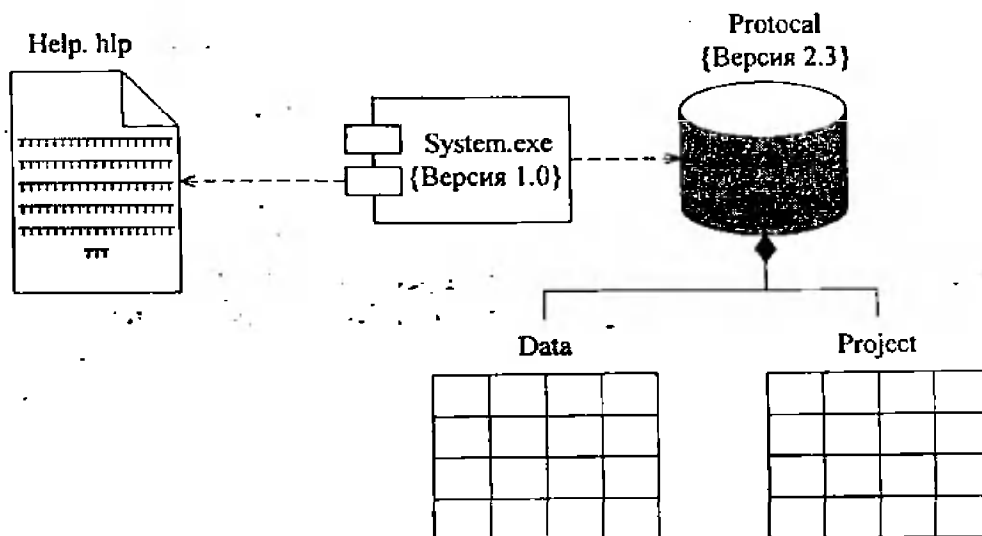


Рис. 7.23. Диаграммы компонентов системы решения комбинаторно-оптимизационных задач

При компонентном подходе на диаграмме компонентов целесообразно показывать интерфейсы, через которые компоненты связаны между собой (рис. 7.24).

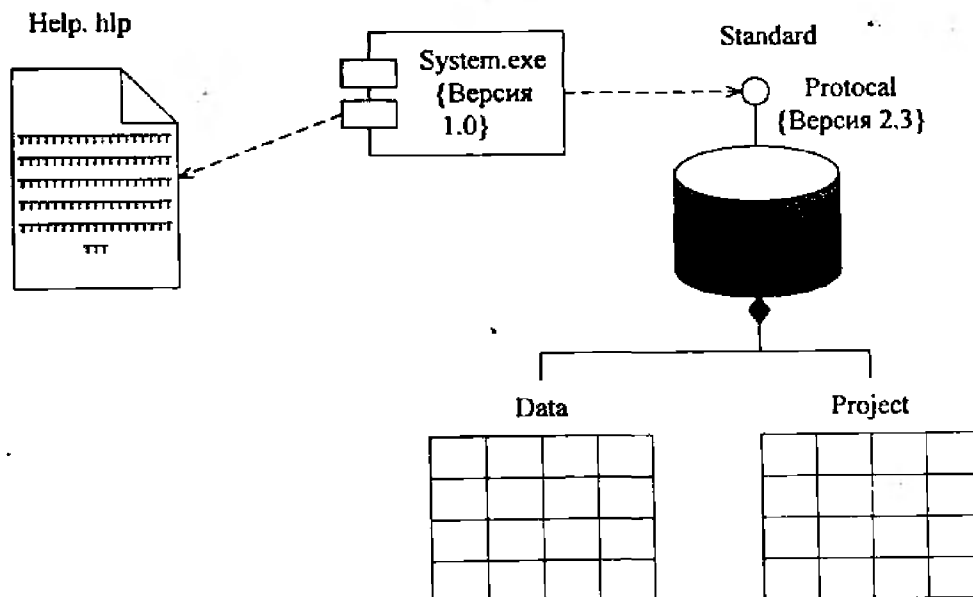


Рис. 7.24. Диаграмма компонентов системы решения комбинаторно-оптимизационных задач с указанием интерфейса

Кроме этого, на диаграмме компонентов допустимо уточнять зависимость между компонентами, используя обозначения обобщения, ассоциации, композиции, агрегатирования и реализации. Так, на рис. 7.23 и 7.24 показано, что база данных включает (отношение композиции) две таблицы.

Используя нотацию UML, можно, построить диаграмму компонентов практически для любого случая, например, для Интернет-приложения. На рис. 7.25 приведен пример диаграммы компонентов клиентской части Интернет-приложения, написанного с использованием Java, которое в процессе работы демонстрирует некоторый рисунок.

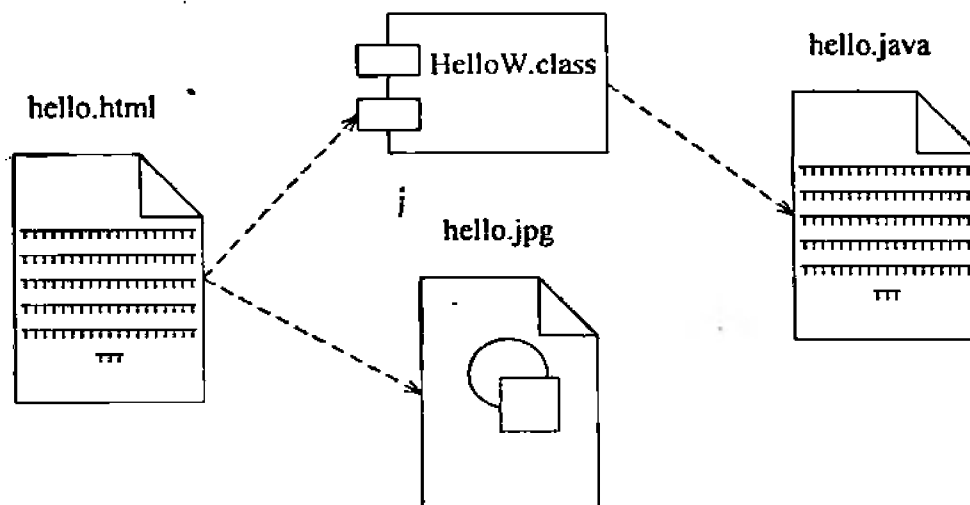


Рис. 7.25. Диаграмма компоновки Internet-приложения, которое выводит фотографию и текст «Hello World»

При «сборке» исполняемых файлов диаграммы компонентов применяют для отображения взаимосвязей файлов, содержащих исходный код. Так, на рис. 7.26 показано, что основной файл Main.cpp зависит от заголовочного файла Model.h, реализация которого находится в файле Model.cpp.

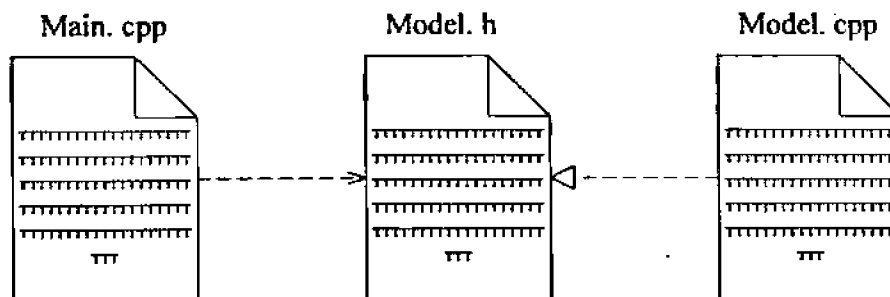


Рис. 7.26. Пример диаграммы компоновки исполняемого файла C++

Для программного обеспечения с архитектурой «клиент-сервер», диаграмму компонентов можно использовать в качестве структурной схемы, определяющей архитектуру разрабатываемого программного обеспечения, так как она позволяет показать связи по управлению частями системы (компонентов). Однако при проектировании такую схему необходимо уточнить, показав более подробно состав компонентов разрабатываемой системы.

7.6. Проектирование размещения программных компонентов для распределенных программных систем

При физическом проектировании распределенных программных систем необходимо определить наиболее оптимальный вариант размещения программных компонентов на реальном оборудовании в локальной или глобальной сетях. Для этого используют специальную модель UML - диаграмму размещения.

Диаграмма размещения отражает физические взаимосвязи между программными и аппаратными компонентами системы. Каждой части аппаратных средств системы, например, компьютеру или датчику, на диаграмме размещения соответствует узел. *Соединения узлов* означают наличие в системе соответствующих коммуникационных каналов. Внутри узлов указывают размещенные на данном оборудовании программные компоненты разрабатываемой программной системы, сохраняя указанные на диаграмме компонентов отношения зависимости.

С точки зрения диаграммы размещения локальная и глобальная сети - это тоже узлы, которые обладают некоторой спецификой. На рис. 7.27 показаны условные обозначения узлов (процессора и устройства) на диаграмме размещения.

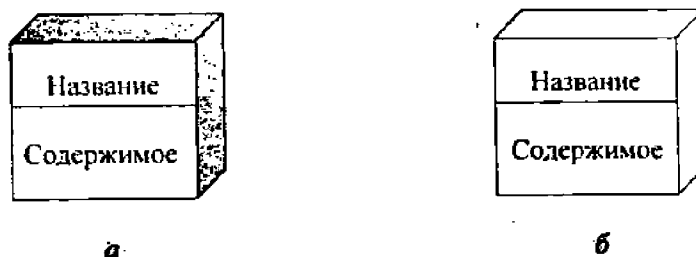


Рис. 7.27. Условные обозначения диаграммы размещения:

а – процессор (компьютер). б – устройство

Пример 7.9. Разработать диаграмму размещения для системы учета успеваемости студентов.

Локальная сеть деканата связывает сервер деканата и компьютеры декана, его заместителей и сотрудников деканата, отвечающих за занесение информации в базу данных. Серверную часть системы и базу данных целесообразно поместить на сервер деканата. На компьютерах локальной сети в этом случае будут функционировать соответствующие клиентские части приложения (рис. 7.28).

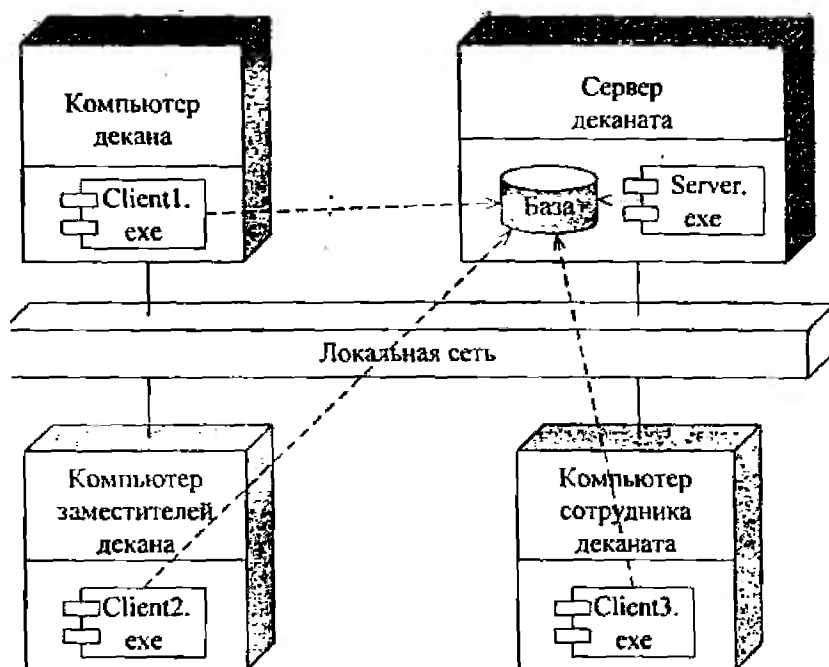


Рис. 7.28. Диаграмма размещения в локальной сети системы учета успеваемости студентов

7.7. Особенность спиральной модели разработки. Реорганизация проекта

Спиральная модель жизненного цикла разработки программного обеспечения предполагает, что процесс разработки программной системы выполняется итерационно. При этом во время проектирования очередной версии может обнаружиться, что хорошо продуманная на начальных итерациях программа утратила свою изначально четкую структуру за счет накопившихся исправлений («заплаток»).

Каждая «заплата» в свое время ставилась, чтобы ликвидировать «небольшое» несоответствие уже реализованной части и той, что находилась в процессе реализации. Возможно также, что часть кодов при этом становилась неиспользуемой, а какие-то коды - не эффективными. Все это вместе приводит к тому, что разобраться в программе становится достаточно сложно. В такой ситуации необходима *реорганизация* программ, т. е. их *перепроектирование без изменения функциональности*. Своевременно выполненная реорганизация позволит сделать структуру программы опять четкой и понятной.

Несмотря на то, что реорганизационные изменения, как правило, невелики, программисты их обычно не любят, так как придерживаются правила «работает — не трогай». Кроме того, сроки обычно ограничены, и тратить время на переделку того, что уже отлажено, кажется нецелесообразным. Практика же показывает, что отказ от реорганизации при накоплении исправлений приводит к усложнению программы и, соответственно, снижению ее технологичности со всеми вытекающими последствиями.

Реорганизацию следует выполнять, если при расширении функциональности обнаруживаются нарушения *основных концепций* проекта или код стал трудным для понимания. В этом случае проектирование нужно приостановить и провести необходимую реорганизацию.

Перепроектирование программы не следует совмещать с увеличением ее функциональности. После реорганизации программу необходимо протестировать, чтобы убедиться, что ничего не было нарушено, а потом уже расширять ее возможности.

Контрольные вопросы и задания

1. Как описывают структуру программного обеспечения при объектном подходе? Что такое «пакет»? Для чего используют диаграммы пакетов?
2. Какие стереотипы классов введены и почему?
3. Разработайте диаграмму пакетов графического редактора, описанного вами при выполнении задания 9 к гл. 6. Какие пакеты включены в эту диаграмму и почему? Какие пакеты будут связаны между собой?
4. Постройте диаграмму последовательности действий для объектов любых предложенных вами пакетов. Какими сообщениями обмениваются объекты? Какую информацию программист получит; анализируя эту диаграмму?
5. Какую диаграмму используют при уточнении взаимодействия объектов? Постройте эту диаграмму для объектов предыдущего задания.
6. Перечислите основные компоненты классов. Как описывают эти компоненты?
7. В каких случаях используют диаграммы состояний объекта? Постройте диаграмму состояний для любого управляющего объекта.
8. Постройте уточненную диаграмму классов по результатам исследования взаимодействия объектов. Какая еще информация необходима для реализации этих классов?
9. Что понимают под диаграммой компонентов? Какую информацию она содержит? В каких случаях целесообразно строить диаграммы компонентов?
10. Какую информацию содержит диаграмма размещения? В каких случаях целесообразно использовать эти диаграммы?

8. РАЗРАБОТКА ПОЛЬЗОВАТЕЛЬСКИХ ИНТЕРФЕЙСОВ

На ранних этапах развития вычислительной техники пользовательский интерфейс рассматривался как средство общения человека с операционной системой и был достаточно примитивным. В основном он позволял запустить задание на выполнение, связать с ним конкретные данные и выполнить некоторые процедуры обслуживания вычислительной установки.

Со временем по мере совершенствования аппаратных средств появилась возможность создания интерактивного программного обеспечения, использующего специальные пользовательские интерфейсы. В настоящее время основной проблемой является разработка интерактивных интерфейсов к сложным программным продуктам, рассчитанным на использование непрофессиональными пользователями. В последние годы были сформулированы основные концепции построения таких пользовательских интерфейсов и предложено несколько методик их создания.

8.1. Типы пользовательских интерфейсов и этапы их разработки

Пользовательский интерфейс представляет собой совокупность программных и аппаратных средств, обеспечивающих взаимодействие пользователя с компьютером. Основу такого взаимодействия составляют диалоги. Под *диалогом* в данном случае понимают *регламентированный* обмен информацией между человеком и компьютером, осуществляемый в реальном масштабе времени и направленный на совместное решение конкретной задачи: обмен информацией и координация действий [35]. Каждый диалог состоит из отдельных процессов ввода-вывода, которые физически обеспечивают связь пользователя и компьютера.

Обмен информацией осуществляется передачей сообщений и управляющих сигналов. *Сообщение* - порция информации, участвующая в диалоговом обмене. Различают:

- входные сообщения, которые генерируются человеком с помощью средств ввода: клавиатуры, манипуляторов, например мыши и т. п.;
- выходные сообщения, которые генерируются компьютером в виде текстов, звуковых сигналов и/или изображений и выводятся пользователю на экран монитора или другие устройства вывода информации (рис. 8.1).

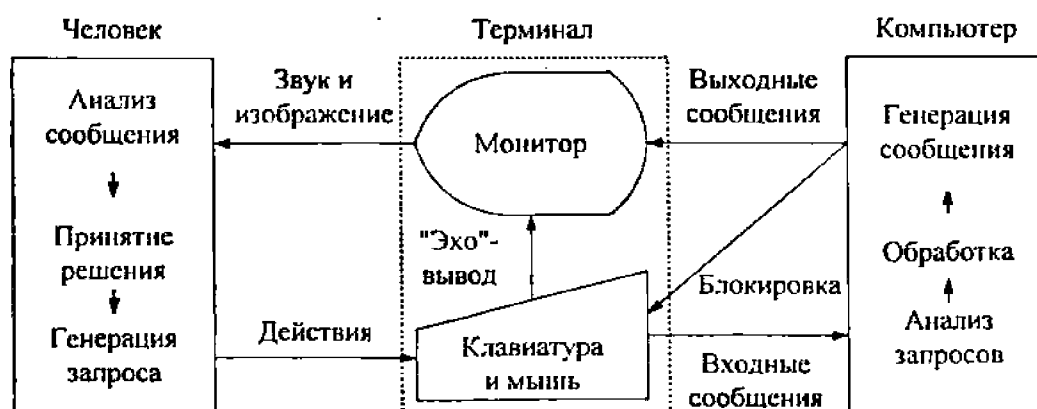


Рис. 8.1. Организация взаимодействия компьютера и пользователя

В основном пользователь генерирует сообщения следующих типов: запрос информации, запрос помощи, запрос операции или функции, ввод или изменение информации, выбор поля

кадра и т. д. В ответ он получает: подсказки или справки, информационные сообщения, не требующие ответа, приказы, требующие действий, сообщения об ошибках, нуждающиеся в ответных действиях, изменение формата кадра и т. д.

Ниже перечислены основные устройства, обеспечивающие выполнение операций ввода-вывода.

Для вывода сообщений:

- монохромные и цветные мониторы - вывод оперативной текстовой и графической информации;
- принтеры - получение «твердой копии» текстовой и графической информации;
- графопостроители - получение твердой копии графической информации;
- синтезаторы речи - речевой вывод;
- звукогенераторы - вывод музыки и т. п.

Для ввода сообщений:

- клавиатура - текстовый ввод;
- планшеты - графический ввод;
- сканеры - графический ввод;
- манипуляторы, световое перо, сенсорный экран - позиционирование и выбор информации на экране и т. п.

Типы интерфейсов. По аналогии с процедурным и объектным подходом к программированию различают процедурно-ориентированный и объектно-ориентированный подходы к разработке интерфейсов (рис. 8.2).

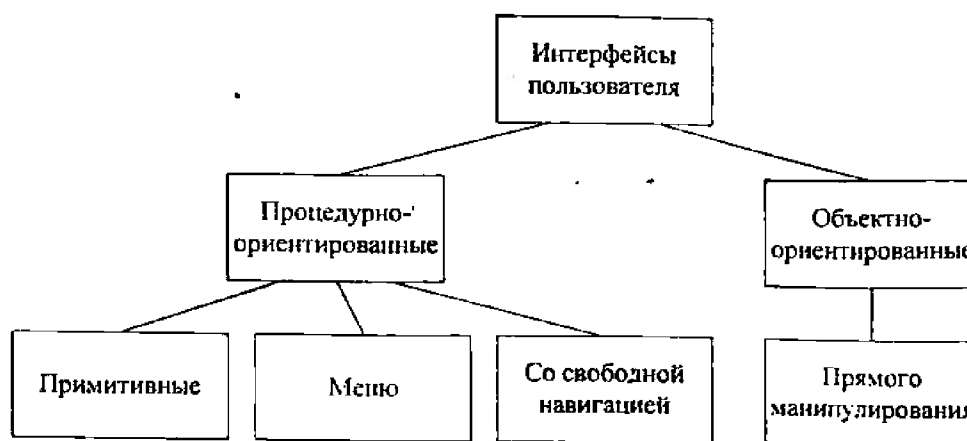


Рис. 8.2. Типы интерфейсов

Процедурно-ориентированные интерфейсы используют традиционную модель взаимодействия с пользователем, основанную на понятиях «процедура» и «операция». В рамках этой модели программное обеспечение предоставляет пользователю возможность выполнения некоторых *действий*, для которых пользователь определяет соответствующие данные и следствием выполнения которых является получение желаемых результатов.

Объектно-ориентированные интерфейсы используют несколько иную модель взаимодействия с пользователем, ориентированную на манипулирование *объектами* предметной области. В рамках этой модели пользователю предоставляется возможность напрямую взаимодействовать с каждым объектом и инициировать выполнение операций, в процессе которых взаимодействуют несколько объектов. Задача пользователя формулируется как целенаправленное изменение некоторого объекта, имеющего внутреннюю структуру, определенное содержание и внешнее символическое или графическое представление. Объект при этом понимается в широком смысле слова, например, модель реальной системы или процесса, база данных, текст и т. п. Пользователю предоставляется возможность создавать объекты, изменять их параметры и связи с другими объектами, а также инициировать взаимодействие этих объектов. Элементы интерфейсов данного

типа включены в пользовательский интерфейс Windows, например, пользователь может «взять» файл и «переместить» его в другую папку. Таким образом, он инициирует выполнение операции перемещения файла.

Применение процедурно-ориентированных интерфейсов в данном случае не означает использования структурного подхода к разработке соответствующего программного обеспечения. Более того, реализация современного процедурно-ориентированного пользовательского интерфейса на базе структурного подхода является очень сложной и трудоемкой задачей.

Таблица 8.1

Процедурно-ориентированные пользовательские интерфейсы	Объектно-ориентированные пользовательские интерфейсы
Обеспечивают пользователей функциями, необходимыми для выполнения задач	Обеспечивают пользователям возможность взаимодействия с объектами
Акцент делается на задачи	Акцент делается на входные данные и результаты
Пиктограммы представляют приложения, окна или операции	Пиктограммы представляют объекты
Содержание папок и справочников отображается с помощью таблиц и списков	Папки и справочники являются визуальными контейнерами объектов

В табл. 8.1 перечислены основные отличия пользовательских моделей интерфейсов процедурного и объектно-ориентированного типов.

Различают **п р о ц е д у р н о – о р и е н т и р о в а н н ы е** интерфейсы трех типов: «примитивные», меню и со свободной навигацией.

Примитивным называют интерфейс, который организует взаимодействие с пользователем в консольном режиме. Обычно такой интерфейс реализует конкретный сценарий работы программного обеспечения, например: ввод данных - решение задачи - вывод результата (рис. 8.3, а). Единственное отклонение от последовательного процесса, которое обеспечивается данным интерфейсом, заключается в организации цикла для обработки нескольких наборов данных (рис. 8.3, б). Подобные интерфейсы в настоящее время используют только в процессе обучения программированию или в тех случаях, когда вся программа реализует одну функцию, например, в некоторых системных утилитах.

Интерфейс-меню в отличие от примитивного интерфейса позволяет пользователю выбирать необходимые операции из специального списка, выводимого ему программой. Эти интерфейсы предполагают реализацию множества сценариев работы, последовательность действий в которых определяется пользователем.

Различают одноуровневые и иерархические меню. Первые используют для сравнительно простого управления вычислительным процессом, когда вариантов немного (не более 5-7), и они включают операции одного типа, например, Создать, Открыть, Закрыть и т. п. Вторые - при большом количестве вариантов или их очевидных различиях, например, операции с файлами и операции с данными, хранящимися в этих файлах.

Интерфейсы данного типа несложно реализовать в рамках структурного подхода к программированию. На рис. 8.4 показана типичная структура алгоритма программы, организующей одноуровневое меню.

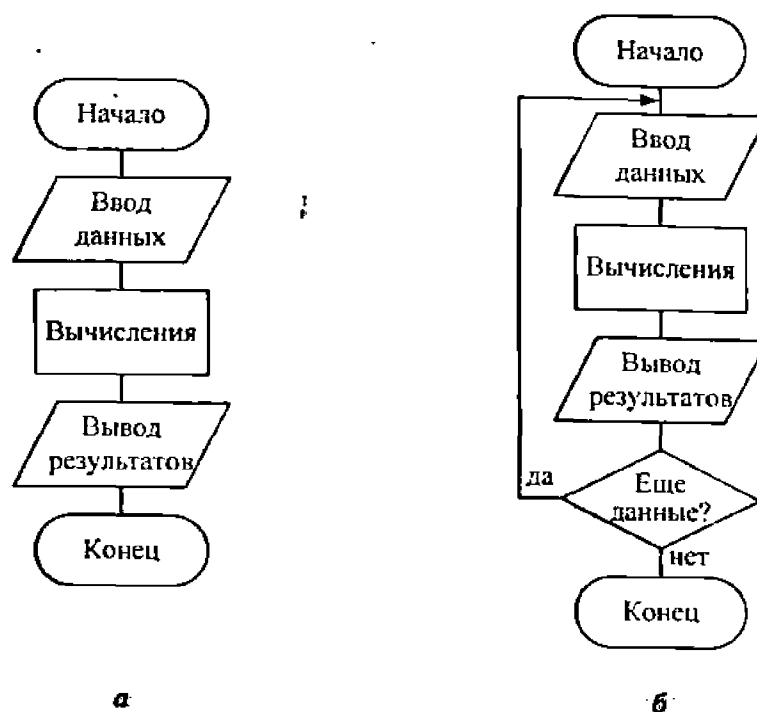


Рис. 8.3. Типичная структура алгоритма программ с примитивным интерфейсом:

а – последовательный; *б* – с возможностью повторения

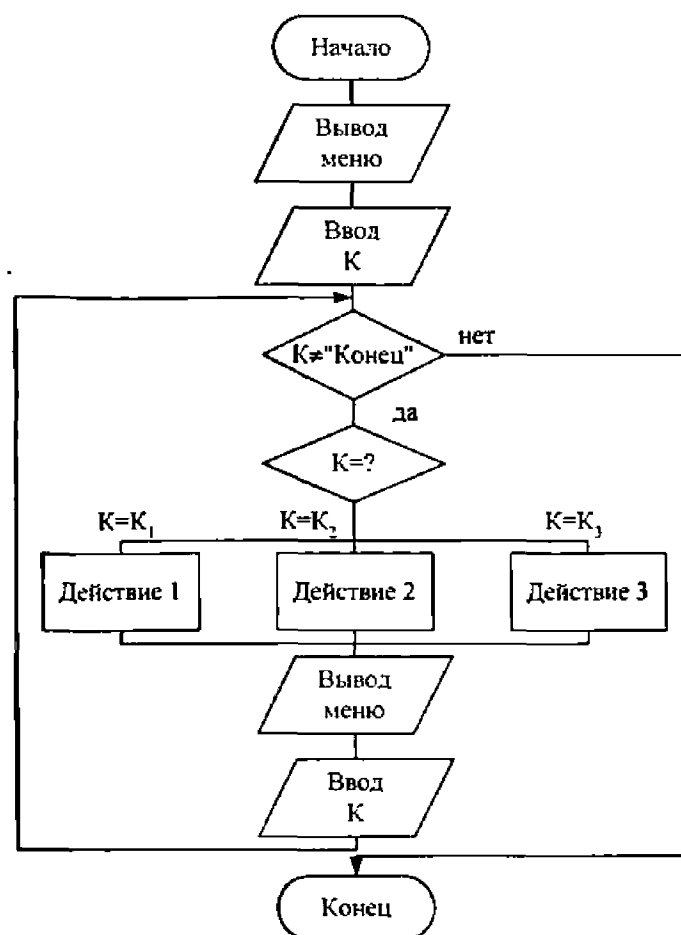


Рис. 8.4. Типичная структура алгоритма программы с одноуровневым меню

Алгоритм программы с многоуровневым меню обычно строится по уровням, причем выбор команды на каждом уровне осуществляется так же, как для одноуровневого меню.

Интерфейс-меню предполагает, что программа находится либо в состоянии Уровень меню, либо в состоянии Выполнение операции. В состоянии Уровень меню осуществляется вывод меню соответствующего уровня и выбор нужного пункта меню, а в состоянии Выполнение операции реализуется сценарий выбранной операции. В порядке исключения иногда пользователю предоставляется возможность завершения операции независимо от стадии выполнения сценария и/или программы, например, по нажатию клавиши Esc.

Древовидная организация меню предполагает строго ограниченную навигацию: либо переходы «вверх» к корню дерева, либо - «вниз» по выбранной ветви. Каждому уровню иерархического меню соответствует свое определенное окно, содержащее пункты данного уровня. При этом возможны два варианта реализации меню: каждое окно меню занимает весь экран или на экране одновременно присутствуют несколько меню разных уровней. Во втором случае окна меню появляются при выборе пунктов соответствующего верхнего уровня - «выпадающие» меню.

В условиях ограниченной навигации независимо от варианта реализации поиск требуемого пункта более чем двухуровневого меню может оказаться непростой задачей.

Интерфейсы-меню в настоящее время также используют редко и только для сравнительно простого программного обеспечения или в разработках, которые должны быть выполнены по структурной технологии и без использования специальных библиотек.

Интерфейсы со свободной навигацией также называют *графическими пользовательскими интерфейсами* (GUI - Graphic User Interface) или интерфейсами WYSIWYG (What You See Is What You Get - что видишь, то и получишь, т. е., что пользователь видит на экране, то он и получит при печати). Эти названия подчеркивают, что интерфейсы данного типа ориентированы на использование экрана в графическом режиме с высокой разрешающей способностью.

Графические интерфейсы поддерживают концепцию интерактивного взаимодействия с программным обеспечением, осуществляя визуальную обратную связь с пользователем и возможность прямого манипулирования объектами и информацией на экране. Кроме того, интерфейсы данного типа поддерживают концепцию совместимости программ, позволяя перемещать между ними информацию (технология OLE, см. § 1.1).

В отличие от интерфейса-меню интерфейс со свободной навигацией обеспечивает возможность осуществления любых допустимых в конкретном состоянии операций, доступ к которым возможен через различные интерфейсные компоненты. Например, окна программ, реализующих интерфейс Windows, обычно содержат:

- меню различных типов: ниспадающее, кнопочное, контекстное;
- разного рода компоненты ввода данных.

Причем выбор следующей операции в меню осуществляется как мышью, так и с помощью клавиатуры.

Существенной особенностью интерфейсов данного типа является способность изменяться в процессе взаимодействия с пользователем, предлагая выбор только тех операций, которые имеют смысл в конкретной ситуации. Реализуют интерфейсы со свободной навигацией, используя событийное программирование и объектно-ориентированные библиотеки, что предполагает применение визуальных сред разработки программного обеспечения.

Объектно-ориентированные интерфейсы пока представлены только *интерфейсом прямого манипулирования*. Этот тип интерфейса предполагает, что взаимодействие пользователя с программным обеспечением осуществляется посредством выбора и перемещения *пиктограмм*, соответствующих объектам предметной области. Для реализации таких интерфейсов также используют событийное программирование и объектно-ориентированные библиотеки.

Сравним четыре указанных типа интерфейсов на конкретном несложном примере.

Пример 8.1. Разработать пользовательский интерфейс программы построения графиков или вывода таблицы функций, техническое задание на которую представлено в § 3.5.

Можно предложить четыре варианта интерфейса, соответствующие рассмотренным выше типам.

Вариант 1. Использование примитивного интерфейса предполагает, что пользователь сразу определяет все параметры, необходимые программе для построения графика или вывода таблицы, вводя их в ответ на соответствующие запросы программы, после чего программа выполняет необходимые вычисления и выводит результат. Если допустить, что программа будет запрашивать подтверждения завершения обработки, то процесс построения графиков/таблиц можно зациклить. В зависимости от используемых средств мы получим сравнительно простую программу, удовлетворяющую функциональным спецификациям, но ориентированную на единственный сценарий: ввод - обработка - вывод (см. рис. 8.3, б). Данный вариант не удобен для пользователя.

Вариант 2. Можно использовать одноуровневое меню, которое будет включать команды: Функция, Отрезок, Шаг, Тип результата, Выполнить и Выход. При выборе первого пункта меню определяется функция, второго - интервал, третьего - шаг, четвертого - тип результата, пятого - осуществляется операция, и, наконец, последний пункт обеспечивает возможность выхода из программы (рис. 8.5).

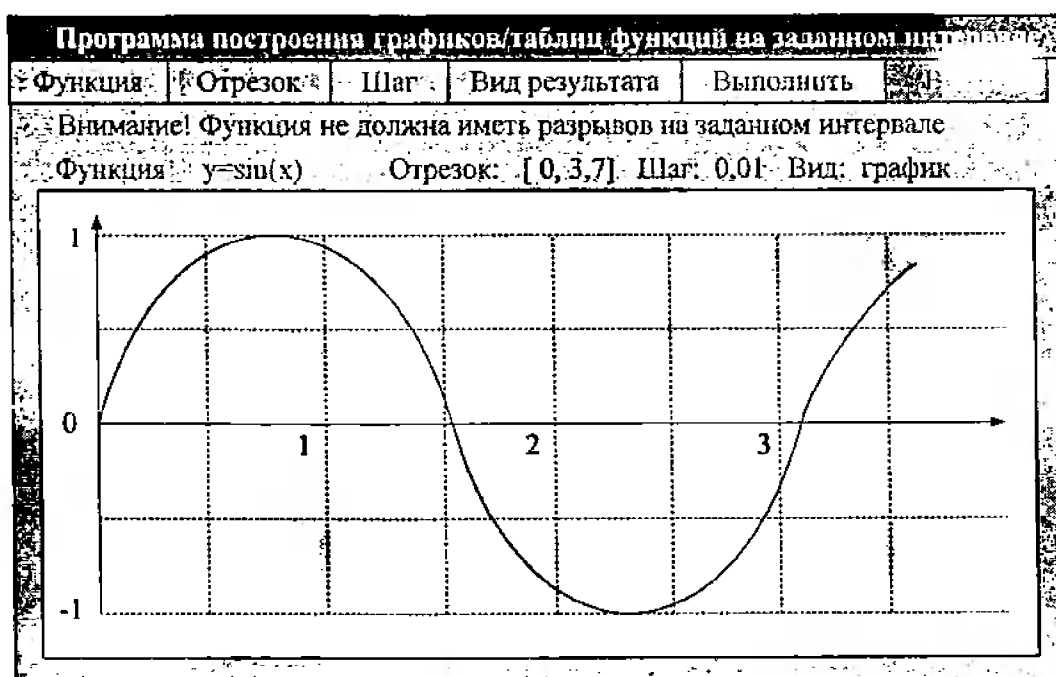


Рис. 8.5. Внешний вид экрана программы построения графиков/таблиц функций (интерфейс – меню)

Очевидно, что в этом случае обеспечивается более гибкое управление для пользователя, так как фактически предусмотрены следующие сценарии работы:

Ввод функции - Ввод отрезка - Ввод шага - Уточнение вида результата: график/таблица - Вывод результата;

Изменение отрезка - Вывод результата;

Изменение шага - Вывод результата;

Изменение вида результата: график/таблица - Вывод результата и др.

Вариант 3. Интерфейс со свободной навигацией для данной программы представлен на рис. 8.6. График строится по нажатию кнопки Построить (естественно, обработчик этого события должен предусматривать анализ данных на полноту и совместимость). Менять данные можно в любой момент и в любом порядке, используя соответствующие компоненты ввода-вывода.

Вариант 4. Интерфейс прямого манипулирования для данной программы представлен на рис. 8.7. Для того чтобы ввести новую формулу, необходимо взять чистый бланк из папки. Бланк раскрывается двойным щелчком мыши, после чего его необходимо заполнить. Затем его можно

«обсчитать», перенеся на пиктограмму компьютера. Заполненные бланки, которые могут еще понадобиться, «кладутся» в папку Функции, остальные - в «корзину».

Менять данные и тип результатов можно в любой момент и в любом порядке, «раскрыв» бланк.

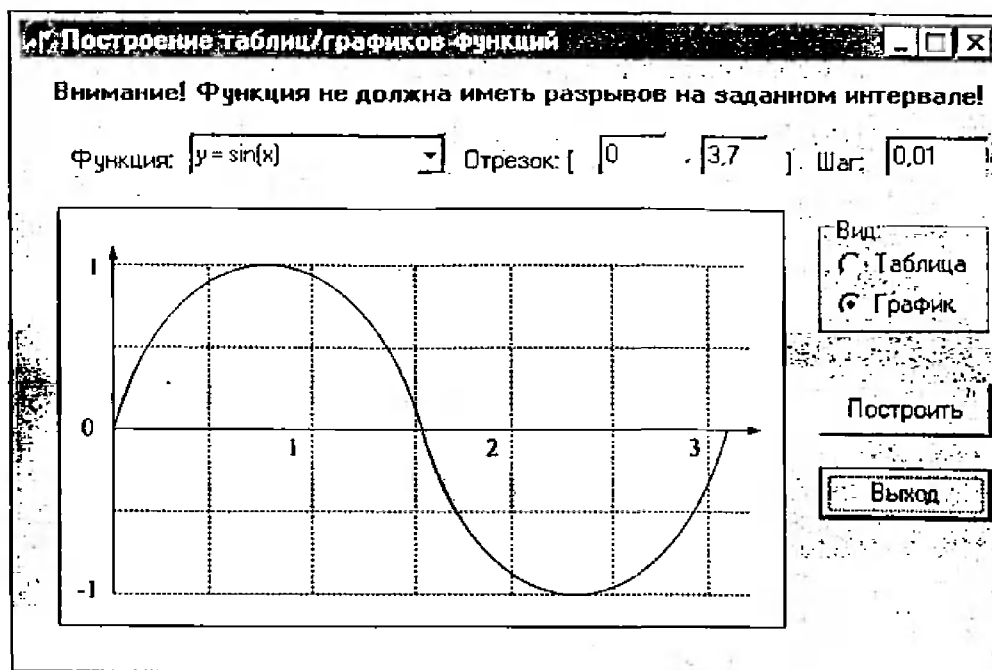


Рис. 8.6. Внешний вид окна программы построения графиков/таблиц функций (интерфейс со свободной навигацией)

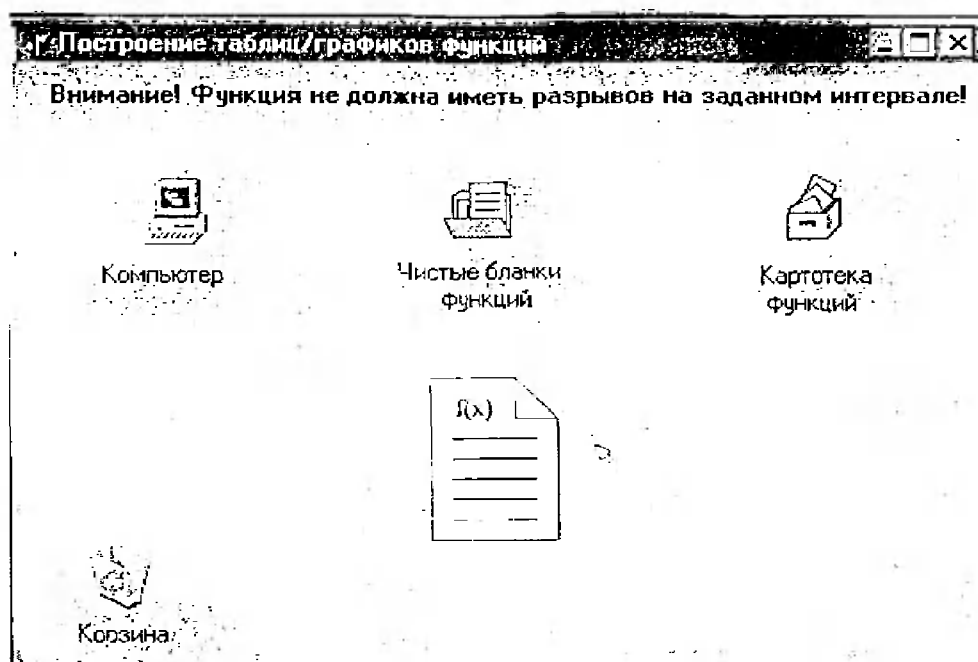


Рис. 8.7. Внешний вид окна программы построения таблиц/графиков функций (интерфейс прямого манипулирования)

Как уже упоминалось в § 3.5, различают также однок документные (SDI - Single Document Interface) и многодокументные (MDI - Multiple Document Interface) интерфейсы. Однок документные

или «однооконные» интерфейсы организуют работу, как следует из названия, только с одним документом, например, текстом или рисунком. Чтобы посмотреть другой текст, необходимо запустить еще одну копию приложения, что допустимо только в мультипрограммной операционной системе. Поэтому такие интерфейсы используют, если одновременная работа с несколькими документами маловероятна.

Многодокументные или «многооконные» интерфейсы соответственно организуют в тех случаях, когда велика вероятность, что пользователю понадобится одновременно работать с несколькими документами. Реализация этих интерфейсов существенно сложнее, а меню должно предусматривать специальные операции управления окнами.

Этапы разработки пользовательского интерфейса. Разработка пользовательского интерфейса включает те же основные этапы, что и разработка программного обеспечения:

- постановка задачи - определение типа интерфейса и общих требований к нему;
- анализ требований и определение спецификаций - определение сценариев использования и пользовательской модели интерфейса;
- проектирование — проектирование диалогов и их реализация в виде процессов ввода-вывода;
- реализация - программирование и тестирование интерфейсных процессов.

8.2. Психофизические особенности человека, связанные с восприятием, запоминанием и обработкой информации

При проектировании пользовательских интерфейсов необходимо учитывать психофизические особенности человека, связанные с восприятием, запоминанием и обработкой информации.

Исследованием принципов работы мозга человека занимается *когнитивная психология*. Специалисты в этой области предлагают упрощенную информационно-процессуальную модель мозга, представленную на рис. 8.8.



Рис. 8.8. Упрощенная информационно-процессуальная модель мозга

Информация о внешнем мире поступает в наш мозг в огромных количествах. Часть мозга, которую условно можно назвать «процессором восприятия», постоянно без участия сознания

перерабатывает ее, сравнивая с прошлым опытом, и помещает в хранилище уже в виде зрительных, звуковых и прочих образов. Любые внезапные или просто значимые для нас изменения в окружении привлекают наше внимание, и тогда интересующая нас информация поступает в кратковременную память. Если же наше внимание не было привлечено, то информация в хранилище пропадает, замещаясь следующими порциями.

В каждый момент времени фокус внимания может фиксироваться в одной точке. Поэтому, если возникает необходимость «одновременно» отслеживать несколько ситуаций, то обычно фокус перемещается с одного отслеживаемого элемента на другой. При этом внимание «рассредоточивается», и какие-то детали могут быть упущены. Например, при «прокрутке» текста или рисунка с использованием линейки прокрутки окна Windows приходится одновременно смотреть на текст, чтобы определить, где остановиться, и на ползунок. Поскольку текст важнее, фокус внимания перестает перемещаться на мышь, и она «соскакивает» с ползунка линейки.

Следует иметь в виду, что обработка процессором восприятия требует некоторого времени и, если сигнал выдается в течение времени, меньшем времени обработки, то наш мозг его не воспринимает.

Существенно и то, что восприятие во многом основано на мотивации. Например, если человек голоден, то он в первую очередь будет замечать все съедобное, а если устал — то, войдя в комнату, он в первую очередь увидит диван или кровать.

Необходимо также учитывать, что в процессе переработки информации мозг сравнивает поступающие данные с предыдущими. Так, если показать человеку последовательность символов:

А, В, С,

то он может принять 13 за В.

При смене кадра мозг на некоторое время блокируется: он «осваивает» новую картинку, выделяя наиболее существенные детали. А значит, если необходима быстрая реакция пользователя, то резко менять картинку не стоит.

Краткосрочная память - самое «узкое» место «системы обработки информации» человека. Ее емкость приблизительно равна 7 ± 2 несвязанных объектов. Краткосрочная память является своего рода оперативной памятью мозга, именно с ней работает процессор познания, но не востребованная информация хранится в ней не более 30 с. Чтобы не забыть какую-нибудь важную для нас информацию, мы обычно повторяем ее «про себя», «обновляя» информацию в краткосрочной памяти. Таким образом, при проектировании интерфейсов следует иметь в виду, что подавляющему большинству людей сложно, например, запомнить и ввести на другом экране число, содержащее более 5 цифр (7 - 2), или некоторое сочетание букв.

Люди вносят в каждую деятельность свое понимание того, как она должна выполняться. Это понимание - *модель деятельности* - базируется на прошлом опыте человека. Множество таких моделей хранится в долговременной памяти человека.

В долговременную память записываются постоянно повторяемые сведения или информация, связанная с сильными эмоциями. Долговременная память человека - хранилище информации с неограниченной емкостью и временем хранения. Однако доступ к этой информации весьма непрост: по всей вероятности, механизмы извлечения информации из памяти имеют ассоциативный характер. Специальная методика запоминания информации (*мнемоника*) использует именно это свойство памяти: для запоминания информации ее «привязывают» к тем данным, которые память уже хранит и позволяет легко получить.

Поскольку доступ к долговременной памяти затруднен, целесообразно рассчитывать не на то, что пользователь вспомнит нужную информацию, а на то, что он ее узнает. Именно поэтому интерфейс типа меню так широко используется.

Особенности восприятия цвета. Цвет в сознании человека ассоциируется с эмоциональным фоном. Известно, что теплые цвета: красный, оранжевый, желтый человека возбуждают, а холодные: синий, фиолетовый, серый - успокаивают. Причем цвет для человека является очень сильным раздражителем, поэтому применять цвета в интерфейсе необходимо крайне осторожно.

Следует иметь в виду, что обилие оттенков привлекает внимание, но быстро утомляет. Поэтому не стоит ярко раскрашивать окна, с которыми пользователь будет долго работать. Необходимо учитывать и индивидуальные особенности восприятия цветов человеком, например, примерно каждый десятый человек плохо различает какие-либо цвета, поэтому в ответственных случаях необходимо предоставить пользователю возможность настройки цветов.

Особенности восприятия звука. В интерфейсах звук обычно используют с разными целями: для привлечения внимания, как фон, обеспечивающий некоторое состояние пользователя, как источник дополнительной информации и т. п. Применяя звук, следует учитывать, что большинство людей очень чувствительны к звуковым сигналам, особенно, если последние указывают на наличие ошибки. Поэтому при создании звукового сопровождения целесообразно предусматривать возможность его отключения.

Субъективное восприятие времени. Человеку свойственно субъективное восприятие времени. Считают, что внутреннее время связано со скоростью и количеством воспринимаемой и обрабатываемой информации. Занятый человек обычно времени не замечает. Зато в состоянии ожидания время тянется бесконечно, что связано с тем, что в это время мозг оказывается в состоянии информационного вакуума. (К аналогичному состоянию приводит и усталость: информация поступает, но больше обрабатывается, а потому и ход времени замедляется.)

Доказано, что при ожидании более 1-2 с пользователь может отвлечься, «потерять мысль», что неблагоприятно сказывается на результатах работы и увеличивает усталость, так как каждый раз после ожидания много сил тратится на включение в работу.

Сократить время ожидания можно, заняв пользователя, но не отвлекая его от работы. Например, можно предоставить ему какую-либо информацию для обдумывания. По возможности целесообразно выводить пользователю промежуточные результаты: во-первых, он будет занят их обдумыванием, во-вторых, по ним он сможет оценить будущие результаты и отменит операцию, если они его не удовлетворяют.

Известны попытки использования для «развлечения» пользователя анимации, например, в Windows при копировании файлов демонстрируется «ролик» с летающими листочками. Однако следует иметь в виду, что, когда какую-либо анимацию смотришь первый раз, то это интересно, а когда в течение получаса наблюдаешь, как «летают» листочки при получении информации из Интернета, то это начинает раздражать.

Чтобы уменьшить раздражение, возникающее при ожидании, необходимо соблюдать основное правило: информировать пользователя, что заказанные им операции потребуют некоторого времени выполнения. Обычно для этого используют индикаторы оставшегося времени, анимированные объекты, как в Интернете, и изменение формы курсора мыши на песочные часы. Очень важно точно обозначить момент, когда система готова продолжать работу. Обычно для этого используют значительные изменения внешнего вида экрана.

В конечном итоге взаимодействие пользователя с интерфейсом будет определяться не только *физическими возможностями и особенностями* человека по восприятию, обработке и запоминанию информации, представленной в различных формах, а также по выполнению им разнообразных действий, но и пользовательской моделью интерфейса.

8.3. Пользовательская и программная модели интерфейса

Существуют три совершенно различные модели пользовательского интерфейса: модель программиста, модель пользователя и программная модель. Программист, разрабатывая пользовательский интерфейс, исходит из того, управление какими операциями ему необходимо реализовать в пользовательском интерфейсе, и как это осуществить, не затрачивая ни существенных ресурсов компьютера, ни своих сил и времени. Его интересуют функциональность, эффективность, технологичность, внутренняя стройность и другие не связанные с удобством пользователя характеристики программного обеспечения. Именно поэтому большинство интерфейсов существующих программ вызывают серьезные нарекания пользователей.

С точки зрения здравого смысла хорошим следует считать интерфейс, при работе с которым пользователь получает именно то, что он ожидал. Представление пользователя о функциях интерфейса можно описать в виде пользовательской модели интерфейса.

Пользовательская модель интерфейса - это совокупность обобщенных представлений конкретного пользователя или некоторой группы пользователей о процессах, происходящих во время работы программы или программной системы. Эта модель базируется на *особенностях* опыта конкретных пользователей, который характеризуется:

- уровнем подготовки в предметной области разрабатываемого программного обеспечения;
- интуитивными моделями выполнения операций в этой предметной области;
- уровнем подготовки в области владения компьютером;
- устоявшимися стереотипами работы с компьютером.

Для построения пользовательской модели необходимо изучить перечисленные выше особенности опыта предполагаемых пользователей программного обеспечения. С этой целью используют опросы, тесты и даже фиксируют последовательность действий, осуществляемых в процессе выполнения некоторых операций, на пленку.

Приведение в соответствие моделей пользователя и программиста, а также построение на их базе программной модели (рис. 8.9) интерфейса задача не тривиальная. Причем, чем сложнее автоматизируемая предметная область, тем сложнее оказывается построить программную модель интерфейса, учитывающую особенности пользовательской модели и не требующую слишком больших затрат как в процессе разработки, так и во время работы. С этой точки зрения объектные интерфейсы кажутся наиболее перспективными, так как в их основе лежит именно отображение объектов предметной области, которыми оперируют пользователи. Хотя на настоящий момент времени их реализация достаточно трудоемка.

При создании программной модели интерфейса также следует иметь в виду, что изменять пользовательскую модель непросто. Повышение профессионального уровня пользователей и их подготовки в области владения компьютером в компетенцию разработчиков программного обеспечения не входит, хотя часто грамотно построенный интерфейс, который адекватно отображает сущность происходящих процессов, способствует росту квалификации пользователей.

Интуитивные модели выполнения операций в предметной области должны стать основой для разработки интерфейса, а потому в большинстве случаев их необходимо не менять, а уточнять и совершенствовать. Именно нежелание или невозможность следования интуитивным моделям выполнения операций приводит к созданию искусственных надуманных интерфейсов, которые негативно воспринимаются пользователями.

Иногда кажется, что единственно доступный для изменения элемент - устоявшийся стереотип работы с компьютером. Однако ломка стереотипов - процедура болезненная. На это стоит решаться, если некоторое революционное изменение значительно расширяет возможности пользователя или облегчает его работу, например, переход к Windows-интерфейсам существенно упростил работу с компьютером огромному числу пользователей-непрофессионалов. Ломая же стереотипы по мелочам или неточно следуя принятой концепции, разработчик рискует оттолкнуть пользователей, которые просто не будут понимать, что происходит. В качестве примера можно вспомнить хотя бы путаницу с вызовом программ двойным щелчком правой клавиши мыши по пиктограмме рабочем столе или одинарным, если пиктограммы вынесена на панель Quick Launch (Быстрый Доступ) Windows.

Критерии оценки интерфейса пользователем. Многочисленные опросы и обследования, проводимые ведущими фирмами по разработке программного обеспечения, показали, что *основными критериями оценки* интерфейсов пользователем являются:

- простота освоения и запоминания операций системы - конкретно оценивают время освоения и продолжительность сохранения информации в памяти;
- скорость достижения результатов при использовании системы - определяется количеством вводимых или выбираемых мышью команд и настроек;
- субъективная удовлетворенность при эксплуатации системы (удобство работы, утомляемость и т. д.).

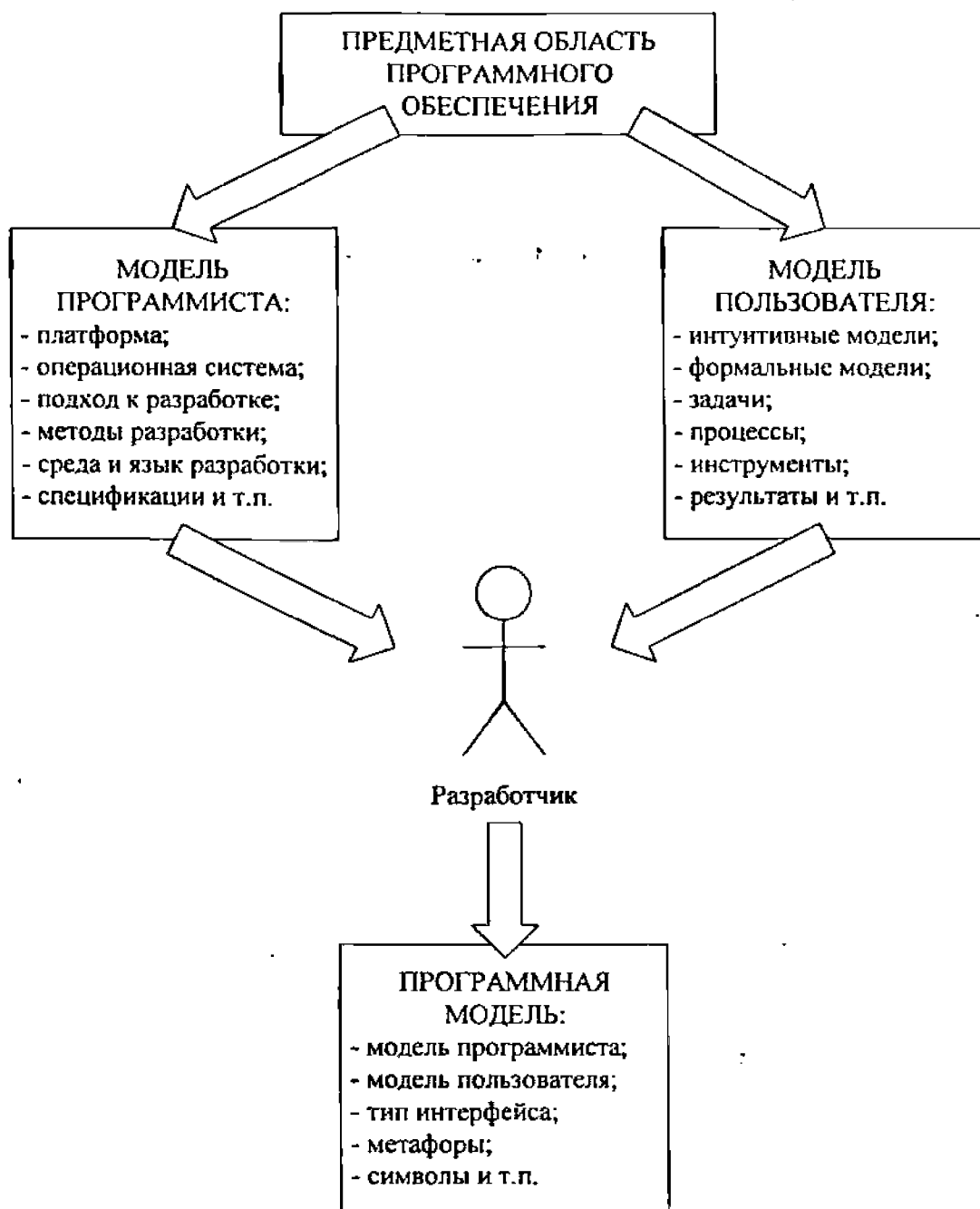


Рис. 8.9. Процесс проектирования пользовательского интерфейса

Причем для пользователей-профессионалов, постоянно работающих с одним и тем же пакетом, на первое место достаточно быстро выходят второй и третий критерии, а для пользователей-непрофессионалов, работающих с программным обеспечением периодически и выполняющих сравнительно несложные задачи - первый и третий.

С этой точки зрения на сегодняшний день наилучшими характеристиками для пользователей-профессионалов обладают интерфейсы со свободной навигацией, а для пользователей-непрофессионалов - интерфейсы прямого манипулирования. Давно замечено, что при выполнении операции копирования файлов при прочих равных условиях большинство профессионалов используют оболочки типа Far, а непрофессионалы - «перетаскивание объектов» Windows.

8.4. Классификации диалогов и общие принципы их разработки

Как отмечалось в § 8.1, диалог - это процесс обмена информацией между пользователем и программной системой, осуществляемый через интерактивный терминал и по определенным правилам.

Различают тип диалога и его форму.

Типы диалога. Тип диалога определяет, кто из «собеседников» управляет процессом обмена информацией. Соответственно различают два типа диалога: управляемые программой и управляемые пользователем.

Диалог, *управляемый программой*, предусматривает наличие жесткого, линейного или древовидного, т. е. включающего возможные альтернативные варианты, сценария диалога, заложенного в программное обеспечение. Такой диалог обычно сопровождают большим количеством подсказок, которые уточняют, какую информацию необходимо вводить на каждом шаге.

Диалог, *управляемый пользователем*, подразумевает, что сценарий диалога зависит от пользователя, который применяет систему для выполнения необходимых ему операций. При этом система обеспечивает возможность реализации различных пользовательских сценариев.

Формы диалога. Никакой диалог невозможен, если не существует языка, понятного «собеседникам». Описание языка, на котором ведется диалог, включает определение его *синтаксиса* - правил, определяющих допустимые конструкции (слова, предложения) языка или его форму, и *семантики* - правил, определяющих смысл синтаксически корректных конструкций языка или его содержание. В зависимости от вида используемых в конкретном случае синтаксиса и семантики различают три формы диалога:

- фразовую,
- директивную,
- табличную.

Фразовая форма предполагает «общение» с пользователем на естественном языке или его подмножестве. Содержание диалога в данной форме составляют повелительные, повествовательные и вопросительные предложения и ответы на вопросы. Общение может осуществляться в свободном формате, но возможна и фиксация отдельных фраз.

Организация диалога на естественном языке на современном уровне - задача не решенная, так как естественный язык крайне сложен и пока не удастся в достаточной степени формализовать его синтаксис и семантику.

Чаще всего используют диалоги, предполагающие односложные ответы, например:

Программа: Введите свой возраст (полных лет):

Пользователь: 48.

В этом случае программа содержит ограниченное описание как синтаксиса, так и семантики используемого *ограниченно-естественного* языка. Для данного примера достаточно определить синтаксис понятия «целое положительное число» и наложить ограничение на значение числа.

Однако существует некоторый опыт создания интерфейсов на базе ограниченного подмножества *предложений* естественного языка в основном для интеллектуальных систем. Синтаксис и семантика языков диалога, реализуемых в таких интерфейсах, достаточно сложны.

При обработке фраз в этих случаях оперируют понятием словоформа. *Словоформа* - отрезок текста между двумя соседними пробелами или знаками препинания. Обработка словоформ вне связи с контекстом называется *морфологическим анализом*.

Выделяют два метода морфологического анализа:

- декларативный - предполагает, что в словаре находятся все возможные словоформы каждого слова, тогда анализ сводится к поиску словоформы в словаре. Данный метод обеспечивает возможность обработки сообщений, состоящих из строчных и прописных букв в произвольной комбинации, причем как латинского, так и русского или других алфавитов;

- процедурный - предполагает выделение в текущей словоформе основы, которую затем идентифицируют.

После распознавания словоформ осуществляют синтаксический анализ сообщения, по результатам которого определяют его синтаксическую структуру, т. е. выполняют разбор предложения.

Далее выполняют *семантический анализ*, т. е. определяют смысловые отношения между словоформами. При этом выявляют главные предикаты, определяющие смысл предложения.

Таким образом, интерфейс, реализующий фразовую форму диалога, должен: преобразовывать сообщения из естественно-языковой формы в форму внутреннего представления и обратно, выполнять анализ и синтез сообщений пользователя и системы, отслеживать и запоминать пройденную часть диалога.

Основными *недостатками* фразовой формы при использовании подмножества естественного языка являются:

- большие затраты ресурсов;
- отсутствие гарантии однозначной интерпретации формулировок;
- необходимость ввода длинных грамматически правильных фраз.

Основное *достоинство* фразовой формы состоит в относительно свободном общении с системой.

Директивная форма предполагает использование команд (директив) *специально разработанного формального языка*. *Командой* в этом случае называют предложение этого языка, описывающее комбинированные данные, которые включают идентификатор иницируемого процесса и, при необходимости, данные для него.

Команду можно вводить:

- в виде строки текста, специально разработанного формата, например, команды MS DOS, которые вводятся в командной строке;
- нажатием некоторой комбинации клавиш клавиатуры, например, комбинации «быстрого доступа» современных Windows-приложений;
- посредством манипулирования мышью, например, «перетаскиванием» пиктограмм;
- комбинацией второго и третьего способов.

Основными достоинствами директивной формы являются:

- сравнительно небольшой объем вводимой информации;
- гибкость - возможности выбора операции в данном случае ограничены только набором допустимых команд;
- ориентация на диалог, управляемый пользователем;
- использование минимальной области экрана или неиспользование ее вообще;
- возможность совмещения с другими формами.

Недостатки директивной формы:

- практическое отсутствие подсказок на экране, что требует запоминания вводимых команд и их синтаксиса;
- почти полное отсутствие обратной связи о состоянии иницированных процессов;
- необходимость навыков ввода текстовой информации или манипуляций мышью;
- отсутствие возможности настройки пользователем.

Исследования показали, что директивная форма удобна для пользователя-профессионала, который обычно быстро запоминает синтаксис часто используемых команды или комбинации клавиш. Основные достоинства формы (гибкость и хорошие временные характеристики) проявляются в этом случае особенно ярко.

Табличная форма предполагает, что пользователь выбирает ответ из предложенных программой. Язык диалога для табличной формы имеет простейший синтаксис и однозначную семантику, что достаточно легко реализовать. Удобна эта форма и для пользователя, так как выбрать всегда проще, чем вспомнить, что особенно существенно для пользователя-непрофессионала или пользователя, редко использующего конкретное программное обеспечение. Однако применение табличной формы возможно не всегда: ее можно использовать только, если

множество возможных ответов на конкретный вопрос конечно. Причем, если количество возможных ответов велико (более 20), то применение табличной формы может оказаться нецелесообразным.

Достоинствами табличной формы являются:

- наличие подсказки, что уменьшает нагрузку на память пользователя, так как данная форма ориентирована не на запоминание, а на узнавание;
- сокращение количества ошибок ввода: пользователь не вводит информацию, а указывает на нее;
- сокращение времени обучения пользователя;
- возможность совмещения с другими формами;
- в некоторых случаях возможность настройки пользователем.

К *недостаткам* данной формы относят.

- необходимость наличия навыков навигации по экрану;
- использование сравнительно большой площади экрана для изображения визуальных компонентов;
- интенсивное использование ресурсов компьютера, связанное с необходимостью постоянного обновления информации на экране.

Следует иметь в виду, что типы и формы диалога выбирают независимо друг от друга: любая форма применима для обоих типов диалогов (рис. 8.10).

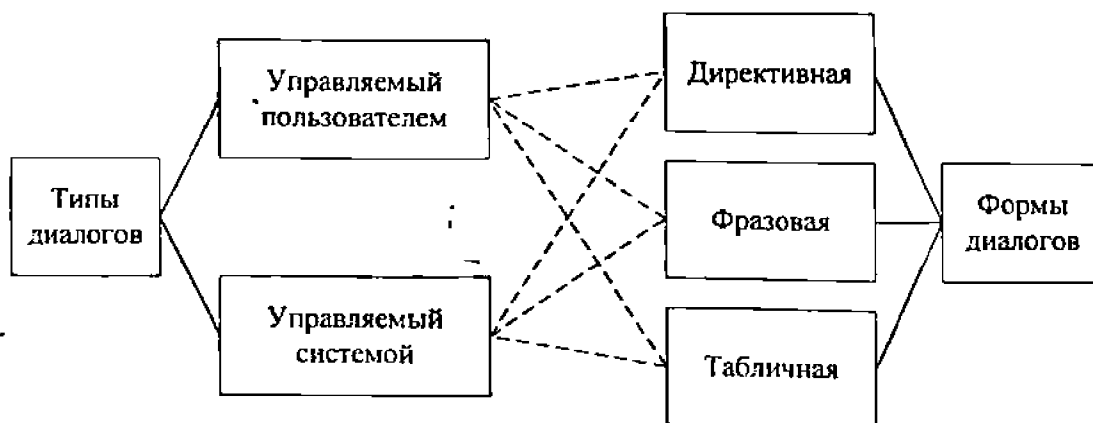


Рис. 8.10. Соответствие типов диалогов и его форм

Однако фразовая форма, которая используется в диалоге, управляемом пользователем, как правило, предполагает более сложные синтаксис и семантику языка диалога, так как программа должна «понимать» пользователя.

Сложное программное обеспечение обычно взаимодействует с пользователем посредством диалогов различных типов и форм в зависимости от решаемых задач. Причем, помимо диалогов, происходящих в процессе нормальной работы программного обеспечения и называемых *синхронными*, предусматривают диалоги, возникающие по инициативе системы или пользователя при нарушении сценария нормального процесса. Такие диалоги называют *асинхронными*. Обычно их используют для выдачи экстренных сообщений от системы или пользователя.

Разработка диалогов. Процесс проектирования и реализации диалогов можно разделить на следующие стадии:

- определение множества необходимых диалогов, их основных сообщений и возможных сценариев — проектирование *абстрактных диалогов*;
- определение типа и формы каждого диалога, а также синтаксиса и семантики используемых языков — проектирование *конкретных диалогов*;

- выбор основных и дополнительных устройств и проектирование процессов ввода-вывода для каждого диалога, а также уточнение передаваемых сообщений - проектирование *технических диалогов*.

В основу абстрактных диалогов должна закладываться идеология технологического процесса, для автоматизации которого предназначается программный продукт. Именно анализируя составляющие автоматизируемого технологического процесса, разработчик определяет сценарии диалогов (см. § 6.2), которые должны быть предусмотрены в программном обеспечении.

Кроме сценариев, при проектировании абстрактных диалогов используют *диаграммы состояний интерфейса* или *графы диалога*.

Граф диалога — ориентированный взвешенный граф, каждой вершине которого сопоставлена конкретная картинка на экране (*кадр*) или определенное состояние диалога, характеризующееся набором доступных пользователю действий. Дуги, исходящие из вершин, показывают возможные изменения состояний при выполнении пользователем указанных действий. В качестве весов дуг указывают условия переходов из состояния в состояние и операции, выполняемые во время перехода.

Таким образом, каждый маршрут на графе соответствует возможному варианту диалога. Причем представление диалога в виде графа в зависимости от стадии разработки может выполняться с разной степенью детализации. По сути граф диалога - это граф состояний конечного автомата, моделирующего поведение программного обеспечения при воздействиях пользователя. Для представления таких графов уже были введены две нотации: нотация диаграмм состояний структурного подхода к разработке (см. рис. 4.3) и нотация диаграмм состояний UML (см. рис. 7.17). Причем нотация UML является более мощной, так как позволяет использовать обобщенные состояния. Поэтому, чтобы не вводить новую нотацию для представления графа диалога, будем использовать обозначения UML.

Пример 8.2. Разработать граф диалога для системы решения комбинаторно-оптимизационных задач.

Так как диалог на верхнем уровне должен обеспечивать реализацию диаграммы вариантов использования, исходный вариант графа диалога строим на основе анализа этой диаграммы (см. рис. 6.4). Можно предположить, что пользователь будет принимать решение о сохранении или удалении результатов после их просмотра, поэтому эти операции естественно объединить в единую группу. Кроме того, в ту же группу целесообразно добавить операцию печати результатов. Аналогично просмотр данных целесообразно объединить с их удалением или корректировкой. Операцию Новое задание целесообразно поместить в отдельную группу (рис. 8.11).

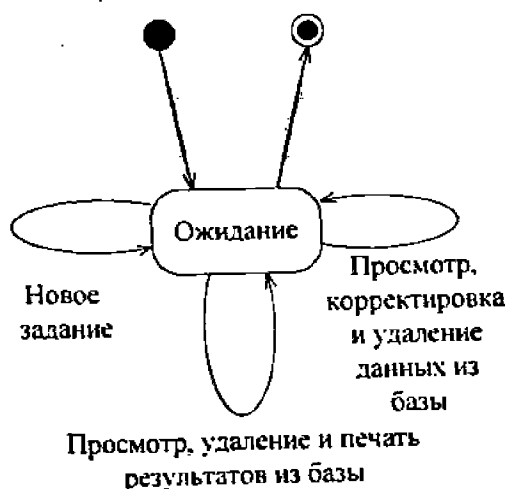


Рис. 8.11. Граф абстрактного диалога системы решения комбинаторно-оптимизационных задач

На верхнем уровне диалог очевидно должен управляться пользователем. Директивная и табличная формы могут использоваться альтернативно, по желанию пользователя, а применение фразовой формы нецелесообразно.

Пример 8.3. Детализировать диалог Новое задание.

В § 6.2 приведен сценарий Выполнения задания, на базе которого можно предложить граф диалога, управляемого системой (рис. 8.12, а). Однако этот же диалог можно представить и в виде диалога, управляемого пользователем (рис. 8.12, б).

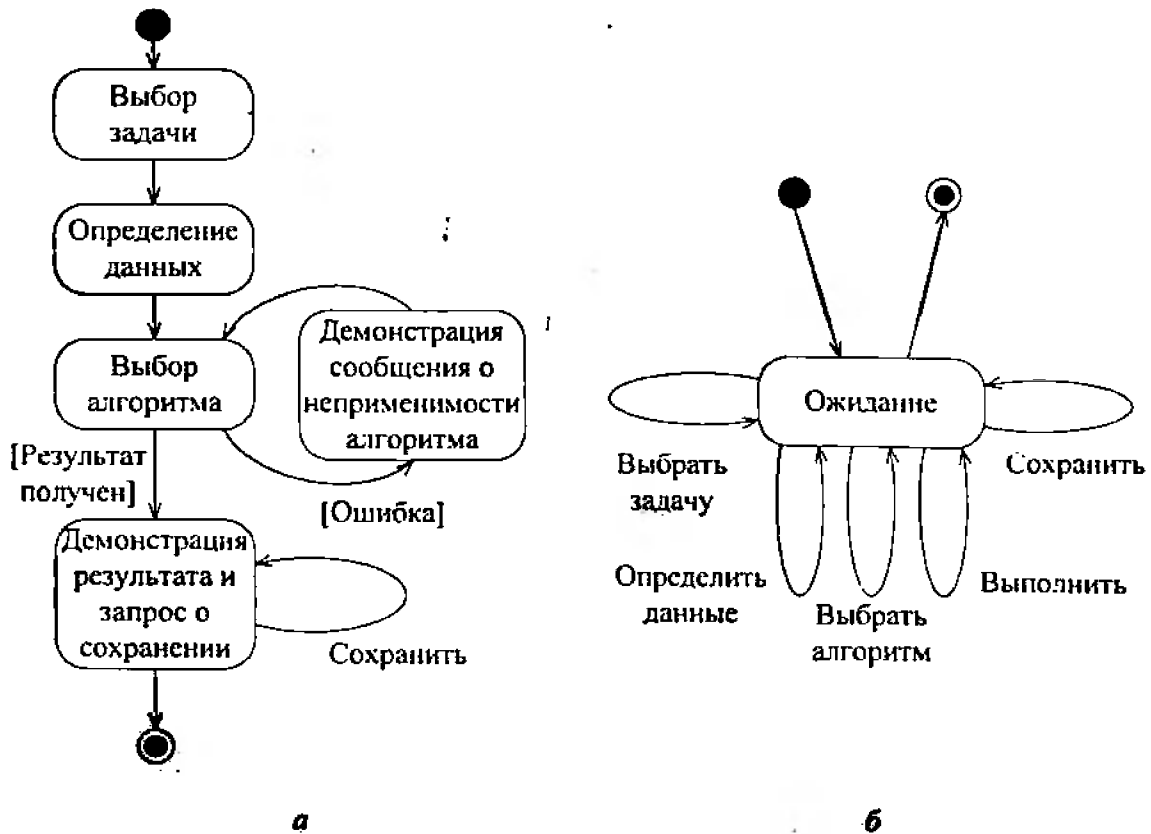


Рис. 8.12. Графы абстрактного диалога Новое задание:
а – диалог, управляемый системой; б – диалог, управляемый пользователем

Анализ графов диалога показывает, что диалог, управляемый системой, в данном случае сильно ограничивает пользователя в выборе вариантов действия, а диалог, управляемый пользователем, предполагает выбор действия после каждого шага, хотя по смыслу эти шаги чаще всего будут выполняться последовательно.

Поэтому для реализации лучше использовать комбинированный вариант, который учитывает наличие сценария, но допускает отклонения от него по желанию пользователя (рис. 8.13).

Теперь необходимо определить, какие формы диалога можно использовать для каждого шага диалога. Первый шаг - Выбор задачи включает три варианта, поэтому имеет смысл использовать табличную форму. Второй шаг - Определение данных не конкретизирован, следовательно, уточнить его форму пока невозможно. Третий шаг - Выбор алгоритма опять же предполагает выбор, причем количество вариантов невелико: целесообразно использовать табличную форму. В остальных случаях также предпочтительной оказывается именно эта форма.

Последний этап проектирования интерфейсов - разработка конкретных операций ввода-вывода для каждого диалога с учетом специфики формы интерфейса. Рассмотрим интерфейсные компоненты, которые могут быть использованы в современных пользовательских интерфейсах.



Рис. 8.13. Граф абстрактного диалога комбинированного типа

8.5. Основные компоненты графических пользовательских интерфейсов

Графические пользовательские интерфейсы поддерживаются операционными системами Windows, Apple Macintosh, OS/2 и т. д. В рамках указанных операционных систем для таких интерфейсов разработаны наборы стандартных компонентов взаимодействия с пользователем. Эти наборы не идентичны, как и основные приемы работы с интерфейсами различных операционных систем.

Пользовательские интерфейсы большинства современных программ строятся по технологии WIMP: W - Windows (окна), I - Icons (пиктограммы), M - Mouse (мышь), P - Pop-up (всплывающие или выпадающие меню). Основными элементами графических интерфейсов, таким образом, являются: окна, пиктограммы, компоненты ввода-вывода и мышь, которую используют в качестве указующего устройства и устройства прямого манипулирования объектами на экране.

Окна. *Окно* - обычно прямоугольная, ограниченная рамкой область физического экрана. Окно может менять размеры и местоположение в пределах экрана. Все окна можно разделить на 5 категорий:

- основные окна (окна приложений);
- дочерние или подчиненные окна;
- окна диалога;
- информационные окна;
- окна меню.

Окно приложения Windows (рис. 8.14) обычно содержит: рамку, ограничивающую рабочую область окна, строку заголовка с кнопкой системного меню и кнопками выбора представления

окна и выхода, строку меню, пиктографическое меню (панель инструментов), горизонтальные и вертикальные полосы прокрутки и строку состояния.

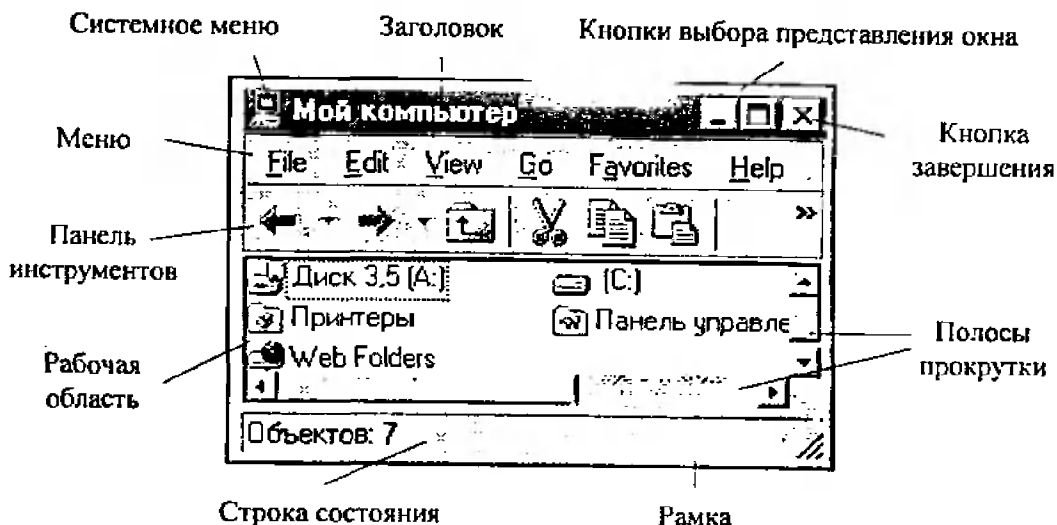


Рис. 8.14. Окно приложения и его элементы

Дочернее окно Windows (рис. 8.15, а) используют в многодокументных программных интерфейсах (MDI), предполагающих, что программное обеспечение должно работать с несколькими документами одновременно. В отличие от окна приложения дочернее окно не содержит меню. В строке заголовка - специальное имя, идентифицирующее связанный с ним документ или файл. Пиктограммы всех дочерних окон одинаковы.

Диалоговое окно Windows (рис. 8.15, б) используют для просмотра и задания различных режимов работы, необходимых параметров или другой информации. Оно может содержать:

- строку заголовка с кнопкой системного меню;
- компоненты, обеспечивающие пользователю возможность ввода или выбора ответа;
- вспомогательные компоненты, обеспечивающую подсказку, например, поле предварительного просмотра или кнопку вызова справки.

Как правило, размер диалогового окна неизменяем, но его можно перемещать по экрану.

Информационные окна бывают двух типов: окна сообщений и окна помощи. *Окна сообщений* (рис. 8.15, в), кроме заголовка с кнопкой системного меню, обычно содержат текст сообщения и одну или несколько кнопок реакции пользователя, например, кнопки Yes и No или кнопки Yes, No и Cancel.

Окно помощи имеет более сложную структуру: оно может содержать меню, полосы прокрутки и информационная область, т. е. по структуре оно аналогично окну приложения, но отличается от него тем, что имеет узко специальное назначение, обеспечивая навигацию по справочной информации.

Окна меню Windows (рис. 8.15, г) можно использовать как открывающиеся панели иерархического меню или как отдельные контекстные меню. Каждой строке окна меню может соответствовать:

- команда;
- меню следующего уровня, что обозначается стрелкой;
- окно диалога, что обозначается тремя точками.

Кроме того, в некоторых строках добавляется указание клавиш быстрого вызова.

Пиктограммы. Пиктограмма представляет собой небольшое окно с графическим изображением, отражающим содержимое буфера, с которым она связана. Различают:

- программные пиктограммы;
- пиктограммы дочерних окон;

- пиктограммы панели инструментов;
- пиктограммы объектов.

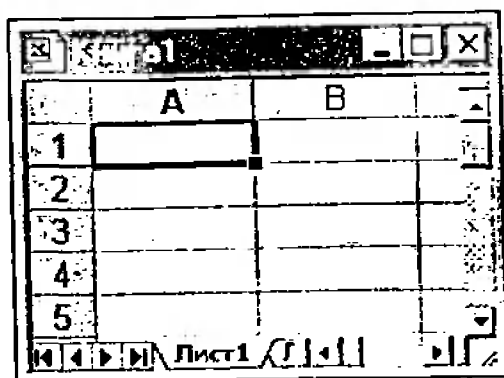
Программными пиктограммами, которые связаны с соответствующей программой, управляет операционная система. Так, можно «свернуть» окно приложения в пиктограмму на панели задач Windows или «развернуть» его обратно «на рабочий стол».

Аналогично многодокументная программная система управляет *пиктограммами дочерних окон*, обеспечивающими доступ к различным документам, одновременно обрабатываемым программной системой.

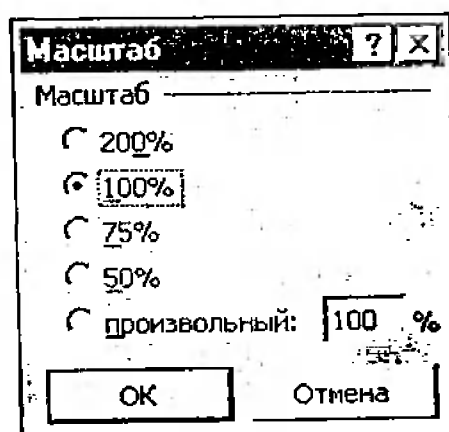
Пиктограммы панели инструментов обычно дублируют доступ к соответствующим функциям через меню, обеспечивая их быстрый вызов.

Пиктограммы объектов используют для прямого манипулирования этими объектами.

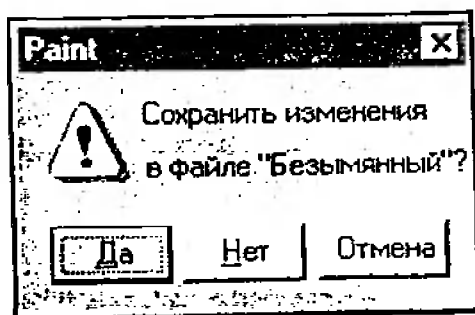
Как правило, все пиктограммы можно перемещать мышью. Кроме того, для облегчения работы с пиктограммами обычно используют «всплывающие» подсказки, которые появляются, если пользователь в течение некоторого времени удерживает мышь над пиктограммой панели инструментов.



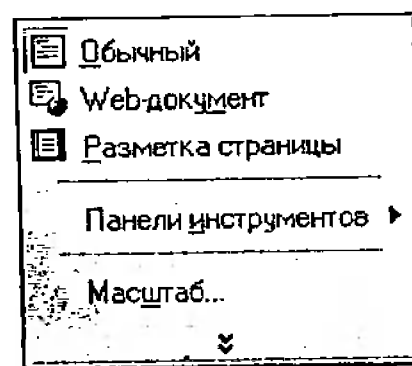
а



б



в



г

Рис. 8.15. Примеры вспомогательных окон Windows:

а – дочернее окно; б – окно диалога; в – окно сообщения; г – окно меню (адаптивное)

Прямое манипулирование изображением. *Прямое манипулирование изображением* - это возможность замены команды воздействия на некоторый объект физическим действием в интерфейсе, осуществляемым с помощью мыши. При этом любая область экрана рассматривается

как адресат, который может быть активизирован при подведении курсора и нажатии клавиши мыши.

По реакции на воздействие различают следующие типы адресатов:

- указание и выбор (развертывание пиктограмм, определение активного окна и т. п.);
- буксировка и «резиновая нить» (перенос объекта или его границ);
- экранные кнопки и «скользящие» барьеры (выполнение дискретных или циклически повторяемых действий, например, выполнение некоторой операции или рисование, подразумеваемых при активизации определенной области экрана - кнопки).

Не последняя роль в графических интерфейсах отводится *динамическим визуальным сигналам*, которые представляют собой изменение изображения на экране. Основная цель этих сигналов заключается в предоставлении пользователям дополнительной информации. Простейшим примером такого сигнала является изменение изображения курсора мыши при выполнении конкретных операций, например, изображение его в форме песочных часов во время обработки. Другой пример - изменение изображения кнопки при нажатии на нее. Хотя в отличие от анимационных интерфейсов прямого манипулирования эти визуальные сигналы играют в графических интерфейсах вспомогательную роль, обеспечивая более реалистическую картинку.

Компоненты ввода-вывода. Как уже упоминалось, в окнах приложения могут размещаться специальные компоненты, используемые для ввода-вывода информации. Интерфейс практически любого современного программного обеспечения включает несколько меню: основное или «ниспадающее» иерархическое меню, пиктографические меню (панели инструментов) и контекстные меню для разных ситуаций. Любое из указанных меню представляет собой компонент ввода-вывода, реализующий диалог с пользователем, используя табличную форму.

Иерархические меню используют, чтобы *организовать* выполняемые программным обеспечением операции, если их число превышает 5-8 (6 в соответствии с рекомендациями фирмы IBM), и обеспечить пользователю их обзор. Панели инструментов и контекстные меню применяют для обеспечения быстрого доступа к часто используемым командам, обеспечивая пользователю возможность относительно свободной навигации.

Кроме меню в интерфейсе используют и другие компоненты ввода-вывода, которые можно разделить на три группы в соответствии с тем, какую форму диалога они реализуют: фразовую, табличную или смешанную. Директивная форма диалога обычно предполагает ввод комбинаций клавиш или перемещение пиктограмм, а потому не требует использования компонентов ввода-вывода. В табл. 8.2 приведены основные компоненты WINP-интерфейса Windows и даны рекомендации по их использованию.

8.6. Реализация диалогов в графическом пользовательском интерфейсе

Как правило, сложное программное обеспечение с развитым пользовательским интерфейсом использует диалоги обоих типов: управляемые пользователем и управляемые системой.

Реализация диалогов, управляемых пользователем. Для реализации диалогов, управляемых пользователем, применяют меню различных видов: основное, панели инструментов, контекстные и кнопочные, т. е. сформированные из отдельных кнопок. Как альтернативу меню целесообразно использовать директивную форму диалога, поставив в соответствие основным командам определенные комбинации клавиш. Кроме того, целесообразно предусмотреть возможность управления меню клавиатурой, что особенно важно, если большую часть времени работы с системой пользователь вводит текст или данные, т. е. взаимодействует с клавиатурой.


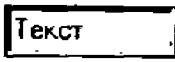



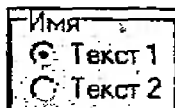
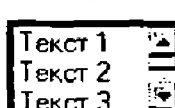
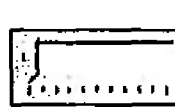



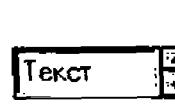
Меню. Меню проектируют на основе графов диалогов разрабатываемого программного обеспечения. При этом, если число операций не превышает 5, то обычно используют кнопки. Если число операций не более 9-10, то - одноуровневое меню. И, наконец, если число реализуемых операций более 10, то используют «ниспадающее» двухуровневое иерархическое меню.

Ниспадающее меню. Первый уровень иерархического меню должен содержать имена основных групп операций. Традиционно первым является пункт Файл, вторым - Правка, третьим -

Вид, а последним - Справка. Такое распределение пунктов специфично для программ обработки данных, размещенных в файлах, например, текстовых и графических редакторов. В последнее время с таким распределением пунктов возникают проблемы, так как большинство программ уже не работает с данными традиционным способом. Так, в примере 8.4 все данные находятся не в файлах, а в базе данных.

Количество уровней иерархического меню не должно превышать 2-3, так как при большем числе уровней требуемую операцию будет сложно искать. Причем желательно, чтобы число операций в окне меню не превышало 7-8, по той же причине.

Таблица 8.2

Компонент	Внешний вид	Реализуемая форма	Особенности использования
Label – метка		Фразовая	Вывод сообщения, как правило, неизменяемого в процессе работы
Edit – однострочный редактор		Фразовая	Ввод-вывод недлинных сообщений: слов, чисел и т. п.
Мемо – многострочный редактор		Фразовая	Ввод-вывод текстовой информации
Button – кнопка		Табличная	Инициация операции
CheckBoxButton – элемент выбора		Табличная	Выбор или отмена опций
RadioGroup – группа радиокнопок		Табличная	Выбор одного из вариантов, если их число не превышает 6-10
ListBox – список		Табличная	Выбор одного или нескольких вариантов
TrackBar – указатель		Табличная	Выбор числового значения из заданного интервала
TabControl, PageControl – закладки		Табличная	Улучшение навигации при большом количестве параметров
DBNavigator – навигатор		Табличная	Навигация по таблицам
ComboBox – список с возможностью добавления		Комбинированная	Выбор из списка или ввод значения
SpinEdit – однострочный редактор с возможностью увеличения и уменьшения значения		Комбинированная	Ввод или изменение значения

Если число операций превышает 70-80, то возникает проблема, как построить наглядное меню с таким большим числом операций. Интересное решение было предложено разработчиками Microsoft Word. Они реализовали *адаптивное* иерархическое меню, где содержимое окна меню второго уровня постоянно меняется, отображая только те операции, которые использует пользователь. Если пользователь не находит нужной операции, то через несколько секунд или при нажатии специальной кнопки Word демонстрирует окно меню полностью.

Пример 8.4. Разработать основное меню системы решения комбинаторно-оптимизационных задач.

Поскольку пользователю может понадобиться сравнить несколько вариантов решений, целесообразно разрабатывать многодокументный интерфейс, каждый документ которого соответствует заданию. В качестве отдельного документа также должны рассматриваться данные, которые могут обрабатываться разными алгоритмами или корректироваться для использования в других задачах.

На рис. 8.11 представлен граф абстрактного диалога системы, на основе которого необходимо построить меню. К операциям, предусмотренным графом диалога, следует добавить служебные операции обслуживания файла задания, управления дочерними окнами и работы со справочной информацией, а затем полученное множество операций разбить на группы.

Вариант 1. Стандартизованный вариант меню для данной системы представлен на рис. 8.16. Пункт Файл объединяет все операции с информационными блоками обоих типов: проектами и данными. Пункт Правка - стандартные операции правки. Пункт Проект - операции управления проектом. Пункт Выполнить - два вида операций выполнения. Пункт Окна - операции управления окнами многодокументного интерфейса. И, наконец, пункт Справка - стандартные операции работы со справочной информацией. Этот вариант, скорее всего, будет интуитивно понятен пользователям, имеющим большой опыт работы со средами программирования, так как он разработан по типу таких сред (сравните, например, с интерфейсом Delphi).

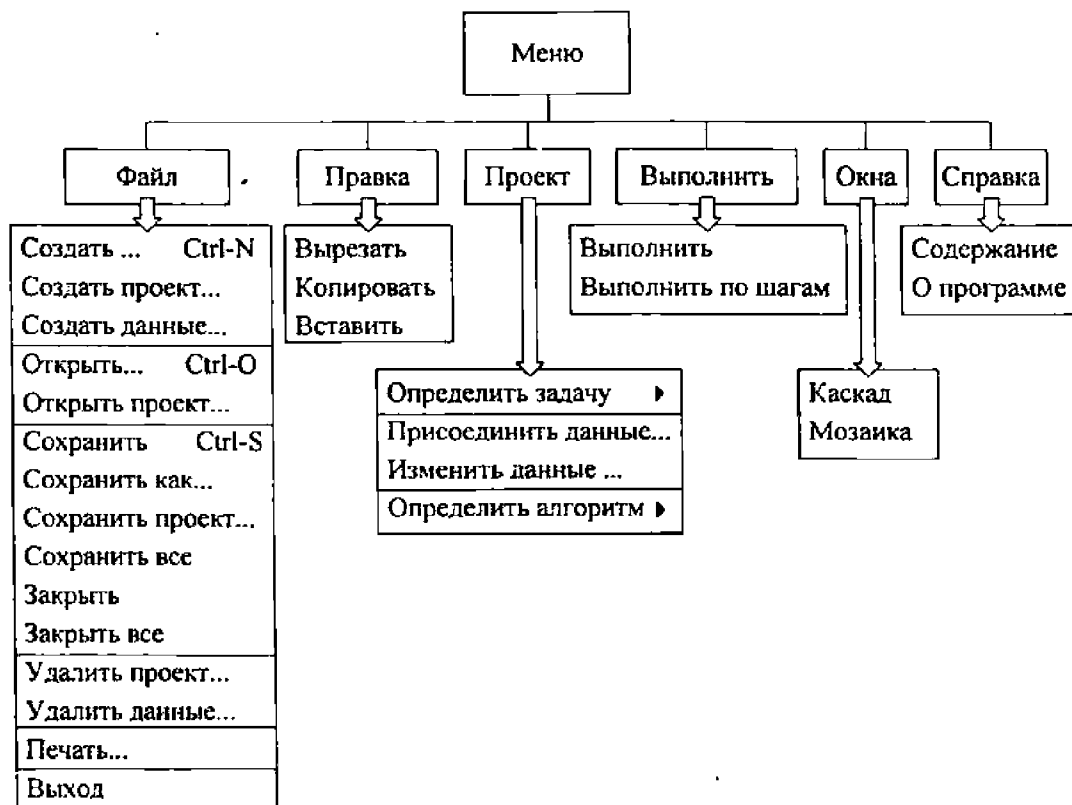


Рис. 8.16. Меню системы решения комбинаторно-оптимизационных задач (вариант 1)

В основе данного интерфейса лежит понятие «проект». Для каждого проекта определяют решаемую задачу, к проекту присоединяют данные (новые или выбранные из уже существующих) и выбирают алгоритм решения задачи. При выполнении проекта результаты заносятся в протокол проекта. Полученный протокол можно сохранить или просто закрыть без сохранения. При необходимости сохраненный протокол можно удалить.

Новые данные можно создавать отдельно от проекта, но при этом необходимо указать задачу. Можно модифицировать данные, в том числе и изменить тип решаемой задачи, и сохранить данные с новым идентификатором. Уже сохраненные данные можно удалить.

Для просмотра результатов необходимо открыть уже выполненные проекты. Их можно распечатать и/или удалить.

Вариант 2. «Нестандартный» вариант, основанный на интуитивной модели пользователя, т. е. концептуальной модели предметной области (см. рис. 6.9), представлен на рис. 8.17. В этом меню два типа блоков данных управляются операциями, отнесенными к разным группам «Задание» и «Данные». В результате удастся частично разгрузить первый пункт меню, но необходимо дублировать операции с блоками данных (создание, открытие, закрытие и т. п.). Чтобы еще сократить количество пунктов в первой и во второй группах, можно использовать адаптивный вариант.

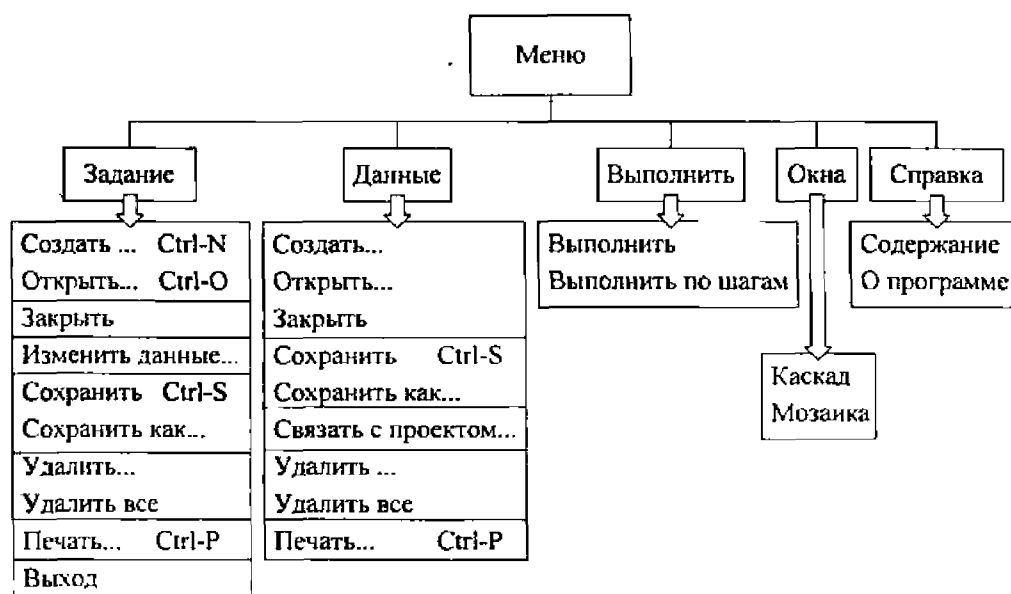


Рис. 8.17. Меню системы решения комбинаторно-оптимизационных задач (вариант 2)

Панель инструментов. На панель инструментов помещают пиктограммы часто используемых операций. Если множество таких операций существенно зависит от специфики выполняемых с разрабатываемым программным обеспечением работ, то целесообразно обеспечить пользователю возможность формирования панелей инструментов по собственному усмотрению. В качестве примера можно посмотреть, как реализована операция настройки (Сервис\Настройка) Microsoft Word.

Контекстные меню. Контекстные меню включают операции, вероятность обращения к которым из данной зоны окна приложения с точки зрения разработчика максимальна. В процессе тестирования «удобства использования» (см. § 9.6) содержание контекстного меню может уточняться. Так же, как и в случае основного меню, нежелательно, если число операций этого меню превышает 6-8. Причем, чтобы облегчить пользователю поиск нужной операции целесообразно операции контекстного меню горизонтальными линиями делить на группы.

Реализация диалогов, управляемых системой. Для реализации диалогов, управляемых системой, обычно используют диалоговые окна. Причем, если число настраиваемых в процессе диалога элементов невелико, и диалогу соответствует последовательный сценарий, то проектируют одно диалоговое окно, включающее все необходимые компоненты. Такое окно часто называют *формой*. Если же диалог имеет сильно разветвленную структуру, в которой следующий вопрос зависит от уже полученных ответов, или число настраиваемых в процессе диалога элементов велико, то для каждого шага диалога проектируют свое диалоговое окно.

Проектирование форм заключается в выборе необходимых компонентов интерфейса и размещении их в пределах диалогового окна. Если количество компонентов более 4-5, то целесообразно их визуальнo разделить, используя рамки.

Проектирование последовательностей диалоговых окон. Как уже упоминалось выше, в основе диалогов, управляемых системой, лежит жестко или нежестко заданный сценарий. Именно этот сценарий должен быть реализован последовательностью диалоговых окон. Независимо от степени жесткости сценария при проектировании такой последовательности необходимо предусмотреть возможность возврата на предыдущий шаг.

Пример 8.5. Реализовать диалог Новое задание системы решения комбинаторно-оптимизационных задач.

Граф диалога представлен на рис. 8.12. Этот управляемый системой диалог допускает возвраты на предыдущие шаги. Соответственно для него последовательность действий определена не жестко. В данном случае можно предложить два варианта реализации диалога: с использованием одной формы и с использованием последовательности диалоговых окон.

Вариант 1. Реализация диалога с использованием формы предполагает, что все шаги выполняются в одном окне. Следовательно, необходимо организовать выбор типа задачи, ввод/выбор данных, выбор алгоритма. После выполнения задания необходимо также предусмотреть возможность его сохранения, сохранения с другим именем и закрытия (рис. 8.18). Результаты целесообразно демонстрировать в отдельном окне, которое будет открываться при нажатии кнопки Показать результаты, так как у каждого типа задачи свои результаты.

Рис. 8.18. Форма, реализующая диалог Новое задание (вариант 1)

Вариант 2. Последовательность диалоговых окон реализует последовательный или древовидный сценарий. Поэтому преобразуем сценарий диалога (см. рис. 8.13), к последовательному с возможностью возврата на один шаг (рис. 8.19).

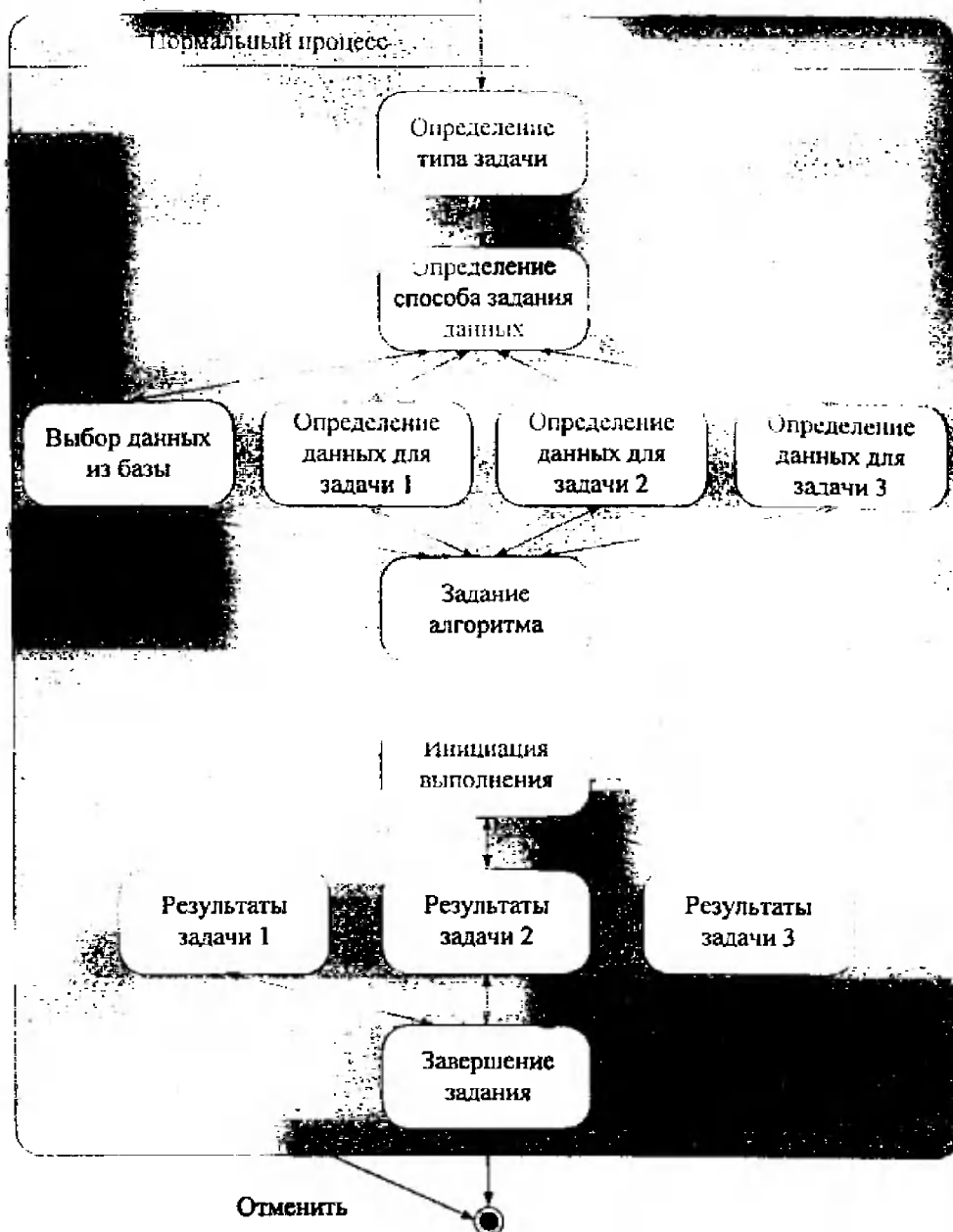


Рис. 8.19. Диаграмма состояний преобразованного диалога Новое задание (вариант 2)

Первое окно реализует выбор типа задачи (рис. 8.20, а). Результат выбора фиксируется в специальном документе - Протоколе. Второе - определение способа задания данных (рис. 8.20, б), третье - непосредственно задание данных в зависимости от выбранного способа (рисунок отсутствует, так как формы определения данных зависят от задачи и их целесообразно проектировать отдельно для каждой задачи вместе с формами вывода результатов). Четвертое - выбор алгоритма (рис. 8.20, в). Пятое - инициацию выполнения (рис. 8.20, г). Шестое - демонстрирует результат (рисунок отсутствует). Седьмое - определяет, что следует сделать с результатом (рис. 8.20, д). Все диалоги строятся по максимально схожей схеме, что упрощает пользователю ориентацию в них.

Оба рассмотренных варианта имеют недостатки. Так, реализация в виде формы содержит слишком много кнопок, регулирующих процесс, а реализация в виде последовательности диалогов

предлагает слишком много шагов, одновременно усложняя доступ к данным и результатам, представляемым в виде отдельных форм.

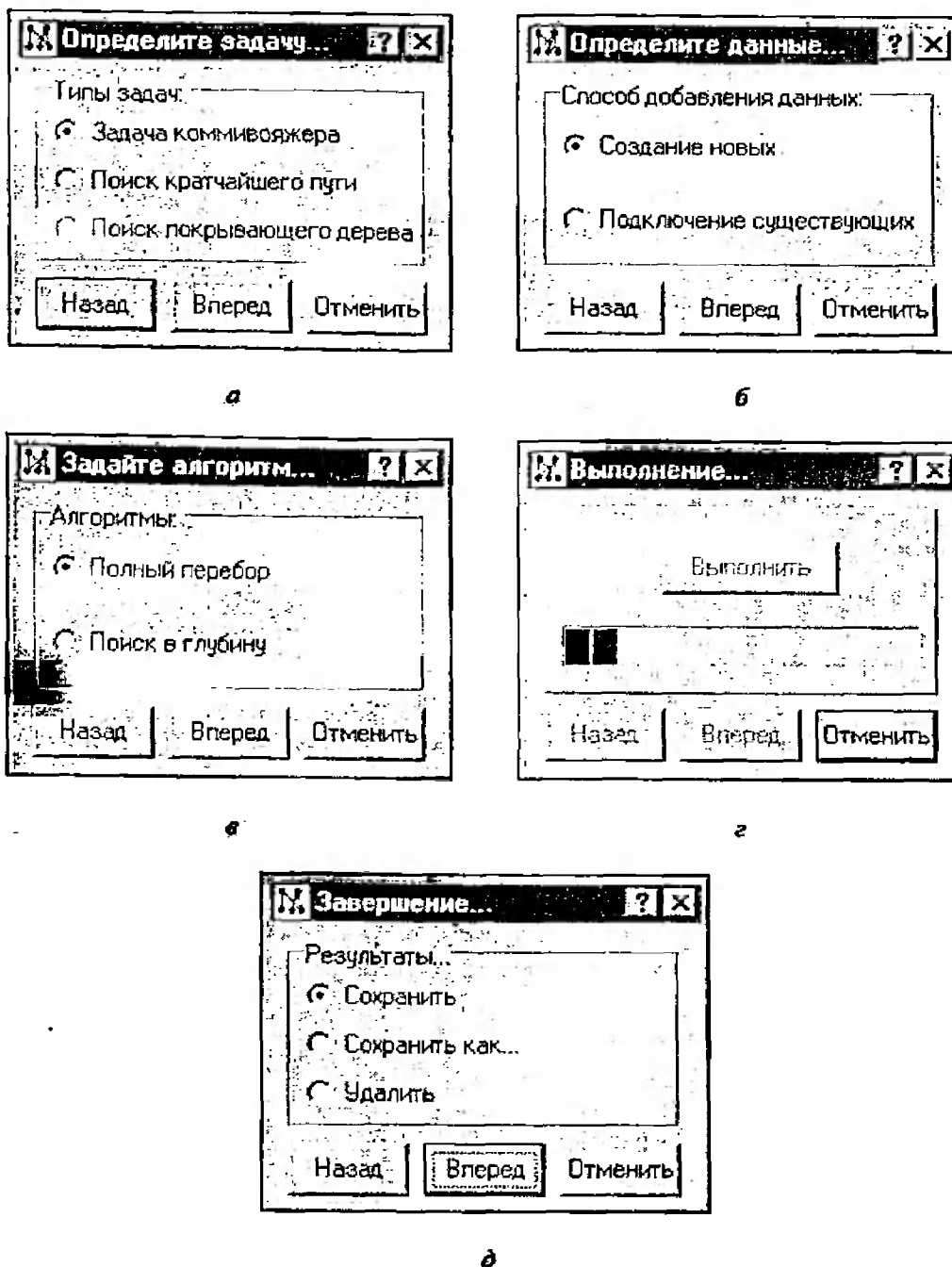


Рис. 8.20. Основные формы диалога Новое задание:

а – диалоговое окно определения типа задачи; *б* – диалоговое окно определения способа задания данных; *в* – диалоговое окно задания алгоритма; *г* – диалоговое окно инициации выполнения; *д* – диалоговое окно завершения задания

Вариант 3. Рассмотрим вариант реализации, который улучшает навигацию за счет использования закладок. Это позволяет визуальнo разнести кнопки, четко обозначив, что кнопки нижнего ряда относятся к протоколу в целом. В таком интерфейсе достаточно просто посмотреть данные и результаты - для этого просто переходим на другую страницу (рис. 8.21).

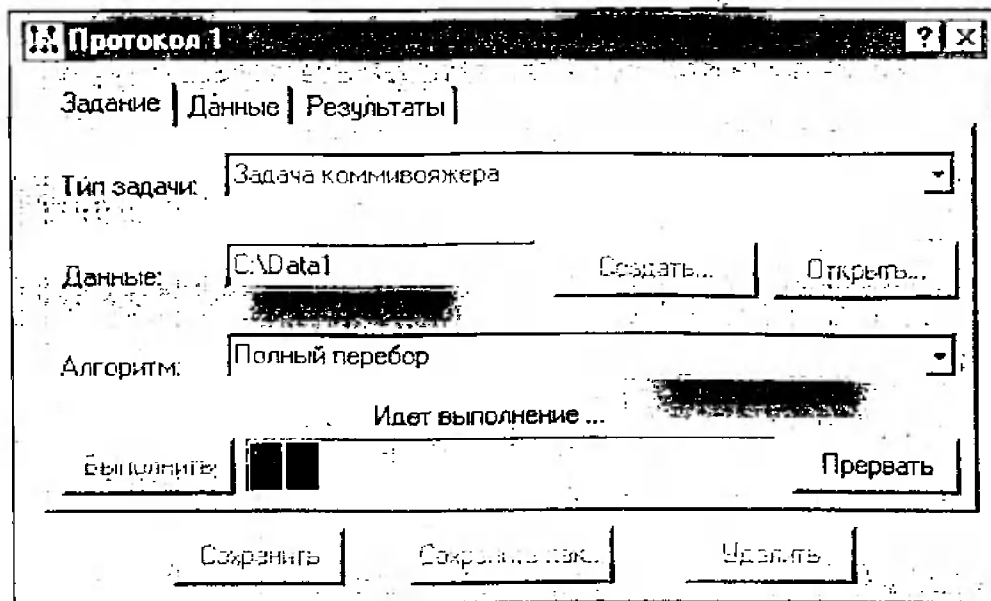


Рис. 8.21. Реализация диалога Новое задание с использованием закладок (вариант 3)

Интерфейс, полученный в результате реализации диалогов, проверяют на полноту, а затем предлагают пользователю для тестирования удобства применения. При наличии сомнений можно предложить несколько вариантов. После одобрения интерфейса его реализуют, кодируя соответствующие процедуры.

8.7. Пользовательские интерфейсы прямого манипулирования и их проектирование

Возможность прямого манипулирования, предусмотренная в WIMP интерфейсах, позволяет разрабатывать для приложений объектно-ориентированные интерфейсы прямого манипулирования.

Интерфейсы данного типа на внешнем уровне используют директивную форму диалога: ввод команды осуществляется при выполнении определенных действий с пиктограммой объекта мышью. Основными элементами этих интерфейсов являются: метафоры, объекты, представления объектов и технология Drag and Drop («перетащи и бросил»).

Метафоры. *Метафора* - мысленный перенос свойств или признаков одного объекта на другой, чем-то аналогичный первому. Использование метафор в интерфейсах предполагает активизацию имеющегося у пользователя опыта (ментальных моделей выполнения аналогичных действий в повседневной жизни или на рабочем месте).

Интерфейс прямого манипулирования должен обеспечивать пользователю среду, содержащую знакомые ему элементы, с которыми пользователь не раз встречался в профессиональной деятельности или в быту, и предоставлять ему возможность манипулирования отдельными объектами. Наличие метафор упрощает для пользователя процесс освоения интерфейса. Например, метафора «Выбрасывание мусора», которую использует Windows для удаления файлов, облегчает пользователю усвоение этой операции.

Использовать метафоры надо очень аккуратно, так как при этом смысл придается всем элементам интерфейса, например, похожие элементы должны вести себя похожим образом, а элементы, выделенные одним цветом, должны находиться в определенной связи друг с другом. Семантическое несоответствие между элементами интерфейса, тем, что от них ожидают, и тем, что они на самом деле выполняют, раздражает и дезориентирует пользователей.

Следует также учитывать, что полное соответствие может обмануть ожидание пользователя, так как все-таки он оперирует не реальными предметами, а их моделями. А значит, его возможности ограничены, о чем необходимо напоминать. Поэтому целесообразно не делать изображения слишком реалистичными.

Метафоры и анимация. При реализации метафор все большая роль уделяется средствам мультимедиа, в основном анимации. В § 8.2 уже упоминалось, что движение привлекает внимание, а резкая смена кадров требует некоторого времени на определения связи данного кадра с предыдущим и на изучение этого кадра. Следовательно, используя мультипликацию, можно не только развлекать пользователя, но и «готовить» его к смене кадров, сокращая время, необходимое на адаптацию к изменившейся ситуации. Например, длинный список можно представить в виде стены, уходящей в бесконечность (по закону перспективы). Тогда «движение» вдоль этой стены, сопровождаемое «естественным» укрупнением названий, позволит рассматривать список, отыскивая необходимую информацию без резкого изменения картинки. При этом *в сознании человека сохраняется идентичность объектов*, а потому он постоянно готов к взаимодействию с ними.

Однако, решая проблемы, связанные с особенностями восприятия человека, анимационные интерфейсы создают дополнительные проблемы для разработчиков и программистов. К ставшим привычными функциональному и интерфейсному уровням программы добавляется еще и визуальный уровень. Программа, реализующая такой интерфейс, никогда не простаивает, так как во время ожидания ввода команды пользователя она продолжает отображать соответствующие кадры. В основе таких программ лежит *временное программирование*. В отличие от событийного программирования, которое позволяет связывать изображение на экране с внешними и внутренними событиями в системе, временное программирование обеспечивает изменение проецируемой *последовательности кадров* в зависимости от состояния моделируемых процессов и действий пользователя.

Объекты интерфейса прямого манипулирования и их представления. Существует три основных типа объектов интерфейсов прямого манипулирования: объекты-данные, объекты-контейнеры и объекты-устройства.

Объекты-данные снабжают пользователя информацией. Это могут быть тексты, изображения, электронные таблицы, музыка, видео и т. п., а также любая их комбинация. В рамках операционной системы таким объектам соответствуют приложения, которые запускаются при раскрытии объекта. В масштабе приложения объекту соответствует одна или несколько форм, в которых содержимое объекта представляется в разных видах. Операции с содержимым объекта реализуются обработчиками событий форм.

Объекты-контейнеры могут манипулировать своими внутренними объектами, в том числе и другими контейнерами, например, копировать их или сортировать в любом порядке. К типичным контейнерам относятся папки, корзины т. п. При раскрытии контейнера демонстрируются сохраняемые им компоненты, и появляется возможность ими манипулировать. Компоненты при этом могут обозначаться пиктограммами или представляться в виде таблицы.

Объекты-устройства часто представляют устройства, существующие в реальном мире: телефоны, факсы, принтеры и т. д., их используют для обозначения этих устройств в абстрактном мире интерфейса. При раскрытии такого объекта, как правило, можно увидеть его настройки.

Итак, каждому объекту соответствует, по крайней мере, одно окно. В исходном состоянии это окно представлено пиктограммой, но при необходимости его можно раскрыть и выполнить требуемые операции, например настройки объекта. Окно объекта в раскрытом состоянии может содержать меню и панели инструментов. Пиктограмме же должно соответствовать контекстное меню, содержащее перечень операций над объектом.

Имя пиктограммы формируют по-своему для каждого типа объектов. Так пиктограммам объектов-данных присваивают имена, соответствующие именам хранимых данных, а тип данных кодируется самой пиктограммой. Имя пиктограммы-контейнера или пиктограммы устройства обозначает сам объект, а потому не зависит от содержимого.

Следует иметь в виду, что различие между типами объектов является условным, так как один и тот же объект в разных ситуациях может вести себя то, как объект-данные, то, как объект-устройство, то, как объект-контейнер. Например, принтер обычно рассматривают как объект-устройство, но он может обладать и свойствами объекта-контейнера, так как может содержать объекты-данные в очереди на печать. Соответственно в Windows объект контейнер/устройство Принтер имеет, помимо пиктограммы (рис. 8.22, а), еще два представления: окно очереди на печать (рис. 8.22, б) и окно настроек (рис. 8.22, в). Имя представления в этом случае целесообразно указывать в заголовке окна объекта.

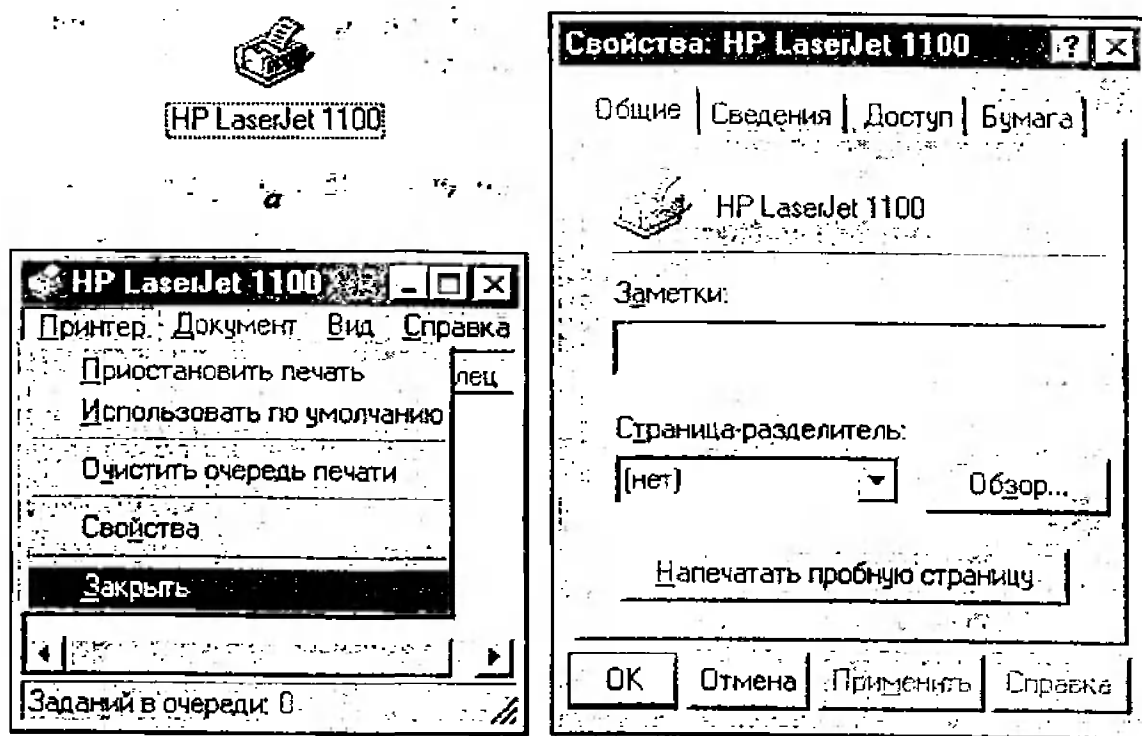


Рис. 8.22. Объект Принтер:

а – пиктограмма; б – окно настроек; в – окно очереди на печать

Технология Drag and Drop. Технология Drag and Drop («перетащи и броси») определяет основные принципы прямого манипулирования, описанные в руководстве по разработке пользовательских интерфейсов фирмы IBM (CUA - Common User Access):

- результат перемещения объекта должен соответствовать ожиданиям пользователя;
- пользователи не должны неожиданно терять информацию;
- пользователь должен иметь возможность отменить неправильное действие.

Эта технология также определяет основные принципы визуализации операции прямого манипулирования:

- исходное выделение - используется в качестве обратной связи пользователю, чтобы сообщить ему, что объект захвачен, в Windows с этой целью используется выделение цветом;
- визуализация перемещения - используется для идентификации выполняемого действия;
- целевое выделение - используется для идентификации пункта назначения, показывая, таким образом, куда «упадет» объект, если его отпустить в текущий момент времени;

- **визуализация действия** - используется для обозначения времени ожидания завершения операции, обычно с этой целью применяют анимацию или изменение формы курсора на «песочные часы».

Следует также иметь в виду, что существует два вида пунктов назначения: один принимает объект, а другой его копию. Например, если пользователь «бросает» документ в «корзину», то уничтожается сам документ, который при этом удаляется с экрана. А если тот же документ он «бросает» на «принтер», то документ не пропадает: на принтер передается копия документа.

В Windows соответствующие действия идентифицируются появлением символа копирования «+» около пиктограммы документа в тот момент, когда он перемещается над пиктограммой устройства, на которое передается копия документа. Если для некоторого устройства возможны оба вида действий, то следует предусмотреть возможность уточнения вида действия. Так Windows в этом случае осуществляет копирование при нажатой клавише CTRL и перемещение в противном случае.

Проектирование интерфейсов прямого манипулирования. Проектирование интерфейсов прямого манипулирования выполняется на основе графов диалога, разработанных для конкретного программного обеспечения, и включает следующие процедуры:

- формирование *множества объектов предметной области*, которое должно быть представлено на экране, причем в качестве основы в этом случае используют не варианты использования, а концептуальную модель предметной области;
- анализ *объектов*, определение их *типов и представлений*, а также перечня *операций* с этими объектами;
- уточнение *взаимодействия объектов* и построение матрицы *прямого манипулирования*;
- определение *визуальных представлений* объектов;
- разработка *меню окон объектов и контекстных меню*;
- создание *прототипа* интерфейса;
- тестирование на *удобство использования*.

Пример 8.6. Разработать пользовательский интерфейс прямого манипулирования для системы решения комбинаторно-оптимизационных задач.

Поскольку единственно напрашивающаяся аналогия - это выполнение операций вручную, интерфейс строим, используя метафору «рабочий стол».

Множество объектов-кандидатов формируем, анализируя концептуальную модель предметной области (см. рис. 6.9) и варианты использования. Для каждого объекта определяем тип и набор операций, связывающих эти объекты с остальными объектами предметной области (табл. 8.3).

Основной объект проектируемой системы - Протокол. Он будет объединять Задание, Данные и Результаты. В процессе работы пользователю понадобится создавать новые Протоколы. При этом целесообразно, чтобы новый Протокол уже содержал бланк Задания. Данные разрешим создавать отдельно, для чего предусмотрим Бланк Данных. После заполнения бланка Данные можно будет включить в Протокол, сохранить в Списке данных, распечатать или выбросить. Результаты будут добавляться в Протокол после решения задачи, т. е. выполнения Задания. Предусмотрим возможность сохранения Протокола в Списке протоколов на любом шаге заполнения, печати и удаления его со стола и из Списка протоколов.

К множеству объектов системы, помимо указанных выше, добавлены объекты-устройства: Компьютер, Принтер, Корзина, необходимые для отображения операций Выполнить, Распечатать, Удалить.

Далее строим *матрицу* (таблицу) *взаимодействия объектов*. В этой таблице по вертикали располагаем объекты, которые согласно метафоре можно перемещать (*исходные*), а по горизонтали - объекты, которые могут служить конечными пунктами перемещения (*конечные*). В самой таблице фиксируем действия, которым будет соответствовать операция перемещения начального объекта на конечный.

Таблица 8.3

Объект	Тип	Действия, связанные с другими объектами	Примечание
Протокол	Контейнер	Сохранить, удалить, распечатать	Включает Задание, Данные и после решения задачи – Результаты
Список протоколов	Контейнер	Очистить	-
Задание	Данные	Выполнить	Бланк задания входит в бланк протокола
Данные	Данные	Связать с протоколом, сохранить, удалить, распечатать	-
Список данных	Контейнер	Очистить	-
Результаты	Данные	-	Если получены, то связаны с Заданием и Данными, т. е. Находятся в Протоколе
Бланк протокола	Данные	-	Бланк протокола содержит бланк задания
Бланк данных	Данные	-	-
Компьютер	Устройство	-	-
Принтер	Устройство	-	-
Корзина	Устройство	-	-

Для рассматриваемого примера исходными объектами являются Протокол, Список протоколов, Задание, Данные, Список протоколов, Бланк задания и Бланк данных (табл. 8.4). Их можно перемещать на объекты-контейнеры: Протокол, Список протоколов, Список данных, а также объекты устройства: Компьютер, Принтер, Корзина и Рабочий стол. Далее анализируем взаимодействие объектов в соответствии с принятой метафорой. Так, если объект Протокол перенести на объект Список протоколов, то по смыслу это можно интерпретировать как желание пользователя добавить Протокол в список. Аналогично заполняем всю таблицу, которая затем будет использоваться при программировании соответствующих событий.

На рис. 8.23 представлен внешний вид интерфейса системы. (В качестве пиктограмм объектов использованы стандартные пиктограммы Windows.)

После этого необходимо разработать представление окон объектов.

Таблица 8.4

Исходные объекты	Операции, выполняемые при перемещении объектов						
	Конечные объекты						
	Протокол	Список протоколо	Список данных	Компьютер	Принтер	Корзина	Рабочий стол
Протокол	-	Добавить протокол в список	-	Выполнить задание*	Распечатать протокол*	Удалить протокол	-
Список протоколо в	-	-	-	-	-	Удалить все протоколы	-
Задание	Занести задание в протокол	-	-	-	-	Удалить задания	-
Данные	Занести данные в протокол	-	Добавить данные в список	-	-	Удалить данные	-
Список данных	-	-	-	-	-	Удалить все данные	-
Бланк протокола	-	-	-	-	-	-	Создать протокол *
Бланк данных	-	-	-	-	-	-	Создать протокол *

* Операция выполняется с копией объекта, сам объект остается на своем месте.

Пример 8.7. Разработать представление окна объекта Протокол.

Объект Протокол является контейнером, который может содержать Задание, Данные и Результаты. В момент создания Протокола автоматически формируется чистый бланк Задания.

На рис. 8.24 показано стандартное представление объекта-контейнера, которое может быть использовано в данном примере. Отдельно изображено, как это окно будет выглядеть в разные моменты времени. Окно содержит меню, включающие пункты: Данные, Правка, Вид и Помощь. Первый пункт позволяет определить данные: Создать или Открыть. Второй - отвечает за работу с общим буфером и содержит пункты Вырезать, Копировать и Вставить. Третий - традиционно управляет видом окна: Крупные значки, Мелкие значки, Список и Таблица. Четвертый - используют для вызова справки.

Поскольку используемая метафора рабочего стола позволяет, необходимо предусмотреть возможность выполнения стандартных клавиатурных команд с объектом Протокол. К ним относят операции Вырезать (Ctrl-X), Копировать (Ctrl-C), Вставить (Ctrl-V).

После завершения проектирования представлений всех объектов создают прототип интерфейса и передают его на тестирование удобства использования.

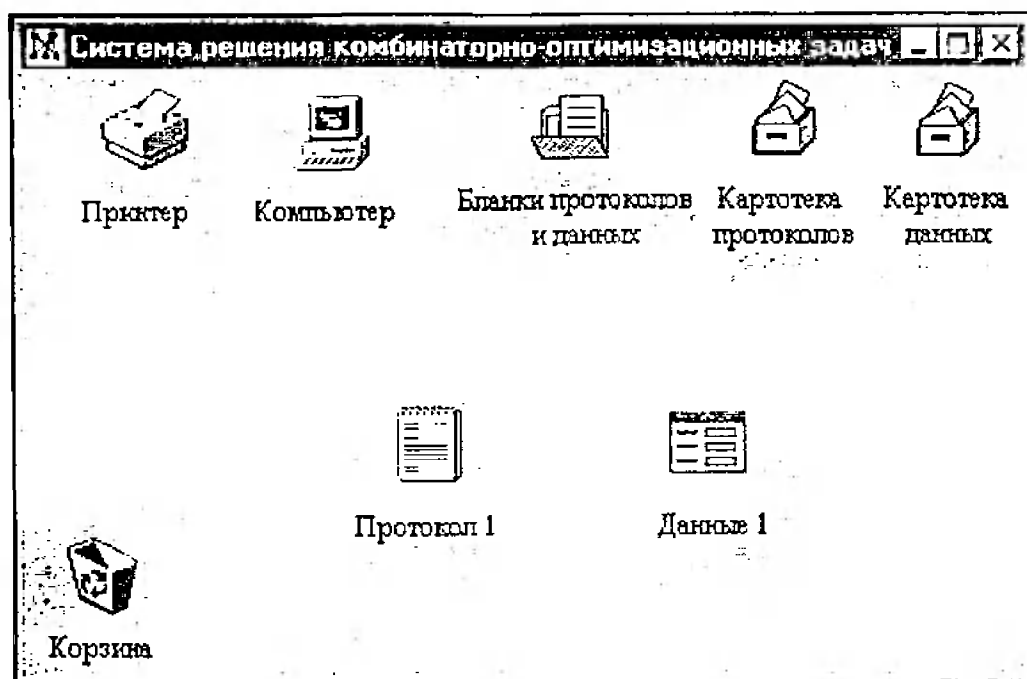
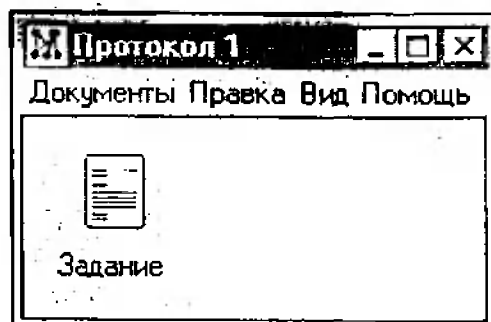
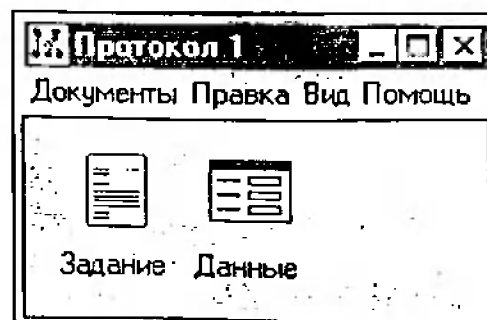


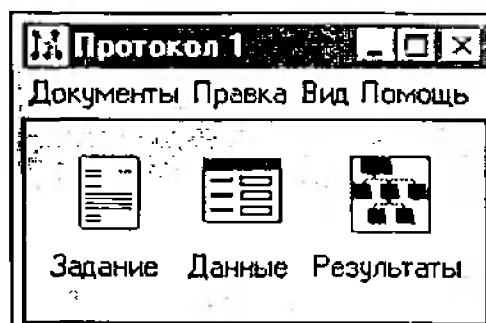
Рис. 8.23. Внешний вид интерфейса прямого манипулирования системой решения комбинаторно-оптимизационных задач



a



б



в

Рис. 8.24. Окно объекта Протокол (вид Крупные значки):

a — в момент создания; *б* — после добавления данных; *в* — после выполнения задания

8.8. Интеллектуальные элементы пользовательских интерфейсов

В последние годы появилось много новых перспективных элементов пользовательских интерфейсов, в основном приносящих в интерфейс элементы искусственного интеллекта, что проявляется в их названиях: Мастер, Советчик, Агент. Сделано множество попыток создания социализированного пользовательского интерфейса. В основе такого интерфейса лежит идея создания персонифицированного, т. е. «имеющего личность», интерфейса. Развлекающие программы, такие как Cats (Кошки) и Dogs (Собаки), реализующие достаточно сложное поведение домашних животных в разных ситуациях, показывают, что технически это вполне решаемая задача. Однако в этой области существуют психологические проблемы. В качестве примера вспомним, что даже «безобидный» Советчик Microsoft Office, рассмотренный ниже, вызывает у многих пользователей резко отрицательную реакцию. Пока попытки создания такой «личности» успеха не имели.

Советчики. Советчики представляют собой форму подсказки. Обычно их можно вызвать с помощью меню справки, командной строки окна или из всплывающего меню. Советчики помогают пользователям в выполнении конкретных задач, но только, если пользователь представляет, что ему нужно сделать. Например, пользователь, работающий в Microsoft Word, собирается вставить в документ рисунок, но не знает как. Он активизирует Помощника-Скрепку и вводит вопрос в специальное поле (рис. 8.25, а). Справочная система анализирует вопрос и формирует список тем, косвенно связанных с интересующей пользователя, в расчете, что пользователь сам выберет нужную справку (рис. 8.25, б).

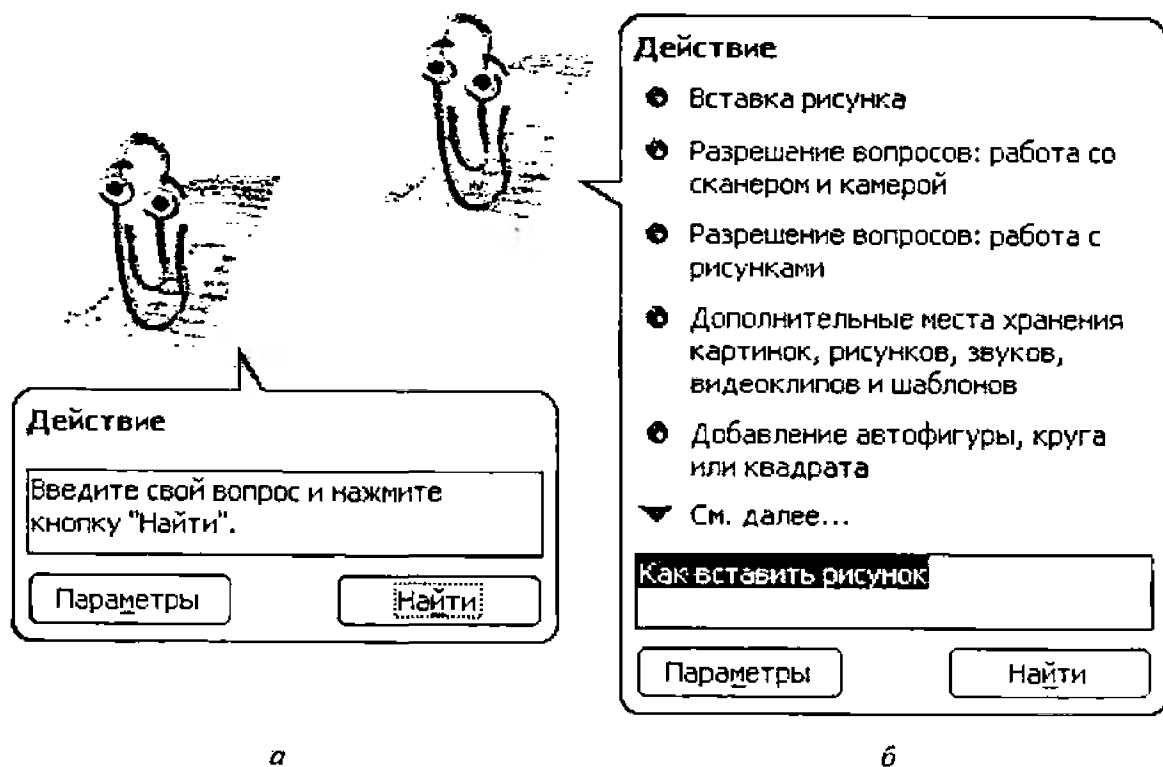


Рис. 8.25. Помощник-Скрепка Microsoft Word 2000:

а – запрос; б – список связанных тем

Мастера. Программу-мастер используют для выполнения общераспространенных, но редко выполняемых отдельным пользователем задач, таких, как установка программ или оборудования. Выполнение подобных действий требует от пользователя принятия сложных взаимосвязанных

решений, последовательность которых и диктует программа-мастер. Интеллектуальные Мастера способны на каждом шаге демонстрировать в окне просмотра результаты ответов пользователя на предыдущие вопросы, помогая последнему сориентироваться в ситуации.

Мастер реализует последовательный или древовидный сценарий диалога, поэтому его целесообразно использовать для решения хорошо структурированных, последовательных задач (рис. 8.26). При этом необходимо:

- предоставить пользователю возможность возврата на предыдущий шаг;
- предусмотреть возможность отмены работы Мастера;
- нумеровать шаги и сообщать пользователю количество шагов Мастера, особенно, если таких шагов больше трех;
- пояснять пользователю каждый шаг;
- по возможности демонстрировать результат уже выполненных операций на каждом шаге.

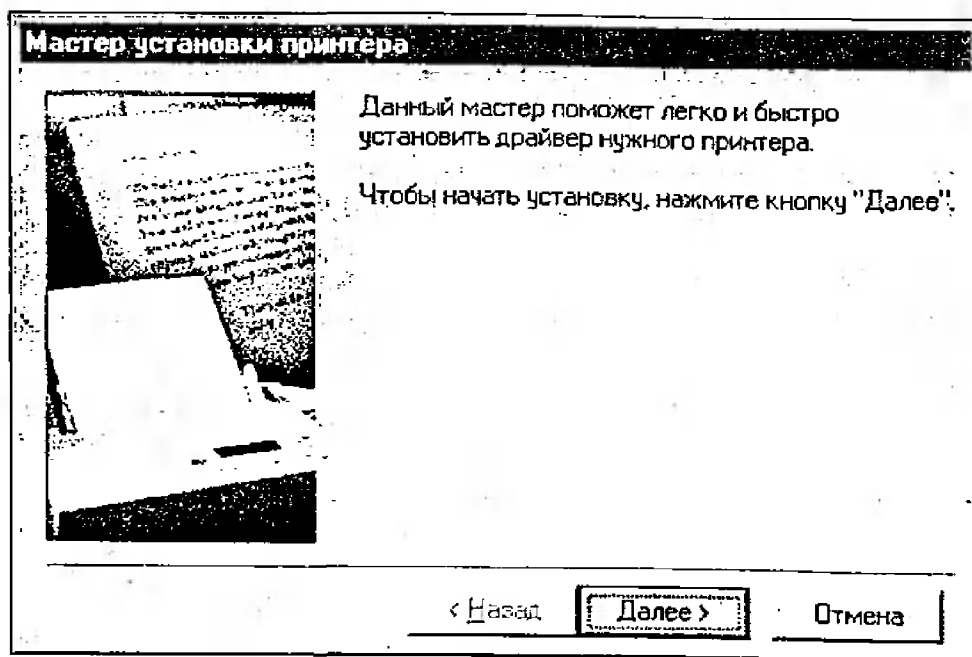


Рис. 8.26. Первое окно мастера Установка принтера Windows' 98

Программные агенты. Наибольший интерес на настоящий момент представляют программные агенты, используемые для выполнения рутинной работы. Такой программный агент является элементом программного обеспечения, которому пользователь может передать часть своих обязанностей. Основными функциями Агентов-Помощников являются: наблюдение, поиск и управление. Различают:

- программы-агенты, настраиваемые на выполнение указанных задач;
- программы-агенты, способные обучаться, например, фиксируя действия пользователя (по типу магнитофона).

Создание агентов последнего типа, например, доступно через механизм макросов Microsoft Office.

Большинство интересных и достаточно сложных программных агентов в настоящее время «живет» в Интернете, где и можно найти последнюю информацию по данной теме.

Контрольные вопросы и задания

1. Назовите основные типы интерфейсов. Чем характеризуется каждый из них? Какими средствами реализуется? Какие типы интерфейсов являются основными в наше время?
2. Перечислите психофизические особенности человека, которые необходимо учитывать при проектировании интерфейсов. Какие ограничения это накладывает на интерфейс?
3. Что понимают под термином «диалог»? Сколько диалогов может реализовывать программное обеспечение?
4. Назовите основные типы диалога и его формы. Какие модели используют для описания диалогов? Что служит исходными данными для проектирования диалогов?
5. Постройте граф диалога для простейшего графического редактора. Почему он имеет такой вид? В каких ситуациях граф диалога имеет вид цепи или дерева?
6. Предложите меню графического редактора. Сравните это меню с меню известных вам графических редакторов. Проанализируйте отличия.
7. Перечислите основные компоненты графических пользовательских интерфейсов. В каких случаях используют каждый из них?
8. Предложите интерфейс прямого манипулирования для графического редактора. В чем состоит основная сложность проектирования таких интерфейсов? В каких случаях их целесообразно использовать?
9. Какие интеллектуальные компоненты пользовательских интерфейсов существуют в настоящее время? Каковы их основные назначения? В каких случаях их целесообразно применять?

9. ТЕСТИРОВАНИЕ ПРОГРАММНЫХ ПРОДУКТОВ

Тестирование - очень важный и трудоемкий этап процесса разработки программного обеспечения, так как правильное тестирование позволяет выявить подавляющее большинство ошибок, допущенных при составлении программ.

Процесс разработки программного обеспечения предполагает три стадии тестирования: автономное, комплексное и системное, каждая из которых соответствует завершению соответствующей части Системы.

Различают два подхода к формированию тестов: структурный и функциональный. Каждый из указанных подходов имеет свои особенности и области применения.

9.1. Виды контроля качества разрабатываемого программного обеспечения

Недостаточно выполнить проектирование и кодирование программного обеспечения, необходимо также обеспечить его соответствие требованиям и спецификациям. Многократно проводимые исследования показали, что чем раньше обнаруживаются те или иные несоответствия или ошибки, тем больше вероятность их правильного исправления (рис. 9.1, а) и ниже его стоимость (рис. 9.1, б) [7].

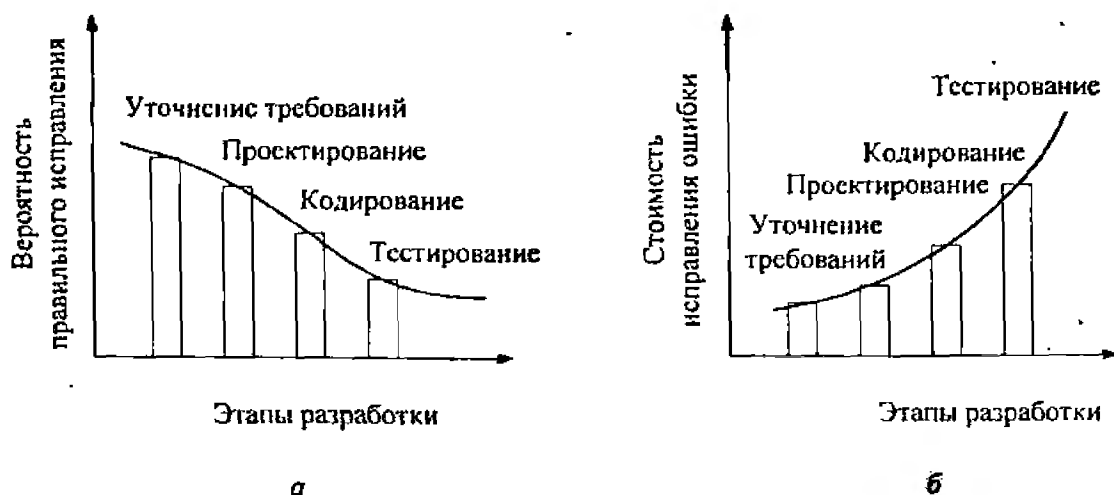


Рис. 9.1. Зависимость вероятности правильного исправления (а) и его стоимости от этапа разработки (б)

Современные технологии разработки программного обеспечения предусматривают раннее обнаружение ошибок за счет выполнения контроля результатов всех этапов и стадий разработки. На начальных этапах такой контроль осуществляют в основном вручную или с использованием CASE-средств, на последних - он принимает форму тестирования.

Тестирование - это процесс выполнения программы, целью которого является выявление ошибок. Никакое тестирование не может доказать отсутствие ошибок в хоть сколько-нибудь сложном программном обеспечении. Для такого программного обеспечения выполнение полного тестирования, т. е. задания всех возможных комбинаций исходных данных, становится невозможным, а, следовательно, всегда имеется вероятность того, что в программном обеспечении остались не выявленные ошибки. Однако соблюдение основных правил тестирования и научно обоснованный подбор тестов может уменьшить их количество.

Примечание. Обычно на вопрос о цели тестирования начинающие программисты отвечают, что целью тестирования является «доказательство правильности программы». Это абсолютно неверное мнение. Г. Майерс [47] предлагает очень удачную аналогию для пояснения этого положения. Представьте себе, что вы пришли на прием к врачу и пожаловались на боль в боку. Врач выслушал вас и направил на обследование. Через некоторое время вы возвращаетесь к врачу с ворохом заключений и результатов анализов, и во всех этих бумагах написано, что все исследуемые параметры у вас в норме. Но бок то болит, значит, что-то не в порядке, хотя анализы этого и не показывают... Так и сложное программное обеспечение, безошибочно работающее на всех тестовых наборах, может содержать и обычно содержит некоторое количество ошибок.

Процесс разработки программного обеспечения, в том виде, как он определяется в современной модели жизненного цикла программного обеспечения, предполагает три стадии тестирования:

- автономное тестирование компонентов программного обеспечения;
- комплексное тестирование разрабатываемого программного обеспечения;
- системное или оценочное тестирование на соответствие основным критериям качества.

Для повышения качества тестирования рекомендуется соблюдать следующие *основные принципы*:

- предполагаемые результаты должны быть известны до тестирования;
- следует избегать тестирования программы автором;
- необходимо досконально изучать результаты каждого теста;
- необходимо проверять действия программы на неверных данных;
- необходимо проверять программу на неожиданные побочные эффекты на неверных данных.

Следует также иметь в виду, что вероятность наличия необнаруженных ошибок в части программы пропорциональны количеству ошибок уже найденных в этой части.

Формирование тестовых наборов. В соответствии с определением тестирования в начале данного параграфа, *удачным* следует считать тест, который обнаруживает хотя бы одну ошибку. С этой точки зрения хотелось бы использовать такие наборы тестов, каждый из которых с максимальной вероятностью может обнаружить ошибку.

Формирование набора тестов имеет большое значение, поскольку тестирование является одним из наиболее трудоемких этапов (от 30 до 60 % общей трудоемкости) создания программного продукта. Причем доля стоимости тестирования в общей стоимости разработки имеет тенденцию возрастать при увеличении сложности программного обеспечения и повышении требований к их качеству.

Существуют два принципиально различных подхода к формированию тестовых наборов: структурный и функциональный.

Структурный подход базируется на том, что *известка структура* тестируемого программного обеспечения, в том числе его алгоритмы («стеклянный ящик»). В этом случае тесты строят так, чтобы проверить правильность реализации заданной логики в коде программы.

Функциональный подход основывается на том, что структура программного обеспечения не известна («черный ящик»). В этом случае тесты строят, опираясь на функциональные спецификации. Этот подход называют также подходом, управляемым данными, так как при его использовании тесты строят на базе различных способов декомпозиции множества данных.

Наборы тестов, полученные в соответствии с методами этих подходов, обычно объединяют, обеспечивая всестороннее тестирование программного обеспечения.

Более подробное рассмотрение перечисленных вопросов начнем с обсуждения методов ручного контроля.

9.2. Ручной контроль программного обеспечения

Ручной контроль, как указано выше, обычно используют на ранних этапах разработки. Все проектные решения, принятые на том или ином этапе, должны анализироваться с точки зрения их правильности и целесообразности как можно раньше, пока их можно легко пересмотреть. Поскольку возможность практической проверки подобных решений на ранних этапах разработки отсутствует, большое значение имеет их обсуждение, которое проводят в разных формах.

Различают статический и динамический подходы к ручному контролю. При *статическом* подходе анализируют структуру, управляющие и информационные связи программы, ее входные и выходные данные. При *динамическом* - выполняют *ручное тестирование*, т. е. вручную моделируют процесс выполнения программы на заданных исходных данных.

Исходными данными для таких проверок являются: техническое задание, спецификации, структурная и функциональная схемы программного продукта, схемы отдельных компонентов и т. д., а для более поздних этапов - алгоритмы и тексты программ, а также тестовые наборы.

Доказано, что ручной контроль способствует существенному увеличению производительности и повышению надежности программ и с его помощью можно находить от 30 до 70 % ошибок логического проектирования и кодирования. Следовательно, один или несколько из методов ручного контроля обязательно должны использоваться в каждом программном проекте.

Основными методами ручного контроля являются:

- инспекции исходного текста,
- сквозные просмотры,
- проверка за столом,
- оценки программ.

Инспекции исходного текста. Инспекции исходного текста представляют собой набор процедур и приемов обнаружения ошибок при изучении текста группой специалистов. В эту группу входят: автор программы, проектировщик, специалист по тестированию и координатор - компетентный программист, но не автор программы. Общая процедура инспекции предполагает следующие операции:

- участникам группы заранее выдается листинг программы и спецификация на нее;
- программист рассказывает о логике работы программы и отвечает на вопросы инспекторов;
- программа анализируется по списку вопросов для выявления исторически сложившихся общих ошибок программирования.

Список вопросов для инспекций исходного текста зависит, как от используемого языка программирования, так и от специфики разрабатываемого программного обеспечения. В качестве примера ниже приведен список вопросов, который можно использовать при анализе правильности программ, написанных на языке Pascal.

I. Контроль обращений к данным

- Все ли переменные инициализированы?
- Не превышены ли максимальные (или реальные) размеры массивов и строк?
- Не перепутаны ли строки со столбцами при работе с матрицами?
- Присутствуют ли переменные со сходными именами?
- Используются ли файлы? Если да, то при вводе из файла проверяется ли завершение файла?
- Соответствуют ли типы записываемых и читаемых значений?
- Использованы ли нетипизированные переменные, открытые массивы, динамическая память?

Если да, то соответствуют ли типы переменных при «наложении» формата? Не выходят ли индексы за границы массивов?

2. Контроль вычислений

- Правильно ли записаны выражения (порядок следования операторов)?
- Корректно ли выполнены вычисления над неарифметическими переменными?
- Корректно ли выполнены вычисления с переменными различных типов (в том числе с использованием целочисленной арифметики)?
- Возможно ли переполнение разрядной сетки или ситуация машинного нуля?
- Соответствуют ли вычисления заданным требованиям точности?
- Присутствуют ли сравнения переменных различных типов?

3. Контроль передачи управления

- Будут ли корректно завершены циклы?
- Будет ли завершена программа?
- Существуют ли циклы, которые не будут выполняться из-за нарушения условия входа? Корректно ли продолжатся вычисления?
- Существуют ли поисковые циклы? Корректно ли отрабатываются ситуации «элемент найден» и «элемент не найден»?

4. Контроль межмодульных интерфейсов

- Соответствуют ли списки параметров и аргументов по порядку, типу, единицам измерения?
- Не изменяет ли подпрограмма аргументов, которые не должны изменяться?
- Не происходит ли нарушения области действия глобальных и локальных переменных с одинаковыми именами?

Кроме непосредственного обнаружения ошибок, результаты инспекции позволяют программисту увидеть другие сделанные им ошибки, получить возможность оценить свой стиль программирования, выбор алгоритмов и методов тестирования. Инспекция является способом раннего выявления частей программы, с большей вероятностью содержащих ошибки, что позволяет при тестировании уделить внимание именно этим частям.

Сквозные просмотры. Сквозной просмотр, как и инспекция, представляет собой набор способов обнаружения ошибок, осуществляемых группой лиц, просматривающих текст программы. Такой просмотр имеет много общего с процессом инспектирования, но отличается процедурой и методами обнаружения ошибок. Группа по выполнению сквозного контроля состоит из трех-пяти человек: председатель или координатор, секретарь, фиксирующий все ошибки, специалист по тестированию, программист и независимый эксперт. Сквозной просмотр предполагает выполнение следующих процедур:

- участникам группы заранее выдают листинг программы и спецификацию на нее;
- участникам заседания предлагают несколько тестов;
- участники заседания мысленно выполняют каждый тест в соответствии с логикой программы, при этом состояние программы (значения переменных) отслеживается на бумаге или доске;
- при необходимости программисту задают вопросы о логике проектирования и принятых допущениях.

В большинстве сквозных просмотров при выполнении самих тестов находят меньше ошибок, чем при опросе программиста.

Проверка за столом. Исторически данный метод ручного тестирования появился первым, так как он не требует наличия группы специалистов. Это - проверка исходного текста или сквозные просмотры, выполняемые одним человеком, который читает текст программы, проверяет его на наличие возможных ошибок по специальному списку часто встречающихся ошибок и «пропускает» через программу тестовые данные. Исходя из принципов тестирования, проверку за столом должен проводить человек, не являющийся автором программы. Метод наименее результативен, так как проверка представляет собой полностью неупорядоченный процесс, при ней отсутствует обмен мнениями и здоровая конкуренция.

Оценка программ. Этот метод непосредственно не связан с тестированием, но его использование также улучшает качество программирования. Его используют для анонимной оценки программы в терминах ее общего качества, простоты эксплуатации и ясности. Цель метода - обеспечить сравнительно объективную оценку и самооценку программистов.

Такая оценка выполняется следующим образом. Выбирается программист, который должен выполнять обязанности администратора процесса. Администратор набирает группу от шести до 20-ти участников, которые должны заниматься разработкой сходных программ. Каждому участнику предлагается представить для рассмотрения две программы, с его точки зрения - наилучшую и наихудшую. Отобранные программы случайным образом распределяются между участниками. Им дают по четыре программы - две наилучшие и две наихудшие, но не говорят, какие программы плохие, а какие - хорошие. Программист просматривает эти программы и заполняет анкету, в которой оценивает качество программ по семибалльной шкале.

После этого результаты оценки сверяют, а проверяющий дает общий комментарий и рекомендации по улучшению программ.

9.3. Структурное тестирование

Структурное тестирование называют также тестированием по «маршрутам», так как в этом случае тестовые наборы формируют путем анализа маршрутов, предусмотренных алгоритмом. Под *маршрутами* при этом понимают последовательности операторов программы, которые выполняются при конкретном варианте исходных данных.

В основе структурного тестирования лежит концепция максимально полного тестирования всех маршрутов программы. Так, если алгоритм программы включает ветвление, то при одном наборе исходных данных может быть выполнена последовательность операторов, реализующая действия, которые предусматривает одна ветвь, а при втором - другая. Соответственно, для программы будут существовать маршруты, различающиеся выбранным при ветвлении вариантом.

Считают, что программа проверена полностью, если с помощью тестов удастся осуществить выполнение программы по всем возможным маршрутам передач управления. Однако нетрудно видеть, что даже в программе среднего уровня сложности число неповторяющихся маршрутов может быть очень велико, и, следовательно, полное или *исчерпывающее* тестирование маршрутов, как правило, невозможно.

Структурный подход к тестированию имеет ряд недостатков. Так тестовые наборы, построенные по данной стратегии:

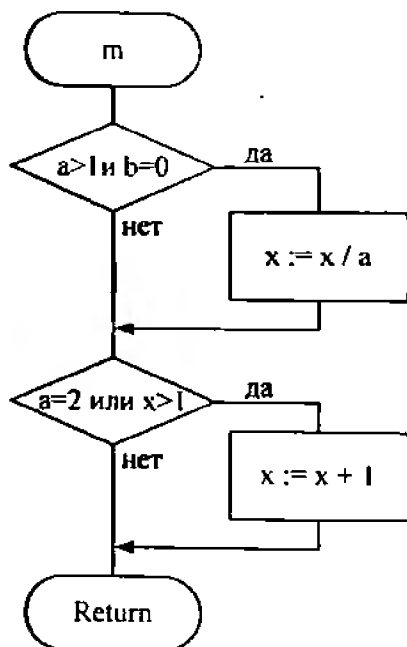
- не обнаруживают пропущенных маршрутов;
- не обнаруживают ошибок, зависящих от обрабатываемых данных, например, в операторе `if (a - b) < eps` - пропуск функции абсолютного значения `abs` проявится только, если $a < b$;
- не дают гарантии, что программа правильна, например, если вместо сортировки по убыванию реализована сортировка по возрастанию.

Для формирования тестов программу представляют в виде графа, вершины которого соответствуют операторам программы, а дуги представляют возможные варианты передачи управления. Ниже приведен текст программы, которая определяет значение x в зависимости от значений параметров процедуры. Алгоритм этой программы представлен на рис. 9.2, а, а соответствующий граф передач управления - на рис. 9.2, б.

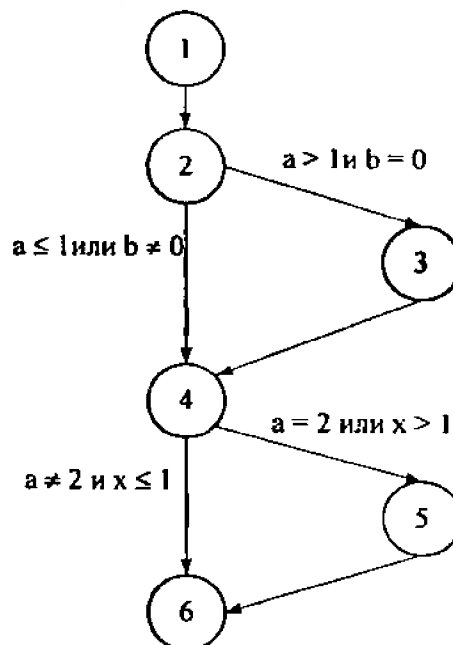
```
Procedure m (a, b: real; var x: real);  
begin  
  if (a=1) and (b=0) then x:=x/a;  
  if (a=2) or (x>1) then x:=x+1;  
end;
```

Формирование тестовых наборов для тестирования маршрутов может осуществляться по нескольким критериям:

- покрытие операторов;
- покрытие решений (переходов);
- покрытие условий;
- покрытие решений/условий;
- комбинаторное покрытие условий.



а



б

Рис. 9.2. Схема алгоритма процедуры примера (а) и ее граф передач управления (б)

Покрывтие операторов. Критерий покрытия операторов подразумевает такой подбор тестов, чтобы каждый оператор программы выполнялся, по крайней мере, один раз. Это необходимое, но недостаточное условие для приемлемого тестирования. Поясним сказанное примером.

Для фрагмента, алгоритм и граф которого представлены на рис. 9.2, можно было бы выполнить каждый оператор один раз, задав в качестве входных данных $a = 2$, $b = 0$, $x = 3$. Но при этом из второго условия следует, что переменная x может принимать любое значение, и в некоторых версиях языка Pascal это значение проверяться не будет (!).

Кроме того:

- если при написании программы в первом условии указано: $(a > 1) \text{ or } (b = 0)$, то ошибка обнаружена не будет;
- если во втором условии вместо $x > 1$ записано $x > 0$, то эта ошибка тоже не будет обнаружена;
- существует путь 1-2-4-6 (см. рис. 9.2, б), в котором x вообще не меняется и, если здесь есть ошибка, она не будет обнаружена.

Таким образом, хотя при тестировании действительно *необходимо* задавать исходные данные так, чтобы все операторы программы были выполнены хотя бы один раз, для проверки программы этого явно недостаточно.

Покрывание решений (переходов). Для реализации этого критерия необходимо такое количество и состав тестов, чтобы результат проверки каждого условия (т.е. решение) принимал значения «истина» или «ложь», по крайней мере, один раз.

Нетрудно видеть, что критерий покрытия решений удовлетворяет критерию покрытия операторов, но является более «сильным».

Программу, алгоритм которой представлен на рис. 9.2, а, можно протестировать по методу покрытия решений двумя тестами, покрывающими либо пути: 1-2-4-6, 1-2-3-4-5-6, либо пути: 1-2-3-4-6, 1-2-4-5-6, например:

$a = 3, b = 0, x = 3$ — путь 1-2-3-4-5-6;
 $a = 2, b = 1, x = 1$ — путь 1-2-4-6.

Однако путь, где x не меняется, будет проверен с вероятностью 50 %: если во втором условии вместо условия $x > 1$ записано $x < 1$, то этими двумя тестами ошибка обнаружена не будет.

Покрывание условий. Критерий покрытия условий является еще более «сильным» по сравнению с предыдущими. В этом случае формируют некоторое количество тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении были выполнены, по крайней мере, один раз.

Однако, как и в случае покрытия решений, этот критерий не всегда приводит к выполнению каждого оператора, по крайней мере, один раз. К критерию требуется дополнение, заключающееся в том, что каждой точке входа управление должно быть передано, по крайней мере, один раз.

Программа, алгоритм которой представлен на рис. 9.2, а, проверяет четыре условия:

1) $a > 1$; 2) $b = 0$; 3) $a = 2$; 4) $x > 1$.

Необходимо реализовать все возможные ситуации:

$a > 1, a \geq 1$, $b = 0, b \neq 0$, $a = 2, a \neq 2$, $x > 1, x \leq 1$.

Тесты, удовлетворяющие этому условию:

$a = 2, b = 0, x = 4$ — путь 1-2-3-4-5-6, условия: 1 - да, 2 - да, 3 - да, 4 - да;

$a = 1, b = 1, x = 1$ — путь 1-2-4-6, условия: 1 - нет, 2 - нет, 3 - нет, 4 - нет.

Критерий покрытия условий часто удовлетворяет критерию покрытия решений, но не всегда. Тесты критерия покрытия условий для ранее рассмотренных примеров покрывают результаты всех решений, но это случайное совпадение. Например, тесты:

$a = 1, b = 0, x = 3$ — путь 1-2-3-6, условия: 1 - нет, 2 - да, 3 - нет, 4 - да;

$a = 2, b = 1, x = 1$ — путь 1-2-3-4-5-6, условия: 1 - да, 2 - нет, 3 - да, 4 - нет

покрывают результаты всех условий, но только два из четырех результатов решений: не выполняется результат «истина» первого решения и результат «ложь» второго.

Основной недостаток метода - недостаточная чувствительность к ошибкам в логических выражениях.

Покрывание решений/условий. Согласно этому методу тесты должны составляться так, чтобы, по крайней мере, один раз выполнились все возможные результаты каждого условия и все результаты каждого решения, и каждому оператору управление передавалось, по крайней мере, один раз.

Анализ, проведенный выше, показывает, что этому критерию удовлетворяют тесты:

$a=2, b=0, x=4$ — путь 1-2-3-4-5-6, условия: 1 - да, 2 - да, 3 - да, 4 - да;
 $a=1, b=1, x=1$ — путь 1-2-4-6, условия: 1 - нет, 2 - нет, 3 - нет, 4 - нет.

Комбинаторное покрытие условий. Этот критерий требует создания такого множества тестов, чтобы все возможные комбинации результатов условий в каждом решении и все операторы выполнялись, по крайней мере, один раз.

Для программы, алгоритм которой представлен на рис. 9.1, необходимо покрыть тестами восемь комбинаций:

- | | |
|---------------------------|---------------------------|
| 1) $a > 1, b = 0$; | 5) $a = 2, x > 1$; |
| 2) $a > 1, b \neq 0$; | 6) $a = 2, x \leq 1$; |
| 3) $a \leq 1, b = 0$; | 7) $a \neq 2, x > 1$; |
| 4) $a \leq 1, b \neq 0$; | 8) $a \neq 2, x \leq 1$. |

Эти комбинации можно проверить четырьмя тестами:

$a=2, b=0, x=4$ — проверяет комбинации (1), (5);
 $a=2, b=1, x=1$ — проверяет комбинации (2), (6);
 $a=1, b=0, x=2$ — проверяет комбинации (3), (7);
 $a=1, b=1, x=1$ — проверяет комбинации (4), (8).

В данном случае то, что четырем тестам соответствует четыре пути, является совпадением. Представленные тесты не покрывают всех путей, например, `acd`. Поэтому иногда необходима реализация восьми тестов.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является набор тестов, который проверяет все результаты каждого решения и передает управление каждому оператору, по крайней мере, один раз.

Для программ, содержащих вычисления, каждое из которых требует проверки более чем одного условия, минимальный набор тестов должен:

- генерировать все возможные комбинации результатов проверок условий для каждого вычисления;
- передавать управление каждому оператору, по крайней мере, один раз.

Термин «возможных» употреблен здесь потому, что некоторые комбинации условий могут быть нереализуемы. Например, для комбинации $k < 0$ и $k > 40$ задать k невозможно.

9.4. Функциональное тестирование

Одним из способов проверки программ является тестирование с управлением по данным или по принципу «черного ящика». В этом случае программа рассматривается как «черный ящик», и целью тестирования является выяснение обстоятельств, в которых поведение программы не соответствует спецификации.

Для обнаружения всех ошибок в программе, используя управление по данным, необходимо выполнить *исчерпывающее* тестирование, т. е. тестирование на всех возможных наборах данных. Для тех же программ, где исполнение команды зависит от предшествующих ей событий, необходимо проверить и все возможные последовательности. Очевидно, что проведение исчерпывающего тестирования для подавляющего большинства случаев невозможно. Поэтому обычно выполняют «разумное» или «приемлемое» тестирование, которое ограничивается прогонами программы на небольшом подмножестве всех возможных входных данных. Этот вариант не дает гарантии отсутствия отклонений от спецификаций.

Правильно выбранный тест должен уменьшать, причем более чем на единицу, число других тестов, которые должны быть разработаны для обеспечения требуемого качества программного обеспечения.

При функциональном тестировании различают следующие методы формирования тестовых наборов:

- эквивалентное разбиение;
- анализ граничных значений;
- анализ причинно-следственных связей;
- предположение об ошибке.

Эквивалентное разбиение. Метод эквивалентного разбиения заключается в следующем. Область всех возможных наборов входных данных программы по каждому параметру разбивают на конечное число групп - *классов эквивалентности*. Наборы данных такого класса объединяют по принципу обнаружения одних и тех же ошибок: если набор какого-либо класса обнаруживает некоторую ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку и наоборот.

Разработку тестов методом эквивалентного разбиения осуществляют в два этапа: на первом выделяют классы эквивалентности, а на втором - формируют тесты.

Выделение классов эквивалентности является эвристическим процессом, однако целесообразным считают выделять в отдельные классы эквивалентности наборы, содержащие допустимые и недопустимые значения некоторого параметра. При этом существует ряд правил:

- если некоторый параметр x может принимать значения в интервале $[1, 999]$, то выделяют один правильный класс $1 \leq x \leq 999$ и два неправильных: $x < 1$ и $x > 999$;
- если входное условие определяет диапазон значений порядкового типа, например, «в автомобиле могут ехать от одного до шести человек», то определяется один правильный класс эквивалентности и два неправильных: ни одного и более шести человек;
- если входное условие описывает множество входных значений и есть основания полагать, что каждое значение программист трактует особо, например, «типы графических файлов: bmp, jpeg, vsd», то определяют правильный класс эквивалентности для каждого значения и один неправильный класс, например, txt;
- если входное условие описывает ситуацию «должно быть», например, «первым символом идентификатора должна быть буква», то определяется один правильный класс эквивалентности (первый символ - буква) и один неправильный (первый символ - не буква);
- если есть основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс разбивается на меньшие классы эквивалентности.

Таким образом, классы эквивалентности выделяют, перебирая ограничения, установленные для каждого входного значения в техническом задании или при уточнении спецификации. Каждое ограничение разбивают на две или более групп. При этом используют специальные бланки - таблицы классов эквивалентности:

Ограничение на значение параметра	Правильные классы эквивалентности	Неправильные классы эквивалентности

Правильные классы включают правильные данные, неправильные классы - неправильные данные. Для правильных и неправильных классов тесты проектируют отдельно. При построении тестов правильных классов учитывают, что каждый тест должен проверять по возможности максимальное количество различных входных условий. Такой подход позволяет минимизировать общее число необходимых тестов. Для каждого неправильного класса эквивалентности формируют свой тест. Последнее обусловлено тем, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

Анализ граничных значений. *Граничные значения* - это значения на границах классов эквивалентности входных значений или около них. Анализ показывает, что в этих местах резко увеличивается возможность обнаружения ошибок. Например, если в программе анализа вида

треугольника было записано $A + B \geq C$ вместо $A + B > C$, то задание граничных значений приведет к ошибке: линия будет отнесена к одному из видов треугольника.

Применение метода анализа граничных значений требует определенной степени творчества и специализации в рассматриваемой проблеме. Тем не менее существует несколько общих правил для применения этого метода:

- если входное условие описывает область значений, то следует построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, например, если описана область $[-1.0, +1.0]$, то должны быть сгенерированы тесты: $-1.0, +1.0, -1.001$ и $+1.001$;

- если входное условие удовлетворяет дискретному ряду значений, то следует построить тесты для минимального и максимального значений и тесты, содержащие значения большие и меньшие этих двух значений, например, если входной файл может содержать от 1 до 255 записей, то следует проверить 0, 1, 255 и 256 записей;

- если существуют ограничения выходных значений, то целесообразно аналогично тестировать и их: конечно не всегда можно получить результат вне выходной области, но тем не менее стоит рассмотреть эту возможность;

- если некоторое входное или выходное значение программы является упорядоченным множеством, например, это последовательный файл, линейный список или таблица, то следует сосредоточить внимание на первом и последнем элементах этого множества.

Помимо указанных граничных значений, целесообразно поискать другие.

Анализ граничных значений, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако следует помнить, что граничные значения могут быть едва уловимы и определение их связано с большими трудностями, что является недостатком этого метода.

Оба описанных метода основаны на исследовании входных данных. Они не позволяют проверять результаты, получаемые при различных сочетаниях данных. Для построения тестов, проверяющих сочетания данных, применяют методы, использующие булеву алгебру.

Анализ причинно-следственных связей. Анализ причинно-следственных связей позволяет системно выбирать высокорезультативные тесты. Метод использует алгебру логики и оперирует понятиями «причина» и «следствие». *Причиной* в данном случае называют отдельное входное условие или класс эквивалентности. *Следствием* - выходное условие или преобразование системы. Идея метода заключается в отнесении всех следствий к причинам, т. е. в уточнении причинно-следственных связей. Данный метод дает полезный побочный эффект, позволяя обнаруживать неполноту и неоднозначность исходных спецификаций.

Построение тестов осуществляют в несколько этапов. Сначала, поскольку таблицы причинно-следственных связей при применении метода к большим спецификациям становятся громоздкими, спецификации разбивают на «рабочие» участки, стараясь по возможности выделять в отдельные таблицы независимые группы причинно-следственных связей. Затем в спецификации определяют множество причин и следствий.

Далее на основе анализа семантического (смыслового) содержания спецификации строят таблицу истинности, в которой каждой *возможной* комбинации причин ставится в соответствие следствие. При этом целесообразно истину обозначать «I», ложь - «O», а для обозначения безразличных состояний условий применять обозначение «X», которое предполагает произвольное значение условия (0 или 1). Таблицу сопровождают примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. При необходимости аналогично строится таблица истинности для класса эквивалентности.

И, наконец, каждую строку таблицы преобразуют в тест. При этом рекомендуется по возможности совмещать тесты из независимых таблиц.

Данный метод позволяет строить высокорезультативные тесты и обнаруживать неполноту и неоднозначность исходных спецификаций. Его недостатком является неадекватное исследование граничных значений.

Предположение об ошибке. Часто программист с большим опытом находит ошибки, «не применяя никаких методов». На самом деле он подсознательно использует метод «предположение об ошибке».

Процедура метода предположения об ошибке в значительной степени основана на интуиции. Основная его идея заключается в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка составить тесты. Другими словами, требуется перечислить те особые случаи, которые могут быть не учтены при проектировании.

Проиллюстрируем применение всех рассмотренных выше методов на примере.

Пример 9.1. Пусть необходимо выполнить тестирование программы, определяющей точку пересечения двух прямых на плоскости. При этом она должна определять параллельность прямой одной из осей координат.

В основе программы лежит решение системы линейных уравнений:

$$\begin{cases} Ax + By = C, \\ Dx + Ey = F. \end{cases}$$

По методу эквивалентных разбиений формируем для каждого коэффициента один правильный класс эквивалентности (коэффициент-вещественное число) и один неправильный (коэффициент- не вещественное число). Откуда генерируем 7 тестов:

- 1) все коэффициенты - вещественные числа (1 тест);
- 2-7) поочередно каждый из коэффициентов - не вещественное число (6 тестов).

По методу граничных значений можно считать, что для исходных данных граничные значения отсутствуют, т. е. коэффициенты - «любые» вещественные числа. Для результатов получаем, что возможны варианты: единственное решение, прямые сливаются - множество решений, прямые параллельны - отсутствие решений. Следовательно, целесообразно предложить тесты с результатами внутри областей возможных значений результатов:

- 8) результат - единственное решение ($\delta \neq 0$);
- 9) результат - множество решений ($\delta = 0$ и $\delta_x = \delta_y = 0$);
- 10) результат - отсутствие решений ($\delta = 0$, но $\delta_x \neq 0$ или $\delta_y \neq 0$);

и с результатами на границе:

- 11) $\delta = 0,01$;
- 12) $\delta = -0,01$;
- 13) $\delta = 0, \delta_x = 0,01, \delta_y = 0$;
- 14) $\delta = 0, \delta_y = -0,01, \delta_x = 0$.

По методу анализа причинно – следственных связей определяем множество условий:

а) для определения типа прямой:

$$\left. \begin{matrix} a = 0 \\ b = 0 \\ c = 0 \end{matrix} \right\} \text{ - для определения типа и существования первой прямой;}$$

$$\left. \begin{matrix} d = 0 \\ e = 0 \\ f = 0 \end{matrix} \right\} \text{ - для определения типа и существования второй прямой;}$$

б) для определения точки пересечения:

$$\delta = 0,$$

$$\delta_x = 0,$$

$$\delta_y = 0.$$

Выделяем три группы причинно-следственных связей (определение типа и существования первой линии, определение типа и существования второй линии, определение точки пересечения) и строим таблицы истинности для определения типа первой прямой (табл. 9.1) и для определения результата (табл. 9.2). В обеих таблицах X означает неопределенное значение. Для второй прямой таблица истинности будет выглядеть аналогично табл. 9.1.

Каждая строка этих таблиц преобразуется в тест. При возможности (с учетом независимости групп) берутся данные, соответствующие строкам сразу двух или всех трех таблиц.

Таблица 9.1

A = 0	B = 0	C = 0	Результат
0	0	X	прямая общего положения
0	1	0	прямая, параллельная оси OX
0	1	1	ось OX
1	0	0	прямая, параллельная оси OY
1	0	1	ось OY
1	1	X	множество точек плоскости

Таблица 9.2

$\delta = 0$	$\delta_x = 0$	$\delta_y = 0$	Единственное решение	Множество решений	Решения нет
0	X	X	1	0	0
1	0	X	0	0	1
1	X	0	0	0	1
1	1	1	0	1	0

В результате к уже имеющимся тестам добавляются:

15-21) проверки всех случаев расположения обеих прямых - 6 тестов по первой прямой совмещают с 6-ю тестами по второй прямой так, чтобы варианты не совпадали (6 тестов);

22) проверка несовпадения условия $\delta_x = 0$ или $\delta_y = 0$ (в зависимости от того, какой тест был выбран по методу граничных условий) — тест также можно совместить с предыдущими 6-ю тестами.

По методу предположения об ошибке добавим тест:

23) все коэффициенты - нули.

Всего получили 23 теста по всем четырем методам. Для каждого теста перед применением необходимо указать ожидаемый результат. Если попробовать вложить независимые проверки, то, возможно, число тестов можно еще сократить.

9.5. Тестирования модулей и комплексное тестирование

Как уже упоминалось в § 2.3 при тестировании модулей программного обеспечения, так же, как при проектировании и кодировании возможно применение как восходящего, так и нисходящего подходов.

Восходящее тестирование. Восходящий подход предполагает, что каждый модуль тестируют отдельно на соответствие имеющимся спецификациям на него, затем собирают оттестированные модули в модули более высокой степени интеграции и тестируют их. При этом проверяют межмодульные интерфейсы, используемые для подключения модулей более низкого уровня иерархии. И так далее, пока не будет собран весь программный продукт (рис. 9.3).

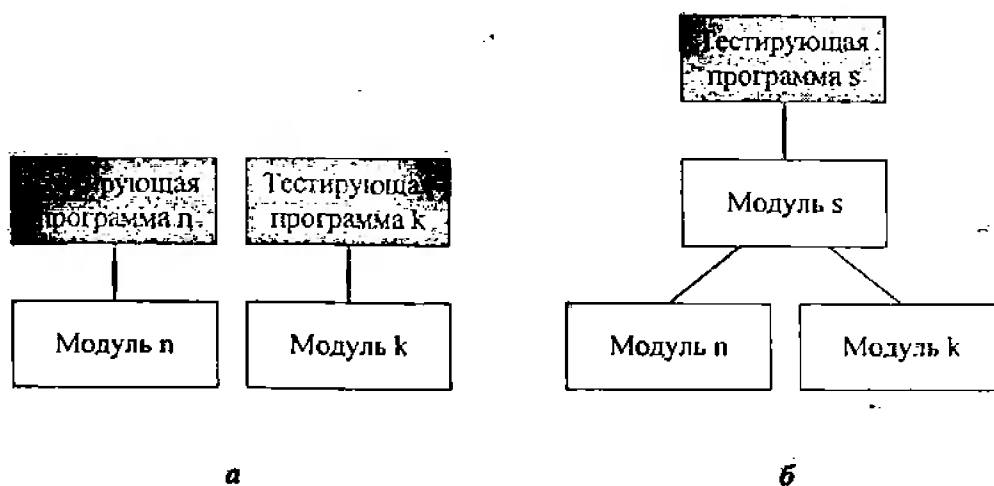


Рис. 9.3. Тестирование программного обеспечения при восходящем подходе:

а – автономное тестирование модулей нижнего уровня; *б* – тестирование следующего уровня

Такой подход обеспечивает полностью автономное тестирование, для которого просто генерировать тестовые последовательности, которые передаются в модуль напрямую. Однако он имеет и существенные недостатки. Во-первых, при восходящем тестировании так же, как при восходящем проектировании, серьезные ошибки в спецификациях, алгоритмах и интерфейсе могут быть обнаружены только на завершающей стадии работы над проектом. Во-вторых, для того, чтобы тестировать модули нижних уровней, необходимо разработать специальные тестирующие программы, которые обеспечивают вызов интересующих нас модулей с необходимыми параметрами. Причем эти тестирующие программы также могут содержать ошибки.

Нисходящее тестирование. Нисходящее тестирование органически связано с нисходящим проектированием и разработкой; как только проектирование какого-либо модуля заканчивается, его кодируют и передают на тестирование.

В этом случае автономно тестируется только основной модуль. При его тестировании все вызываемые им модули заменяют модулями, которые в той или иной степени имитируют поведение вызываемых модулей (рис. 9.4). Такие модули принято называть «заглушками». В отличие от тестирующих программ заглушки очень просты, например, они могут просто фиксировать, что им передано управление. Часто заглушки просто возвращают какие-либо фиксированные данные.

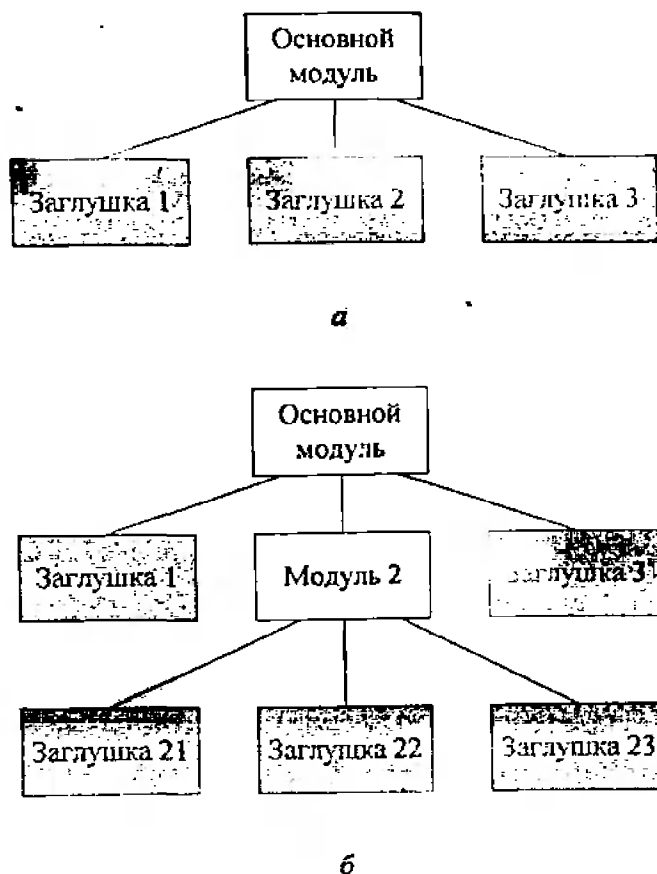


Рис. 9.4. Начальные этапы тестирования:

а – основного модуля; б – двух модулей

Как только тестирование основного модуля завершено, к нему подключают модули, непосредственно им вызываемые, и необходимые заглушки, а затем проводят их совместное тестирование. Далее последовательно подключают следующие модули, пока не будет собрана вся система. Желательная последовательность сборки обсуждалась в § 2.3, хотя на практике ее редко удается соблюдать.

Основной *недостаток* нисходящего тестирования - отсутствие автономного тестирования модулей. Поскольку модуль получает данные не непосредственно, а через вызывающий модуль, то гораздо сложнее обеспечить его «достаточное» тестирование.

Основным *достоинством* данного метода является ранняя проверка основных решений и качественное многократное тестирование сопряжения модулей в контексте программного обеспечения. При нисходящем тестировании есть возможность согласования с заказчиком внешнего вида (интерфейса) программного обеспечения.

Комбинированный подход. Чаще всего применяют комбинированный подход: модули верхних уровней тестируют нисходящим способом, а модули нижних уровней - восходящим. Этот способ позволяет с одной стороны начать с тестирования интерфейса, с другой - обеспечивает качественное автономное тестирование модулей низших уровней.

Тестирование программного обеспечения специалистами. Согласно основным принципам нежелательно тестирование программного обеспечения его автором, поэтому эту работу, как правило, выполняют другие программисты, желательно - специалисты в этой области.

Задачей специалиста по тестированию является обнаружение максимального количества несоответствий тестируемого модуля и спецификаций на него. Для выполнения этой задачи специалист по тестированию формирует тесты, используя как структурный, так и функциональный подходы, обеспечивая всестороннее тестирование.

Каждое отклонение от спецификации обязательно документируют, заполняя специальный протокол (рис. 9.5). Наиболее интересными полями протокола являются тип проблемы и ее описание.

Название компании _____		Конфиденциально	Отчет о проблеме
№ _____			
Программа _____	Выпуск _____	Версия _____	
Тип проблемы (1-6) _____	Степень важности (1-3) _____	Приложения (Д/Н) _____	
1 - Ошибка кодирования	1 - Фатальная	Если да, то какие	
2 - Ошибка проектирования	2 - Серьезная		
3 - Предложение	3 - Незначительная		
4 - Расхождение с документацией			
5 - Взаимодействие с аппаратурой			
6 - Вопрос			
ПРОБЛЕМА			
Можете ли вы воспроизвести проблемную ситуацию? (Д/Н) _____			
Подробное описание проблемы и способа ее воспроизведения: _____			

Предлагаемое исправление (необязательно): _____			

Отчет представлен сотрудником _____		Дата ____ / ____ / ____	
Заполняется разработчиком			
Функциональная область _____		Ответственный _____	
Комментарии _____			
Состояние (1-2) _____		Приоритет (1-5) _____	
1 - Открыто 2 - Закрыто			
Резолюция (1-9) _____		Иправленная версия _____	
1 - Рассматривается	6 - Не может быть исправлено		
2 - Исправлено	7 - Отозвано составителем		
3 - Не воспроизводится	8 - Нужна дополнительная информация		
4 - Отложено	9 - Не согласен с предложением		
5 - Соответствует проекту			
Рассмотрено _____	Дата ____ / ____ / ____		
Проконтролировано _____	Дата ____ / ____ / ____		
Считать отложенным (Д/Н) _____			

Рис. 9.5. Бланк отчета об обнаруженном несоответствии

В поле тип проблемы указывают один из вариантов:

1 - ошибка кодирования - программа ведет себя не так, как следует из общепринятых представлений, например, $2 + 2 = 5$ - на что разработчик может выдать резолюцию «соответствует проекту»;

2 - ошибка проектирования — программа ведет себя в соответствии с проектом, но специалист по тестированию не согласен с данным решением в проекте - на что разработчик может отреагировать, наложив резолюцию «не согласен с предложением»; . 3 - предложение - предложение по улучшению проекта;

4 - расхождение с документацией - обнаружено, что программа ведет себя не так, как указано в документации;

5 - взаимодействие с аппаратурой - обнаружены проблемы при использовании определенного вида аппаратуры;

6 - вопрос - программа делает что-то не совсем понятное.

Описание проблемы должно быть коротким и понятным, например:

«Система не запоминает настройки принтера, выполняемые в окне настройки».

Если программист исправляет ошибку, то тестирование повторяют сначала, так как при исправлении ошибки программист может внести в программу новые ошибки. Такое тестирование называют «регрессионным».

Комплексное тестирование. Особенностью комплексного тестирования является то, что структурное тестирование для него практически не применимо. В основном на данной стадии используют тесты, построенные по методам эквивалентных классов, граничных условий и предположении об ошибках.

Критерии завершения тестирования и отладки. Одним из самых сложных является вопрос о том, когда следует завершать тестирование, поскольку невозможно гарантировать, что в разрабатываемом программном обеспечении не осталось ошибок.

Предложено очень много критериев. Все критерии можно разделить на три группы:

- основанные на методологиях проектирования тестов — определенное количество тестов, полученных по методам анализа причинно-следственных связей, анализа граничных значений и предположения об ошибке, перестают выявлять ошибки;

- основанные на оценке возможного количества ошибок - возможное количество ошибок оценивают экспертно, или по специальным методикам [21], а затем завершают тестирование при нахождении примерно 93-95% ошибок;

- основанные на исследовании результатов тестирования - строят график зависимости количества обнаруженных ошибок от времени тестирования, если он напоминает график, представленный на рис. 9.6, то тестирование можно завершать.

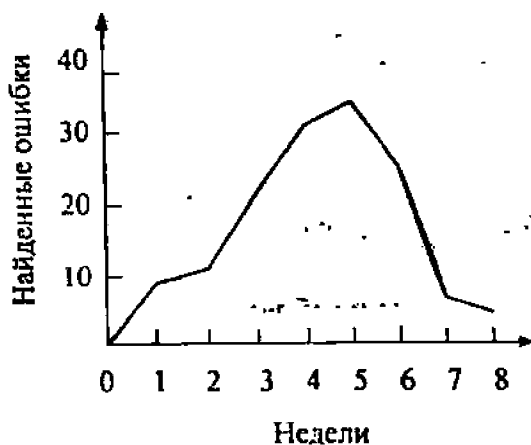


Рис. 9.6. Пример графика обнаружения ошибок

Часто тестирование завершают потому, что закончилось время, отведенное на выполнение данного этапа. В этом случае тестирование сворачивают, обходясь минимальным вариантом. Минимальное тестирование [31] предполагает:

- тестирование граничных значений;
- тщательную проверку руководства;
- тестирование минимальных конфигураций технических средств;
- тестирование возможности редактирования команд и повторения их в любой последовательности;
- тестирование устойчивости к ошибкам пользователя.

Часть ошибок при этом остаются неисправленными «отложенными» до выпуска следующей версии.

9.6. Оценочное тестирование

После завершения комплексного тестирования приступают к оценочному тестированию, целью которого является тестирование программы на соответствие основным требованиям. Эта стадия тестирования особенно важна для программных продуктов, предназначенных для продажи на рынке.

Оценочное тестирование, которое также называют «тестированием системы в целом», включает следующие виды:

- тестирование удобства использования - последовательная проверка соответствия программного продукта и документации на него основным положениям технического задания;
- тестирование на предельных объемах - проверка работоспособности программы на максимально больших объемах данных, например, объемах текстов, таблиц, большом количестве файлов и т. п.;
- тестирование на предельных нагрузках - проверка выполнения программы на возможность обработки большого объема данных, поступивших в течение короткого времени;
- тестирование удобства эксплуатации - анализ психологических факторов, возникающих при работе с программным обеспечением; это тестирование позволяет определить, удобен ли интерфейс, не раздражает ли цветовое или звуковое сопровождение и т. п.;
- тестирование защиты - проверка защиты, например, от несанкционированного доступа к информации;
- тестирование производительности - определение пропускной способности при заданной конфигурации и нагрузке;
- тестирование требований к памяти - определение реальных потребностей в оперативной и внешней памяти;
- тестирование конфигурации оборудования - проверка работоспособности программного обеспечения на разном оборудовании;
- тестирование совместимости - проверка преемственности версий: в тех случаях, если очередная версия системы меняет форматы данных, она должна предусматривать специальные конвекторы, обеспечивающие возможность работы с файлами, созданными предыдущей версией системы;
- тестирование удобства установки - проверка удобства установки;
- тестирование надежности - проверка надежности с использованием соответствующих математических моделей [66];
- тестирование восстановления - проверка восстановления программного обеспечения, например системы, включающей базу данных, после сбоев оборудования и программы;
- тестирование удобства обслуживания - проверка средств обслуживания, включенных в программное обеспечение;
- тестирование документации - тщательная проверка документации, например, если документация содержит примеры, то их все необходимо попробовать;
- тестирование процедуры - проверка ручных процессов, предполагаемых в системе.

Естественно, целью всех этих проверок является поиск несоответствий техническому заданию. Считают, что только после выполнения всех видов тестирования программный продукт может быть представлен пользователю или к реализации. Однако на практике обычно выполняют не все виды оценочного тестирования, так как это очень дорого и трудоемко. Как правило, для каждого типа программного обеспечения выполняют те виды тестирования, которые являются для него наиболее важными. Так базы данных обязательно тестируют на предельных объемах, а системы реального времени - на предельных нагрузках.

Контрольные вопросы и задания

1. Что является целью тестирования программ? Почему?
2. Перечислите известные вам виды контроля качества программного обеспечения. На каких этапах применяют каждый из них?
3. Какие подходы к тестированию вы знаете? В чем они заключаются?
4. Почему функциональное тестирование называют «тестированием по методу черного ящика»? Перечислите методы функционального тестирования и определите, в каких случаях следует использовать каждый из них.
5. Почему структурное тестирование называют «тестированием по методу белого или прозрачного ящика»? Перечислите методы структурного тестирования и определите возможности каждого из них. Какой метод структурного тестирования обеспечивает наибольшую вероятность обнаружения ошибок?
6. Используя методы обоих подходов, сформируйте пакет тестов для тестирования программы, вычисляющей действительные корни квадратного уравнения. Какие методы вы использовали и почему?
7. Чем нисходящее тестирование отличается от восходящего? Что понимают под комплексным тестированием и чем оно отличается от тестирования компонент? Когда можно прекращать тестирование компонентов?
8. Перечислите виды тестирования системы в целом. В каких случаях применяют каждый из них?

10. ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Отладка программы — один из самых сложных этапов разработки программного обеспечения, требующий глубокого знания:

- специфики управления используемыми техническими средствами,
- операционной системы,
- среды и языка программирования,
- реализуемых процессов,
- природы и специфики различных ошибок,
- методик отладки и соответствующих программных средств.

Обсуждению последних двух вопросов и посвящается данная глава.

10.1. Классификация ошибок

Отладка — это процесс локализации и исправления ошибок, обнаруженных при тестировании программного обеспечения. *Локализацией* называют процесс определения оператора программы, выполнение которого вызвало нарушение нормального вычислительного процесса. Для исправления ошибки необходимо определить ее *причину*, т. е. определить оператор или фрагмент, содержащие ошибку. Причины ошибок могут быть как очевидны, так и очень глубоко скрыты.

В целом сложность отладки обусловлена следующими причинами:

- требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики различных ошибок, методик отладки и соответствующих программных средств;
- психологически дискомфортна, так как необходимо искать собственные ошибки и, как правило, в условиях ограниченного времени;
- возможно взаимовлияние ошибок в разных частях программы, например, за счет затирания области памяти одного модуля другим из-за ошибок адресации;
- отсутствуют четко сформулированные методики отладки.

В соответствии с этапом обработки, на котором проявляются ошибки, различают (рис. 10.1): *синтаксические ошибки* — ошибки, фиксируемые компилятором (транслятором, интерпретатором) при выполнении синтаксического и частично семантического анализа программы; *ошибки компоновки* — ошибки, обнаруженные компоновщиком (редактором связей) при объединении модулей программы; *ошибки выполнения* — ошибки, обнаруженные операционной системой, аппаратными средствами или пользователем при выполнении программы.



Рис. 10.1. Классификация ошибок по этапу обработки программы

Синтаксические ошибки. Синтаксические ошибки относят к группе самых простых, так как синтаксис языка, как правило, строго формализован, и ошибки сопровождаются развернутым комментарием с указанием ее местоположения. Определение причин таких ошибок, как правило, труда не составляет, и даже при нечетком знании правил языка за несколько прогонов удастся удалить все ошибки данного типа.

Следует иметь в виду, что чем лучше формализованы правила синтаксиса языка, тем больше ошибок из общего количества может обнаружить компилятор и, соответственно, меньше ошибок будет обнаруживаться на следующих этапах. В связи с этим говорят о языках программирования с защищенным синтаксисом и с незащищенным синтаксисом. К первым, безусловно, можно отнести Pascal, имеющий очень простой и четко определенный синтаксис, хорошо проверяемый при компиляции программы, ко вторым - Си со всеми его модификациями. Чего стоит хотя бы возможность выполнения присваивания в условном операторе в Си, например:

`if (c = n) x = 0; /*` в данном случае не проверятся равенство `c` и `n`, а выполняется присваивание `c` значения `n`, после чего результат операции сравнивается с нулем, если программист хотел выполнить не присваивание, а сравнение, то эта ошибка будет обнаружена только на этапе выполнения при получении результатов, отличающихся от ожидаемых `*/`

Ошибки компоновки. Ошибки компоновки, как следует из названия, связаны с проблемами, обнаруженными при разрешении внешних ссылок. Например, предусмотрено обращение к подпрограмме другого модуля, а при объединении модулей данная подпрограмма не найдена или не стыкуются списки параметров. В большинстве случаев ошибки такого рода также удастся быстро локализовать и устранить.

Ошибки выполнения. К самой непредсказуемой группе относятся ошибки выполнения. Прежде всего они могут иметь разную природу, и соответственно по-разному проявляться. Часть ошибок обнаруживается и документируется операционной системой. Выделяют четыре способа проявления таких ошибок:

- появление сообщения об ошибке, зафиксированной схемами контроля выполнения машинных команд, например, переполнении разрядной сетки, ситуации «деление на ноль», нарушении адресации и т. п.;
- появление сообщения об ошибке, обнаруженной операционной системой, например, нарушении защиты памяти, попытке записи на устройства, защищенные от записи, отсутствии файла с заданным именем и т. п.;
- «зависание» компьютера, как простое, когда удастся завершить программу без перезагрузки операционной системы, так и «тяжелое», когда для продолжения работы необходима перезагрузка;
- несовпадение полученных результатов с ожидаемыми.

Примечание. Отметим, что, если ошибки этапа выполнения обнаруживает пользователь, то в двух первых случаях, получив соответствующее сообщение, пользователь в зависимости от своего характера, степени необходимости и опыта работы за компьютером, либо попытается понять, что произошло, ища свою вину, либо обратится за помощью, либо постарается никогда больше не иметь дела с этим продуктом. При «зависании» компьютера пользователь может даже не сразу понять, что происходит что-то не то, хотя его печальный опыт и заставляет волноваться каждый раз, когда компьютер не выдает быстрой реакции на введенную команду, что также целесообразно иметь в виду. Также опасны могут быть ситуации, при которых пользователь получает неправильные результаты и использует их в своей работе.

Причины ошибок выполнения очень разнообразны, а потому и локализация может оказаться крайне сложной. Все возможные причины ошибок можно разделить на следующие группы:

- неверное определение исходных данных,
- логические ошибки,
- накопление погрешностей результатов вычислений (рис. 10.2).



Рис. 10.2. Классификация ошибок этапа выполнения по возможным причинам

Неверное определение исходных данных происходит, если возникают любые ошибки при выполнении операций ввода-вывода: ошибки передачи, ошибки преобразования, ошибки перезаписи и ошибки данных. Причем использование специальных технических средств и программирование с защитой от ошибок (см. § 2.7) позволяет обнаружить и предотвратить только часть этих ошибок, о чем безусловно не следует забывать.

Логические ошибки имеют разную природу. Так они могут следовать из ошибок, допущенных при проектировании, например, при выборе методов, разработке алгоритмов или определении структуры классов, а могут быть непосредственно внесены при кодировании модуля. К последней группе относят:

- *ошибки некорректного использования переменных*, например, неудачный выбор типов данных, использование переменных до их инициализации, использование индексов, выходящих за границы определения массивов, нарушения соответствия типов данных при использовании явного или неявного переопределения типа данных, расположенных в памяти при использовании нетипизированных переменных, открытых массивов, объединений, динамической памяти, адресной арифметики и т. п.;

- *ошибки вычислений*, например, некорректные вычисления над неарифметическими переменными, некорректное использование целочисленной арифметики, некорректное преобразование типов данных в процессе вычислений, ошибки, связанные с незнанием приоритетов выполнения операций для арифметических и логических выражений, и т. п.;

- *ошибки межмодульного интерфейса*, например, игнорирование системных соглашений, нарушение типов и последовательности при передачи параметров, несоблюдение единства единиц

измерения формальных и фактических параметров, нарушение области действия локальных и глобальных переменных;

- другие *ошибки кодирования*, например, неправильная реализация логики программы при кодировании, игнорирование особенностей или ограничений конкретного языка программирования.

Накопление погрешностей результатов числовых вычислений возникает, например, при некорректном отбрасывании дробных цифр чисел, некорректном использовании приближенных методов вычислений, игнорировании ограничения разрядной сетки представления вещественных чисел в ЭВМ и т. п.

Все указанные выше причины возникновения ошибок следует иметь в виду в процессе отладки. Кроме того, сложность отладки увеличивается также вследствие влияния следующих факторов:

- опосредованного проявления ошибок;
- возможности взаимовлияния ошибок;
- возможности получения внешне одинаковых проявлений разных ошибок;
- отсутствия повторяемости проявлений некоторых ошибок от запуска к запуску – так называемые стохастические ошибки;
- возможности устранения внешних проявлений ошибок в исследуемой ситуации при внесении некоторых изменений в программу, например, при включении в программу диагностических фрагментов может аннулироваться или измениться внешнее проявление ошибок;
- написания отдельных частей программы разными программистами.

10.2. Методы отладки программного обеспечения

Отладка программы в любом случае предполагает обдумывание и логическое осмысление всей имеющейся информации об ошибке. Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации. При этом используют различные методы:

- ручного тестирования;
- индукции;
- дедукции;
- обратного прослеживания.

Метод ручного тестирования. Это - самый простой и естественный способ данной группы. При обнаружении ошибки необходимо выполнить тестируемую программу вручную, используя тестовый набор, при работе с которым была обнаружена ошибка.

Метод очень эффективен, но не применим для больших программ, программ со сложными вычислениями и в тех случаях, когда ошибка связана с неверным представлением программиста о выполнении некоторых операций.

Данный метод часто используют как составную часть других методов отладки.

Метод индукции. Метод основан на тщательном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке. Если компьютер просто «зависает», то фрагмент проявления ошибки вычисляют, исходя из последних полученных результатов и действий пользователя. Полученную таким образом информацию организуют и тщательно изучают, просматривая соответствующий фрагмент программы. В результате этих действий выдвигают гипотезы об ошибках, каждую из которых проверяют. Если гипотеза верна, то детализируют информацию об ошибке, иначе - выдвигают другую гипотезу. Последовательность выполнения отладки методом индукции показана на рис. 10.3 в виде схемы алгоритма.

Самый ответственный этап - выявление симптомов ошибки. Организуя данные об ошибке, целесообразно записать все, что известно о ее проявлениях, причем фиксируют, как ситуации, в которых фрагмент с ошибкой выполняется нормально, так и ситуации, в которых ошибка

проявляется. Если в результате изучения данных никаких гипотез не появляется, то необходима дополнительная информация об ошибке. Дополнительную информацию можно получить, например, в результате выполнения схожих тестов.

В процессе доказательства пытаются выяснить, все ли проявления ошибки объясняет данная гипотеза, если не все, то либо гипотеза не верна, либо ошибок несколько.

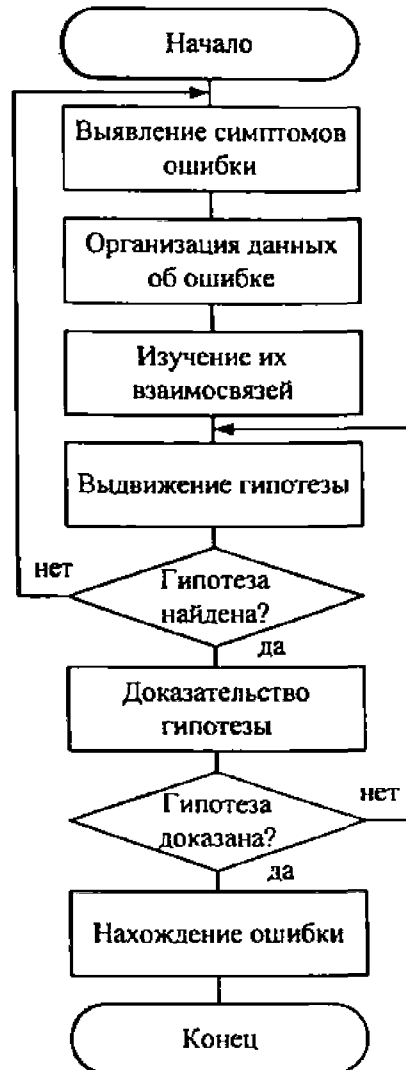


Рис. 10.3. Схема процесса отладки методом индукции

Метод дедукции. По методу дедукции вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки. Затем анализируя причины, исключают те, которые противоречат имеющимся данным. Если все причины исключены, то следует выполнить дополнительное тестирование исследуемого фрагмента. В противном случае наиболее вероятную гипотезу пытаются доказать. Если гипотеза объясняет полученные признаки ошибки, то ошибка найдена, иначе - проверяют следующую причину (рис. 10.4).

Метод обратного прослеживания. Для небольших программ эффективно применение метода обратного прослеживания. Начинают с точки вывода неправильного результата. Для этой точки строится гипотеза о значениях основных переменных, которые могли бы привести к получению имеющегося результата. Далее, исходя из этой гипотезы, делают предложения о значениях переменных в предыдущей точке. Процесс продолжают, пока не обнаружат причину ошибки.

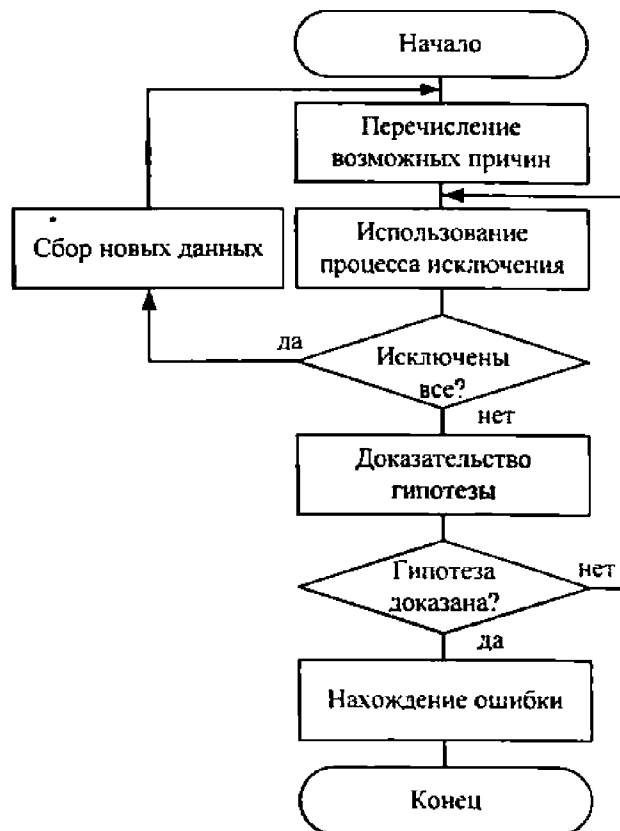


Рис. 10.4. Схема процесса отладки методом дедукции

10.3. Методы и средства получения дополнительной информации

Для получения дополнительной информации об ошибке можно выполнить добавочные тесты или использовать специальные методы и средства:

- отладочный вывод;
- интегрированные средства отладки;
- независимые отладчики.

Отладочный вывод. Метод требует включения в программу дополнительного отладочного вывода в *узловых* точках. Узловыми считают точки алгоритма, в которых основные переменные программы меняют свои значения. Например, отладочный вывод следует предусмотреть до и после завершения цикла изменения некоторого массива значений. (Если отладочный вывод предусмотреть в цикле, то будет выведено слишком много значений, в которых, как правило, сложно разбираться.) При этом предполагается, что, выполнив анализ выведенных значений, программист уточнит момент, когда были получены неправильные значения, и сможет сделать вывод о причине ошибки.

Данный метод не очень эффективен и в настоящее время практически не используется, так как в сложных случаях в процессе отладки может потребоваться вывод большого количества - «трассы» значений многих переменных, которые выводятся при каждом изменении. Кроме того, внесение в программы дополнительных операторов может привести к изменению проявления ошибки, что нежелательно, хотя и позволяет сделать определенный вывод о ее природе.

Примечание. Ошибки, исчезающие при включении в программу или удалению из нее каких-либо «безобидных» операторов, как правило, связаны с «затиранием» памяти. В результате

добавления или удаления операторов область затирания может сместиться в другое место и ошибка либо перестанет проявляться, либо будет проявляться по-другому.

Интегрированные средства отладки. Большинство современных сред программирования (Delphi, Builder C++, Visual Studio и т. д.) включают средства отладки, которые обеспечивают максимально эффективную отладку. Они позволяют:

- выполнять программу по шагам, причем как с заходом в подпрограммы, так и выполняя их целиком;
- предусматривать точки останова;
- выполнять программу до оператора, указанного курсором;
- отображать содержимое любых переменных при пошаговом выполнении;
- отслеживать поток сообщений и т. п.

На рис. 10.5 показан вид программы в момент перехода в режим пошагового выполнения по достижении точки останова в Delphi. В этот момент программист имеет возможность посмотреть значения интересующих его переменных.

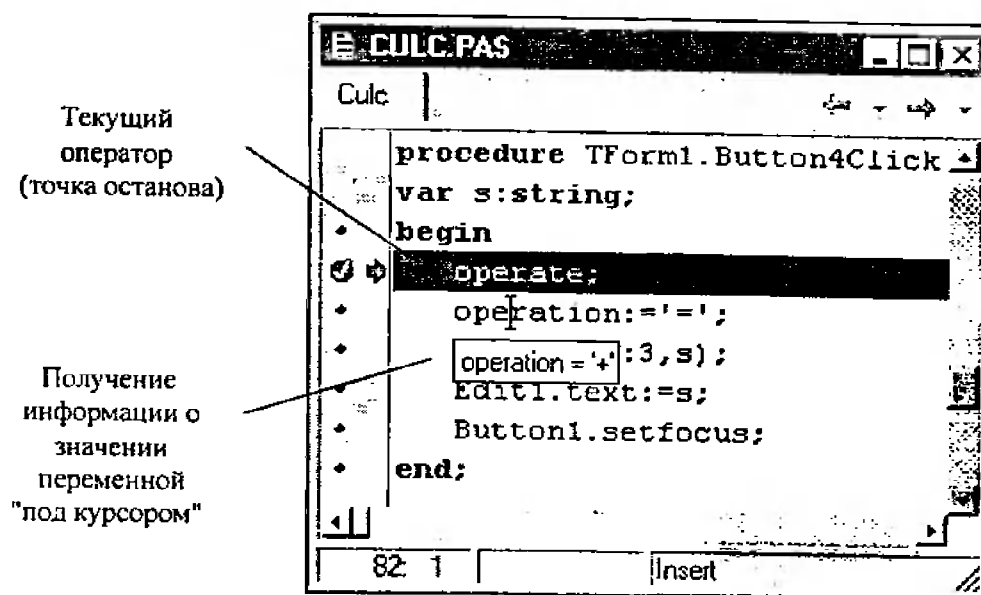


Рис. 10.5. Переход в режим пошагового выполнения по достижении точки останова в Delphi (в рамке указано значение переменной «под курсором»)

Применять интегрированные средства в рамках среды достаточно просто. Используют разные приемы в зависимости от проявлений ошибки. Если получено сообщение об ошибке, то сначала уточняют, при выполнении какого оператора программы оно получено. Для этого устанавливают точку останова в начало фрагмента, в котором проявляется ошибка, и выполняют операторы в пошаговом режиме до проявления ошибки.

Аналогично поступают при «зависании» компьютера.

Если получены неправильные результаты, то локализовать ошибку обычно существенно сложнее. В этом случае сначала определяют фрагмент, при выполнении которого получаются неправильные результаты. Для этого последовательно проверяют интересующие значения в узловых точках. Обнаружив значения, отличающиеся от ожидаемых, по шагам трассируют соответствующий фрагмент до выявления оператора, выполнение которого дает неверный результат.

Для уточнения природы ошибки возможен анализ машинных кодов, флагов и представления программы и значений памяти в 16-ричном виде (рис. 10.6).

Причину ошибки определяют, используя один из методов, рассмотренных в § 10.2. При этом для проверки гипотез также можно использовать интегрированные средства отладки.

Относительный адрес	Оператор Delphi Pascal	Команда ассемблера	Содержимое регистров	Значения флагов
CPU				
\$004408D8		Thread # \$FFE23E09	EAX 00000000	CR0 0
	CULC.PAS.82: operate;		EBX 00BB16	PF 1
00440A5F	call operate		ECX 00000000	AF 0
	CULC.PAS.83: operation:= '=';		EDX 00BB3D	ZF 1
00440A64	mov byte ptr [operation]		ESI 0067F5	SF 0
	CULC.PAS.84: str(sum:6:3,s);		EDI 0067F5	TF 0
00440A6B	fld qword ptr [Sum]		EBP 0067F3	IF 1
00440A71	add esp, -\$0c		ESP 0067F2	DF 0
00440A74	fstp tbyte ptr [esp]		EIP 00440A	OF 0
00440A77	wait		EFL 000002	IO 0
00440A78	lea ecx, [ebp-\$00000104]		CS 0167	NF 0
00440A7E	mov edx, \$00000003		DS 016F	RF 0
00440A83	mov eax, \$00000006		SS 016F	VM 0
00440A88	call @Str2Ext			AC 0
00410000	7B F8 00 74 04 33 C0 5D	хш.т.3А	0067F2C4	0067F3B0
00410008	C3 B0 01 5D C3 8D 40 00	Г°.JГKQ	0067F2C0	00000000
00410010	55 8B EC 53 56 8B F0 33	U<MSY< p	0067F2BC	00C0C0C0
00410018	DB E8 AE 8D FF FF 8B 15	МиджЯя<	0067F2B8	00000000
			0067F2B4	6BCC8260

Содержимое памяти
в 16-ричном коде

Содержимое памяти
в символьном виде

Содержимое стека
в 16-ричном коде

Рис. 10.6. Вид экрана при отладке программы в 16-ричном коде (режим CPU)

Отладка с использованием независимых отладчиков. При отладке программ иногда используют специальные программы - отладчики, которые позволяют выполнить любой фрагмент программы в пошаговом режиме и проверить содержимое интересующих программиста переменных. Как правило такие отладчики позволяют отлаживать программу только в машинных командах, представленных в 16-ричном коде.

10.4. Общая методика отладки программного обеспечения

Суммируя все сказанное выше, можно предложить следующую методику отладки программного обеспечения, написанного на универсальных языках программирования для выполнения в операционных системах MS DOS и Win32:

1 этап - изучение проявления ошибки - если выдано какое-либо сообщение или выданы неправильные или неполные результаты, то необходимо их изучить и попытаться понять, какая ошибка могла так проявиться. При этом используют индуктивные и дедуктивные методы отладки. В результате выдвигают версии о характере ошибки, которые необходимо проверить. Для этого можно применить методы и средства получения дополнительной информации об ошибке.

Если ошибка не найдена или система просто «зависла», переходят ко второму этапу.

2 этап - локализация ошибки - определение конкретного фрагмента, при выполнении которого произошло отклонение от предполагаемого вычислительного процесса. Локализация может выполняться:

- путем отсечения частей программы, причем, если при отсечении некоторой части программы ошибка пропадает, то это может означать как то, что ошибка связана с этой частью, так и то, что внесенное изменение изменило проявление ошибки;

- с использованием отладочных средств, позволяющих выполнить интересующих нас фрагмент программы в пошаговом режиме и получить дополнительную информацию о месте проявления и характере ошибки, например, уточнить содержимое указанных переменных.

При этом если были получены неправильные результаты, то в пошаговом режиме проверяют ключевые точки процесса формирования данного результата.

Как подчеркивалось выше, ошибка не обязательно допущена в том месте, где она проявилась. Если в конкретном случае это так, то переходят к следующему этапу.

3 этап - определение причины ошибки - изучение результатов второго этапа и формирование версий возможных причин ошибки. Эти версии необходимо проверить, возможно, используя отладочные средства для просмотра последовательности операторов или значений переменных.

4 этап - исправление ошибки - внесение соответствующих изменений во все операторы, совместное выполнение которых привело к ошибке.

5 этап - повторное тестирование - повторение всех тестов с начала, так как при исправлении обнаруженных ошибок часто вносят в программу новые.

Следует иметь в виду, что процесс отладки можно существенно упростить, если следовать основным рекомендациям структурного подхода к программированию:

- программу наращивать «сверху-вниз», от интерфейса к обрабатывающим подпрограммам, тестируя ее по ходу добавления подпрограмм;

- выводить пользователю вводимые им данные для контроля и проверять их на допустимость сразу после ввода;

- предусматривать вывод основных данных во всех узловых точках алгоритма (ветвлениях, вызовах подпрограмм).

Кроме того, следует более тщательно проверять фрагменты программного обеспечения, где уже были обнаружены ошибки, так как вероятность ошибок в этих местах по статистике выше. Это вызвано следующими причинами. Во-первых, ошибки чаще допускают в сложных местах или в тех случаях, если спецификации на реализуемые операции недостаточно проработаны. Во-вторых, ошибки могут быть результатом того, что программист устал, отвлекся или плохо себя чувствует. В-третьих, как уже упоминалось выше, ошибки часто появляются в результате исправления уже найденных ошибок.

Возвращаясь к рис. 10.2, можно отметить, что проще всего обычно искать ошибки определения данных и ошибки накопления погрешностей: их причины локализованы в месте проявления. Логические ошибки искать существенно сложнее. Причем обнаружение ошибок проектирования требует возврата на предыдущие этапы и внесения соответствующих изменений в проект. Ошибки кодирования бывают как простые, например, использование неинициализированной переменной, так и очень сложные, например, ошибки, связанные с затиранием памяти.

Затиранием памяти называют ошибки, приводящие к тому, что в результате записи некоторой информации не на свое место в оперативной памяти, затираются фрагменты данных или даже команд программы. Ошибки подобного рода обычно вызывают появление сообщения об ошибке. Поэтому определить фрагмент, при выполнении которого ошибка проявляется, несложно. А вот определение фрагмента программы, который затирает память - сложная задача, причем, чем длиннее программа, тем сложнее искать ошибки такого рода. Именно в этом случае часто прибегают к удалению из программы частей, хотя это и не обеспечивает однозначного ответа на вопрос, в какой из частей программы находится ошибка. Эффективнее попытаться определить операторы, которые записывают данные в память не по имени, а по адресу, и последовательно их проверить. Особое внимание при этом следует обращать на корректное распределение памяти под данные.

Контрольные вопросы

1. Какой процесс называют отладкой? В чем его сложность?
2. Назовите основные типы ошибок. Как они проявляются при выполнении программы?
3. Перечислите основные методы отладки. В чем заключается различие между ними? Возьмите любую программу, содержащую ошибки, и попробуйте найти ошибку, используя каждый из перечисленных методов. Какой метод для вас проще и естественней и почему?
4. Какие средства получения дополнительной информации об ошибках вы знаете? Вспомните, какие ошибки вы искали дольше всего и почему. В каких случаях дополнительная информация позволяет найти ошибку?

11. СОСТАВЛЕНИЕ ПРОГРАММНОЙ ДОКУМЕНТАЦИИ

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 не допустимо.

При подготовке документации не следует забывать, что она разрабатывается для того, чтобы ее можно было использовать, и потому она должна содержать все необходимые сведения.

11.1. 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. 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) 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 тестовых 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 проекта.

Прочие документы (коды вида документа - 90-99) могут составляться на любых стадиях разработки, т. е. на стадиях эскизного, технического и рабочего проектов.

Допускается объединять отдельные виды эксплуатационных документов, кроме формуляра и ведомости. Необходимость объединения указывается в техническом задании, а имя берут у одного из объединяемых документов. Например, в настоящее время часто используется эксплуатационный документ, в который отчасти входит руководство системного программиста, программиста и оператора. Он называется «Руководство пользователя» (см. § 11.3).

Рассмотрим наиболее важные программные документы более подробно.

11.2. 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 выглядеть следующим образом:

- 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:

- постановка 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 и для специалистов того же уровня квалификации. Следовательно, в ней уместно использовать специальную терминологию, ссылаться на специальную литературу и т. п.

11.3. Руководство пользователя

Как уже указывалось выше, 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 узко специальной 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.

11.4. 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.

11.5. Основные правила оформления программной документации

При оформлении текстовых и графических материалов, входящих в программную документацию следует придерживаться действующих стандартов. Некоторые положения этих стандартов приведены ниже.

Оформление текстового и графического материала. Текстовые документы оформляют на листах формата А4, причем графический материал допускается представлять на листах формата А3. Поля на листе определяют в соответствии с общими требованиями: левое - не менее 30, правое - не менее 10, верхнее - не менее 15, а нижнее - не менее 20 мм. В текстовых редакторах для оформления записки параметры страницы заказывают в зависимости от устройства печати. При ручном оформлении документов параметры страницы выбирают из соображений удобства.

Нумерация всех страниц - сквозная. Номер проставляется сверху справа арабской цифрой. Страницами считают, как листы с текстами и рисунками, так и листы приложений. Первой страницей считается титульный лист. Номер страницы на титульном листе не проставляют.

Наименование разделов пишут прописными буквами в середине строки. Расстояние между заголовками и текстом, а также между заголовками раздела и подразделов должно быть равно:

- при выполнении документа машинописным способом - двум интервалам;
- при выполнении рукописным способом - 10 мм;
- при использовании текстовых редакторов - определяется возможностями редактора.

Наименования подразделов и пунктов следует размещать с абзацного отступа и печатать вразрядку с прописной буквы, не подчеркивая и без точки в конце. Расстояние между последней строкой текста предыдущего раздела и последующим заголовком при расположении их на одной странице должно быть равно:

- при выполнении документа машинописным способом - трем интервалам;
- при выполнении рукописным способом - не менее 15 мм;
- при использовании текстовых редакторов - определяется возможностями редактора.

Разделы и подразделы нумеруются арабскими цифрами с точкой. Разделы должны иметь порядковые номера 1, 2, и т. д. Номер подраздела включает номер раздела и порядковый номер подраздела, входящего в данный раздел, разделенные точкой. Например: 2.1, 3.5. Ссылки на пункты, разделы и подразделы указывают, используя порядковый номер раздела или пункта, например, «в разд. 4», «в п. 3.3.4».

Текст разделов печатают через 1,5-2 интервала. При использовании текстовых редакторов высота букв и цифр должна быть не менее 1,8 мм (шрифты №11-12).

Перечисления следует нумеровать арабскими цифрами со скобкой, например: 2), 3) и т. д. - с абзацного отступа. Допускается выделять перечисление простановкой дефиса перед пунктом текста или символом, его заменяющим, в текстовых редакторах.

Оформление рисунков, схем алгоритмов, таблиц и формул. В соответствии с ГОСТ 2.105-79 «Общие требования к текстовым документам» иллюстрации (графики, схемы, диаграммы) могут быть приведены как в основном тексте, так и в приложении. Все иллюстрации именуют рисунками. Все рисунки, таблицы и формулы нумеруют арабскими цифрами последовательно (сквозная нумерация) или в пределах раздела (относительная нумерация). В приложении - в пределах приложения.

Каждый рисунок должен иметь подрисуночную подпись - название, помещаемую под рисунком, например:

Рис.12. Форма окна основного меню

На все рисунки, таблицы и формулы в записке должны быть ссылки в виде: «(рис. 12)» или «форма окна основного меню приведена на рис. 12».

Если позволяет место, рисунки и таблицы должны размещаться сразу после абзаца, в котором они упоминаются в первый раз, или как можно ближе к этому абзацу на следующих страницах.

Если рисунок занимает более одной страницы, на всех страницах, кроме первой, проставляется номер рисунка и слово «Продолжение». Например:

Рис. 12. Продолжение

Рисунки следует размещать так, чтобы их можно было рассматривать без поворота страницы. Если такое размещение невозможно, рисунки следует располагать так, чтобы для просмотра надо было повернуть страницу по часовой стрелке. В этом случае верхним краем является левый край страницы. Расположение и размеры полей сохраняются.

Схемы алгоритмов должны быть выполнены в соответствии со стандартом ЕСПД. Толщина сплошной линии при вычерчивании схем алгоритмов должна составлять от 0,6... 1,5 мм. Надписи на схемах должны быть выполнены чертежным шрифтом, высота букв и цифр должна быть не менее 3,5 мм.

Номер таблицы размещают в правом верхнем углу или перед заголовком таблицы, если он есть. Заголовок, кроме первой буквы, выполняют строчными буквами.

Ссылки на таблицы в тексте пояснительной записки указывают в виде слова «табл.» и номера таблицы. Например:

Результаты тестов приведены в табл. 4.

Номер формулы ставится с правой стороны страницы в крутых скобках на уровне формулы. Например:

$$z: =\sin (x)+\ln (y); \quad (12)$$

Ссылка на номер формулы дается в скобках. Например: «расчет значений проводится по формуле (12)».

Оформление приложений. Каждое приложение должно начинаться с новой страницы с указанием в правом углу слова «ПРИЛОЖЕНИЕ» прописными буквами и иметь тематический заголовок. При наличии более одного приложения все они нумеруются арабскими цифрами: ПРИЛОЖЕНИЕ 1, ПРИЛОЖЕНИЕ 2 и т. д. Например:

Титульный лист расчетно-пояснительной записки

Рисунки и таблицы, помещаемые в приложении, нумеруют арабскими цифрами в пределах каждого приложения с добавлением буквы «П». Например:

Рис. П. 12 - 12-й рисунок приложения;
Рис. III .2 - 2-й рисунок 1 -го приложения.

Если в приложении приводится текст программы, то каждый файл оформляют как рисунок с наименованием файла и его назначением, например:

Рис. П2.4. Файл `menuran.pas`- программа движения курсора
основного меню.

Оформление списка литературы. Список литературы должен включать все использованные источники. Сведения о книгах (монографиях, учебниках, пособиях, справочниках и т. д.) должны содержать: фамилию и инициалы автора, заглавие книги, место издания, издательство, год издания. При наличии трех и более авторов допускается указывать фамилию и инициалы только первого из них со словами «и др.». Издательство надо приводить полностью в именительном падеже: допускается сокращение названия только двух городов: Москва (М.) и Санкт-Петербург (СПб.).

Сведения о статье из периодического издания должны включать: фамилию и инициалы автора, наименование статьи, издания (журнала), серии (если она есть), год выпуска, том (если есть), номер издания (журнала) и номера страниц, на которых помещена статья.

При ссылке на источник из списка литературы (особенно при обзоре аналогов) надо указывать порядковый номер по списку литературы, заключенный в квадратные скобки; например: [5].

11.6. Правила оформления расчетно-пояснительных записок при курсовом проектировании

При оформлении пояснительных записок следует придерживаться ГОСТ 7.32-91 (ИСО 5966-82) «Отчет по научно-исследовательской работе. Структура и правила оформления». В соответствии с этим стандартом текстовый документ подобного типа должен включать:

- титульный лист,
- реферат,
- содержание,
- введение,
- основную часть,
- заключение,
- список использованных источников, в том числе литературы,
- приложения.

Титульный лист оформляют в соответствии с ГОСТ 19.104—78 «Единая система программной документации. Основные надписи» (рис. 11.1).

Вторая страница - реферат или аннотация на разрабатываемый программный продукт. Реферат в сжатом виде должен содержать сведения об объеме документа, количестве иллюстраций, таблиц приложений и т. п., а также перечень ключевых слов и основные положения документа. Например, для отчета по научно-исследовательской работе: описание объекта исследования, цели работы, методы исследования и аппаратура, полученные результаты, рекомендации по внедрению и т. д. В аннотации также в сжатом виде описывают назначение и особенности разработки, но она обычно не включает сведений об объеме и т. п.

Министерство образования Российской Федерации
Московский государственный технический университет имени Н.Э. Баумана

Факультет Информатика и системы управления
Кафедра Компьютерные системы и сети

СИСТЕМА УЧЕТА ТЕКУЩЕЙ УСПЕВАЕМОСТИ СТУДЕНТОВ

Расчетно-пояснительная записка
к курсовой работе

Листов 25

Руководитель,
канд. техн. наук, доцент _____ Петров П.П.

Исполнитель,
студент гр. ИУ6-31 _____ Иванов И.И.

2002

Рис. 11.1. Пример титульного листа расчетно-пояснительной записки

Третья страница - содержание, включающее: введение, наименование всех разделов, подразделов, пунктов, заключение, списки литературы и приложений *с указанием номеров страниц*. Ни аннотация или реферат, ни самосодержание в оглавлении не упоминают.

Затем следуют разделы документа в порядке, определенном логикой изложения материала. Далее могут следовать приложения, содержащие материал, не вошедший в документ по причине его ограниченного объема, но интересный для более глубокого понимания излагаемого материала.

В качестве примера рассмотрим оглавление пояснительной записки к проекту по курсу «Технология программирования».

Содержание

Введение	4
1.Выбор технологии, языка и среды программирования.....	6
2.Анализ и уточнение требований к программному продукту	7
2.1.Анализ процесса обработки информации и выбор структур данных для ее хранения	7
2.2.Выбор методов и разработка основных алгоритмов решения задачи	9
3.Разработка структурной схемы программного продукта	11
4.Проектирование интерфейса пользователя	13
4.1.Построение графа диалога	13
4.2.Разработка форм ввода-вывода информации	14
5.Проектирование классов предметной области	17
5.1.Построение диаграммы классов	17
5.2.Уточнение структуры классов предметной области и разработка алгоритмов методов	19
6.Выбор стратегии тестирования и разработка тестов	21
Заключение	24
Список литературы	25
Приложение 1. Техническое задание	26
Приложение 2. Руководство пользователя	30

Контрольные вопросы

1.Назовите основные виды программной документации. Охарактеризуйте каждый из них. В каких случаях их используют?

2.Что должно описываться в пояснительной записке? Кому она предназначена? Почему в пояснительной записке обычно описывают не только принятые решения, но и отвергнутые варианты?

3.На кого рассчитано руководство пользователя? Что оно должно содержать? В каких ситуациях вы читаете руководство пользователя? Вспомните прочитанные вами руководства пользователя. Что вам в них не понравилось?

ПРИЛОЖЕНИЕ

Система условных обозначений унифицированного языка моделирования (UML)

Унифицированный язык моделирования UML-фактически стандартное средство описания проектов, создаваемых с использованием объектно-ориентированного подхода. В модель проекта программного обеспечения по замыслу авторов языка может входить большое количество диаграмм различных типов, использующих единую систему обозначений. Среди диаграмм наиболее часто используемыми являются:

- *диаграммы вариантов использования* или *прецедентов* (uses case diagrams) - показывают основные функции системы для каждого типа пользователей;
- *диаграммы классов* (class diagrams): контекстные, описания интерфейсов и реализации - демонстрируют отношения классов между собой;
- *диаграммы деятельности* (activity diagrams) - представляют собой схему потоков управления для решения некоторой задачи по отдельным действиям, допускают наличие параллельных и/или альтернативных действий;
- *диаграммы взаимодействия* (interaction diagrams) двух альтернативных типов:
 - а) *диаграммы последовательности действий* (sequence diagrams) - отображают упорядоченное по времени взаимодействие объектов в процессе выполнения варианта использования,
 - б) *диаграммы кооперации* (collaboration diagrams) – предоставляют ту же информацию, что и диаграммы последовательности действий, но в форме, позволяющей лучше представить ответственности классов в целом;
- *диаграммы состояний объекта* (statechart diagrams) - показывают состояния объекта и условия переходов из одного состояния в другое;
- *диаграммы пакетов* (package diagrams) - демонстрируют связи наборов классов, объединенных в пакеты, между собой;
- *диаграммы компонентов* (component diagrams) - показывают, из каких программных компонентов состоит программное обеспечение и как эти компоненты связаны между собой;
- *диаграммы размещения* (deployment diagrams) - позволяют связать программные и аппаратные компоненты системы.

Дополнениями к диаграммам служат формализованные и неформализованные текстовые описания, комментарии и словари.

При построении этих и других диаграмм используют унифицированную систему обозначений. Обозначения диаграмм прецедентов приведены в табл. П.1, диаграмм классов и пакетов - в табл. П.2, диаграмм взаимодействия - в табл. П.3, диаграмм деятельности и состояний объекта - в табл. П.4, диаграмм компонентов и размещения - в табл. П.5. При необходимости допускается использование обозначений одних диаграмм на других.

Таблица П.1





Компонент модели	Условное обозначение	Компонент модели	Условное обозначение
Действующее лицо		Связь	
Вариант использования или прецедент		Связи «Расширение» и «Использование»	

Таблица П.2

Компонент модели	Условное обозначение	Компонент модели	Условное обозначение
Класс со скрытыми секциями		Абстрактный класс	
Класс с раскрытыми секциями		Абстрактная операция класса	
Класс (пиктограмма)		Параметризованный класс (шаблон)	
Активный класс		Настроенный класс (шаблон)	
Видимость атрибутов класса	+ Общий - Скрытый # Защищенный	Пакет	
Граничный класс		Пакет с раскрытой секцией	
Управляющий класс		Пакет (пиктограмма)	
Класс-сущность		Интерфейс	
Обобщение		Реализация интерфейса классом	
Двунаправленная ассоциация		Реализация интерфейса пакетом	
Однонаправленная ассоциация		Реализация интерфейса (раскрытая)	
Агрегация		Использование интерфейса классом	
Композиция		Использование интерфейса пакетом	
Отношение ассоциации класса		Зависимость классов	
Примечание		Связь пакетов	

Таблица П.3









Компонент модели	Условное обозначение	Компонент модели	Условное обозначение
Объект		Фокус управления	
Линия жизни		Разрушение объекта	
Вызов процедуры		Асинхронный поток	
Синхронный поток управления		Возврат управления	

Таблица П.4




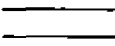
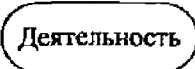
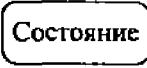


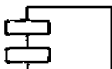


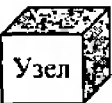
Компонент модели	Условное обозначение	Компонент модели	Условное обозначение
Начало		Переход	
Конец		Линейки синхронизации	
Деятельность		Состояние	
Выбор		Составное состояние	

Таблица П.5

Компонент модели	Условное обозначение	Компонент модели	Условное обозначение
Программный компонент		База данных	
Текстовый файл		Таблица	
DLL		Узел (компьютер)	

СПИСОК ЛИТЕРАТУРЫ

1. *Агабеков Л. К., Иванова Г. С.* Программирование на C++. Ч. 1. Средства процедурного программирования: Учеб. пособие. - М.: Изд-во МГТУ им. Баумана, 1999.
2. *Агабеков Л. Е., Иванова Г. С.* Программирование на C++. Ч. 2. Средства объектно-ориентированного программирования: Учеб. пособие. - М.: Изд-во МГТУ им. Баумана, 1996.
3. *Анишина М.Л.* Страсти по качеству ПО. Открытые системы, № 6, 1998.
4. *Артемьев В. И., Строганов В. Ю.* Организация диалога в САПР. Разработка САПР: В 10 кн. Кн. 5.-М.: Высш. шк., 1991.
5. *Бадд Т.* Объектно-ориентированное программирование в действии: Пер. с англ. - СПб.: Питер, 1997.
6. *Бозм Б.* Инженерное проектирование программного обеспечения. — М.: Радио и связь, 1985.
7. *Бозм Б., Браун Дж., Каспар Х. и др.* Характеристики качества программного обеспечения. - М.: Мир, 1981.
8. *Брукс Ф.* Мифический человеко-месяц или как создаются программные системы. - СПб.: Символ-Плюс, 1999.
9. *Бутаков К.А.* Методы создания качественного программного обеспечения ЭВМ. - М.: Энергоатомиздат, 1984.
10. *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд.: Пер. с англ. - М.: Бином, СПб.: Невский диалект, 1998.
11. *Буч Г., Рамбо Д., Джейкобсон А.* Язык UML. Руководство пользователя. - М.: ДМК Пресс, 2001.
12. *Вендров А.М.* Один из подходов к выбору средств проектирования баз данных и приложений. // СУБД. 1995. № 3.
13. *Вендров А.М.* CASE-технологии. Современные методы и средства проектирования информационных систем. - М.: Финансы и статистика, 1998.
14. *Вендров А.М.* Проектирование программного обеспечения экономических информационных систем: Учеб. - М.: Финансы и статистика, 2000.
15. *Вирт И.* Алгоритмы и структуры данных: Пер. с англ. - М.: Мир, 1989.
16. *Гейн К., Сарсон Т.* Системный структурный анализ: средства и методы. - М.: «Эйтекс», 1992.
17. *Гримм С.Дж.* Как писать руководства для пользователей. - М.: Радио и связь, 1985.
18. *Грис Д.* Наука программирования. - М.: Мир, 1984.
19. *Дал У., Дейкстра Э., Хоор К.* Структурное программирование: Пер. с англ. - М.: Мир, 1975.
20. *Зелковец М., Шоу А., Гэннон Дж.* Принципы разработки программного обеспечения. - М.: Мир, 1982.
21. *Зиглер К.* Методы проектирования программных систем. - М.: Мир, 1985.
22. *Зиндер Е.З.* Бизнес реинжиниринг и технологии системного проектирования: Учеб. пособие. - М.: Центр информационных технологий, 1996.
23. *Йордан Э.* Структурное программирование и проектирование программ. - М.: Мир, 1979.
24. *Иванова Г.С.* Основы программирования: Учеб. для вузов. - М.: Изд-во МГТУ им. Баумана, 2001.
25. *Иванова Г.С.* Программирование на Ассемблере ПЭВМ: Метод, указания по выполнению лабораторных работ. - М.: Изд-во МГТУ им. Н.Э. Баумана, 1991.
26. *Иванова Г.С., Коновалов С.М., Петрова Г.Б.* Работа на ЭВМ: Метод, указания по выполнению вычислительной практики. - М.: МГТУ им. Н.Э. Баумана, 1991.
27. *Иванова Г.С., Ничушкина Т.Н., Овчинников В. А.* Выбор структур данных для представления графов при решении комбинаторно-оптимизационных задач. // Вестник МГТУ, серия «Приборостроение», спец. выпуск «Информатика». 2001. №2.
28. *Иванова Г.С., Мартынюк В.А., Петрова Г.Б.* Отладка программ на Ассемблере: Метод, указания. - М.: МВТУ им. Н.Э. Баумана, 1984.

29. *Иванова Г.С., Ничушкина Т.Н., Пугачев Е.К.* Объектно-ориентированное программирование: Учеб. для вузов. - М.: Изд-во МГТУ им. Баумана, 2001.
30. *Калянов Г.Н.* Консалтинг при автоматизации предприятий. Подходы, методы, средства. - М.: СИНТЕГ, 1997.
31. *Канер С., Фояк Д., Нгуен Е.К.* Тестирование программного обеспечения. - Киев: «ДиаСофт», 2000.
32. *Кватрани Т.* Rational Rose 2000 и UML. Визуальное моделирование. - М.: ДМК Пресс, 2001.
33. *Кинг Д.* Создание эффективного программного обеспечения. - М.: Мир, 1991.
34. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ: Пер. с англ. - М.: МЦНМО, 2000.
35. *Коутс Р., Влейминк И.* Интерфейс «человек-компьютер». - М.: Мир, 1990.
36. *Корячко В.Л., Курейчик В.М., Норенков И.П.* Теоретические основы САПР. - М.: Энергоатомиздат, 1987.
37. *Ларман К.* Применение UML и шаблонов проектирования. - М.: Издательский дом «Вильямс», 2001.
38. *Леоненков А.В.* Самоучитель UML. - СПб.: БХВ-Петербург, 2001.
39. *Липаев В.В.* Тестирование программ. - М.: Радио и связь, 1986.
40. *Липаев В.В., Позин Б.А., Штрик А.А.* Технология сборочного программирования. - М.: Радио и связь, 1992.
41. *Липаев В.В.* Управление разработкой программных комплексов. - М.: Финансы и статистика, 1993.
42. *Липаев В.В., Филинов К.Н.* Мобильность программ и данных в открытых информационных системах. - М.: Научная книга, 1997.
43. *Липаев В.В.* Надежность программных средств. - М.: «Синтег», 1998.
44. *Лисков Б., Гатэг Дж.* Использование абстракций и спецификаций при разработке программ. - М.: Мир, 1989.
45. *Ломако Е.И., Гуков Л.И., Морозова А.В.* Макетирование, проектирование и реализация диалоговых информационных систем. - М.: Финансы и статистика, 1993.
46. *Майерс Г.* Надежность программного обеспечения. - М.: Мир, 1980.
47. *Майерс Г.* Искусство тестирования программ. - М.: Финансы и статистика, 1982.
48. *Маклаков С. В.* CASE-средства разработки информационных систем BPWin, ERWin. - М.: Диалог МИФИ, 2000.
49. *Мандел Т.* Разработка пользовательского интерфейса. - М.: ДМК Пресс, 2001.
50. *Марка Д.А., МакГоуэн К.* Методология структурного анализа и проектирования. - М.: МетаТехнология, 1993.
51. *Мартин Дж.* Организация баз данных в вычислительных системах. - М.: Мир, 1980.
52. *Международные стандарты, поддерживающие жизненный цикл программных средств.* - М.: МП «Экономика», 1996.
53. *Новожинов Ю.В.* Объектно-ориентированные технологии разработки сложных программных систем. - М.: ДМК Пресс, 1996.
54. *Овчинников В.А.* Алгоритмизация комбинаторно-оптимизационных задач при проектировании ЭВМ или систем: Учеб. для вузов. - М.: Изд-во МГТУ им. Баумана, 2001.
55. *Программные системы / Бахманн П., Френцель М., Ханцшманн К. и др.* — М.: Мир, 1988.
56. *Проектирование пользовательского интерфейса на персональных компьютерах. Стандарт фирмы IBM.* - Вильнюс, DBS LTD, 1992.
57. *Разработка САПР. В 10 кн. Кн. 3. Проектирование программного обеспечения САПР: Прак. пособие/ Б. С. Федоров, Б.Н. Гуляев; под ред. А.В. Петрова.* - М.: Высш. шк., 1990.
58. *Росс Д.* Структурный анализ: язык для передачи понимания // Требования и спецификации в разработке программ. - М.: Мир, 1984.
59. *Савельев А.Я.* Прикладная теория цифровых автоматов: Учеб. для вузов. - М.: Высш. шк., 1987.

60. *Тассел Д. Ван.* Стил, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1985.
61. *Тейер Т., Липов М., Нельсон Э.* Надежность программного обеспечения. - М.: Мир, 1981.
62. Требования и спецификации в разработке программ. - М.: Мир, 1984.
63. *Фокс Д.* Программное обеспечение и его разработка. М.: Мир, 1985.
64. *Хьюз Дж., Мичтом Дж.* Структурный подход к программированию: Пер. с англ. - М.: Мир, 1980.
65. *Чен П.* Модель «сущность-связь» - шаг к единому представлению данных // СУБД. 1995. №3. С. 137-158.
66. *Шураков В.В.* Надежность программного обеспечения систем обработки данных: Учеб. - М.: Финансы и статистика, 1987.