bhv®

Александр Леоненков

САМОУЧИТЕЛЬ

UML

2-е издание

Объектно-ориентированный анализ

Проектирование программных систем

Диаграммы логического и физического моделирования

Документирование проекта в нотации UML

UML в Rational Rose 2002

Язык объектных ограничений OCL

Освой UML самостоятельно

Александр Леоненков

САМОУЧИТЕЛЬ UML

2-е издание

Санкт-Петербург

«БХВ-Петербург»

2004

УДК 681.3.068+800.92UML
ББК 32.973.26-018.1
Л47

Леоненков А. В.

Л47 Самоучитель UML. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 432 с.: ил.

ISBN 5-94157-342-1

Цель книги — помочь менеджерам и руководителям проектов, руководителям информационных служб, бизнес-аналитикам, корпоративным программистам и ведущим разработчикам самостоятельно освоить базовые концепции и понятия наиболее перспективной и современной методологии разработки корпоративных информационных систем для последующего применения полученных знаний в ходе выполнения реальных проектов и совершенствования бизнес-процессов с использованием соответствующих CASE-средств. Рассматриваются основы современной технологии унифицированного анализа и проектирования программных систем на языке UML. Подробно излагаются базовые понятия языка UML, необходимые для построения объектно-ориентированных моделей корпоративных программных систем с использованием специальной графической нотации. Приводятся конкретные рекомендации по изображению канонических диаграмм UML и рассматриваются особенности разработки моделей с помощью CASE-средства IBM Rational Rose 2002. Описывается нотация OCL — языка объектных ограничений, что делает книгу уникальной среди аналогичных изданий.

Для программистов

УДК 681.3.068+800.92UML
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Андрей Смышляев*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.02.04.
Формат 70×100[1]/[16]. Печать офсетная. Усл. печ. л. 46,44.
Тираж 3 000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-342-1

© Леоненков А. В., 2004
© Оформление, издательство "БХВ-Петербург", 2004

Содержание

Предисловие 9

Структура книги 10
Рекомендации по изучению языка UML 11
Благодарности 12

ЧАСТЬ I. ОСНОВЫ UML 15

Глава 1. Введение 17

1.1. Методология процедурно-ориентированного программирования 17
1.2. Методология объектно-ориентированного программирования 22
1.3. Методология объектно-ориентированного анализа и проектирования 30
1.4. Методология системного анализа и системного моделирования 34

Глава 2. Исторический обзор развития методологии объектно-ориентированного анализа и проектирования сложных систем 39

2.1. Предыстория. Математические основы 39
2.2. Диаграммы структурного системного анализа 51
2.3. Основные этапы развития UML 63

Глава 3. Основные компоненты языка UML 70

3.1. Назначение языка UML 72
3.2. Общая структура языка UML 76
3.3. Пакеты в языке UML 79
3.4. Основные пакеты метамодели языка UML 82
3.5. Специфика описания метамодели языка UML 92
3.6. Особенности изображения диаграмм языка UML 97

ЧАСТЬ II. ДИАГРАММЫ КОНЦЕПТУАЛЬНОГО, ЛОГИЧЕСКОГО И ФИЗИЧЕСКОГО МОДЕЛИРОВАНИЯ 103

Глава 4. Диаграмма вариантов использования (use case diagram) 105

4.1. Вариант использования 106
4.2. Актеры 108
4.3. Примечания 111
4.4. Отношения на диаграмме вариантов использования 112
4.5. Расширение языка UML для бизнес-моделирования 118
4.6. Текстовые сценарии вариантов использования 121
4.7. Пример построения диаграммы вариантов использования для системы управления банкоматом 123
4.8. Рекомендации по разработке диаграмм вариантов использования 127

Глава 5. Диаграмма классов (class diagram) 133

5.1. Класс 134
5.2. Отношения между классами 145
5.3. Расширение языка UML для построения моделей программного обеспечения и бизнес-систем 158
5.4. Шаблоны или параметризованные классы 162
5.5. Пример построения диаграммы классов системы управления банкоматом 164
5.6. Рекомендации по построению диаграмм классов 165

Глава 6. Диаграмма кооперации (collaboration diagram) 169

6.1. Кооперация 170
6.2. Объекты 175
6.3. Связи 180
6.4. Сообщения 182
6.5. Пример построения диаграммы кооперации системы управления банкоматом 188
6.6. Заключительные рекомендации по построению диаграмм кооперации 190

Глава 7. Диаграмма последовательности (sequence diagram) 193

7.1. Объекты 193
7.2. Сообщения на диаграмме последовательности 197
7.3. Пример построения диаграммы последовательности 203

7.4. Пример построения диаграммы последовательности системы управления банкоматом ... 206
7.5. Заключительные рекомендации по построению диаграмм последовательности ... 208

Глава 8. Диаграмма состояний (statechart diagram) ... 210

8.1. Конечные автоматы ... 212
8.2. Состояние ... 215
8.3. Переход ... 219
8.4. Составное состояние и подсостояние ... 225
8.5. Исторические состояния ... 228
8.6. Сложные переходы ... 230
8.7. Пример построения диаграммы состояний системы управления банкоматом ... 235
8.8. Заключительные рекомендации по построению диаграмм состояний ... 237

Глава 9. Диаграмма деятельности (activity diagram) ... 240

9.1. Состояние действия ... 241
9.2. Переходы ... 243
9.3. Дорожки ... 249
9.4. Объекты ... 251
9.5. Пример построения диаграммы деятельности системы управления банкоматом ... 255
9.6. Рекомендации по построению диаграмм деятельности ... 257

Глава 10. Диаграмма компонентов (component diagram) ... 260

10.1. Компоненты ... 262
10.2. Интерфейсы ... 265
10.3. Зависимости ... 266
10.4. Пример построения диаграммы компонентов системы управления банкоматом ... 270
10.5. Рекомендации по построению диаграммы компонентов ... 272

Глава 11. Диаграмма развертывания (deployment diagram) ... 275

11.1. Узел ... 276
11.2. Соединения и зависимости ... 280
11.3. Пример построения диаграммы развертывания системы управления банкоматом ... 282
11.4. Рекомендации по построению диаграммы развертывания ... 284

ЧАСТЬ III. АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ НОТАЦИИ ЯЗЫКА UML И CASE-СРЕДСТВА IBM RATIONAL ROSE 2002 289

Глава 12. Общая характеристика инструментария IBM Rational Rose 2002 291

12.1. Общая характеристика CASE-средства IBM Rational Rose 2002 292
12.2. Особенности рабочего интерфейса IBM Rational Rose 2002 293
12.3. Назначение операций главного меню 299
12.4. Назначение операций стандартной панели инструментов 312

Глава 13. Начало работы над проектом в среде IBM Rational Rose 2002 315

13.1. Разработка диаграммы вариантов использования в среде IBM Rational Rose 315
13.2. Разработка диаграммы классов в среде IBM Rational Rose 325
13.3. Разработка диаграммы кооперации в среде IBM Rational Rose 333
13.4. Разработка диаграммы последовательности в среде IBM Rational Rose 340

Глава 14. Завершение работы над проектом в среде IBM Rational Rose 2002 346

14.1. Разработка диаграммы состояний в среде IBM Rational Rose 346
14.2. Разработка диаграммы деятельности в среде IBM Rational Rose 352
14.3. Разработка диаграммы компонентов в среде IBM Rational Rose 357
14.4. Разработка диаграммы развертывания в среде IBM Rational Rose 362
14.5. Генерации программного кода в среде IBM Rational Rose 366

Заключение 375

Приложение 379

Язык объектных ограничений 379
Выражения языка OCL 381
Основные типы значений и операций в языке OCL 383

Операции над отдельными типами значений 384
Допустимые выражения в языке OCL 391
Неопределенные значения выражений 392
Совокупности допустимых значений в языке OCL....................392
Операции над совокупностями значений 393
Некоторые операции с множествами, последовательностями и комплектами 396
Операции преобразования типов 397
Примеры записи выражений языка OCL 398

Глоссарий 401

Литература 417

Предметный указатель 421

Предисловие

Книга посвящена рассмотрению основ Унифицированного языка моделирования или, сокращенно, языка UML (Unified Modeling Language), который предназначен для описания, визуализации и документирования объектно-ориентированных систем и бизнес-процессов с ориентацией на их последующую реализацию в виде программного обеспечения.

Чем обусловлена необходимость создания и изучения еще одного языка, который вовсе не является языком программирования в привычном смысле этого слова?

Прежде всего — это возрастающая сложность прикладных программ и высокая стоимость их разработки и сопровождения. Действительно, в разработке современных корпоративных информационных систем принимают участие десятки и сотни различных специалистов, для которых построение предварительной модели системы до начала написания соответствующего программного кода стало настоятельной необходимостью. Разработка программных систем на заказ также приводит к необходимости поддержания единого стиля для различных версий программ при их постоянной доработке и модификации.

Основное требование к такой предварительной модели программной системы — модель должна быть понятна как заказчику, так и всем специалистам проектной группы, включая и программистов. Оказалось, что разработка соответствующего языка моделирования или нотации является непростым делом. Потребовалось несколько лет, прежде чем усилия группы специалистов ведущих фирм-производителей программного и аппаратного обеспечения привели к появлению языка UML.

Какими бы свойствами ни обладал новый язык, важной особенностью для его жизнеспособности является реализация и поддержка соответствующей нотации в коммерческих программных продуктах. В последние годы наблюдается активный интерес к языку UML, о чем свидетельствуют десятки

коммерческих программных инструментариев, предназначенных для автоматизации разработки программного обеспечения на основе построения предварительной объектно-ориентированной модели предметной области. Одним из наиболее мощных инструментальных средств этого класса попрежнему остается "широко известный в узких корпоративных кругах" IBM Rational Rose.

Хотя в настоящее время в России получили широкое применение три нотации визуального моделирования: IDEF (Icam DEFinition), ARIS (Architecture of Integrated Information Systems) и UML, именно последняя из них все более активно используется корпоративными программистами для разработки графических моделей при выполнении программных проектов. При этом новые инструментарии разработки приложений и соответствующие среды визуального программирования MS Visual Studio 6/.NET и Borland Delphi/C++Builder/Jbuilder не только поддерживают нотацию языка UML в качестве базового средства моделирования программных систем, но и позволяют получить исполнимые модули программ на основе разработанной графической модели.

Следует отметить еще одну область, в которой все более активно используются графические нотации — это выполнение работ по приведению системы менеджмента качества в соответствии со стандартом ISO 9001:2000 в рамках сертификации предприятий и компаний. В этой области язык UML применяется для визуального моделирования и документирования бизнес-процессов, в результате чего разработанные диаграммы и пояснения к ним предоставляются международным сертификационным органам для получения соответствующего сертификата.

Структура книги

Во втором издании книги были исправлены некоторые неточности в тексте и на графических диаграммах, уточнена семантика отдельных терминов и даны более ясные формулировки основных элементов языка UML. При этом материал книги соответствует последней версии языка UML 1.5. Для иллюстрации рассматриваемых понятий языка UML приводится описание сквозного примера модели системы управления банкоматом.

В основу книги положены две основные идеи. С одной стороны, рассмотреть все базовые конструкции языка UML, необходимые для самостоятельной разработки концептуальных, логических и физических моделей программного обеспечения. С другой стороны, донести до читателя основы методологии моделирования сложных систем, без понимания которой вряд ли возможно адекватно и безошибочно использовать богатейший потенциал возможностей языка UML.

Материал книги делится на три части. Первая часть знакомит с основными теоретическими понятиями, которые необходимы для правильного понимания назначения и возможностей языка UML. Здесь также приводится исторический обзор развития технологий программирования и методологии объектно-ориентированного анализа и проектирования. Поскольку UML не является формальным языком с фиксированным синтаксисом, описание языка рассматривается как некоторая открытая модель с определенными базовыми семантическими конструкциями и неформальными правилами их расширения.

Вторая часть является центральной в книге и содержит описание назначения элементов всех канонических диаграмм языка UML (версия 1.5), которые являются основой построения концептуальных, логических и физических моделей. В отдельных главах второй части последовательно рассматриваются:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграмма кооперации;
- диаграмма последовательности;
- диаграмма состояний;
- диаграмма деятельности;
- диаграмма компонентов;
- диаграмма развертывания.

Для каждой из диаграмм описываются базовые элементы графической нотации, необходимые для изображения различных элементов диаграмм, а также рассматривается соответствующая диаграмма сквозного примера модели системы управления банкоматом с использованием соответствующей графической нотации.

Третья часть содержит описание особенностей реализации языка UML в уже упомянутом CASE-инструментарии — IBM Rational Rose 2002. В заключении сделана попытка оценить перспективы дальнейшего развития языка UML и технологии компонентной разработки приложений.

Рекомендации по изучению языка UML

Тематика книги, возможно, производит нетрадиционное впечатление для многих программистов, занятых разработкой конкретных приложений. Однако современные тенденции развития индустрии создания программного обеспечения складываются таким образом, что именно язык UML де-факто оказывается общепризнанным стандартом в области разработки моделей

систем и процессов с его последующей реализацией в соответствующих инструментальных средствах.

В то же время, следует отметить одну важную особенность современной программной инженерии — это тенденция специализации в области разработки программ. Речь идет о появлении таких специалистов, как системный аналитик, менеджер проекта, бизнес-аналитик и архитектор системы, которые, наряду со знанием языков программирования, должны владеть методологией объектно-ориентированного анализа и моделирования предметной области. Для системного инженера и интегратора также важно разбираться в возможностях реализации конкретных проектов и использовать общепринятую систему обозначений для решения своих задач. Наконец, программисты, занятые в масштабных проектах, должны четко понимать функциональные аспекты будущей программной системы. Для всех этих категорий специалистов и предназначена данная книга, поскольку язык UML позволяет организовать эффективное и понятное для всех общение.

Вполне вероятно, что со временем элементы нотации языка UML будут использоваться в учебных программах для обучения студентов самых различных специальностей. Во всяком случае, многие преподаватели давно осознали ограниченность существующих отечественных стандартов на разработку программной документации, которые не отражают современных тенденций развития программной инженерии. Как студенты, так и преподаватели найдут в книге интересный материал для размышления, который позволит понять целый ряд особенностей и перспектив профессионального образования в области современных информационных технологий.

Для понимания основных конструкций языка UML достаточно общей эрудиции и некоторого знакомства с одним из языков программирования. Читатели, которые ставят перед собой такую цель, могут бегло просмотреть материал первой части и сразу приступить к изучению отдельных диаграмм. Однако для творческого овладения методологией объектно-ориентированного анализа и моделирования необходима, как представляется, определенная математическая культура и знание некоторых общих понятий прикладного системного анализа. Соответствующий теоретический материал приводится в первой части книги и далее используется по мере изложения элементов конкретных диаграмм.

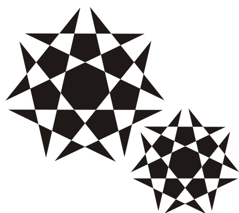
Благодарности

Автор искренне благодарит К. Н. Ильинского, Е. В. Кондукову, Е. В. Строганову, А. М. Коновалова и И. А. Корнеева за предоставленные в разное время материалы, которые были использованы при написании книги. Автор искренне признателен директору Школы IT-менеджмента АНХ при Правительстве РФ (**www.itmane.ru**) А. И. Соколову, а также Л. А. Ермакову, В. А. Перекрестову

и В. В. Фамильнову за оказанную поддержку в процессе работы над книгой. В предоставлении демонстрационной версии IBM Rational Rose 2002 неоценимую помощь оказали сотрудники компании Интерфейс Ltd., которым автор выражает свою признательность.

Написание современной книги немыслимо без использования ресурсов Интернета. В этой связи хотелось бы выразить особую признательность директору Междисциплинарного Центра СПбГУ (**www.icape.nw.ru**) профессору Н. В. Борисову за предоставленную возможность электронной коммуникации.

В заключение следует специально отметить одно немаловажное обстоятельство, которое усложняет понимание и распространение идей визуального моделирования среди отечественных системных аналитиков, менеджеров проектов, системных инженеров и программистов. Речь идет о неустановившейся терминологии в этой области и о неоднозначности перевода отдельных терминов, имеющих зачастую многозначное толкование в том или ином конкретном контексте. С этой целью названия наиболее важных понятий и их краткая характеристика отдельно приводятся в *Глоссарии* в конце книги. В любом случае автор будет признателен за все отзывы и конструктивные предложения, связанные с содержанием книги и проблематикой моделирования в контексте языка UML, которые можно отправлять по адресу: **uml@itmane.ru**.



Часть I

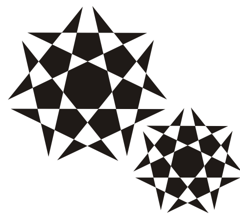
Основы UML

Глава 1. Введение

Глава 2. Исторический обзор развития методологии объектно-ориентированного анализа и проектирования сложных систем

Глава 3. Основные компоненты языка UML

Глава 1



Введение

Мир компьютерных и информационных технологий без преувеличения можно назвать наиболее динамичной областью современных знаний. Практически каждый год появляются новые модели процессоров и комплектующих, новые версии операционных систем и программного обеспечения. Все это происходит на фоне постоянного усложнения не только отдельных физических и программных компонентов, но и лежащих в их основе концепций и идей.

Кажется, еще совсем недавно профессиональному программисту было достаточно в совершенстве владеть одним-двумя языками программирования, чтобы разрабатывать серьезные программные приложения. Выбор платформы и операционной системы, как правило, не являлся серьезной проблемой. Разработка блок-схем алгоритмов и программ представлялась лишней тратой времени, и в лучшем случае выполнялась с целью документирования проекта. А сопровождение программы, хотя и было сопряжено с объективными трудностями, могло быть реализовано простым добавлением или изменением исходного кода.

1.1. Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин или компьютеров ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых преобразовать в понятные компьютеру инструкции, и любая вычислительная задача может быть решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специальные языки программирования, позволившие преобразовывать отдельные вычислительные операции в соответствующий программный код.

Основой данной методологии разработки программ являлась *процедурная* или *алгоритмическая организация* структуры программного кода. Это было

настолько естественно для решения вычислительных задач, что ни у кого не вызывала сомнений целесообразность такого подхода. Исходным понятием этой методологии являлось понятие *алгоритма*, под которым, в общем случае, понимается некоторое предписание выполнить точно определенную последовательность действий, направленных на достижение заданной цели или решение поставленной задачи. Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или корней линейной системы уравнений.

Примечание

Принято считать, что сам термин алгоритм, а точнее его более ранний вариант — *алгорифм*, происходит от имени средневекового математика Аль-Хорезми, который в 825 г. описал правила выполнения арифметических действий в десятичной системе счисления.

С этой точки зрения вся история математики тесно связана с разработкой тех или иных алгоритмов решения актуальных для своей эпохи задач. Более того, само понятие алгоритма стало предметом соответствующей теории — *теории алгоритмов*, которая занимается изучением общих свойств алгоритмов. Со временем содержание этой теории стало настолько абстрактным, что понимание соответствующих результатов стало доступно только специалистам. Как дань этой традиции какой-то период языки программирования назывались алгоритмическими, а первое графическое средство документирования программ получило название *блок-схемы алгоритма*. Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90, который регламентировал использование условных обозначений в схемах алгоритмов, программ, данных и систем.

Однако потребности практики не всегда требовали установления вычислимости конкретных функций или разрешимости отдельных задач. В языках программирования возникло и закрепилось новое понятие *процедуры*, которое конкретизировало общее понятие алгоритма применительно к решению задач на компьютерах. Так же, как и алгоритм, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая получила название процедуры.

Со временем разработка больших программ превратилась в серьезную проблему и потребовала их разбиения на более мелкие фрагменты. Основой для такого разбиения стала *процедурная декомпозиция*, при которой отдельные части программы или *модули* представляли собой совокупность процедур для решения некоторой совокупности задач. Главная особенность процедурного программирования заключается в том, что программа всегда имеет начало во времени или начальную процедуру (начальный блок) и окончание

(конечный блок). При этом вся программа может быть представлена визуально в виде направленной последовательности графических примитивов или блоков (рис. 1.1).

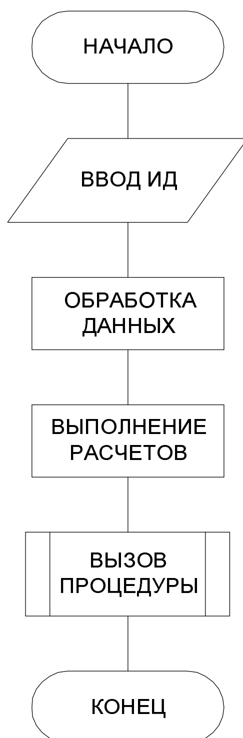


Рис. 1.1. Графическое представление программы в виде последовательности процедур

Важным свойством таких программ является необходимость завершения всех действий предшествующей процедуры для начала действий последующей процедуры. Изменение порядка выполнения таких действий даже в пределах одной процедуры потребовало включения в языки программирования специальных условных операторов типа `If-then-else` и `Goto` для реализации ветвления вычислительного процесса в зависимости от промежуточных результатов решения задачи.

Примечание

Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода `goto` способно серьезно осложнить

понимание кода. Соответствующие программы стали сравнивать со спагетти, называя их *bowl of spagetti*, имея в виду многочисленные переходы от одного фрагмента программы к другому или, что еще хуже, возврат от конечных операторов программы к ее начальным операторам. Ситуация казалась настолько драматичной, что в литературе зазвучали призывы исключить оператор `goto` из языков программирования. Именно с этого времени принято считать хорошим стилем программирование без `goto`.

Рассмотренные идеи способствовали становлению определенной системы взглядов на процесс разработки программ и написания программных кодов, которая получила название *методологии структурного программирования*. Основой данной методологии является процедурная декомпозиция программной системы и организация отдельных модулей в виде совокупности выполняемых процедур. В рамках данной методологии получило развитие *нисходящее проектирование* программ или программирование "сверху-вниз". Период наибольшей популярности идей структурного программирования приходится на конец 70-х — начало 80-х годов прошлого столетия.

Как вспомогательное средство структуризации программного кода было рекомендовано использование отступов в начале каждой строки, которые должны выделять вложенные циклы и условные операторы. Все это призвано способствовать пониманию или читабельности самой программы. Данное правило со временем было реализовано в большинстве популярных средств разработки. Листинг 1.1 фрагмента программы на языке Pascal иллюстрирует эту особенность написания программ.

Листинг 1.1. Пример фрагмента программы на языке Pascal, разработанной с использованием правил структурного программирования

```
Procedure FirstOpt;
  Begin
    FuncRaz(Frec, Rn);
    for i:=1 to N do
       RvarRec[i]:= Rn[i];
    FvarRec:= Frec;
    NumIt:=0;
      Repeat
        NumIt:=NumIt+1;
        V:= Frec;
        for j:=1 to K do
           for l:=1 to M do
              begin
                 S:=0.0;
```

```
            T:=0.0;
            for i:=1 to N do
               begin
                  T:=T+sqr(W1[i,j])*Xpr[i,l];
                  S:=S+sqr(W1[i,j])
               end;
            Zentr[j,l]:=T/S
         end;
      for j:=1 to K do
         for i:=1 to N do
            begin
               S:=0.0;
               P:=0.0;
               Q:=0.0;
               for l:=1 to M do
                  S:=S+sqr(Xpr[i,l]-Zentr[j,l]);
                P:=1.0/S;
            end;
      Q:=0.0;
      D:=0;
      for i:=1 to N do
         for j:=1 to K do
            if Abs(W1[i,j]-W2[i,j]) >= Eps then D:=1;
      for i:=1 to N do
         for j:=1 to K do
            W1[i,j]:=W2[i,j]
    Until (D=0)or(NumIt=NumMax)
 End;
```

В этот период основным показателем сложности разработки программ считали ее размер. Вполне серьезно обсуждались такие оценки сложности программ, как количество строк программного кода. Правда, при этом делались некоторые предположения относительно синтаксиса самих строк, которые должны были удовлетворять определенным правилам. Общая трудоемкость разработки программ оценивалась специальной единицей измерения — "человеко-месяц" или "человеко-год". А профессионализм программиста напрямую связывался с количеством строк программного кода, который он мог написать и отладить в течение, скажем, месяца.

Примечание

Сейчас попытки оценить профессионализм программиста количеством строк программного кода могут вызвать лишь улыбку собеседника. Действительно, пользуясь встроенными мастерами современных инструментариев разработки (MS Visual C++ или Borland Delphi), даже новичок может за считанные секунды последовательным нажатием кнопок диалоговых меню создать работоспособное приложение, содержащее сотни строк программного кода и состоящее из десятка отдельных файлов проекта.

1.2. Методология объектно-ориентированного программирования

Со временем ситуация стала существенно изменяться. Оказалось, что трудоемкость разработки программных приложений на начальных этапах программирования оценивалась значительно ниже реально затрачиваемых усилий, что служило причиной дополнительных расходов и затягивания окончательных сроков готовности программ. В процессе разработки приложений изменялись функциональные требования заказчика, что еще более отдаляло момент окончания работы программистов. Увеличение размеров программ приводило к необходимости привлечения дополнительных программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы.

Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. Если в эпоху "больших машин" основными потребителями программного обеспечения были крупные предприятия, компании и учреждения, то позже появились персональные компьютеры и быстро стали повсеместным атрибутом мелкого и среднего бизнеса. Вычислительные и расчетно-алгоритмические задачи в этой области традиционно занимали второстепенное место, а на первый план выступили задачи обработки и манипулирования данными.

Стало очевидным, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов прошлого столетия возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Такой методологией стало *объектно-ориентированное программирование* (ООП).

Фундаментальными понятиями ООП являются понятия класса и объекта. При этом под *классом* понимают некоторую абстракцию совокупности объектов, которые имеют общий набор свойств и обладают одинаковым поведением. Каждый *объект* в этом случае рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют полностью одинаковых

свойств или не обладают одинаковым поведением, по определению не могут быть отнесены к одному классу.

Примечание

Приведенное выше определение класса является общим. В последующих главах по мере изучения материала этот термин будет уточняться на основе установления семантических связей с другими понятиями объектно-ориентированного анализа и проектирования.

Важной особенностью классов является возможность их организации в виде некоторой *иерархической* структуры, которая по внешнему виду напоминает схему классификации понятий формальной логики. В этой связи следует заметить, что каждое понятие в логике имеет некоторый объем и содержание. При этом под *объемом* понятия понимают все другие мыслимые понятия, для которых исходное понятие может служить определяющей категорией или главной частью. *Содержание* понятия составляет совокупность всех его признаков или атрибутов, отличающих данное понятие от всех других. В формальной логике имеет место закон обратного отношения: если содержание понятия *A* содержится в содержании понятия *B*, то объем понятия *B* содержится в объеме понятия *A*.

Иерархия понятий строится следующим образом. В качестве наиболее общего понятия или категории берется понятие, имеющее наибольший объем и, соответственно, наименьшее содержание — это самый высокий уровень абстракции для данной иерархии. Затем данное общее понятие некоторым образом конкретизируется, тем самым уменьшается его объем и увеличивается содержание. Появляется менее общее понятие, которое на схеме иерархии будет расположено на уровень ниже исходного понятия. Этот процесс конкретизации понятий может быть продолжен до тех пор, пока на самом нижнем уровне не будет получено понятие, дальнейшая конкретизация которого в данном контексте либо невозможна, либо нецелесообразна.

Примерами наиболее общих понятий могут служить такие абстрактные категории, как система, структура, интеллект, информация, сущность, связь, состояние, событие и многие другие. В процессе изучения этих категорий появляются новые особенности их содержания и объема. Именно по этим причинам всегда трудно дать им точное определение. В качестве примеров конкретных понятий можно привести понятие книги, которую читатель держит в руках, или понятие микропроцессора Intel Pentium IV.

Примечание

Как будет видно из дальнейшего изложения, иерархическая схема организации понятий во многом похожа на рассматриваемую далее иерархию классов, но не тождественна ей. В общем случае взаимосвязи между классами могут иметь и другие качественные особенности. С другой стороны, иерархия понятий является более общей категорией по сравнению с иерархией уровней абстракции классов ООП.

Основными принципами ООП являются наследование, инкапсуляция и полиморфизм. Первый принцип, в соответствии с которым знание о более общей категории разрешается применять для более частной категории, называется *наследованием*. Наследование тесно связано с иерархией классов, которая определяет, какие классы следует считать наиболее абстрактными и общими по отношению к другим классам. При этом, если некоторый общий или родительский класс (предок) обладает фиксированным набором свойств и поведением, то производный от него класс (потомок) должен содержать этот же набор свойств и поведение, а также дополнительные, которые будут характеризовать уникальность полученного таким образом класса. В этом случае говорят, что производный класс наследует свойства и поведение родительского класса.

Для иллюстрации принципа наследования можно привести следующий пример. Рассмотрим в качестве общего класс "Автомобиль". Этот класс определяется как некоторая абстракция свойств и поведения всех реально существующих автомобилей. При этом свойствами класса "Автомобиль" могут быть такие общие свойства, как наличие двигателя, трансмиссии, колес и рулевого управления. Если в качестве производного класса рассмотреть класс "Легковой автомобиль", то все выделенные выше свойства будут присущи и этому классу. Можно сказать, что класс "Легковой автомобиль" наследует свойства родительского класса "Автомобиль". Однако кроме перечисленных свойств класс-потомок будет содержать дополнительные свойства, например, такое как наличие салона с количеством посадочных мест 2—5.

В свою очередь, класс "Легковой автомобиль" может быть классом-предком для других классов, которые вполне могут соответствовать, например, моделям конкретных фирм-производителей. С этой точки зрения можно рассматривать класс "Легковой автомобиль производства ВАЗ". Поскольку Волжский автомобильный завод выпускает несколько моделей автомобилей, одним из производных классов для предыдущего класса может быть конкретная модель автомобиля, например, ВАЗ-2110.

В этом контексте конкретный автомобиль, изготовленный на Волжском автомобильный заводе, имеет свои особенности, включая уникальный заводской номер, который отличает один автомобиль от другого. В этом случае автомобиль с идентификационным номером, например, XTA-211000V0001294, будет представлять собой один из объектов или экземпляров класса "Модель ВАЗ-2110".

Описанная выше информация о соотношении классов в нашем примере обладает одним серьезным недостатком, а именно, отсутствием наглядности. В этой связи возникает вопрос: а возможно ли представить иерархию наследования классов в визуальной форме? Традиционно для изображения понятий в формальной логике использовались окружности или прямоугольники. Используя эту графическую нотацию, иерархия классов для рассмотренного

примера может быть представлена в виде вложенных прямоугольников, каждый из которых соответствует отдельному классу (рис. 1.2).

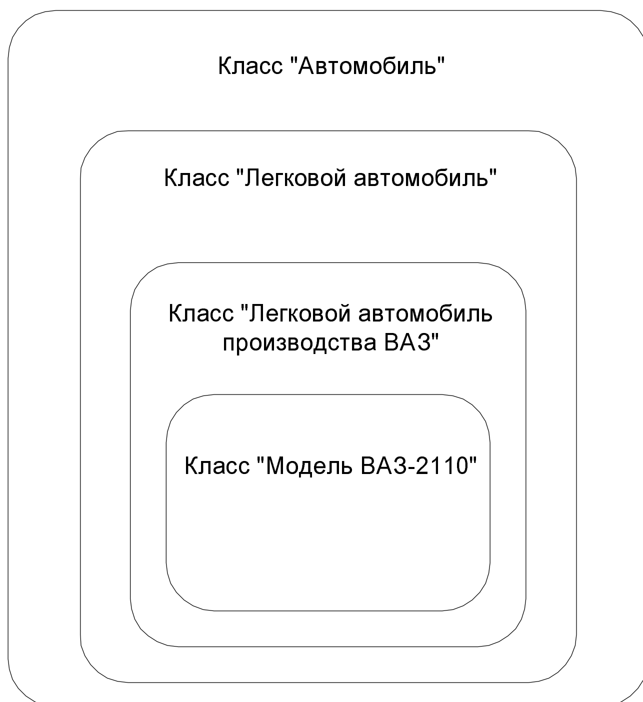


Рис. 1.2. Иерархия вложенности классов для примера "Автомобиль"

Появление объектно-ориентированных языков программирования было связано с необходимостью реализации концепции классов и объектов на синтаксическом уровне. С точки зрения ООП, класс является дальнейшим расширением *структуры* (structure) или *записи* (record). Включение в известные языки программирования C и Pascal классов и некоторых других возможностей привело к появлению, соответственно, C++ и Object Pascal, которые, на сегодня, являются наиболее распространенными языками разработки приложений. Распространению C++ и Object Pascal способствовало то обстоятельство, что язык C++ был выбран в качестве базового для программного инструментария MS Visual C++, а язык Object Pascal — для популярного средства быстрой разработки приложений Borland Delphi.

За короткий период оба инструментария превратились в мощные системы разработки программ с соответствующими библиотеками стандартных классов, содержащих сотни различных свойств и методов. Применительно к среде MS Visual C++ 6/.NET такая библиотека имеет специальное название — MFC (Microsoft Foundation Classes), т. е. фундаментальные классы

от Microsoft. При этом производные классы наследуют свойства и методы родительских классов. Ниже приводится фрагмент иерархии классов MFC в том виде, как он изображен в соответствующей документации (рис. 1.3).

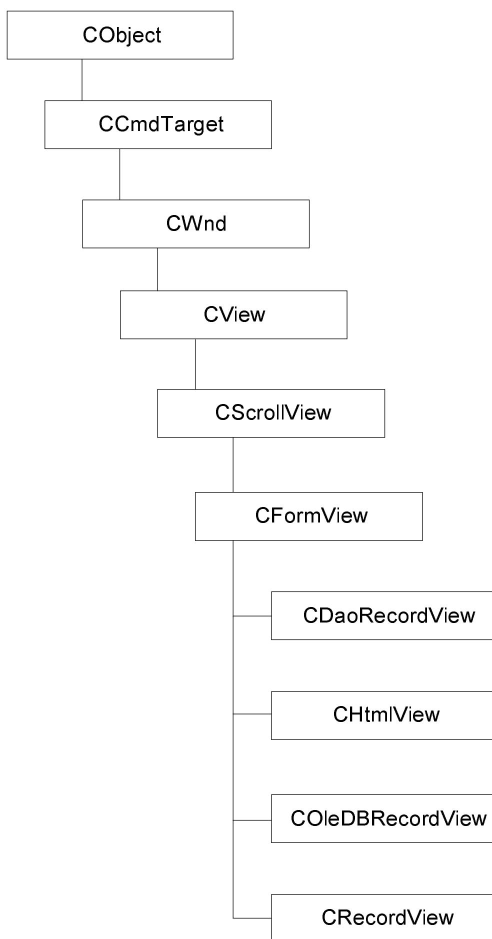


Рис. 1.3. Фрагмент иерархии классов MFC, используемых в среде программирования MS Visual C++ 6/.NET

Процесс разработки программ в среде Borland Delphi также тесно связан с использованием библиотеки стандартных классов — VCL (Visual Component Library) или библиотеки визуальных компонентов. Эта библиотека тоже построена по иерархическому принципу, в соответствии с которым компоненты нижележащих уровней наследуют свойства и методы вышележащих компонентов. Для этого случая также приводится фрагмент иерархии классов VCL (рис. 1.4).

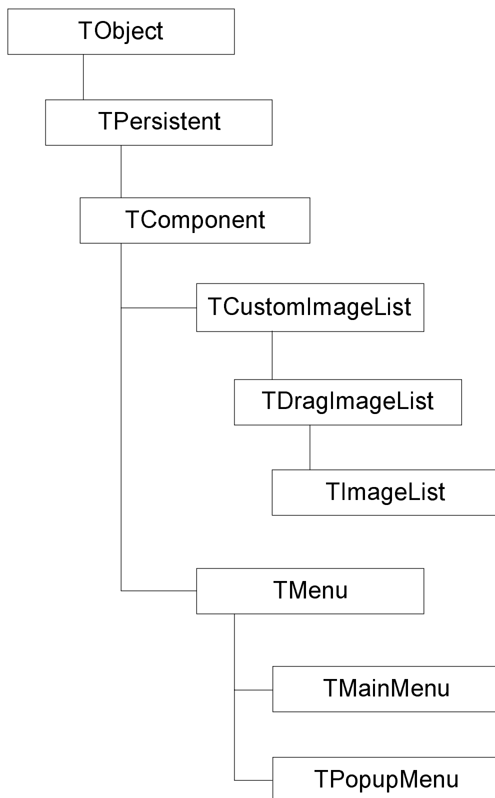


Рис. 1.4. Фрагмент иерархии классов VCL, используемых в среде программирования Borland Delphi 6/7

Даже этих простых примеров достаточно, чтобы понять следующий факт. А именно то, что для одной и той же общей концепции иерархии классов используются совершенно различные графические средства. В первом случае — вложенные прямоугольники, во втором — связные прямоугольники. В действительности, различных способов изображения классов предложено гораздо больше, небольшая часть которых будет рассмотрена далее. Однако уже сейчас важно осознать, что подобную ситуацию следовало бы унифицировать, т. е. использовать для этой цели некоторую единую систему обозначений.

Следующий принцип ООП — *инкапсуляция*. Этот термин характеризует сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей. Действительно, взаимодействующему с классом клиенту нет необходимости знать, каким образом реализован тот или иной метод класса, услугами которого он решил воспользоваться. Конкретная реализация присущих классу свойств и методов, которые определяют поведение этого класса, является собственным делом данного класса. Более того, отдельные свойства и методы класса вообще

могут быть невидимы за пределами этого класса, что является базовой идеей введения различных категорий видимости для компонентов класса.

Если продолжить рассмотрение примера с классом "Легковой автомобиль", то нетрудно проиллюстрировать инкапсуляцию следующим образом. Основным субъектом, который взаимодействует с объектами этого класса, является водитель. Вполне очевидно, что не каждый водитель в совершенстве знает внутреннее устройство легкового автомобиля. Более того, отдельные детали этого устройства сознательно скрыты в корпусе двигателя или коробке передач. А в случае нарушения работы автомобиля, являющейся причиной неадекватности его поведения, необходимый ремонт выполняет профессиональный механик.

Инкапсуляция ведет свое происхождение от деления модулей в некоторых языках программирования на две части (или секции): интерфейс и реализацию. При этом в интерфейсной секции модуля описываются все объявления функций и процедур, а возможно и типов данных, доступных за пределами данного модуля. Другими словами, указанные процедуры и функции являются способами оказания услуг внешним клиентам. В другой секции модуля, называемой реализацией, содержится программный код, который определяет конкретные способы реализации объявленных в интерфейсной части процедур и функций.

Принцип разделения модуля на интерфейс и реализацию отражает суть наших представлений об окружающем мире. В интерфейсной части указывается вся информация, необходимая для взаимодействия с любыми другими объектами. Реализация скрывает или маскирует от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов (рис. 1.5).

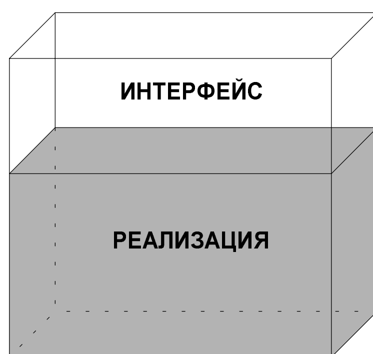


Рис. 1.5. Иллюстрация сокрытия внутренних деталей структуры классов

Примечание

Степень затемнения фона на приведенном выше рисунке имеет более глубокий смысл, чем может показаться на первый взгляд. Если ассоциировать реа-

> лизацию программного модуля с водой в аквариуме, то видимость объектов, находящихся в воде, будет зависеть от степени ее чистоты или загрязнения. В ООП существуют различные варианты доступа к свойствам и методам классов, которые получили название *кванторов видимости* свойств и методов. В этом случае использование различных форм видимости для компонентов классов удобно ассоциировать с прозрачностью фона рисунка или видимостью в воде аквариума. Более детальное рассмотрение различных форм видимости приводится в *части II* книги.

Третьим принципом ООП является полиморфизм. Под *полиморфизмом* (греч. poly — много, morfos — форма) понимают свойство некоторых объектов принимать различные внешние формы в зависимости от обстоятельств. Применительно к ООП полиморфизм означает, что действия, выполняемые одноименными методами, могут отличаться в зависимости от того, к какому из классов относится тот или иной метод.

Рассмотрим, например, три объекта соответствующих классов: двигатель автомобиля, электрический свет в комнате и персональный компьютер. Для каждого из них можно определить операцию `выключить()`. Однако результат выполнения этой операции будет отличаться для каждого из рассмотренных объектов. Так, для двигателя автомобиля вызов метода `двигательАвтомобиля.выключить()` означает прекращение подачи топлива и его остановку. Вызов метода `комната.электрическийСвет.выключить()` означает простой щелчок выключателя, после чего комната должна погрузиться в темноту. В последнем случае действие `персональныйКомпьютер.выключить()` может быть причиной потери данных, если выполняется нерегламентированным образом.

Примечание

> В рассмотренном выше примере использовалась одна из принятых нотаций в некоторых языках программирования (например, в Object Pascal и C++) для обозначения принадлежности метода тому или иному классу. В соответствии с этой нотацией, сначала указывается имя класса, в котором определен метод, а затем через точку имя самого метода. Если метод определен в некотором подклассе, то должна быть указана вся цепочка классов, начиная с наиболее общего из них. При этом характерным признаком метода является пара скобок, которые используются для указания списка аргументов или формальных параметров данного метода.

Для операции `выключить()`, рассматриваемой в том или ином контексте, можно определить дополнительные параметры, такие как время выключения, некоторое условие нахождения объекта в предварительно включенном состоянии и прочее. С этой целью после имени операции указываются скобки, в которых могут быть перечислены эти дополнительные параметры или аргументы. В случае отсутствия аргументов считается, что список параметров пуст. Однако скобки все равно записываются и указывают на тот факт, что соответствующее имя является именем операции или метода, в отличие от свойств или атрибутов класса, которые записываются без скобок.

Полиморфизм объектно-ориентированных языков связан с *перегрузкой* функций, но не тождествен ей. Важно иметь в виду, что имена методов и свойств тесно связаны с классами, в которых они описаны. Это обстоятельство обеспечивает определенную надежность работы программы, поскольку исключает случайное применение метода для решения несвойственной ему задачи.

Широкое распространение методологии ООП оказало влияние на процесс разработки программ. В частности, процедурно-ориентированная декомпозиция программ уступила место *объектно-ориентированной декомпозиции*, при которой отдельными структурными единицами программы стали являться не процедуры и функции, а классы и объекты с соответствующими свойствами и методами. Как следствие, программа перестала быть последовательностью предопределенных на этапе кодирования действий, а стала *событийно управляемой*. Последнее обстоятельство доминирует и при разработке широкого круга современных приложений. В этом случае каждая программа представляет собой бесконечный цикл ожидания некоторых заранее определенных событий. Инициаторами событий могут быть другие программы или пользователи. При наступлении отдельного события, например, нажатия клавиши на клавиатуре или щелчка кнопкой мыши программа выходит из состояния ожидания и реагирует на это событие вполне адекватным образом. Реакция программы при этом тоже связывается с последующими событиями.

Наиболее существенным обстоятельством в развитии методологии ООП явилось осознание того факта, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Действительно, до того, как начать программирование классов, их свойств и методов, необходимо определить, чем же являются сами эти классы. Более того, нужно дать ответы на такие вопросы, как: сколько и какие классы нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания классам требуемого поведения, а также установить взаимосвязи между классами.

Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной предметной области, для которой разрабатывается программа. Все эти обстоятельства привели к появлению специальной методологии, получившей название методологии объектно-ориентированного анализа и проектирования (ООАП).

1.3. Методология объектно-ориентированного анализа и проектирования

Необходимость анализа предметной области до начала написания программы была осознана давно при разработке масштабных проектов. Процесс разработки баз данных существенно отличается от написания программного

кода для решения вычислительной задачи. Главное отличие заключается в том, что при проектировании базы данных возникает необходимость в предварительной разработке *концептуальной схемы* или *модели*, которая отражала бы общие взаимосвязи предметной области и особенности организации соответствующей информации. При этом под *предметной областью* принято понимать ту часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы. Другими словами, предметная область включает в себя только те объекты и взаимосвязи между ними, которые необходимы для описания требований и условий решения некоторой задачи.

Выделение исходных или базовых компонентов предметной области, необходимых для решения той или иной задачи, представляет, в общем случае, нетривиальную задачу. Сложность проявляется в неформальном характере процедур или правил, которые можно применять для этой цели. Более того, эта работа должна выполняться совместно со специалистами или экспертами, хорошо знающими предметную область. Например, если разрабатывается база данных для обслуживания пассажиров крупного аэропорта, то в проектировании концептуальной схемы базы данных должны принимать участие штатные сотрудники данного аэропорта. Эти сотрудники должны хорошо знать весь процесс обслуживания пассажиров или данную предметную область.

Для выделения или идентификации компонентов предметной области было предложено несколько способов и правил. Сам этот процесс получил название *концептуализации* предметной области. На предварительном этапе концептуализации конструктивную помощь могут оказать, так называемые, CRC-карточки (Component, Responsibility, Collaborator — компонента, обязанность, сотрудники). При этом под *компонентой* понимают некоторую абстрактную единицу, которая обладает функциональностью, т. е. может выполнять определенные действия, связанные с решением поставленных задач. Для каждой выделенной компоненты предметной области разрабатывается собственная CRC-карточка (рис. 1.6).

КОМПОНЕНТ (название)	СПИСОК
Описание обязанностей, выполняемых данным компонентом	всех взаимодействующих с ним компонентов

Рис. 1.6. Общий вид CRC-карточки для описания компонентов предметной области

Построение концептуальной модели предметной области с помощью CRC-карточек предполагает выявление и последующую конкретизацию всех компонентов, которые оказывают существенное влияние на решение поставленной задачи. При этом каждый из компонентов служит прототипом некоторого класса при программной реализации соответствующего проекта. Другими словами, отдельные компоненты выбираются таким образом, чтобы при последующей работе над проектом их было удобно представить в форме классов. В этом случае немаловажное значение приобретает и сам способ представления информации о концептуальной схеме предметной области.

Появление методологии ООАП потребовало, с одной стороны, разработки различных средств концептуализации предметной области, а с другой стороны, соответствующих специалистов, которые владели бы этой методологией. Именно на этом этапе появляется необходимость в относительно новом типе специалиста, который получил название *аналитика* или *архитектора*. Наряду со специалистами по предметной области аналитик участвует в построении концептуальной схемы будущей программы, которая затем преобразуется программистами в код.

Разделение процесса разработки сложных программных приложений на отдельные этапы способствовало становлению концепции жизненного цикла программы. Под *жизненным циклом* (ЖЦ) программы понимают совокупность взаимосвязанных и следующих во времени этапов, начиная от разработки требований к ней и заканчивая полным отказом от ее использования. Стандарт ISO/IEC 12207, хотя и описывает общую структуру ЖЦ программы, но не конкретизирует детали выполнения тех или иных этапов. Согласно принятым взглядам ЖЦ программы состоит из следующих этапов:

- анализ предметной области и формулировка требований к программе;
- проектирование структуры программы;
- реализация программы в кодах (собственно программирование);
- внедрение программы;
- сопровождение программы;
- отказ от использования программы.

На этапе анализа предметной области и формулировки требований осуществляется определение функций, которые должна выполнять разрабатываемая программа, а также концептуализация предметной области. Эту работу выполняют аналитики совместно со специалистами предметной области. Результатом является некоторая концептуальная схема, содержащая описание основных компонентов и тех функций, которые они должны выполнять.

Этап проектирования структуры программы заключается в разработке детальной схемы, на которой указываются классы, их свойства и методы, а также различные взаимосвязи между ними. Как правило, на этом этапе могут участвовать в работе аналитики, архитекторы и отдельные квалифи-