Т. Бадд.

# Объектно-ориентированное программирование.

PDF created with pdfFactory Pro trial version www.pdffactory.com

# Введение.

I. **Введение и общий замысел.** Глава 1 дает неформальное определение базовых концепций объектно-ориентированного программирования. Глава 2 вводит принцип *разработки на основе обязанностей*. Эти две главы являются фундаментальными, и их следует изучить подробно. В частности, я настоятельно рекомендую выполнить по крайней мере одно упражнение с CRC-карточками из главы 2. Техника CRC-карточек, по моему мнению, является одной из лучших для определения функциональности, ответственности и инкапсуляции при базовой разработке проекта.

II. **Классы, методы и сообщения** Главы 3 и 4 определяют синтаксис, используемый в языках Smalltalk, C++, Java, Objective-C и Object Pascal для задания классов, методов и посылки сообщений. Глава 3 заостряет внимание на статических свойствах (классах и методах), в то время как глава 4 описывает динамические аспекты (создание объектов и пересылку сообщений). Главы 5 и 6 развивают эти идеи. Здесь же начинаются обучающие примеры — образцы программ, разработанных в объектно-ориентированной манере и иллюстрирующих различные черты объектной техники.

III. **Наследование и повторное использование кода** Главы 7, 8 и 9 вводят концепцию наследования и объясняют ее применение для обеспечения повторного использования кода. Пример из главы 8, написанный на языке Java, иллюстрирует также применение стандартного прикладного программного интерфейса (API — application program interface). В главе 9 противопоставляются наследование и композиция в качестве альтернативных техник обеспечения повторного использования кода.

IV. **Более подробно о наследовании.** В главах с 10 по 13 концепция наследования анализируется более детально. Введение наследования оказывает влияние на почти все аспекты языка программирования, которое зачастую не сразу очевидно для начинающего. В главе 10 обсуждается поиск методов и их связывание с сообщениями. Там же иллюстрируется тот факт, что подклассы и подтипы — это не одно и то же. В главе 11 обсуждается семантика переопределения методов и отмечаются две совершенно различные интерпретации этого понятия. В главе 12 продолжается тема переопределения и исследуются некоторые следствия наследования применительно к механизмам управления памятью, присваивания и сравнения. Наконец, в главе 13 изучается множественное наследование.

V. **Полиморфизм.** В значительной степени мощь объектно-ориентированного программирования проистекает из применения различных форм полиморфизма. В главе 14 читатель знакомится с основными механизмами полиморфизма в объектно-ориентированных языках и двумя показательными обучающими примерами. Первый пример в главе 15 рассматривает создание библиотек общего назначения. Конкретная библиотека, а именно недавно разработанная стандартная библиотека шаблонов (STL — Standard Template Library) для языка C++, обсуждается в главе 16.

VI. **Разработка программного обеспечения.** В главе 17 обсуждается ряд стандартных тем компьютерной инженерии в контексте объектно-ориентированного программирования. Глава 18 знакомит с несколькими относительно новыми концепциями — *средой разработки приложений и шаблонами разработки*. Оба подхода основаны на использовании наборов классов. Наконец, в главе 19 приводится конкретный пример среды разработки.

VII. **Продвинутое изучение.**Концепция классов при внимательном рассмотрении не столь проста, как нас пытались убедить в главе 3. В главе 20 рассмотрены более глубокие аспекты объектно-ориентированного программирования. Там же

PDF created with pdfFactory Pro trial version www.pdffactory.com

обсуждаются делегирование (являющееся примером объектно-ориентированного программирования без классов) и понятие метакласса (на уровне собственно языка программирования). В главе 21 в общих чертах описаны разнообразные техники реализации, применяющиеся при создании объектно-ориентированных языков.

В десятинедельном курсе, который я читаю в университете штата Орегон, приблизительно одну неделю я посвящаю каждому из основных направлений, описанных выше. В то же самое время студенты работают над не слишком большим проектом. Конкретный объектно-ориентированный язык разработки они выбирают сами. Семестр заканчивается представлением дизайна проекта и его реализацией.

Первое издание книги я закончил главой «Дополнительная информация». К сожалению, объектно-ориентированное программирование развивается так быстро, что любая дополнительная информация почти сразу устаревает. Поэтому я не включил во второе издание главу с таким названием. Вместо этого я попытаюсь поддерживать страничку Web с последними сведениями.

### *Как получить исходные тексты*

Исходные тексты обучающих примеров, представленных в книге, можно получить анонимно, обратившись через ftp по адресу ftp.cs.orst.edu, каталог /pub/budd/oopintro. В том же каталоге можно будет найти дополнительную информацию, например список ошибок, обнаруженных в книге, упражнения, копии «прозрачек», которые я использую в своем курсе. Все это можно также увидеть через World Wide Web на моих личных домашних страницах по адресу http://www.cs.orst.edu/~budd/oopintro. Вопросы вы можете посылать электронной почтой по адресу budd@cs.orst.edu или обычной почтой: Professor Timothy A. Budd, Department of Computer Science, Oregon State University, Corvallis, Oregon, 97331.

### *Что требуется знать для чтения книги*

Я предполагаю, что читатель знаком хотя бы с одним традиционным языком программирования, например Pascal или C. Мои курсы были вполне успешно восприняты студентами последнего года undegraduate level и первого graduate level. В некоторых случаях (особенно в последней четверти книги) более глубокие знания окажутся полезны, но они не являются обязательными. Например, студент, который специализируется на разработке программного обеспечения, легче воспримет материал главы 17, а обучающийся построению компиляторов сочтет главу 21 вполне понятной. Тематику обеих глав можно упростить при необходимости.

## *Глава 1 : Объектно- ориентированное мышление*

Объектно-ориентированное программирование (ООП) стало чрезвычайно популярно в последние несколько лет. Производители программного обеспечения бросаются создавать объектно-ориентированные версии своих продуктов. Появилось несчетное количество книг и специальных выпусков академических (и не только) журналов, посвященных этому предмету. Студенты стремятся к записи «компетентен в объектно-ориентированном программировании» в своих характеристиках. Чтобы оценить эту безумную активность, отметим, что объектно-ориентированное программирование приветствуется с б'ольшим энтузиазмом, чем тот, который мы видели ранее при провозглашении таких революционных идей, как «структурное программирование» или «экспертные системы».

PDF created with pdfFactory Pro trial version www.pdffactory.com

Моя цель в первой главе состоит в том, чтобы исследовать и объяснить основные принципы объектно-ориентированного программирования, а также проиллюстрировать следующие утверждения.

- ООП — это революционная идея, совершенно непохожая на что-либо выдвигавшееся в программировании.
- ООП — это эволюционный шаг, естественным образом вытекающий из предшествующей истории.

## 1.1. Почему ООП так популярно?

Я перечислю некоторые (на мой взгляд — самые главные) причины огромной популярности объектно-ориентированного программирования в последнее десятилетие:

- надежда, что ООП может просто и быстро привести к росту продуктивности и улучшению надежности программ, помогая тем самым разрешить кризис в программном обеспечении;
- желание перейти от существующих языков программирования к новой технологии;
- вдохновляющее сходство с идеями, родившимися в других областях.

Объектно-ориентированное программирование является лишь последним звеном в длинной цепи решений, которые были предложены для разрешения «кризиса программного обеспечения». Положа руку на сердце: кризис программного обеспечения просто означает, что наше воображение и те задачи, которые мы хотим решить с помощью компьютеров, почти всегда опережают наши возможности.

Несмотря на то что объектно-ориентированное программирование действительно помогает при создании сложных программных систем, важно помнить, что ООП не является «серебряной пулей» (термин, ставший популярным благодаря Фреду Бруксу [Brooks 1987]), которая запросто справляется с чудовищем. Программирование по-прежнему является одной из наиболее трудных задач, взваливаемых на себя человеком. Чтобы стать профессионалом в программировании, необходимы талант, способность к творчеству, интеллект, знания, логика, умение строить и использовать абстракции и, самое главное, опыт — даже в том случае, когда используются лучшие средства разработки.

Я подозреваю, что есть и другая причина особой популярности таких языков программирования, как C++ и Object Pascal (по контрасту со Smalltalk и Beta). Она состоит в том, что и администрация и разработчики надеются, что программист на языках C или Pascal может перейти на C++ или Object Pascal с той же легкостью, с которой происходит добавление нескольких букв на титульный лист сертификата о специальности. К сожалению, так происходит не всегда. Объектно-ориентированное программирование является новым пониманием того, что собственно называется вычислениями, а также того, как мы можем структурировать информацию внутри компьютера. Чтобы стать профессионалом в технике ООП, требуется полная переоценка привычных методов разработки программ.

## 1.2. Язык и мышление

*Человеческие существа не общаются непосредственно с объективным миром и с обществом в том смысле, как это обычно понимается. Они в значительной мере зависят от того конкретного языка, который стал их средой общения. Это совершенная*

PDF created with pdfFactory Pro trial version www.pdffactory.com

*иллюзия — полагать, что кто-то может согласовать себя с сущностью реальности без использования языка и что язык — всего лишь случайное средство решения конкретных задач общения или мышления. Суть вопроса в том, что «реальный мир» в значительной степени неосознанно строится на языковых привычках группы людей... Мы видим, слышим и испытываем остальные ощущения так, как мы это делаем, в значительной степени потому, что языковые обычаи нашего общества предрасполагают к определенному выбору способа интерпретации.*

Эдвард Сапир (цитировано по [Whorf 1956]).

Цитата подчеркивает тот факт, что язык, на котором мы говорим, непосредственно влияет на способ восприятия мира. Это справедливо не только для естественных языков, подобных тем, что изучались в начале двадцатого века американскими лингвистами Эдвардом Сапиром и Ли Ворфом, но также и для искусственных языков, наподобие тех, что мы используем в программировании.

### 1.2.1. Эскимосы и снег

Примером, почти повсеместно цитируемым (хотя зачастую ошибочно — см. [Pillum 1991]) в качестве иллюстрации того, как язык влияет на мышление, является тот «факт», что в эскимосских (или юитских) языках имеется множество слов для описания различных типов снежного покрова — мокрого, плотного, подмерзшего и т. д. Это-то как раз не является удивительным. Любое сообщество с общими интересами естественным образом разрабатывает специализированный словарь необходимых понятий.

Что действительно важно — не слишком абсолютизировать вывод, который мы можем сделать из этого простого наблюдения. Главное не в том, что глаз эскимосов в каком-то существенном аспекте отличается от моего собственного или что эскимосы могут видеть вещи, которые я не способен различать. С течением времени, с помощью тренировки, я бы стал ничуть не хуже различать разнообразные типы снежного покрова. Однако язык, на котором я говорю (а именно английский), не вынуждает меня заниматься этим, и тем самым указанные способности не являются для меня естественными.

Таким образом, различные языки (например, эвенкийский) могут привести (но не обязательно требуют этого) к тому, чтобы смотреть на мир с разных сторон.

Чтобы эффективно использовать ООП, требуется глядеть на мир иным способом. Само по себе применение объектно-ориентированного языка программирования (такого, как C++) не вынуждает стать объектно-ориентированным программистом. Использование объектно-ориентированного языка упрощает разработку объектно-ориентированных приложений, но, как было остроумно замечено, «программа фортрановского типа может быть написана на любом языке».

### 1.2.2.Пример из области программирования

Связь между языком и мышлением для естественных языков, о которой мы говорили, является еще более заметной для искусственных компьютерных языков. Язык программирования, в терминах которого разработчик думает о проблеме, вносит особые оттенки и, вообще говоря, изменяет даже сам алгоритм.

Приведем пример, иллюстрирующий связь между компьютерным языком и способом решения задачи. Некоторое время назад один студент, работающий в области

PDF created with pdfFactory Pro trial version www.pdffactory.com

генетических исследований, столкнулся с необходимостью анализа последовательностей ДНК. Проблема могла быть сведена к относительно простой задаче. Молекула ДНК представляется в виде вектора из N целочисленных значений, где N очень велико (порядка десятков тысяч). Нужно было проверить, не является ли какой-либо участок длины M (M — фиксированная константа порядка 5–10) повторяющимся в последовательности ДНК.

ACTCGGATCTTGCATTTCGGCAATTGGACCCTGACTTGGCCA...

Программист, не долго думая, написал простую и прямолинейную программу на Fortran — нечто вроде

```
        DO 10 I = 1, N-M
        DO 10 J = 1, N-M
        FOUND=.TRUE.
        DO 20 K = 1, M
20      IF (X(I+K-1).NE.X(J+K-1)) FOUND=.FALSE.
        IF (FOUND) ...
10      CONTINUE
```

Он был неприятно разочарован, когда пробные запуски программы показали, что она потребует многих часов для завершения работы. Студент обсудил эту проблему со студенткой, которая оказалась профессионалом в программировании на языке APL. Она сказала, что могла бы попробовать написать программу для решения этой задачи. Студент был в сомнении: Fortran известен как один из наиболее «эффективных» компилируемых языков, а APL реализовывался с помощью интерпретатора. Таким образом, тот факт, что APL-программист способен составить алгоритм, который требует для работы минуты, а не часы, был воспринят с определенной дозой недоверия.

APL-программистка переформулировала задачу. Вместо того чтобы работать с вектором из N элементов, она представила данные в виде матрицы, имеющей приблизительно N строк и M столбцов:

```
A  C  T  C  G  G  позиции 1 — M
C  T  C  G  G  A  позиции 2 — M+1
T  C  G  G  A  T  позиции 3 — M+2
C  G  G  A  T  T  позиции 4 — M+3
G  G  A  T  T  C  позиции 5 — M+4
G  A  T  T  C  T  позиции 6 — M+5
  .  .  .
T  G  G  A  C  C
G  G  A  C  C  C
```

Затем студентка отсортировала матрицу по строкам. Если какой-то фрагмент оказывается повторяющимся, то в отсортированной матрице две соседние строки должны оказаться идентичными.

```
       .     .     .
T      G     G     A     C     C
T      G     G     A     C     C
       .     .     .
```

Проверка этого условия оказывается тривиальной задачей. Причина, по которой APL-программа оказалась быстрее, не имела ничего общего со скоростью работы APL по сравнению с Fortran. Главным было то, что программа на Fortran использовала алгоритм со сложностью O(M r N 2), в то время как алгоритм сортировки APL-программы требовал примерно O(M r N log N) операций.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Ключевой момент этой истории не в том, что APL является лучшим языком программирования, чем Fortran, но в том, что APL-программист естественным образом пришел к более удачному решению. В частности, из-за того, что на языке APL очень неудобно организовывать циклы, а сортировка является тривиальной операцией — ей соответствует встроенный оператор языка. Таким образом, раз уж сортировку можно столь легко использовать, хороший APL-программист всегда старается найти для нее новое применение. В этом смысле язык программирования, на котором записывается решение задачи, напрямую влияет на ход мыслей программиста, заставляя его рассматривать задачу под определенным углом.

### 1.2.3. Принцип Чёрча и гипотеза Ворфа

Легко поверить в утверждение, что язык, на котором высказывается идея, направляет мышление. Однако есть более сильное утверждение, известное среди лингвистов как гипотеза Сапира–Ворфа. Она идет еще дальше, хотя и является спорной [Pullum 1991].

Гипотеза Сапира–Ворфа утверждает, что индивидуум, использующий некоторый язык, в состоянии вообразить или придумать нечто, не могущее быть переведенным или даже понятым индивидуумами из другой языковой среды. Такое происходит, если в языке второго индивидуума нет эквивалентных слов и отсутствуют концепции или категории для идей, вовлеченных в рассматриваемую мысль. Интересно сравнить данную идею с прямо противоположной концепцией в информатике — а именно принципом Чёрча.
В 30-х годах у математиков пробудился большой интерес к различным формализмам, которые могут быть использованы при вычислениях. Эти идеи получили развитие в 40–50-х годах, когда они привлекли внимание молодого сообщества специалистов по информатике. Примерами таких систем являются модели, предложенные Чёрчем [Church 1936], Постом [Post 1936], Марковым [Markov 1951], Тьюрингом [Turing 1936], Клини [Kleene 1936] и другими. В одно время приводилось множество аргументов, доказывающих, что каждая из этих систем может быть использована для моделирования остальных. Часто такие доводы были двухсторонними, показывая, что обе модели эквивалентны с некой общей точки зрения. Все это привело логику Алонзо Чёрча к гипотезе, которая теперь связана с его именем.

**Принцип Чёрча**: Любое вычисление, для которого существует эффективная процедура, может быть реализовано на машине Тьюринга.

По самой своей природе это утверждение недоказуемо, поскольку мы не имеем строгого определения термина «эффективная процедура». Тем не менее до сих пор не было найдено контрпримера, и убедительность очевидности, по-видимому, благоприятствует принятию этого утверждения [1].

---

[1] Создание математического формализма вычислимости было связано с необходимостью определить понятие алгоритма. Пока исследования в этой области шли успешно, каждая новая формализованная последовательность вычислений получала имя «алгоритм» просто по определению. Когда же математики столкнулись с задачами, для которых пришлось доказывать *отсутствие* алгоритма, потребовалось формальное определение. В настоящий момент *принято считать*, что алгоритмом является последовательность действий, которая может быть сведена к программе, выполняемой с помощью машины Тьюринга. Или, в эквивалентной форме: последовательность действий, которая может быть сведена к программе для машины Поста, или конечного автомата Маркова, или же к последовательности рекурсивных функций Клини и Чёрча, является алгоритмом. Доказано, что все эти формальные системы вычислимости являются эквивалентными. Тем самым принцип Чёрча является *аксиомой*, не требующей доказательства, которая формализует понятие алгоритма («эффективной процедуры») и в силу статуса аксиомы опровергающего контрпримера иметь не может. — *Примеч. перев.*

PDF created with pdfFactory Pro trial version www.pdffactory.com

Признание принципа Чёрча имеет важное и глубокое следствие для языков программирования. Машины Тьюринга являются изумительно простыми механизмами. От языка программирования требуется немного, чтобы смоделировать такое устройство. В 1960-х годах, к примеру, было показано, что машина Тьюринга может быть смоделирована на любом языке программирования, в котором содержатся условные операторы и операторы цикла [Bohm 1966]. Этот не совсем правильно понимаемый результат был одним из основных доводов в защиту утверждения о том, что знаменитый оператор goto является ненужным.

Если мы признаем принцип Чёрча, то любой язык, на котором можно смоделировать машину Тьюринга, является достаточно мощным, чтобы осуществить любой реализуемый алгоритм. Для решения проблемы надо построить машину Тьюринга, которая выдаст желаемый результат, — согласно принципу Чёрча такая машина должна существовать для каждого алгоритма. Затем остается только смоделировать машину Тьюринга на вашем любимом языке программирования. Тем самым споры об относительной «мощности» языков программирования — если под мощностью мы понимаем «способность решать задачи», — оказываются бессмысленными. Позднее Алан Перлис ввел удачный термин для подобных аргументов, назвав их «тьюринговская пропасть», поскольку из них так сложно выбраться, в то время как сами они столь фундаментально бесполезны.

Заметим, что принцип Чёрча является в определенном смысле точной противоположностью гипотезы Сапира–Ворфа. Принцип Чёрча утверждает, что по своей сути все языки программирования идентичны. Любая идея, которая выражается на одном языке, может (согласно теории) быть реализована на другом. Гипотеза же Сапира–Ворфа, как вы помните, утверждает, что существуют идеи, не согласующиеся с этим принципом.

Многие лингвисты отвергают гипотезу Сапира–Ворфа и вместо этого принимают «тьюринговский эквивалент» для естественных языков: любая идея в принципе может быть выражена на любом языке. Например, несмотря на то что язык людей, живущих в жарком климате, не содержит готовых понятий для типов снежного покрова, в принципе южане тоже могут стать специалистами в области гляциологии. Аналогично объектно-ориентированная техника не снабжает (в теории) вас новой вычислительной мощностью, которая позволила бы решить проблемы, недоступные для других средств. Но объектно-ориентированный подход делает задачу проще и приводит ее к более естественной форме. Это позволяет обращаться с проблемой таким образом, который благоприятствует управлению большими программными системами.

Итак, как для компьютерных, так и для естественных языков справедливо: язык направляет мысли, но не предписывает их.

## 1.3. Новая парадигма

Объектно-ориентированное программирование часто называют новой парадигмой программирования. Другие парадигмы программирования: директивное (языки типа Pascal или C), логическое (языки типа Prolog) и функциональное (языки типа Lisp, FP или Haskell) программирование.

Интересно исследовать слово «парадигма». Следующий фрагмент взят из толкового словаря *American Heritage Dictionary of the English Language*:

**par-a-digm** (сущ.) 1. Список всех вариантов окончаний слова, рассматриваемый как иллюстративный пример того, к какому спряжению или склонению оно относится. 2.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Любой пример или модель (от латинского paradigma и греческого paradeigma — модель, от paradeiknunai — сравнивать, выставлять).

На первый взгляд, склонение и спряжение слов (например, латинских) имеет мало общего с компьютерными языками. Чтобы понять связь, мы должны заметить, что слово «парадигма» пришло в программирование из оказавшей большое влияние книги «Структура научных революций», написанной историком науки Томасом Куном [Kuhn 1970]. Кун использовал этот термин во втором значении, чтобы описывать набор теорий, стандартов и методов, которые совместно представляют собой способ организации научного знания — иными словами, способ видения мира. Основное положение Куна состоит в том, что революции в науке происходят, когда старая парадигма пересматривается, отвергается и заменяется новой.

Именно в этом смысле — как модель или пример, а также как организующий подход — это слово использовал Роберт Флойд, лауреат премии Тьюринга 1979 года, в лекции «Парадигмы программирования» [Floyd 1979]. Парадигмы в программировании — это способ концептуализации, который определяет, как проводить вычисления и как работа, выполняемая компьютером, должна быть структурирована и организована.

Хотя сердцевина объектно-ориентированного программирования — техника организации вычислений и данных является новой, ее зарождение можно отнести по крайней мере к временам Линнея (1707–1778), если не Платона. Парадоксально, но стиль решения задач, воплощенный в объектно-ориентированной технике, нередко используется в повседневной жизни. Тем самым новички в информатике часто способны воспринять основные идеи объектно-ориентированного программирования сравнительно легко, в то время как люди, более осведомленные в информатике, зачастую становятся в тупик из-за своих представлений. К примеру, Алан Кей обнаружил, что легче обучать языку Smalltalk детей, чем профессиональных программистов [Kay 1977].

При попытках понять, что же в точности имеется в виду под термином *объектно-ориентированное программирование*, полезно посмотреть на ООП с разных точек зрения. В нескольких следующих разделах кратко очерчиваются три разных аспекта объектно-ориентированного программирования. Каждый из них по-своему объясняет, чем замечательна эта идея.

## 1.4. Способ видения мира

Чтобы проиллюстрировать некоторые основные идеи объектно-ориентированного программирования, рассмотрим ситуацию из обыденной жизни, а затем подумаем, как можно заставить компьютер наиболее близко смоделировать найденное решение.

Предположим, я хочу послать цветы своей бабушке (которую зовут Элси) в ее день рождения. Она живет в городе, расположенном за много миль от меня, так что вариант, когда я сам срываю цветы и кладу их к ее порогу, не подлежит обсуждению. Тем не менее послать ей цветы — это достаточно простая задача: я иду в ближайший цветочный магазин, хозяйку которого (какое совпадение) зовут Фло (florist — цветочница), называю ей тип и количество цветов, которые бы я хотел послать моей бабушке, и (за приемлемую цену) я могу быть уверен, что цветы будут доставлены в срок, по нужному адресу.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 1.4.1 Агенты, обязанности, сообщения и методы

Рискуя быть обвиненным в тавтологии, все-таки хочу подчеркнуть, что механизм, который я использовал для решения этой проблемы, состоял в поиске подходящего агента (а именно, Фло) и передаче ей сообщения, содержащего мой запрос. *Обязанностью* Фло является удовлетворение моего запроса. Имеется некоторый *метод* — то есть алгоритм, или последовательность операций, который используется Фло для выполнения запроса. Мне не надо знать, какой конкретный метод она использует для выполнения моего запроса, и в действительности зачастую я и не хочу это знать. Все дальнейшее обычно скрыто от моего взгляда.

Однако если бы я исследовал этот вопрос, я, возможно, обнаружил бы, что Фло пошлет свое сообщение хозяину цветочного магазина в городе, где живет моя бабушка. Хозяин цветочного магазина в свою очередь примет необходимые меры и подготовит распоряжение (сообщение) для человека, ответственного за доставку цветов, и т. д. Тем самым мой запрос в конечном счете будет удовлетворен через последовательность запросов, пересылаемых от одного агента к другому.

Итак, первым принципом объектно-ориентированного подхода к решению задач является способ задания действий.

Действие в объектно-ориентированном программировании инициируется посредством передачи сообщений агенту (объекту), ответственному за действие. Сообщение содержит запрос на осуществление действия и сопровождается дополнительной информацией (аргументами), необходимой для его выполнения. Получатель (receiver) — это агент, которому посылается сообщение. Если он принимает сообщение, то на него автоматически возлагается ответственность за выполнение указанного действия. В качестве реакции на сообщение получатель запустит некоторый метод, чтобы удовлетворить принятый запрос.

Мы заметили, что существует важный принцип маскировки информации в отношении пересылки сообщений. А именно: клиенту, посылающему запрос, не требуется знать о фактических средствах, с помощью которых его запрос будет удовлетворен. Существует и другой принцип, также вполне человеческий, который мы видели в неявной форме при пересылке сообщений. Если имеется работа, которую нужно выполнить, то первая мысль клиента — найти кого-либо еще, кому можно было бы ее поручить. Такая вполне нормальная реакция почти полностью атрофировалась у программиста, имеющего большой опыт в традиционном программировании. Ему трудно представить, что он (или она) не должен все полностью программировать сам, а может обратиться к услугам других. Важная часть объектно-ориентированного программирования — разработка повторно используемых компонент, и первым шагом в этом направлении является желание попробовать этот путь.

Скрытие информации является важным принципом и в традиционных языках программирования. Тогда в чем пересылка сообщений отличается от обычного вызова процедуры? В обоих случаях имеется последовательность точно определенных действий, которые будут инициированы в ответ на запрос. Однако имеются два существенных отличия.

Первое из них состоит в том, что у сообщения имеется вполне конкретный получатель — агент, которому послано сообщение. При вызове процедуры нет столь явно выделенного получателя. (Хотя, конечно, мы можем принять соглашение, согласно которому

PDF created with pdfFactory Pro trial version www.pdffactory.com

получателем сообщения является первый аргумент в вызове процедуры — примерно так и реализуются получатели сообщений).

Второе отличие состоит в том, что интерпретация сообщения (а именно метод, вызываемый после приема сообщения) зависит от получателя и является различной для различных получателей. Я могу передать мое сообщение, к примеру, моей жене Бет, и она его поймет, и как результат действие будет выполнено (а именно цветы будут доставлены бабушке). Однако метод, который использует Бет для выполнения запроса (весьма вероятно, просто переадресовав его хозяйке цветочного магазина Фло), будет иным, чем тот, который применит Фло в ответ на тот же самый запрос. Если я попрошу о том же Кена, моего зубного врача, у него может не оказаться подходящего метода для решения поставленной задачи. Если предположить, что Кен вообще воспримет этот запрос, то он с большой вероятностью выдаст надлежащее диагностическое сообщение об ошибке.

Вернемся в нашем обсуждении на уровень компьютеров и программ. Различие между вызовом процедуры и пересылкой сообщения состоит в том, что в последнем случае существует определенный получатель и интерпретация (то есть выбор подходящего метода, который запускается в ответ на сообщение) может быть различной для разных получателей. Обычно конкретный получатель неизвестен вплоть до выполнения программы, так что определить, какой метод будет вызван, заранее невозможно. В таком случае говорят, что имеет место позднее связывание между сообщением (именем процедуры или функции) и фрагментом кода (методом), используемым в ответ на сообщение. Эта ситуация противопоставляется раннему связыванию (на этапе компилирования или компоновки программы) имени с фрагментом кода, что происходит при традиционных вызовах процедур.

### 1.4.2. Обязанности и ответственности

Фундаментальной концепцией в объектно-ориентированном программировании является понятие обязанности или ответственности за выполнение действия. Мой запрос выражает только стремление получить желаемый результат (а именно доставить цветы бабушке). Хозяйка цветочного магазина свободна в выборе способа, который приведет к желаемому результату, и не испытывает препятствий с моей стороны в этом аспекте.

Обсуждая проблему в терминах обязанностей, мы увеличиваем уровень абстрагирования. Это позволяет иметь большую независимость между агентами — критический фактор при решении сложных задач. В главе 2 мы будем подробно исследовать, как можно использовать обязанности в разработке программного обеспечения. Полный набор обязанностей, связанных с определенным объектом, часто определяется с помощью термина протокол.

Различие между взглядом на программное обеспечение со стороны традиционного, структурного подхода и объектно-ориентированной точкой зрения на него может быть выражено в форме пародии на хорошо известную цитату:

Задавайтесь вопросом не о том, что вы можете сделать для своих структур данных, а о том, что структуры данных могут сделать *для вас*.

### 1.4.3. Классы и экземпляры

Хотя я имел дело с Фло лишь несколько раз, у меня имеется примерное представление о ее реакции на мой запрос. Я могу сделать определенные предположения, поскольку имею

PDF created with pdfFactory Pro trial version www.pdffactory.com

общую информацию о людях, занимающихся разведением цветов, и ожидаю, что Фло, будучи представителем этой категории, в общих чертах будет соответствовать шаблону. Мы можем использовать термин Florist для описания категории (или *класса*) всех людей, занимающихся цветоводством, собрав в нее (категорию) все то общее, что им свойственно. Эта операция является вторым принципом объектно-ориентированного программирования:

Все объекты являются представителями, или экземплярами, классов. Метод, активизируемый объектом в ответ на сообщение, определяется классом, к которому принадлежит получатель сообщения. Все объекты одного класса используют одни и те же методы в ответ на одинаковые сообщения.

Проблема сообщества объектно-ориентированных программистов заключается в распространенности различных терминов для обозначения сходных идей. Так, в языке Object Pascal класс называется «объектом» (тип данных object), а надклассы (которые вкратце будут описаны ниже) известны как родительский класс, класс-предок и т. д. Словарь-глоссарий в конце этой книги поможет вам разобраться с нестандартными терминами. Мы будем использовать соглашение, общее для объектно-ориентированных языков программирования: всегда обозначать классы идентификаторами, начинающимися с заглавной буквы. Несмотря на свою распространенность, данное соглашение не является обязательным для большинства языков программирования.

### 1.4.4. Иерархии классов и наследование

О Фло у меня имеется больше информации, чем содержится в категории Florist. Я знаю, что она разбирается в цветах и является владелицей магазина (shopkeeper). Я догадываюсь, что, вероятно, меня спросят о деньгах в процессе обработки моего запроса и что после оплаты мне будет выдана квитанция. Все вышеперечисленное справедливо также для зеленщиков, киоскеров, продавцов магазинов и т. д. Поскольку категория Florist является более узкой, чем Shopkeeper, то любое знание, которым я обладаю о категории Shopkeeper, справедливо также и для Florist, и, в частности, для Фло.

Один из способов представить организацию моего знания о Фло — это иерархия категорий (рис. 1.1). Фло принадлежит к категории Florist; Florist является подкатегорией категории Shopkeeper. Далее, представитель Shopkeeper заведомо является человеком, то есть принадлежит к категории Human — тем самым я знаю, что Фло с большой вероятностью является двуногим существом. Далее, категория Human включена в категорию млекопитающих (Mammal), которые кормят своих детенышей молоком, а млекопитающие являются подкатегорией животных (Animal) и, следовательно, дышат кислородом. В свою очередь животные являются материальными объектамидуумов с различными линиями наследования. Классы представляются в виде иерархической древовидной структуры, в которой более абстрактные классы (такие, как Material Object или Animal) располагаются в корне дерева, а более специализированные классы и в конечном итоге индивидуумы располагаются на его конце, в ветвях. Рисунок 1.2 показывает такую иерархию классов для Фло. Эта же самая иерархия включ ает в себя мою жену Бет, собаку Флеш, Фила — утконоса, живущего в зоопарке, а также цветы, которые я послал своей бабушке.

PDF created with pdfFactory Pro trial version www.pdffactory.com

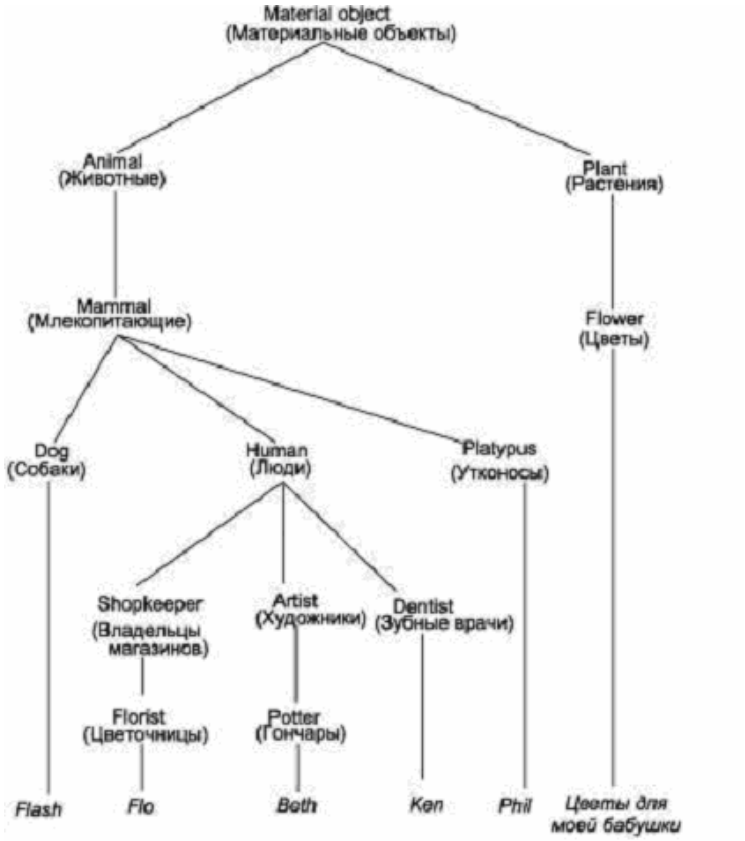



**Рис. 1.2.** Иерархическое дерево классов, представляющих различные материальные объекты

Поскольку Фло — человек, та информация о ней, которой я обладаю, применима также, к примеру, к моей жене Бет. Те данные, которыми я располагаю в силу принадлежности последней к классу млекопитающих, имеют также отношение к моей собаке Флеш. Информация об объектах как о вещах материальных имеет смысл в отношени Фло, и ее цветов. Мы выражаем все это в виде идеи наследования:
Классы могут быть организованы в иерархическую структуру с *наследованием* свойств. *Дочерний класс* (или *подкласс*) наследует атрибуты родительского класса (или надкласса), расположенного выше в иерархическом дереве [1] . *Абстрактный родительский класс* — это класс, не имеющий экземпляров (его примером может служить Mammal на рис. 1.2). Он используется только для порождения подклассов.

### 1.4.5. Связывание и переопределение методов

Утконос Фил представляет собой проблему для нашей простой структуры. Я знаю, что млекопитающие являются живородящими, но Фил определенно является млекопитающим, хотя он (точнее, его подруга Филлис) кладет яйца. Чтобы включить его в нашу схему, мы должны найти технику для представления исключений из общего правила.

Мы сделаем это, допустив правило, что информация, содержащаяся в подклассе, может -

---

[1] Здесь придется попросить читателя вернуться к рис. 1.2 и обратить внимание на то, что согласно принятой схеме дерево растет сверху вниз. — *Примеч. ред.*

PDF created with pdfFactory Pro trial version www.pdffactory.com

переопределять информацию, наследуемую из родительского класса. Очень часто при реализация такого подхода метод, соответствующий подклассу, имеет то же имя, что и соответствующий метод в родительском классе. При этом для поиска метода, подходящего для обработки сообщения, используется следующее правило: Поиск метода, который вызывается в ответ на определенное сообщение, начинается с методов, принадлежащих классу получателя. Если подходящий метод не найден, то поиск продолжается для родительского класса. Поиск продвигается вверх по цепочке родительских классов до тех пор, пока не будет найден нужный метод или пока не будет исчерпана последовательность родительских классов. В первом случае выполняется найденный метод, во втором — выдается сообщение об ошибке. Если выше в иерархии классов существуют методы с тем же именем, что и текущий, то говорят, что данный метод переопределяет наследуемое поведение.

Даже если компилятор не может определить, какой именно метод будет вызываться во время выполнения программы, то во многих языках программирования уже на этапе компилирования, а не при выполнении программы можно определить, что подходящего метода нет вообще, и выдать сообщение об ошибке. Мы будем обсуждать реализацию механизма переопределения в различных языках программирования в главе 11.

Тот факт, что моя жена Бет и хозяйка цветочного магазина Фло будут реагировать на мое сообщение с применением различных методов, является одним из примеров полиморфизма. Мы будем обсуждать эту важную составную часть объектно-ориентированного программирования в главе 14. То, что я, как уже говорилось, не знаю и не хочу знать, какой именно метод будет использован Фло для выполнения моего запроса, является примером маскировки *информации*, которая анализируется в главе 17.

### 1.4.6. Краткое изложение принципов

Алан Кей, которого кое-кто называет отцом объектно-ориентированного программирования, считает следующие положения фундаментальными характеристиками ООП [Kay 1993]:

1. Все является объектом.
2. Вычисления осуществляются путем взаимодействия (обмена данными) между объектами, при котором один объект требует, чтобы другой объект выполнил некое действие. Объекты взаимодействуют, посылая и получая сообщения. Сообщение — это запрос на выполнение действия, дополненный набором аргументов, которые могут понадобиться при выполнении действия.
3. Каждый объект имеет независимую память, которая состоит из других объектов.
4. Каждый объект является представителем класса, который выражает общие свойства объектов (таких, как целые числа или списки).
5. В классе задается поведение (функциональность) объекта. Тем самым все объекты, которые являются экземплярами одного класса, могут выполнять одни и те же действия.
6. Классы организованы в единую древовидную структуру с общим корнем, называемую иерархией наследования. Память и поведение, связанное с экземплярами определенного класса, автоматически доступны любому классу, расположенному ниже в иерархическом дереве.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## 1.5. Вычисление и моделирование

Взгляд на программирование, проиллюстрированный на примере с цветами, весьма отличается от привычного понимания того, что такое компьютер. Традиционная модель, описывающая выполнение программы на компьютере, базируется на дуализме процесс

состояние. С этой точки зрения компьютер является администратором данных, следующим некоторому набору инструкций. Он странствует по пространству памяти, изымает значения из ее ячеек (адресов памяти), некоторым образом преобразует полученные величины, а затем помещает их в другие ячейки (рис. 1.3). Проверяя значения, находящиеся в различных ячейках, мы определяем состояние машины или же результат вычислений. Хотя эта модель и может рассматриваться как более или менее точный образхранения, почтовых ящиках или ячейках памяти, содержащих значения, мало что из житейского опыта может подсказать, как следует структурировать задачу.

Хотя антропоморфные описания, подобные тем, что цитировались выше в тексте Ингалса, могут шокировать людей, фактически они являются отражением огромной выразительной силы метафор. Журналисты используют метафоры каждый день, подобно тому, как это сделано в нижеследующем фрагменте из NewsWeek: В отличие от обычного метода программирования — то есть написания программы строчка за строчкой, — «объектно-ориентированная» система компьютеров NeXT предлагает строительные блоки большего размера, которые разработчик может быстро собирать воедино, подобно тому, как дети складывают мозаику.

Возможно, именно это свойство — в большей степени, чем другие — вызывает часто наблюдаемый эффект, когда новичков от информатики легче учить понятиям объектно-ориентированного программирования, чем уже сложившихся профессионалов. Молодежь быстро адаптируется к соответствующим обыденной жизни метафорам, с которыми они себя чувствуют комфортно, в то время как «ветераны» обременены стремлением представить себе процесс вычислений, соответствующий традиционным взглядам на программирование.

### 1.5.2. Как избежать бесконечной регрессии

Конечно, объекты не могут во всех случаях реагировать на сообщение только тем, что вежливо обращаются к другим с просьбой выполнить некоторое действие. Это приведет к бесконечному циклу запросов, как если бы два джентльмена так и не вошли в дверь, уступая друг другу дорогу. На некоторой стадии по крайней мере некоторые объекты должны выполнять какую-то работу перед пересылкой запроса другим агентам. Эти действия выполняются по-разному в различных объектно-ориентированных языках программирования.

В языках, где директивный и объектно-ориентированный подходы уживаются вместе (таких, как C++, Object Pascal и Objective-C), реальные действия выполняются методами, написанными на основном (не объектно-ориентированном) языке. В чисто объектно-ориентированных языках (таких, как Smalltalk и Java) это выполняется с помощью «примитивных» или «встроенных» операций, которые обеспечиваются исполнительной системой более низкого уровня.

## 1.6. Барьер сложности

PDF created with pdfFactory Pro trial version www.pdffactory.com

На заре информатики, большинство программ писалось на ассемблере. Они не соответствуют сегодняшним стандартам. По мере того как программы становились все сложнее, разработчики обнаружили, что они не в состоянии помнить всю информацию, необходимую для отладки и совершенствования их программного обеспечения. Какие значения находятся в регистрах? Вступает ли новый идентификатор в конфликт с определенными ранее? Какие переменные необходимо инициализировать перед тем, как передать управление следующему коду?

Появление таких языков программирования высокого уровня, как Fortran, Cobol и Algol, разрешило некоторые проблемы (было введено автоматическое управление локальными переменными и неявное присваивание значений). Одновременно это возросла вера пользователей в возможности компьютера. По мере того как предпринимались попытки решить все более сложные проблемы с его использованием, возникали ситуации, когда даже лучшие программисты не могли удержать все в своей памяти. Привычными стали команды программистов, работающих совместно.

### 1.6.1. Нелинейное увеличение сложности

По мере того как программные проекты становились все сложнее, было замечено интересное явление. Задача, для решения которой одному программисту требовалось два месяца, не решалась двумя программистами за месяц. Согласно замечательной фразе Фреда Брукса, «рождение ребенка занимает девять месяцев независимо от того, сколько женщин занято этим» [Brooks 1975].

Причиной такого нелинейного поведения является сложность. В частности, взаимосвязи между программными компонентами стали сложнее, и разработчики вынуждены были постоянно обмениваться между собой значительными объемами информации. Брукс также сказал:

Поскольку конструирование программного обеспечения по своей внутренней природе есть задача системная (требует сложного взаимодействия участников), то расходы на обмен данными велики. Они быстро становятся доминирующими и нивелируют уменьшение индивидуальных затрат, достигаемое за счет разбиения задачи на фрагменты. Добавление новых людей удлиняет, а не сокращает расписание работ.

Порождает сложность не просто большой объем рассматриваемых задач, а уникальное свойство программных систем, разработанных с использованием традиционных подходов, — большое число перекрестных ссылок между компонентами (именно это делает их одними из наиболее сложных людских творений). Перекрестные ссылки в данном случае обозначают зависимость одного фрагмента кода от другого.

Действительно, каждый фрагмент программной системы должен выполнять некую реальную работу — в противном случае он был бы не нужен. Если эта деятельность оказывается необходимой для других частей программы, то должен присутствовать поток данных либо из, либо в рассматриваемую компоненту. По этой причине полное понимание фрагмента программы требует знаний как кода, который мы рассматриваем, так и кода, который пользуется этим фрагментом. Короче говоря, даже относительно независимый раздел кода нельзя полностью понять в изоляции от других.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 1.6.2. Механизмы абстрагирования

Программисты столкнулись с проблемой сложности уже давно. Чтобы полностью понять важность объектно-ориентированного подхода, нам следует рассмотреть разнообразные механизмы, которые использовались программистами для контроля над сложностью. Главный из них — это абстрагирование, то есть способность отделить логический смысл фрагмента программы от проблемы его реализации. В некотором смысле объектно-ориентированный подход вообще не является революционным и должен рассматриваться как естественный результат исторического развития: от процедур к модулям, далее к абстрактным типам данных и наконец к объектам.

#### *Процедуры*

Процедуры и функции были двумя первыми механизмами абстрагирования, примененными в языках программирования. Процедуры позволяют сконцентрировать в одном месте работу, которая выполняется многократно (возможно, с небольшими вариациями), и затем многократно использовать этот код, вместо того чтобы писать его снова и снова. Кроме всего прочего, процедуры впервые обеспечили возможность маскировки информации. Программист мог написать процедуру или набор процедур, которые потом использовались другими людьми. Последние не обязаны были знать детали использованного алгоритма — их интересовал только интерфейс программы. Но процедуры не решили всех проблем. В частности, они не обладали эффективным механизмом маскировки деталей организации данных и только отчасти снимали проблему использования разными программистами одинаковых имен.

#### *Пример: стек*

Чтобы проиллюстрировать эти проблемы, рассмотрим ситуацию, когда программисту нужно реализовать управление простым стеком. Следуя старым добрым принципам разработки программного обеспечения, наш программист прежде всего определяет внешний интерфейс — скажем, набор из четырех процедур init, push, pop и top. Затем он выбирает подходящий метод реализации. Здесь есть из чего выбрать: массив с указателем на вершину стека, связный список и т. д. Наш бесстрашный разработчик выбирает один из методов, а затем приступает к кодированию, как показано в листинге 1.1.

Легко увидеть, что данные, образующие стек, не могут быть сделаны локальными для какой-то из четырех процедур, поскольку эти данные являются общими для всех из них. Но если у нас есть только локальные или глобальные переменные (как это имеет место для Fortran или было в C, до того как ввели модификатор static), то данные стека должны содержаться в глобальных переменных. Однако если переменные являются глобальными, то нет способа ограничить доступ к ним или их видимость. Например, если стек реализован как массив с именем datastack, то об этом должны знать все остальные программисты, поскольку они могут захотеть создать переменные с таким же именем, чего делать ни в коем случае нельзя. Запрет на использование имени datastack необходим, даже если сами данные важны только для подпрограмм работы со стеком и не будут использоваться за пределами этих четырех процедур. Аналогично имена init, push, pop и top являются теперь зарезервированными и не должны встречаться в других частях программы (разве что с целью вызова процедур), даже если эти части не имеют ничего общего с подпрограммами, обслуживающими стек.

PDF created with pdfFactory Pro trial version www.pdffactory.com

**Листинг 1.1.** Процедуры не годятся для маскировки информации

```
int datastack[100];
int datatop = 0;
void init()
{
    datatop = 0;
}
void push(int val)
{    if (datatop < 100)
        datastack [datatop++] = val;
}
int top()
{
    if (datatop > 0)
        return datastack [datatop – 1];
    return 0;
}
int pop()
{
    if (datatop > 0)
        return datastack  [--datatop];
    return 0;
}
```

### *Область видимости для блоков*

Механизм видимости для блоков, использованный в языке Алгол и его преемниках (таких, как Pascal), предлагает чуть больший контроль над видимостью имен, чем просто различие между локальными и глобальными именами. Кажется, мы могли бы надеяться, что это решит проблему скрытия информации. К сожалению, проблема остается. В любой области, в которой разрешен доступ к именам четырех процедур, видны также и их общие данные. Чтобы решить эту дилемму, требуется разработать иной механизм структурирования.

```
begin
  var
    datastack : array [1..100] of integer;
    datatop : integer;
  procedure init; . . .
  procedure push(val : integer); . . .
  function pop : integer; . . .
. . .
end;
```

## Модули

В некотором смысле модули можно рассматривать просто как улучшенный метод создания и управления совокупностями имен и связанными с ними значениями. Наш пример со стеком является типичным в том аспекте, что имеется определенная информация (интерфейсные процедуры), которую мы хотим сделать широко и открыто используемой, в то время как доступ к некоторым данным (собственно данные стека) должен быть ограничен. Если рассматривать модуль как абстрактную концепцию, сведенную к своей простейшей форме, то ее суть состоит в разбиении пространства имен на две части. Открытая (public) часть является доступной извне модуля, закрытая (private)

PDF created with pdfFactory Pro trial version www.pdffactory.com

часть доступна только внутри модуля. Типы, данные (переменные) и процедуры могут быть отнесены к любой из двух частей.

Дэвид Парнас [Parnas 1972] популяризовал понятие модулей. Он сформулировал следующие два принципа их правильного использования:

1. Пользователя, который намеревается использовать модуль, следует снабдить всей информацией, необходимой, чтобы делать это корректно, и не более того.
2. Разработчика следует снабдить всей информацией, необходимой для создания модуля, и не более того.

Эта философия в значительной мере напоминает военную доктрину «необходимого знания»: если вам не нужно знать определенную информацию, вы и не должны иметь к ней доступа. Это явное, намеренное и целенаправленное утаивание информации называется маскировкой информации (information hiding).

Модули решают некоторые, но не все проблемы разработки программного обеспечения. Например, они позволяют нашему программисту скрыть детали реализации стека, но что делать, если другие пользователи захотят иметь два (или более) стека?

В качестве более сложного примера предположим, что программист заявляет, что им разработан новый тип числовых объектов, названный Complex. Он определил арифметические операции для комплексных величин — сложение, вычитание, умножение и т. д. и ввел подпрограммы для преобразования обычных чисел в комплексные и обратно. Имеется лишь одна маленькая проблема: можно манипулировать только с одним комплексным числом.

Комплексные числа вряд ли будут полезны при таком ограничении, но это именно та ситуация, в которой мы оказываемся в случае простых модулей. Последние, взятые сами по себе, обеспечивают эффективный механизм маскировки информации, но они не позволяют осуществлять размножение экземпляров, под которым мы понимаем возможность сделать много копий областей данных. Чтобы справиться с проблемой размножения, специалистам по информатике потребовалось разработать новую концепцию.

## Абстрактные типы данных

Абстрактный тип данных задается программистом. С данными абстрактного типа можно манипулировать так же, как и с данными типов, встроенных в систему. Как и последним, абстрактному типу данных соответствует набор (возможно, бесконечный) допустимых значений и ряд элементарных операций, которые могут быть выполнены над данными. Пользователю разрешается создавать переменные, которые принимают значения из допустимого множества, и манипулировать ими, используя имеющиеся операции. К примеру, наш бесстрашный программист может определить свой стек как абстрактный тип данных и стековые операции как единственные действия, которые допускается производить над отдельными экземплярами стеков.

Модули часто используются при реализации абстрактных типов данных. Непосредственной логической взаимосвязи между понятиями модуля и абстрактного типа данных нет. Эти две идеи близки, но не идентичны. Чтобы построить абстрактный тип данных, мы должны уметь:

PDF created with pdfFactory Pro trial version www.pdffactory.com

1. Экспортировать определение типа данных.
2. Делать доступным набор операций, использующихся для манипулирования экземплярами типа данных.
3. Защищать данные, связанные с типом данных, чтобы с ними можно было работать только через указанные подпрограммы.
4. Создавать несколько экземпляров абстрактного типа данных.

В нашем определении модули служат только как механизм маскировки информации и тем самым непосредственно связаны только со свойствами 2 и 3 из нашего списка. Остальные свойства в принципе могут быть реализованы с использованием соответствующей техники программирования. Пакеты, которые встречаются в таких языках программирования, как CLU или Ada, тесно связаны с перечисленными выше требуемыми свойствами абстрактных типов данных.

В определенном смысле объект — это просто абстрактный тип данных. Говорили, к примеру, что программисты на языке Smalltalk пишут наиболее «структурированные» программы, потому что они не имеют возможности написать что-либо кроме определений абстрактных типов данных. Истинная правда, что объект является абстрактным типом данных, но понятия объектно-ориентированного программирования, хотя и строятся на идеях абстрактных типов данных, добавляют к ним важные новшества по части разделения и совместного использования программного кода.

## Объекты: сообщения, наследование и полиморфизм

Объектно-ориентированное программирование добавляет несколько новых важных идей к концепции абстрактных типов данных. Главная из них — пересылка сообщений. Действие инициируется по запросу, обращенному к конкретному объекту, а не через вызов функции. В значительной степени это просто смещение ударения: традиционная точка зрения делает основной упор на операции, в то время как ООП на первое место ставит собственно значение. (Вызываете ли вы подпрограмму push со стеком и значением в качестве аргументов, или же вы просите объект stack поместить нужное значение к нему внутрь?) Если бы это было все, что имеется в объектно-ориентированном программировании, эта техника не рассматривалась бы как принципиальное нововведение. Но к пересылке сообщений добавляются мощные механизмы переопределения имен и совместного/многократного использования программного кода.

Неявной в идее пересылки сообщений является мысль о том, что интерпретация сообщения может меняться для различных объектов. А именно поведение и реакция, инициируемые сообщением, зависят от объекта, который получает сообщение. Тем самым push может означать одно действие для стека и совсем другое для блока управления механической рукой. Поскольку имена операций не обязаны быть уникальными, могут использоваться простые и явные формы команд. Это приводит к более читаемому и понятному коду.

Наконец, объектно-ориентированное программирование добавляет механизмы наследования и полиморфизма. Наследование позволяет различным типам данных совместно использовать один и тот же код, приводя к уменьшению его размера и повышению функциональности. Полиморфизм перекраивает этот общий код так, чтобы удовлетворить конкретным особенностям отдельных типов данных. Упор на независимость индивидуальных компонент позволяет использовать процесс пошаговой сборки, при которой отдельные блоки программного обеспечения разрабатываются, программируются и отлаживаются до того, как они объединяются в большую систему.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Все эти идеи будут описаны более подробно в последующих главах.

## 1.7. Многократно используемое программное обеспечение

Десятилетиями люди спрашивали себя, почему создание программного обеспечения не может копировать процесс конструирования материальных объектов. К примеру, когда мы строим здание, автомобиль или электронное устройство, мы обычно соединяем вместе несколько готовых компонент вместо того, чтобы изготовлять каждый новый элемент с нуля. Можно ли конструировать программное обеспечение таким же образом?

Многократное использование программного обеспечения — цель, к которой постоянно стремятся и редко достигают. Основная причина этого — значительная взаимозависимость большей части программного обеспечения, созданного традиционными способами. Как мы видели в предыдущих разделах, трудно извлечь из проекта фрагменты программного обеспечения, которые бы легко использовались в не имеющем к нему отношения новом программном продукте (каждая часть кода обычно связана с остальными фрагментами). Эти взаимозависимости могут быть результатом определения структуры данных или следствием особенностей функционирования.

Например, организация записей в виде таблицы и осуществление операции ее индексированного просмотра являются обычным и в программировании. Тем не менее до сих пор подпрограммы поиска в таблицах зачастую пишутся «с нуля» для каждого нового приложения. Почему? Потому что в привычных языках программирования формат записи для элементов таблицы жестко связан с более общим кодом для вставки и просмотра. Трудно написать код, который бы работал для произвольной структуры данных и любого типа записей.

Объектно-ориентированное программирование обеспечивает механизм для отделения существенной информации (занесение и получение записей) от специализированной (конкретный формат записей). Тем самым при использовании объектно-ориентированной техники мы можем создавать большие программные компоненты, пригодные для повторного использования. Многие коммерческие пакеты программных компонентов, пригодных для многократного использования, уже имеются, и разработка повторно используемых программных компонентов становится быстро развивающейся отраслью индустрии программного обеспечения.

## 1.8. Резюме

Объектно-ориентированное программирование — это не просто несколько новых свойств, добавленных в уже существующие языки. Скорее — это новый шаг в осмыслении процессов декомпозиции задач и разработки программного обеспечения.

ООП рассматривает программы как совокупность свободно (гибко) связанных между собой агентов, называемых объектами. Каждый из них отвечает за конкретные задачи. Вычисление осуществляется посредством взаимодействия объектов. Следовательно, в определенном смысле программирование — это ни много ни мало, как моделирование мира.

Объект получается в результате инкапсуляции состояния (данных) и поведения (операций). Тем самым объект во многих отношениях аналогичен модулю или абстрактному типу данных.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Поведение объекта диктуется его классом. Каждый объект является экземпляром некоторого класса. Все экземпляры одного класса будут вести себя одинаковым образом (то есть вызывать те же методы) в ответ на одинаковые запросы.

Объект проявляет свое поведение путем вызова метода в ответ на сообщение. Интерпретация сообщения (то есть конкретный используемый метод) зависит от объекта и может быть различной для различных классов объектов.

Объекты и классы расширяют понятие абстрактного типа данных путем введения наследования. Классы могут быть организованы в виде иерархического дерева наследования. Данные и поведение, связанные с классами, которые расположены выше в иерархическом дереве, доступны для нижележащих классов. Происходит наследование поведения от родительских классов.

С помощью уменьшения взаимозависимости между компонентами программного обеспечения ООП позволяет разрабатывать системы, пригодные для многократного использования. Такие компоненты могут быть созданы и отлажены как независимые программные единицы, в изоляции от других частей прикладной программы.

Многократно используемые программные компоненты позволяют разработчику иметь дело с проблемами на более высокой ступени абстрагирования. Мы можем определять и манипулировать объектами просто в терминах сообщений, которые они распознают, и работы, которую они выполняют, игнорируя детали реализации.

## Что читать дальше

Я отметил ранее, что Алан Кей считается отцом объектно-ориентированного программирования. Подобно многим простым высказываниям, данное утверждение выдерживает критику лишь отчасти. Сам Кей [Kay 1993] считает, что его вклад состоит преимущественно в разработке языка Smalltalk на основе более раннего языка программирования Simula, созданного в Скандинавии в 60-х годах [Dahl 1966, Kirkerud 1989]. История свидетельствует, что большинство принципов объектно-ориентированного программирования было полностью разработано создателями языка Simula, но этот факт в значительной степени игнорировался профессионалами до тех пор, пока они (принципы) не были вновь открыты Кеем при разработке языка программирования Smalltalk. Пользующийся широкой популярностью журнал Byte в 1981 году сделал многое для популяризации концепций, разработанных Кеем и его командой из группы Xerox PARC.

Термин «кризис программного обеспечения», по-видимому, был изобретен Дугом Мак-Илроем во время конференции НАТО 1968 года по программным технологиям. Забавно, что мы находимся в этом кризисе и сейчас, по прошествии половины срока существования информатики как независимой дисциплины. Несмотря на окончание холодной войны, выход из кризиса программного обеспечения не ближе к нам, чем это было в 1968 году — см., к примеру, статью Гиббса «Хронический кризис программного обеспечения» в сентябрьском выпуске Scientific American за 1994 год [Gibbs 1994].

До некоторой степени кризис программного обеспечения — в значительной мере иллюзия. Например, задачи, рассматривавшиеся как чрезвычайно сложные пять лет назад, редко считаются таковыми сегодня. Проблемы, которые мы желаем решить сейчас, ранее считались непреодолимыми — по-видимому, это показывает, что разработка программного обеспечения год от года прогрессирует.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Цитата американского лингвиста Эдварда Сапира (стр. 21) взята из статьи «Связь поведения и мышления с языком», перепечатанной в сборнике «Мышление и реальность» [Whorf 1956]. В нем содержится несколько интересных работ по связям между языком и процессом мышления. Я настоятельно рекомендую каждому серьезному студенту, занимающемуся компьютерными языками, прочитать эти статьи. Некоторые из них имеют удивительно близкое отношение к искусственным языкам.

Другая интересная книга — это «Эффект алфавита» Роберта Логана [Logan 1986], которая объясняет в лингвистических терминах, почему логика и наука были разработаны на Западе, в то время как в течение веков Китай имел опережающую технологию. В более современном исследовании о влиянии естественного языка на информатику Дж. Маршалл Унгер [Unger 1987] описывает влияние японского языка на известный проект Пятого поколения компьютеров.

Всеми признанное наблюдение, что язык эскимосов имеет много слов для обозначения типов снега, было развенчано Джоффри Паллумом в его сборнике статей по лингвистике [Pullum 1991]. В статье в Atlantic Monthly «Похвала снегу» (январь 1995) Каллен Мерфи указывал, что набор слов, используемый для обсуждения «снежной» тематики людьми, говорящими по-английски, по крайней мере столь же разнообразен, как и термины эскимосов. При этом, естественно, имеются в виду люди, для которых различия в типах снега существенны (преимущественно это ученые, которые проводят исследования в данной области).

В любом случае данное обстоятельство не имеет значения для нашего обсуждения. Определенно истинно, что группы индивидуумов с общими интересами стремятся разработать свой собственный специализированный словарь и, будучи однажды созданным, он имеет тенденцию направлять мысли своих творцов по пути, который не является естественным для людей за пределами группы. Именно такова ситуация с ООП. Хотя объектно-ориентированные идеи могут, при надлежащей дисциплине, быть использованы и без объектно-ориентированных языков, использование их терминов помогает направить ум программиста по пути, который не очевиден без терминологии ООП.

Мой рассказ является слегка неточным в отношении принципа Чёрча и машин Тьюринга. Чёрч фактически делал свое утверждение относительно рекурсивных функций [Church 1936], которые впоследствии оказались эквивалентными вычислениям, проводимым с помощью машин Тьюринга [Turing 1936]. В той форме, в которой мы его формулируем здесь, этот принцип был описан Клини, и им же было дано то название, под которым принцип теперь известен. Роджерс приводит хорошую сводку аргументов в защиту эквивалентности различных моделей вычислений [Rogers 1967].

Если вы помните, именно шведский ботаник Карл Линней разработал идеи родов, видов и т. д. Это является прототипом схемы иерархической организации, иллюстрирующей наследование, поскольку абстрактная классификация описывает характеристики, свойственные всем классификациям. Большинство иерархий наследования следуют модели Линнея.

Критика процедур как методики абстрагирования (поскольку они не способны обеспечить надлежащий механизм маскировки данных) была впервые проведена Вилльямом Вульфом и Мери Шоу [Wulf 1973] при анализе многочисленных проблем, связанных с использованием глобальных переменных. Эта аргументация была впоследствии расширена Дэвидом Хансоном [Hanson 1981].

PDF created with pdfFactory Pro trial version www.pdffactory.com

Подобно многим словам, которые нашли себе место в общепринятом жаргоне, термин «объектно-ориентированный» используется гораздо шире своего фактического значения. Тем самым на вопрос: «Что такое объектно-ориентированное программирование?» очень непросто ответить. Бьорн Страуструп [Stroustrup 1988] не без юмора заметил, что большинство аргументов сводится к следующему силлогизму:

- X — это хорошо.
- Объектная ориентированность — это хорошо.
- Следовательно, X является объектно-ориентированным.

Роджер Кинг аргументированно настаивал, что его кот является объектно-ориентированным. Кроме прочих своих достоинств, кот демонстрирует характерное поведение, реагирует на сообщения, наделен унаследованными реакциями и управляет своим вполне независимым внутренним состоянием.

Многие авторы пытались дать строгое определение тех свойств языка программирования, которыми он должен обладать, чтобы называться объектно-ориентированным, — см., к примеру, анализ, проведенный Джозефиной Микалеф [Micallef 1988] или Питером Вегнером [Wegner 1986].

Вегнер, к примеру, различает языки, основанные на объектах, которые поддерживают только абстрагирование (такие, как Ada), и объектно-ориентированные языки, которые поддерживают наследование.

Другие авторы — среди них наиболее заметен Брэд Кокс [Cox 1990] — определяют термин ООП значительно шире. Согласно Коксу объектно-ориентированное программирование представляет собой метод или цель (objective) программирования путем сборки приложений из уже имеющихся компонент, а не конкретную технологию, которую мы можем использовать, чтобы достичь этой цели. Вместо выпячивания различий между подходами мы должны объединить воедино любые средства, которые оказываются многообещающими на пути к новой Индустриальной Революции в программировании. Книга Кокса по ООП [Cox 1986], хотя и написана на заре развития объектно-ориентированного программирования, и в силу этого отчасти устаревшая в отношении деталей, тем не менее является одним из наиболее читаемых манифестов объектно-ориентированного движения.

## Упражнения

1. В объектно-ориентированной иерархии наследования каждый следующий уровень является более специализированной формой предыдущего. Приведите пример иерархии из повседневной жизни с этим свойством. Некоторые из иерархий, обнаруживаемые в реальной жизни, не являются иерархиями наследования. Укажите пример иерархии без свойства наследования.
2. Посмотрите значение слова парадигма по крайней мере в трех словарях. Соотнесите эти определения с языками программирования.
3. Возьмите задачу из реального мира (аналогичную пересылке цветов, рассмотренной ранее) и опишите ее решение в терминах агентов (объектов) и обязанностей.
4. Если вы знакомы с двумя (или более) различными языками программирования, приведите пример, когда один язык направляет мысль программиста к определенному решению, а другой — стимулирует альтернативное решение.

PDF created with pdfFactory Pro trial version www.pdffactory.com

5. Если вы знакомы с двумя (или более) естественными языками, опишите ситуацию, когда один язык направляет говорящего в одном направлении, в то время как другой язык приводит к иному ходу мысли.

## *Глава 2 : Объектно- ориентированное проектирование*

Когда программисты спрашивают друг друга: «Чем же, в конце концов, является объектно-ориентированное программирование?», ответ чаще всего подчеркивает синтаксические свойства таких языков, как C++ или Object Pascal, по сравнению с их более ранними, не объектно-ориентированными версиями, то есть C или Pascal. Тем самым обсуждение обычно переходит на такие предметы, как классы и наследование, пересылка сообщений, виртуальные и статические методы. Но при этом опускают наиболее важный момент в объектно-ориентированном программировании, который не имеет ничего общего с вопросами синтаксиса.

Работа на объектно-ориентированном языке (то есть на языке, который поддерживает наследование, пересылку сообщений и классы) не является ни необходимым, ни достаточным условием для того, чтобы заниматься объектно-ориентированным программированием. Как мы подчеркнули в главе 1, наиболее важный аспект в ООП — техника проектирования, основанная на выделении и распределении обязанностей. Она была названа проектированием на основе обязанностей или проектированием на основе ответственности (responsibility-driven design) [Wirfs-Brock 1989b, Wirfs-Brock 1990].

### 2.1. Ответственность подразумевает невмешательство

Как может констатировать любой, кто помнит себя ребенком (или кто воспитывает детей), ответственность — обоюдоострый меч. Когда вы заставляете какой-либо объект (является ли он ребенком, или программной системой) быть ответственным за конкретные действия, вы ожидаете с его стороны определенного поведения, по крайней мере пока не нарушены правила. Но, в равной степени важно, что ответственность подразумевает определенный уровень независимости или невмешательства. Если вы скажете своей дочке, что она отвечает за уборку своей комнаты, вы, как правило, не стоите рядом с ней и не наблюдаете за выполнением работы — это противоречило бы понятию ответственности. Вместо этого вы рассчитываете, что после выдачи распоряжения будет получен желаемый результат.

Аналогично в случае примера с цветами из главы 1, когда я передаю запрос хозяйке цветочного магазина с просьбой переслать цветы, я не задумываюсь о том, как мой запрос будет обслужен. Хозяйка цветочного магазина, раз уж она взяла на себя ответственность, действует без вмешательства с моей стороны.

Разница между традиционным и объектно-ориентированным программированием во многих отношениях напоминает различие между активным наблюдением за тем, как ребенок выполняет работу, и передачей (делегированием) ребенку ответственности за эту деятельность. Традиционное программирование основывается в основном на приказаниях чему-либо сделать что-то — к примеру, модифицировать запись или обновить массив данных. Тем самым фрагмент кода привязан посредством передачи управления и соглашений о структуре данных ко многим другим разделам программной системы. Такие зависимости могут возникать через использование глобальных переменных, значений указателей или попросту из-за неправильного применения или зависимой реализации других фрагментов кода. Проектирование, основанное на ответственности, старается

PDF created with pdfFactory Pro trial version www.pdffactory.com

отсекать эти связи или по крайней мере сделать их настолько слабыми, насколько это возможно.

С первого взгляда идея кажется не более сложной, чем понятия маскировки информации и модульности, которые важны при программировании в целом, в том числе и при использовании традиционных языков. Но проектирование, основанное на распределении ответственности, поднимает маскировку данных с уровня техники до уровня искусства. Принцип маскировки информации становится жизненно важным при переходе от программирования «в малом» к программированию «в большом».

Одно из основных преимуществ ООП наблюдается, когда программные подсистемы многократно используются в разных проектах. Например, программа, управляющая моделированием (подобно той, которую мы будем разрабатывать в главе 6), может имитировать как движение бильярдных шаров по столу, так и перемещение рыбы в цистернах. Эта способность кода к многократному использованию неявным образом подразумевает, что в программном обеспечении почти не должно быть проблемно-зависимых компонентов — оно должно полностью делегировать ответственность за специфичное поведение к фрагментам конкретной системы. Умению создавать подобный многократно используемый код не так просто научиться — оно требует опыта, тщательного исследования учебных примеров (парадигм, в исходном значении этого слова) и использования языков программирования, в которых такое делегирование является естественным и легко выражаемым. В последующих главах мы приведем несколько примеров.

## 2.2. Программирование «в малом» и «в большом»

О разработке индивидуального проекта часто говорят как о программировании «в малом», а о реализации большого проекта как о программировании «в большом».

Для программирования «в малом» характерны следующие признаки:

- Код разрабатывается единственным программистом или, возможно, небольшой группой программистов. Отдельно взятый индивидуум может понять все аспекты проекта, от и до.
- Основная проблема при разработке состоит в проектировании программы и написании алгоритмов для решения поставленной задачи.

С другой стороны, программирование «в большом» наделяет программный проект следующими свойствами:

- Программная система разрабатывается большой командой программистов. При этом одна группа может заниматься проектированием (или спецификацией) системы, другая — осуществлять написание кода отдельных компонент, а третья — объединять фрагменты в конечный продукт. Нет единственного человека, который знал бы все о проекте.
- Основная проблема в процессе разработки программного обеспечения — управление проектом и обмен информацией между группами и внутри групп.

В то время как начинающий студент обычно знакомится с программированием «в малом», особенности многих объектно-ориентированных языков наилучшим образом понимаются при встрече с проблемами, типичными для программирования «в большом». Тем самым

PDF created with pdfFactory Pro trial version www.pdffactory.com

некоторое представление о трудностях, возникающих при разработке больших систем, является полезным для понимания ООП.

## 2.3. Почему надо начинать с функционирования?

Из-за чего процесс проектирования начинают с анализа функционирования или поведения системы? Простой ответ состоит в том, что поведение системы обычно известно задолго до остальных ее свойств.

Предшествовавшие методы разработки программного обеспечения концентрировались на таких идеях, как характеристики основных данных или же общая структура вызова функций. Но структурные элементы приложения могут быть определены только после интенсивного анализа задачи. Соответственно процесс формальной спецификации часто заканчивался созданием документа, который не понимали ни программисты, ни клиенты. Но поведение — это нечто, что может быть описано в момент возникновения идеи программы и (в отличие от формальной спецификации системы) выражено в терминах, имеющих значение как для программиста, так и для клиента.

Мы проиллюстрируем проектирование на основе обязанностей (или RDD-проектирование — Responsibility-Driven-Design) на учебном примере.

## 2.4. Учебный пример: проектирование на основе обязанностей

Представьте себе, что вы являетесь главным архитектором программных систем в ведущей компьютерной фирме. В один прекрасный день ваш начальник появляется в офисе с идеей, которая, как он надеется, будет очередным успехом компании. Вам поручают разработать систему под названием Interactive Intelligent Kitchen Helper (Интерактивный разумный кухонный помощник) (рис. 2.1)

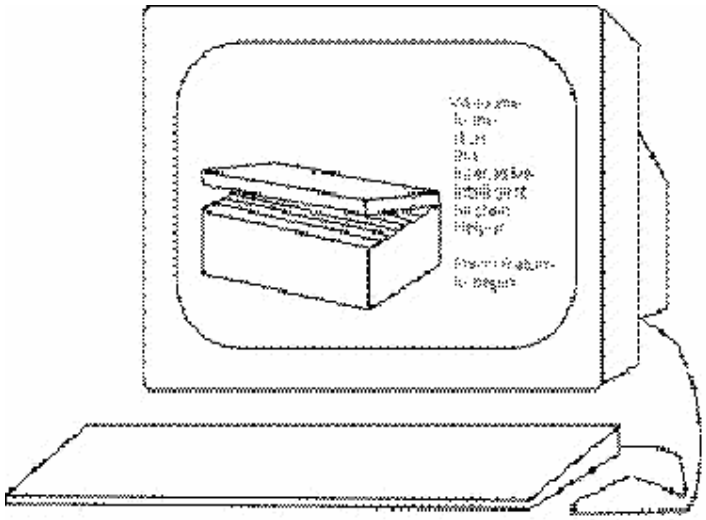


**Рис. 2.1.** Внешний вид программы «Интерактивный разумный кухонный помощник»

Задача, поставленная перед вашей командой программистов, сформулирована в нескольких скупых словах (написанных на чем-то, что оказывается использованной обеденной салфеткой, причем почерком, принадлежащим вашему начальнику).

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 2.4.1. Интерактивный разумный кухонный помощник

Программа «Интерактивный разумный кухонный помощник» (Interactive Intelligent Kitchen Helper, IIKH) предназначена для персональных компьютеров. Ее цель — заменить собой набор карточек с рецептами, который можно встретить почти в каждой кухне. Но помимо ведения базы данных рецептов, IIKH помогает в планировании питания на длительный период — например, на неделю вперед. Пользователь программы IIKH садится за компьютер, просматривает базу данных рецептов и в диалоговом режиме определяет меню на весь требуемый период.

Как это обычно бывает при первоначальном описании многих программных систем, спецификация для IIKH в значительной степени двусмысленна в отношении ряда важных пунктов. Кроме того, проект и разработка программной системы IIKH потребует совместных усилий нескольких программистов. Тем самым первоначальная цель команды разработчиков состоит в том, чтобы сделать ясными двусмысленные места и наметить разбиение проекта на компоненты, с тем чтобы распределить их между отдельными членами команды.

Краеугольным камнем в ООП является характеристика программного обеспечения в терминах *поведения*, то есть в терминах действий, которые должны быть выполнены. Мы увидим воплощение в жизнь этого принципа на многих уровнях процесса разработки IIKH. Первоначально команда попытается охарактеризовать на очень высоком уровне абстрагирования поведение приложения в целом. Затем она займется описанием поведения различных программных подсистем. И только тогда, когда все аспекты поведения будут выделены и описаны, программисты-разработчики приступят к этапу кодирования. В следующих разделах мы будем отслеживать этапы работы команды программистов при создании данного приложения.

### 2.4.2. Работа по сценарию

Первой задачей является уточнение спецификации. Как мы уже заметили, исходные спецификации почти всегда двусмысленны и непонятны во всем, кроме наиболее общих положений. На этом этапе ставится несколько целей. Одной из них является лучшее понимание и ощущение того, чем будет конечный продукт (принцип «посмотри и почувствуй» для проектирования системы). Затем эта информация может быть возвращена назад клиенту (в данном случае вашему начальнику), чтобы увидеть, находится ли она в соответствии с исходной концепцией. Вероятно и, возможно, неизбежно то, что спецификации для конечного продукта будут изменяться во время разработки программной системы, и поэтому важно, чтобы проект мог легко включать в себя новые идеи, а также чтобы потенциально возможные исправления были выявлены как можно раньше — см. раздел 2.6.2. «Готовность к изменениям». На этом же этапе проводится обсуждение структуры будущей программной системы. В частности, действия, осуществляемые программной системой, разбиваются на компоненты.

### 2.4.3. Идентификация компонент

Создание сложной физической системы, подобной зданию или двигателю автомобиля, упрощается с помощью разбиения проекта на структурные единицы. Точно так же разработка программного обеспечения облегчается после выделения отдельных компонент программы. Компонента — это просто абстрактная единица, которая может выполнять определенную работу (то есть иметь определенные обязанности). На этом этапе нет необходимости знать в точности то, как задается компонента или как она будет

PDF created with pdfFactory Pro trial version www.pdffactory.com

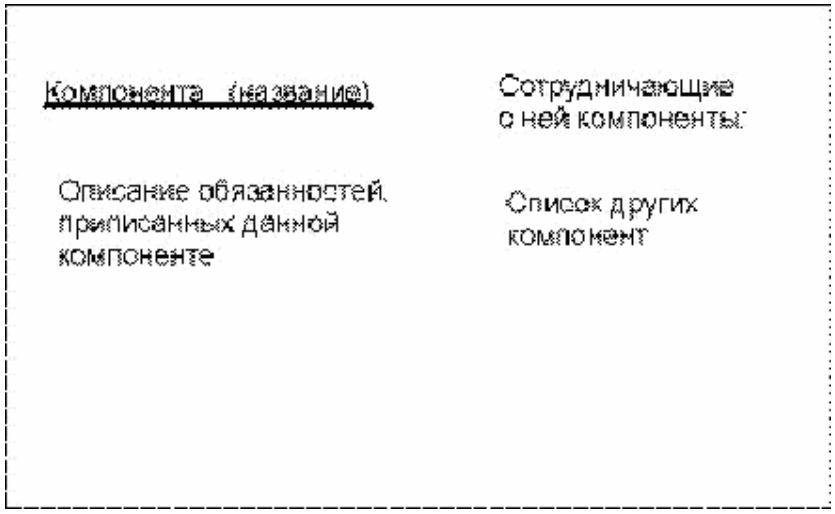
выполнять свою работу. Компонента может в конечном итоге быть преобразована в отдельную функцию, структуру или класс, или же в совокупность других компонент (шаблон). На этом уровне разработки имеются две важные особенности:

- компонента должна иметь небольшой набор четко определенных обязанностей;
- компонента должна взаимодействовать с другими компонентами настолько слабо, насколько это возможно.

Позднее мы поговорим о второй особенности подробнее. Сейчас мы просто занимаемся определением обязанностей компонент.

## 2.5. CRC-карточка — способ записи обязанностей

Чтобы выявить отдельные компоненты и определить их обязанности, команда программистов прорабатывает сценарий системы. То есть воспроизводится запуск приложения, как если бы оно было уже готово. Любое действие, которое может произойти, приписывается некоторой компоненте в качестве ее обязанности.




В качестве составной части этого процесса полезно изображать компоненты с помощью небольших индексных карточек. На лицевой стороне карточки написаны имя компоненты, ее обязанности и имена других компонент, с которыми она должна взаимодействовать. Такие карточки иногда называются CRC-карточками от слов Component, Responsibility, Collaborator (компонента, обязанность, сотрудники) [Beck 1989]. По мере того как для компонент выявляются обязанности, они записываются на лицевой стороне CRC-карточки.

### 2.5.1. Дайте компонентам физический образ

При проработке сценария полезно распределить CRC-карточки между различными членами проектной группы. Человек, имеющий карточку, которая представляет определенную компоненту, записывает ее обязанности и исполняет функции заменителя программы во время моделирования сценария. Он описывает действия программной системы, передавая «управление» следующему члену команды, когда программная система нуждается в услугах других компонент.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Преимущество CRC-карточек в том, что они широко доступны, недороги и с них можно стирать информацию. Это стимулирует экспериментирование, поскольку альтернативные проекты могут быть испробованы, изучены и отброшены с минимальными затратами. Физическое разделение карточек стимулирует интуитивное понимание важности логического разделения компонент, что помогает сделать упор на связности внутри модулей и зацеплении между модулями (которые вкратце будут описаны ниже). Небольшой размер индексной карточки служит хорошей оценкой примерной сложности отдельного фрагмента — компоненты, которой приписывается больше задач, чем может поместиться на ее карточке, вероятно, является излишне сложной, и должно быть найдено более простое решение. Может быть, следует пересмотреть разделение обязанностей или разбить компоненту на две.

### 2.5.2. Цикл «что/кто»

Как мы заметили в начале нашего обсуждения, выделение компонент производится во время процесса мысленного представления работы системы. Часто это происходит как цикл вопросов «что/кто». Во-первых, команда программистов определяет: что требуется делать? Это немедленно приводит к вопросу: кто будет выполнять действие? Теперь программная система в значительной мере становится похожа на некую организацию, скажем, карточный клуб. Действия, которые должны быть выполнены, приписываются некоторой компоненте в качестве ее обязанностей.

Популярная наклейка от жевательной резинки утверждает, что время от времени может и должно спонтанно происходить необъяснимое. (Наклейка от жевательной резинки использует чуть более короткую фразу.) Мы знаем, однако, что в реальной жизни это вряд ли справедливо. Если происходит некоторое действие, должен быть и агент, которому предписано выполнять это действие. Точно так же как в карточном клубе каждое действие приписано определенным индивидуумам, при организации объектно-ориентированной программы каждое действие является обязанностью некоторой компоненты. Секрет хорошего объектно-ориентированного проекта состоит в том, чтобы установить агента для каждого действия.

### 2.5.3. Документирование

На этом этапе следует начать разработку документации. Два документа должны являться существенными составными частями любой программной системы: руководство пользователя и проектная документация системы. Работа над каждым из них может начинаться до того, как написана первая строчка программного кода.

Руководство пользователя описывает взаимодействие с системой с точки зрения пользователя. Это — отличное средство проверки того, что концепция команды программистов-разработчиков соответствует мнению клиента. Поскольку решения, принятые в процессе проработки сценария, соответствуют действиям, которые потребуются от пользователя при использовании программы, то написание руководства пользователя естественным образом увязывается с процессом проработки сценария.

Перед тем как написан какой-либо кусок кода, мышление команды программистов во многом похоже на сознание конечных пользователей. То есть именно на этом этапе разработчики могут наиболее легко предугадать те вопросы, на которые новичку-пользователю понадобятся ответы.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Второй существенный документ — проектная документация. Она протоколирует основные решения, принятые при планировании программы, и, следовательно, должна создаваться в тот момент, когда эти решения еще свежи в памяти создателей, а не годом позже. Зачастую много проще написать общее глобальное описание программной системы в начале разработки. Затем, естественно, совершается переход к уровню отдельных компонент или модулей.

Хотя в равной мере важно документировать программу на уровне модулей, слишком большое внимание к деталям организации каждого фрагмента сделает сложным для последующих программистов, осуществляющих сопровождение программной системы, формирование общей картины приложения.

CRC-карточки являются одним из видов проектной документации, но многие другие важные решения не отражены в них. Аргументы за и против каждой важной альтернативы при проектировании должны записываться, равно как и факторы, которые повлияли на конечное решение. Должен вестись протокол или дневник хода проекта. Как руководство пользователя, так и проектная документация уточняются и изменяются в процессе работы в точном соответствии с тем, как модифицируется собственно программа.

## 2.6. Компоненты и поведение

Вернемся к программе IIKH. Команда разработчиков решает, что когда система начинает работу, пользователь видит привлекательное информационное окно (см. рис. 2.1). Ответственность за его отображение приписана компоненте, названной Greeter. Некоторым, пока еще не определенным образом (с помощью всплывающих меню, кнопок, нажатия на клавиши клавиатуры или использования сенсорного экрана) пользователь выбирает одно из нескольких действий.
Первоначально планируется только пять действий:

1. Просмотреть базы данных с рецептами, но без ссылок на какой-то конкретный план питания.
2. Добавить новый рецепт в базу данных.
3. Редактировать или добавить комментарии к существующему рецепту.
4. Пересмотреть существующий план в отношении некоторых продуктов.
5. Создать новый план питания.

Эти действия естественным образом разбиваются на две группы. Первые три действия связаны с базой данных рецептов, последние два — с планированием питания. В результате команда принимает следующее решение: создать компоненты, соответствующие этим двум обязанностям. Продолжая прорабатывать сценарий, планирование питания на время игнорируем и переходим к уточнению действий, связанных с компонентой Recipe Database. На рис. 2.2 показан

PDF created with pdfFactory Pro trial version www.pdffactory.com

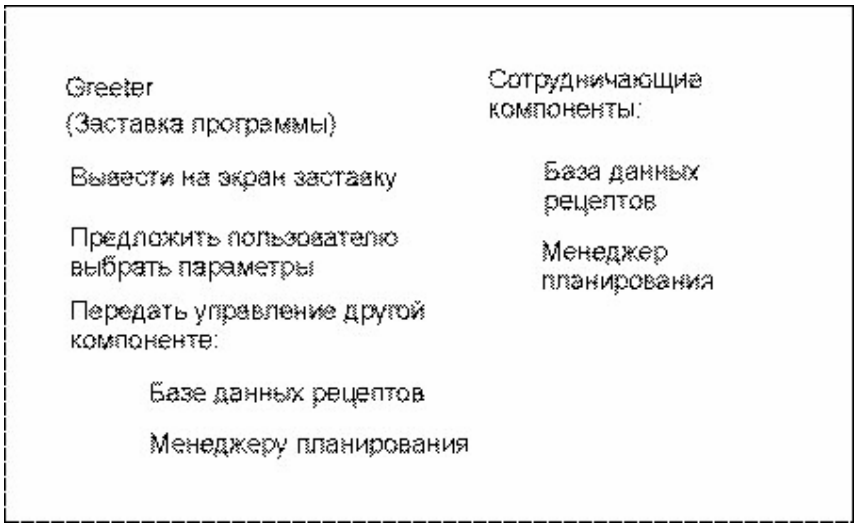


**Рис. 2.2.** CRC-карточка для класса заставки Greeter

первоначальный вид CRC-карточки для компоненты Greeter.

В широком смысле обязанность компоненты, работающей с базой данных, — просто поддерживать записи с рецептами.

Мы уже выделили три аспекта этой компоненты: Recipe Database должна обеспечивать просмотр библиотеки существующих рецептов, редактирование рецептов, включение новых рецептов в базу данных.

### 2.6.1. Отложенные решения

В конце концов придется решить, как пользователь станет просматривать базу данных. Например, должен ли он сначала входить в список категорий таких, как «супы», «салаты», «горячие блюда», «десерты»?

С другой стороны, может ли пользователь задавать ключевые слова для ограничения области поиска, включая список ингредиентов («миндаль», «клубника», «сыр»)? Или же использовать список предварительно заданных ключевых слов («любимые пирожные Боба»)? Следует ли применять полосы прокрутки (scroll bars) или имитировать закладки в виртуальной книжке? Размышлять об этих предметах доставляет удовольствие, но важно то, что нет необходимости принимать конкретные решения на данном этапе проектирования (см. раздел 2.6.2. «Готовность к изменениям»). Поскольку они влияют только на отдельную компоненту и не затрагивают функционирование остальных частей системы, то все, что надо для продолжения работы над сценарием, — это информация о том, что пользователь может выбрать конкретный рецепт.

### 2.6.2. Готовность к изменениям

Как было сказано, единственное, что является постоянным в жизни, — неизбежность изменений. То же самое справедливо для программного обеспечения. Независимо от того как тщательно вы пытаетесь разработать исходные спецификации и проект программной системы, почти наверняка изменения в желаниях или потребностях пользователя будут вызывать соответствующие исправления в программе (зачастую в течение всего жизненного цикла системы). Разработчики должны предвидеть это и планировать свои действия соответствующим образом:

PDF created with pdfFactory Pro trial version www.pdffactory.com

- Главная цель состоит в том, что изменения должны затрагивать как можно меньше компонент. Даже принципиальные новшества во внешнем виде или функционировании приложения должны затронуть один или два фрагмента кода.
- Старайтесь предсказать наиболее вероятные источники изменений и позвольте им влиять на возможно меньшее количество компонент программы. Наиболее общими причинами изменений являются интерфейс, форматы обмена информацией, вид выходных данных.
- Старайтесь изолировать и уменьшить зависимость программного обеспечения от аппаратуры. Например, интерфейс просмотра рецептов в вашем приложении может частично зависеть от аппаратного обеспечения системы, на которой работает программа. Последующие версии должны переноситься на различные платформы. Хороший проект должен предвидеть подобное изменение.
- Уменьшение количества связей между фрагментами программы снижает взаимозависимость между ними и увеличивает вероятность того, что каждую компоненту удастся изменить с минимальным воздействием на другие.
- Аккуратно заносите записи о процессе разработке и о дискуссиях, проводившихся вокруг принципиальных решений, в проектную документацию. Почти наверняка коллектив, отвечающий за сопровождение системы и разработку следующих версий, будет отличаться от команды, разработавшей первоначальную версию программы. Проектная документация позволит в последствии узнать о мотивах принятых решений и поможет избежать затрат времени на обсуждение вопросов, которые уже были разрешены.

### 2.6.3. Продолжение работы со сценарием

Каждый кулинарный рецепт будет идентифицироваться с конкретной программной компонентой. Если рецепт выбран пользователем, управление передается объекту, ассоциированному с рецептом. Рецепт должен содержать определенную информацию. В основном она состоит из списка ингредиентов и действий, необходимых для трансформации составляющих в конечный продукт. Согласно нашему сценарию компонента-рецепт должна также выполнять и другие действия. Например, она будет отображать рецепт на экране. Пользователь получит возможность снабжать рецепт аннотацией, а также менять список ингредиентов или набор инструкций. С другой стороны, пользователь может потребовать распечатать рецепт. Все эти действия являются обязанностью компоненты Recipe. (Временно мы продолжим описание Recipe как отдельно взятого объекта. На этапе проектирования мы можем рассматривать его как прототип многочисленных объектов-рецептов. Позднее мы вернемся к обсуждению альтернативы «одиночная компонента — множество компонент».)

Определив вчерне, как осуществить просмотр базы данных, вернемся к ее блоку управления и предположим, что пользователь хочет добавить новый рецепт. В блоке управления базой данных неким образом определяется, в какой раздел поместить новый рецепт (в настоящее время детали нас не интересуют), запрашивается имя рецепта и выводится окно для набора текста. Таким образом, эту задачу естественно отнести к той компоненте, которая отвечает за редактирование рецептов.

Вернемся к блоку Greeter. Планирование меню, как вы помните, было поручено программной компоненте Plan Manager. Пользователь должен иметь возможность сохранить существующий план. Следовательно, компонента Plan Manager может запускаться либо в результате открытия уже существующего плана, либо при создании нового. В последнем случае пользователя необходимо попросить ввести временные интервалы (список дат) для нового плана. Каждая дата ассоциируется с отдельной

PDF created with pdfFactory Pro trial version www.pdffactory.com

компонентой типа Date. Пользователь может выбрать конкретную дату для детального исследования — в этом случае управление передается соответствующей компоненте Date. Компонента Plan Manager должна уметь распечатывать меню питания на планируемый период. Наконец, пользователь может попросить компоненту Plan Manager сгенерировать список продуктов на указанный период.

В компоненте Date хранятся следующие данные: список блюд на соответствующий день и (необязательные) текстовые комментарии, добавленные пользователем (информация о днях рождения, юбилейные даты, напоминания и т. д.). Что должна уметь компонента? Прежде всего выводить на экран вышеперечисленные данные. Кроме того, в ней должна быть предусмотрена функция печати. В случае желания пользователя более детально ознакомиться с тем или иным блюдом, следует передать управление компоненте Meal.

В компоненте Meal хранится подробная информация о блюде. Не исключено, что у пользователя окажется несколько рецептов одного блюда. Поэтому необходимо добавлять и удалять рецепты. Кроме того, желательно иметь возможность распечатать информацию о том или ином блюде. Разумеется, должен быть обеспечен вывод на экран. Пользователю, вероятнее всего, захочется обратиться к еще каким-нибудь рецептам — следовательно, необходимо наладить контакт с базой данных рецептов. Раз так, компоненты Meal и база данных должны взаимодействовать между собой.

Далее команда разработчиков продолжает исследовать все возможные сценарии. Необходимо предусмотреть обработку исключительных ситуаций. Например, что происходит, если пользователь задает ключевое слово для поиска рецепта, а подходящий рецепт не найден? Как пользователь сможет прервать действие (например, ввод нового рецепта), если он не хочет продолжать дальше? Все это должно быть изучено. Ответственность за обработку подобных ситуаций следует распределить между компонентами.

Изучив различные сценарии, команда разработчиков в конечном счете решает, что все действия могут быть надлежащим образом распределены между шестью компонентами (рис. 2.3). Компонента Greeter взаимодействует только с Plan Manager и Recipe Database. Компонента Plan Manager «зацепляется» только с Date, а та в свою очередь — с Meal. Компонента Meal обращается к Recipe Manager и через посредство этого объекта к конкретным рецептам.

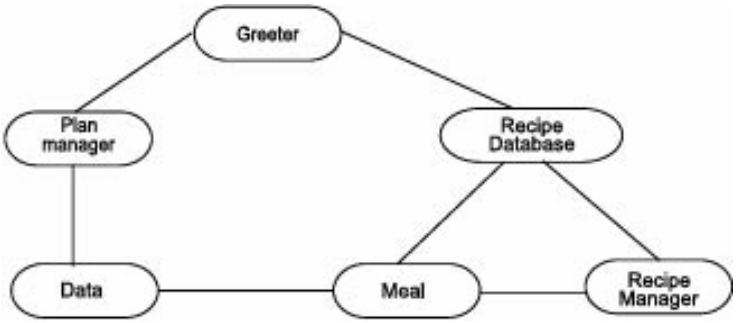



**Рис. 2.3.** Взаимосвязь между компонентами программы IIKH

### 2.6.4. Диаграммы взаимодействия

Схема, изображенная на рис. 2.3, хорошо подходит для отображения статических связей между компонентами. Но она не годится для описания динамического взаимодействия во время выполнения программы. Для этих целей используются диаграммы взаимодействия.

PDF created with pdfFactory Pro trial version www.pdffactory.com

На рис. 2.4 показана часть диаграммы взаимодействия для программы IIKH. Время движется сверху вниз. Каждая компонента представлена вертикальной линией. Сообщение от одной компоненты к другой изображается горизонтальной стрелкой между вертикальными линиями. Возврат управления (и, возможно, результата) в компоненту представлен аналогичной стрелкой. Некоторые авторы используют для этой цели пунктирную стрелку. Комментарии справа от рисунка более подробно объясняют взаимодействие.

Благодаря наличию оси времени диаграмма взаимодействия лучше описывает последовательность событий в процессе работы программы. Поэтому диаграммы взаимодействия являются полезным средством документирования для сложных программных систем.

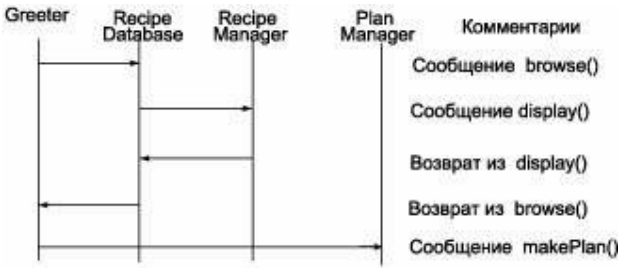



**Рис. 2.4.** Пример диаграммы взаимодействия

## 2.7. Компоненты программы

В этом разделе мы исследуем компоненты программы более подробно. За этим внешне простым понятием прячется много нетривиальных аспектов (что, впрочем, справедливо почти для всех понятий за исключением совсем элементарных).

### 2.7.1. Поведение и состояние

Мы уже видели, что компоненты характеризуются своим поведением, то есть тем, что они должны делать. Но компоненты также хранят определенную информацию. Возьмем, к примеру, компоненту-прототип Recipe из программы IIKH. Можно представить ее себе как пару «поведение—состояние».

- Поведение компоненты — это набор действий, ею осуществляемых. Полное описание поведения компоненты иногда называют протоколом. Например, в протоколе, поддерживаемом компонентой Recipe, значится, что она осуществляет редактирование инструкций по приготовлению блюд, отображает их на экране, распечатывает рецепты.
- Говоря о состоянии компоненты, имеют в виду ее внутреннее содержание. Для Recipe состояние включает в себя ингредиенты и инструкции по приготовлению блюд. Состояние не является статическим и может изменяться с течением времени. Например, редактируя текст, пользователь изменяет состояние рецепта.

Не все компоненты обязаны иметь состояние. Например, у компоненты Greeter, вероятно, не будет внутренних данных, поскольку ей ни к чему помнить какую-либо информацию. Однако большинство компонент характеризуется и поведением, и состоянием.

### 2.7.2. Экземпляры и классы

PDF created with pdfFactory Pro trial version www.pdffactory.com

Разделив понятия о состоянии и поведении, мы можем теперь затронуть тему, которую ранее избегали. Вероятно, в реальном приложении будет много рецептов. Однако все они будут вести себя одинаково. Отличается только состояние: список ингредиентов и инструкций по приготовлению. На ранних стадиях разработки нас должно интересовать поведение, общее для всех рецептов. Детали, специфические для отдельного рецепта, не важны.

Термин класс используется для описания множества объектов с похожим поведением. Мы увидим в последующих главах, что класс применяется как синтаксический механизм почти во всех объектно-ориентированных языках. Конкретные представители класса называются экземплярами. Заметим, что поведение ассоциировано с классом, а не с индивидуальными представителями. То есть все экземпляры класса воспринимают одни и те же команды и выполняют их сходным способом. С другой стороны, состояние является индивидуальным. Мы видим это на примере различных экземпляров класса Recipe. Все они могут выполнять одни и те же действия (редактирование, вывод на экран, печать), но используют различные данные. Мы рассмотрим концепцию класса более подробно в главе 3.

### 2.7.3. Зацепление и связность

Двумя важными понятиями при разработке программ являются зацепление (coupling) и связность (cohesion). Связность — это мера того, насколько отдельная компонента образует логически законченную, осмысленную единицу. Высокая связность достигается объединением в одной компоненте соотносящихся (в том или ином смысле) друг с другом функций. Наиболее часто функции оказываются связанными друг с другом при необходимости иметь доступ к общим данным. Именно это объединяет разные части компоненты Recipe.

С другой стороны, зацепление характеризует взаимозависимость между компонентами программы. В общем случае желательно уменьшить степень зацепления как только возможно, поскольку связи между компонентами программы препятствуют их модификации и мешают дальнейшей разработке или повторному использованию в других программах.

В частности, зацепление возникает, если одна программная компонента должна иметь доступ к данным (состоянию) другой компоненты. Следует избегать подобных ситуаций. Возложите обязанность осуществлять доступ к данным на компоненту, которая ими владеет. Например, в случае с нашим проектом кажется, что ответственность за редактирование рецептов должна лежать на компоненте Recipe Database, поскольку именно в ней впервые возникает в этом необходимость. Но тогда объект Recipe Database должен напрямую манипулировать состоянием отдельных рецептов (их внутренними данными: списком ингредиентов и инструкциями по приготовлению). Лучше избежать столь тесного зацепления, передав обязанность редактирования непосредственно рецепту.

Более подробно о связности и зацеплении, а также о соответствующей технике программирования рассказывается в главе 17.

### 2.7.4. Интерфейс и реализация модуля — принципы Парнаса

Идея характеризации компонент программы через их поведение имеет одно чрезвычайно важное следствие. Программист знает, как использовать компоненту, разработанную другим программистом, и при этом ему нет необходимости знать, как она реализована.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Например предположим, что шесть компонент приложения IIKH создаются шестью программистами. Программист, разрабатывающий компоненту Meal, должен обеспечить просмотр базы данных с рецептами и выбор отдельного рецепта при составлении блюда. Для этого компонента Meal просто вызывает функцию browse, привязанную к компоненте Recipe Database. Функция browse возвращает отдельный рецепт Recipe из базы данных.

Все это справедливо вне зависимости от того, как конкретно реализован внутри Recipe Database просмотр базы данных.

Мы прячем подробности реализации за фасадом интерфейса. Происходит маскировка информации. Говорят, что компонента инкапсулирует поведение, если она умеет выполнять некоторые действия, но подробности, как именно это делается, остаются скрытыми. Это естественным образом приводит к двум различным представлениям о программной системе. Вид со стороны интерфейса — это лицевая сторона; ее видят другие программисты. В интерфейсной части описывается, что умеет делать компонента. Вид со стороны реализации — это «изнанка», видимая только тем, кто работает над конкретной компонентой. Здесь определяется, как компонента выполняет задание.

Разделение интерфейса и реализации является, возможно, наиболее важной идеей в программировании. Ее непросто объяснить студентам. Маскировка информации имеет значение в основном только в контексте программных проектов, в которых занято много людей. При таких работах лимитирующим фактором часто является не количество привлеченных людей, а частота обмена информацией и данными как между программистами, так и между разрабатываемыми ими программными системами. Как будет показано ниже, компоненты часто разрабатываются параллельно разными программистами в изоляции друг от друга.

Интерес к многократному использованию программных компонент общего назначения в разных проектах возрастает. Для осуществления подобного, связи между различными частями системы должны быть минимальны и прозрачны.

Как мы уже отмечали в предыдущей главе, эти идеи были сформулированы специалистом по информатике Дэвидом Парнасом в виде правил, часто называемых принципами Парнаса:

- Разработчик программы должен предоставить пользователю всю информацию, которая нужна для эффективного использования приложения, и ничего кроме этого.
- Разработчик программного обеспечения должен знать только требуемое поведение компоненты и ничего кроме этого.

Следствие принципа отделения интерфейса от реализации состоит в том, что программист может экспериментировать с различными алгоритмами, не затрагивая остальные компоненты программы.

## 2.8. Формализация интерфейса

Продолжим разработку программы IIKH. На следующих нескольких этапах уточняется описание компонент. Сначала формализуются способы взаимодействия.

Следует определить, как будет реализована каждая из компонент. Компонента, характеризуемая только поведением (не имеющая внутреннего состояния), может быть

PDF created with pdfFactory Pro trial version www.pdffactory.com

оформлена в виде функции. Например, компоненту, заменяющую все заглавные буквы в текстовой строке на строчные, разумнее всего сделать именно так. Компоненты с многими функциями лучше реализовать в виде классов. Каждой обязанности, перечисленной на CRC-карточке компоненты, присваивается имя. Эти имена станут затем названиями функций или процедур. Вместе с именами определяются типы аргументов, передаваемых функциям. Затем описывается (вся) информация, содержащаяся внутри компоненты. Если компоненте требуются некие данные для выполнения конкретного задания, их источник (аргумент функции, глобальная или внутренняя переменная) должен быть явно описан.

### 2.8.1. Выбор имен

Имена, связанные с различными действиями, должны тщательно подбираться. Шекспир сказал, что переименование не меняет сути объекта [1], но определенно не все имена будут вызывать в воображении слушателя одинаковые мысленные образы.

Как давно известно правительственным чиновникам, неясные и используемые в переносном смысле имена придают отпугивающий вид даже простейшим действиям. Выбор удобных имен необычайно важен. Они должны быть внутренне совместимы, значимы, коротки, содержательны. Часто значительное время тратится на нахождение правильного набора имен для выполняемых заданий и объектов. Являясь далеко не бесплодным и не бесполезным процессом, надлежащий выбор имен на ранней стадии проектирования значительно упрощает и облегчает дальнейшую разработку.

Были предложены следующие положения общего характера, регулирующие этот процесс [Keller 1990]:

- Используйте имена, которые можно произнести вслух. Основное правило: если вы не можете громко прочитать имя, забудьте о нем.
- Применяйте заглавные буквы или символы подчеркивания для того, чтобы отметить начало нового слова в составном имени: CardReader или Card_reader вместо нечитаемого cardreader.
- Тщательно проверяйте сокращения. Сокращение, ясное для одного человека, может быть загадочным для другого. Обозначает ли имя TermProcess последний процесс в цепочке (terminal process), или нечто, что прекращает выполнение процесса (terminate process), или же процесс, связанный с терминалом компьютера?
- Избегайте многозначности имен. Имя empty для функции — обозначает ли оно проверку того, что некоторый объект пуст, или же она удаляет все значения из объекта (делает его пустым)?
- Не используйте цифры в именах. Их легко спутать с буквами (0 как O, 1 как l, 2 как Z, 5 как S).
- Присваивайте функциям, которые возвращают логические (булевские) значения, такие имена, чтобы было ясно, как интерпретировать true и false. Например, PrinterIsReady ясно показывает, что значение true соответствует принтеру в рабочем состоянии, в то время как PrinterStatus является гораздо менее точным.
- Дайте дорогостоящим (с точки зрения компьютерных ресурсов) и редко используемым операциям уникальные, четко выделяемые имена. При таком подходе уменьшается вероятность использования «не тех» функций.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Как только для всех действий выбраны имена, CRC-карточка для каждой компоненты переписывается заново с указанием имен функций и списка формальных аргументов. Пример CRC-карточки для компоненты Date приведен на рис. 2.5. Что осталось не установленным, так это то, как именно каждая компонента будет выполнять указанные действия.

---

[1] «Что значит имя? Роза пахнет розой, хоть розой назови ее, хоть нет. Ромео под любым названьем был бы тем верхом совершенств, какой он есть». — Вильям Шекспир, «Ромео и Джульетта», действие II, сцена 2 (пер. Бориса Пастернака).

Необходимо еще раз «прокрутить» сценарий более детально, чтобы гарантировать, что все действия учтены и вся необходимая информация имеется и доступна для соответствующих компонент.


**Рис. 2.5.** Обновленная CRC-карточка для компоненты Date

## 2.9. Выбор представления данных

На данном этапе, если только это не было сделано раньше, происходит разделение команды разработчиков на группы, каждая из которых отвечает за конкретные компоненты программы. Задача теперь состоит в переходе от описания компоненты к конкретному коду. Главное здесь — проектирование структур данных, которые будут использоваться каждой из подсистем для хранения внутренней информации, необходимой для выполнения предписанных обязанностей.

Именно на этом этапе в игру вступают классические структуры данных, используемые в информатике. Выбор структуры данных является важным, центральным моментом с точки зрения проектирования. Если представление данных выбрано правильно, то код, используемый компонентой при выполнении ее обязанностей, становится почти самоочевидным.

Структуры данных должны точно соответствовать рассматриваемой задаче. Неправильный выбор структуры может привести к сложным и неэффективным программам.

На этом же этапе описание поведения компонент должно быть преобразовано в алгоритмы. Реализованные функции затем сопоставляются с потребностями компоненты,

PDF created with pdfFactory Pro trial version www.pdffactory.com

являющейся клиентом данного фрагмента, чтобы гарантировать, что все ее запросы оказываются выполненными и что все необходимые для ее работы данные являются доступными.

## 2.10. Реализация компонент

Когда проект в целом определен и разбит на подсистемы, следующим шагом является реализация компонент. Если предыдущие этапы были выполнены корректно, каждая обязанность или поведение будут кратко охарактеризованы. Задачей данного этапа является воплощение желаемых действий на компьютерном языке. В следующем разделе мы опишем некоторые из наиболее типичных эвристических подходов, используемых с этой целью.

Если это не было сделано ранее (например, как часть этапа спецификации всей системы), то теперь можно решить, как будут устроены внутренние детали отдельных компонент. В случае нашего примера на данном этапе следует подумать, как пользователь будет просматривать базу данных рецептов.

По мере того как программные проекты с большим числом разработчиков становятся нормой, все реже встречается ситуация, когда один–единственный программист отвечает за всю систему. Наиболее важные для программиста качества — это способность понимать, как отдельный фрагмент кода подключается к более высокому программному уровню, и желание работать совместно с остальными членами команды.

Часто в процессе реализации одной компоненты становится ясно, что некоторая информация или действия должны быть присвоены совсем другой компоненте, которая работала бы «за сценой», не видимо для пользователя. Такие компоненты иногда называют суфлерами. Мы встретим соответствующие примеры в некоторых последующих главах.

Важной частью анализа и кодирования на этом этапе является полная характеризация и документирование необходимых предварительных условий, которые требуются программной компоненте для выполнения задания. Также следует проверить, правильно ли работает программная компонента, если вызвать ее с правильными входными значениями. Это подтвердит корректность алгоритмов, использованных при реализации компоненты.

## 2.11. Интеграция компонент

Когда индивидуальные подсистемы разработаны и протестированы, они должны быть интегрированы в конечный продукт. Это делается поэтапно. Начиная с элементарной основы, к системе постепенно добавляются (и тестируются) новые элементы. При этом для еще не реализованных частей используются так называемые заглушки (stubs) — подпрограммы без какого-либо поведения или с ограниченной функциональностью.

Например, при разработке программы IIKH было бы разумным начать интегрирование с компоненты Greeter. Чтобы протестировать ее в изоляции от остальных блоков программы, потребуются заглушки для управляющего кода базы данных с рецептами Recipe Database и блока управления планированием питания Meal Plan. Заглушки просто должны выдавать информационные сообщения и возвращать управление. Таким образом, команда разработчиков компоненты Greeter сможет протестировать различные аспекты

PDF created with pdfFactory Pro trial version www.pdffactory.com

данной компоненты (например, проверить, вызывает ли нажатие клавиши нужную реакцию). Отладку отдельных компонент часто называют тестированием блоков.

Затем заглушки заменяются более серьезным кодом. Например, вместо заглушки для компоненты Recipe Database можно вставить реальную подсистему, сохранив заглушки для остальных фрагментов. Тестирование продолжается до тех пор, пока не станет ясно, что система работает правильно. Этот процесс называют тестированием системы в целом.

Когда все заглушки заменены работающими компонентами, приложение завершено. Процесс тестирования заметно облегчается, если число связей между компонентами невелико — в этом случае не придется писать множество программ-заглушек.

Во время интеграции системы вполне возможно, что ошибка, проявляющаяся в одной из программных систем, вызвана некорректным кодом в другом фрагменте. Тем самым ошибки, выявляемые в процессе интеграции, приводят к необходимости исправлять некоторые компоненты. Вслед за этим измененные компоненты должны вновь тестироваться изолированно перед очередной попыткой интеграции. Повторная прогонка разработанных ранее тестовых примеров, выполняемая после изменений в компоненте программы, иногда называется регрессионным тестированием (regression testing).

## 2.12. Сопровождение и развитие

Хотелось бы, чтобы с передачей пользователю функционирующего приложения, работа команды разработчиков завершалась. К сожалению, такого почти никогда не происходит. Необходимо дополнительное сопровождение программного обеспечения. Вот некоторые причины, вызывающие его неизбежность:

- В переданном продукте могут быть обнаружены ошибки. Они должны быть исправлены либо через «заплатки» (patches) к существующей версии, либо в новой версии.
- Изменение требований — возможно, из-за новых государственных постановлений или стандартизации.
- Переход на другое аппаратное обеспечение. Например, в результате переноса системы на другие платформы или поддержки новых устройства ввода (световое перо или сенсорный экран). Может измениться технология вывода: скажем, вы перешли от текстового интерфейса к графическому.
- Изменение запросов пользователей. Пользователи могут требовать увеличения возможностей программы, снижения цены, более простого использования и т. д. Как правило, такие «повышенные» требования диктуют конкурирующие продукты.
- Потребность в улучшенной документации.

Хороший проект предусматривает неизбежность изменений и подготавливает их с самого начала.

## Упражнения

1. Опишите распределение обязанностей в организации, которая включает по крайней мере шесть членов. Рассмотрите учебное заведение (студенты, преподаватели, директор, гардеробщик), фирму (совет директоров, президент, рабочий) и клуб (президент, вице-президент, рядовой член). Опишите обязанности каждого члена организации и его сотрудников (если они есть).

PDF created with pdfFactory Pro trial version www.pdffactory.com

2. Создайте с помощью диаграммы взаимодействия сценарий для организации из упражнения 1.
3. Для типичной карточной игры опишите программную систему, которая будет взаимодействовать с пользователем в качестве противоположного партнера. Типичные компоненты должны включать игровой стол и колоду карт.
4. Опишите программную систему для управления ATM (Automatic Teller Machine). Поскольку слово Teller достаточно многозначно (рассказчик, счетчик голосов при выборах, кассир в банке, диктор радиолокационной станции ПВО и т. д.), то у вас имеется большая свобода в выборе предназначения этой машины. Приведите диаграммы взаимодействия для различных сценариев использования этой машины.

# *Глава 3 : Классы и методы*

Хотя термины, которые используются в объектно-ориентированных языках, отличаются, понятия классов, экземпляров, пересылки сообщений, методов и наследования являются общими. Эти термины были введены в главе 1. Как уже отмечалось, использование различных терминов для одних и тех же понятий широко распространено в объектно-ориентированных языках. Мы будем применять единую и, как мы надеемся, ясную терминологию для всех языков программирования. В вводных разделах будем отмечать специфику конкретных языков и различные синонимы для наших терминов. Обращайтесь к разделу "Глоссарий" за разъяснением незнакомых понятий.

В этой главе речь идет о статических атрибутах классов; в главе 4 мы рассмотрим их динамическое использование. Здесь же мы проиллюстрируем механизмы объявления класса и определения методов. В главе 4 объясняется, как создаются экземпляры класса и как им передаются сообщения. Мы отложим анализ механизмов наследования до главы 7.

Важный момент, который необходимо прояснить, — различие между объявлением класса или объекта и его порождением. В первом случае (который является темой этой главы) просто указывается тип данных. Объявление характеризует объект (внутренние переменные, типы поведения), но само по себе не создает новых данных. Порождение, то есть создание новых экземпляров класса, может рассматриваться как разновидность объявления переменной и будет являться темой следующей главы. Различие между этими двумя понятиями отчасти маскируется тем фактом, что в языках с контролем типов данных (таких, как C++) определение, порождающее переменную, выглядит так же, как и чистое объявление.

## 3.1. Инкапсуляция

В главе1 мы заметили, что объектно-ориентированное программирование и, в особенности, объекты могут рассматриваться со многих точек зрения. В этой главе мы будем представлять себе объекты как абстрактные типы данных.
В программировании, основанном на абстракции данных, информация сознательно прячется в небольшой части программы. В частности, каждый объект из набора абстрактных типов данных, разрабатываемого программистом, имеет два "лица". Это аналогично дихотомии принципов Парнаса, которые обсуждались в главе 2. С внешней точки зрения (клиента или пользователя) абстрактный тип данных представляет собой всего лишь совокупность операций, которые определяют поведение абстракций. С

PDF created with pdfFactory Pro trial version www.pdffactory.com

противоположной стороны, за фасадом интерфейса, программист, который определяет абстрактный тип, видит значения переменных, которые используются для поддержки внутреннего состояния объекта.
Например, для абстрактного типа данных stack пользователь видит только описание допустимых операций — скажем, push, pop, top. С другой стороны, программисту, реализующему stack, необходимо манипулировать с конкретными структурами данных (рис. 3.1). Конкретные детали инкапсулированы в более абстрактный объект.

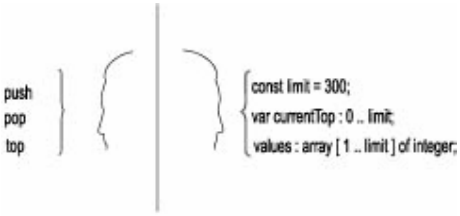



**Рис. 3.1.** Интерфейс и реализация для типа stack

Мы использовали термин экземпляр для обозначения представителя класса. Соответственно мы будем использовать термин переменная экземпляра для обозначения внутренней переменной, содержащейся в экземпляре. Каждый экземпляр имеет свою собственную совокупность переменных. Эти значения не должны изменяться клиентами напрямую, а только с помощью методов, ассоциированных с классами.

Объект является, таким образом, комбинацией состояния и поведения. Состояние описывается переменными экземпляра, в то время как поведение характеризуется методами. Снаружи клиенты могут узнать только о поведении объектов. Изнутри доступна полная информация о том, как методы обеспечивают необходимое поведение, изменяют состояние и взаимодействуют с другими объектами.

## 3.2. Разновидности классов

Классы в объектно-ориентированном программировании имеют несколько различных форм и используются для разных целей. Следующие категории охватывают большую часть классов:

- управление данными;
- источники данных или посредники в передаче данных;
- классы для просмотра данных;
- вспомогательные, или упрощающие проектирование, классы.

Этот список не является исчерпывающим, однако он вполне подходит для учебных целей.

Большинство объектно-ориентированных приложений включают как классы вышеперечисленных категорий, так и другие. Если оказывается, что класс "разрывается" между двумя категориями, то зачастую его можно разбить на два класса.

Классы-администраторы данных Data Managers, часто получающие имена Data или State, — это классы, основной обязанностью которых является поддержка данных или информации о состоянии чего-либо. Например, для абстрактной модели игры в карты основная задача класса Card состоит в том, чтобы хранить масть и ранг (достоинство) карты. Классы-администраторы данных обычно являются фундаментальными строительными блоками проекта, а их прототипами в спецификации проекта являются существительные.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Источники данных Data Sources — это классы, которые генерируют данные (например, случайные числа). Посредники при передаче данных Data Sinks, естественно, служат для приема и дальнейшей передачи данных (например, запись в файл). В отличие от администраторов данных, источники и посредники не хранят внутри себя данные в течение неопределенного времени, но генерируют их по запросу (источники данных) или обрабатывают их при вызове (посредники данных).

Классы для просмотра данных View и Observer также незаменимы практически в любом приложении. Все программы так или иначе осуществляют вывод информации (как правило, на экран). Соответствующий программный код нередко является сложным, часто модифицируется и в значительной степени не зависит от выводимых данных. Поэтому хорошим тоном в программировании считается изоляция внутренних данных от методов, осуществляющих вывод информации.

Полезно отделять собственно объект (называемый часто моделью) от его изображения (визуального представления). Благодаря этому принципу системы, обеспечивающие графический вывод информации, в значительной степени могут быть упрощены. В идеальном случае модель не требует и не содержит информации о своем визуальном представлении. Это упрощает многократное использование кода, поскольку одна и та же модель может применяться во многих приложениях. Модель зачастую имеет более одного визуального представления. Например, финансовая информация может быть представлена в виде гистограмм, круговых диаграмм, таблиц или рисунков. При этом сама информация остается неизменной.

К вспомогательным классам (Facilitator и Helper) разумно отнести те классы, которые не содержат полезной информации, но облегчают выполнение сложных заданий. Например, при отображении игральной карты мы используем вспомогательный класс, рисующий линии и текст на устройстве вывода. Другой служебный класс может, например, обслуживать связный список карт (колоду).

### 3.3. Пример: игра в карты

Мы используем программную абстракцию типичной игры в карты, чтобы познакомить вас с различными объектно-ориентированными языками программирования, которые рассматриваются в этой книге. В следующей главе мы используем разработанный здесь класс Card для написания пасьянса "косынка". Класс Card, подобно настоящим игральным картам, мало что знает о своем предполагаемом использовании и может применяться в карточной игре любого типа.

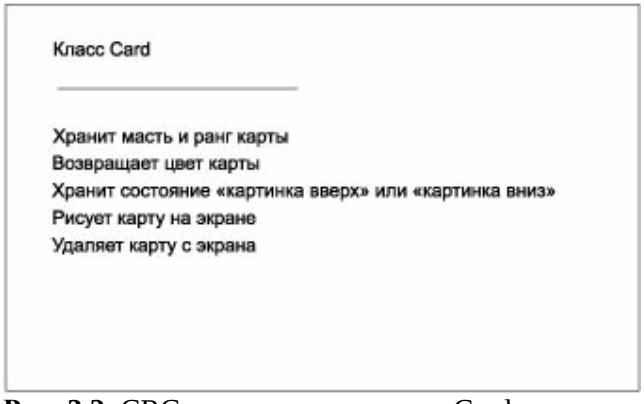



**Рис. 3.2.** CRC-карточка для класса Card

PDF created with pdfFactory Pro trial version www.pdffactory.com

На рис. 3.2 показана CRC-карточка, которая описывает поведение игральной карты. Обязанности класса Card очень ограничены. В своей основе он является просто администратором данных, который хранит и возвращает значения масти и ранга, а также рисует карту.

Как мы отмечали в главе 2, CRC-карточки много раз уточняются и перерисовываются, медленно эволюционируя от естественного языка к программному коду. Как мы помним, на следующем этапе каждый метод снабжается именем и списком аргументов. Не исключено, что описание вылезет за карточку, и тогда их придется скреплять скрепками (имеет смысл вообще отказаться от карточек, заменив их чем-то вроде отчетов).

CRC-карточка, изображенная на рис. 3.3, соответствует следующему этапу. Заметьте, что даже если обязанность состоит всего лишь в возврате значения (например, признака "картинка вверх"), мы все равно определяем функцию для посредничества в выполнении запроса. Имеются как практические, так и теоретические соображения в пользу этого. Мы вернемся к ним в главе 17.

Как было предложено ранее, можно выделить и записать на оборотной стороне CRC-карточки значения данных, которые должны содержаться в каждом экземпляре класса игральной карты. Следующий этап состоит в переводе поведения и состояния, описанных на CRC-карточке, в выполняемый код. Мы рассмотрим этот этап после того, как исследуем дихотомию между объявлением и определением.

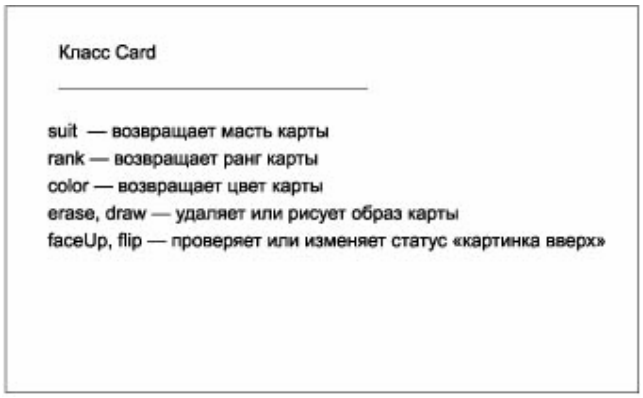



**Рис. 3.3.** Уточнение CRC-карточки для класса Card

## 3.4. Интерфейс и реализация

В главе 1 мы рассказали об истории объектно-ориентированного программирования, о том, что оно основывается на предшествующих идеях модульности и маскировки информации. В процессе эволюции некоторые соображения и концепции были отброшены, когда оказалось, что они расходятся с концепцией объектно-ориентированного проектирования, но другие понятия сохранились. В частности, принципы Парнаса применимы к объектно-ориентированной технологии в той же мере, как и к модульному подходу. Мы можем следующим образом перефразировать идеи Парнаса в терминах объектов:

- Объявление класса должно обеспечивать клиента всей информацией, необходимой для успешной работы с классом, и никакой другой.
- Методам должна быть доступна вся информация, необходимая для выполнения их обязанностей, и никакая другая.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Принципы Парнаса делят мир объекта на две части. Имеется внешний образ, наблюдаемый пользователем объекта, — мы будем называть это представление об объекте интерфейсом (interface), поскольку оно описывает, как объект взаимодействует с внешним миром. Обратная сторона объекта связана с его реализацией (implementation). Пользователю разрешен доступ только к тому, что описано в интерфейсной части. Реализация определяет, как достигается выполнение обязанностей, заявленных в интерфейсной части.

За исключением языка Smalltalk все языки программирования, которые мы рассматриваем, поддерживают разбиение класса на блок интерфейса и блок реализации. Мы опишем соответствующий механизм в разделах, посвященных особенностям каждого языка. Заметьте, что разделение интерфейса и реализации не является в точности инкапсуляцией данных, рассмотренной ранее. Первое является абстрактным понятием, второе — механизмом его воплощения. Другими словами, модули используются в процессе реализации объектов, принадлежащих к абстрактным типам данных, но модули сами по себе не являются абстрактными типами данных.

## 3.5. Классы и методы в ООП

В следующих разделах подробно описывается механизм определения классов и методов для каждого из языков программирования, которые мы рассматриваем. Обратите внимание, что некоторые объектно-ориентированные языки рассматривают классы как специализированную форму записей, в то время как другие языки используют иные подходы.

### 3.5.1. Классы и методы в языке Object Pascal

По крайней мере два различных языка носят имя Object Pascal. Исходным является язык, созданный Ларри Теслером из компании Apple Computer [Tesler 1985]. Язык был построен на основе модулей из языка Apple Pascal. Второй вариант языка Object Pascal первоначально назывался Turbo Pascal [Turbo 1988, O'Brian 1989]. Его разработала и распространяла компания Borland International. Первый упомянутый язык очень часто встречается на компьютерах Macintosh, второй большей частью связан с IBM PC. Язык, созданный компанией Borland, вновь привлек внимание к Object Pascal. Сейчас этот язык используется в качестве фундамента в среде Delphi для разработки Windows-приложений [Borland 1995]. В него были введены новые свойства, отсутствовавшие в исходном языке Turbo Pascal. В данной книге мы постараемся описать обе версии языка, отмечая особо, где и в чем они различаются.

В языке Object Pascal модуль называется библиотекой процедур (unit). В отличие от языков C++ и Objective-C библиотека процедур содержится в едином файле, а не разбивается на два. Тем не менее библиотека процедур состоит из интерфейса (interface) и реализации (implementation). Библиотека процедур может подключать другие библиотеки. Этот процесс делает доступными свойства, описанные в разделе интерфейса подключаемой библиотеки.

Часть библтотеки для класса Card в языке Object Pascal версии фирмы Apple показана в листинге 3.1. Раздел interface аналогичен описаниям функций в Pascal. Он может содержать подразделы, обозначаемые ключевыми словами const, type и var. Здесь же задаются необъектные типы данных (такие, как перечисляемые типы suits и colors).

PDF created with pdfFactory Pro trial version www.pdffactory.com

Описание класса напоминает запись (record), за исключением того, что класс может содержать заголовки процедур и функций наряду с полями данных. Последние должны быть перечислены перед объявлениями функций. Поле данных и методы должны иметь разные имена, поэтому поле данных называется suitValue, а функция — suit. В одной библиотеке можно определить несколько классов.

Описание класса изучается пользователями намного чаще, чем собственно код. По этой причине для облегчения понимания в описании используются комментарии. Описания данных должны отделяться от описания методов. Методы группируются в соответствии с абстрактной классификацией их поведения. В пределах каждой группы методы можно перечислять в алфавитном порядке. Полезно использовать табуляцию, это поможет пользователю быстро найти имена методов.

**Листинг 3.1.** Интерфейсный раздел библиотеки для языка Object Pascal фирмы Apple

```
unit card;
interface
type
  suits = (Heart, Club, Diamond, Spade);
  colors = (Red, Black);
  Card = object
      (* поля данных *)
      suitValue: suits;
      rankValue: integer;
      faceUp          : boolean;
  (* инициализация *)
  procedure setRankAndSuit (c : integer; s : suits);
  (* рабочие функции *)
  function  color  : colors;
  procedure draw (win : windows; x, y : integer);
  function  faceUp : boolean;
  procedure flip;
  function  rank   : integer;
  function  suit   : suits;
end;
implementation
    ...
end.
```

**Листинг 3.2.** Интерфейсный раздел библиотеки для языка Deplhi Pascal

```
implementation
const
  CardWidth  = 65;
  CardHeight = 75;
  function Card.color : colors;
  begin
    case suit of
    Diamond: color:= Red;
    Heart:   color:= Red;
    Spade:   color:= Black;
    Club:    color:= Black;
    end;
  end;
...
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
end.
```

### 3.5.2. Классы и методы в языке Smalltalk

Описание языка Smalltalk почти неразрывно связано с пользовательским интерфейсом среды Smalltalk. Таким образом, объяснение того, как в языке Smalltalk создаются новые классы, должно обязательно начинаться с описания программы просмотра или броузера Smalltalk. Не только собственно броузер является достаточно сложным, но и детали его реализации отличаются для различных систем. Поэтому наше обсуждение необходимым образом будет поверхностным. Читатель, заинтересованный в более подробной информации, должен обратиться к руководству по той версии языка Smalltalk, которую он использует [Goldberg 1984, LaLonde 1990b, Korienek 1993, Smith 1995].

Для пользователя броузер представляет собой большое окно, разделенное на пять окошек — четыре маленьких и одно большое (рис. 3.4). Каждое из верхних окон имеет полосы прокрутки. Нижнее окно используется для высвечивания и редактирования информации. Броузер управляется посредством мыши, которая должна иметь три кнопки: левая кнопка используется для операций выбора и редактирования, средняя и правая кнопки вызывают меню с операциями.

Классы в языке Smalltalk сгруппированы в категории. В первом окне прокручивается список всех категорий, известных системе Smalltalk. Хотя можно создать новую категорию, для наших целей достаточно выбрать существующую категорию и сконструировать относящийся к ней новый класс. Выбор элемента "Graphics-Primitives" в первом окне приведет к выполнению двух действий: во втором окне отобразится список всех классов, относящихся к данной категории, а в большом окне редактирования появится текст сообщения, вызываемого при создании новых классов.

После редактирования с использованием мыши пользователь может заменить это сообщение на:

```
Object subclass: #Card
instanceVariableNames: 'suit rank'
classVariableNames: ''
poolDictionaries: ''
category: 'Graphics-Primitives'
```

В данный момент мы будем рассматривать это сообщение просто в качестве иллюстрации того, что Card создается как подкласс Object.

Каждый экземпляр класса Card содержит два поля данных. Как и в языке Delphi Pascal, все классы должны быть подклассами уже существующих классов. Класс Object является наиболее общим порождающим классом.

Заметим, что имена полей данных не связаны с каким-либо конкретным типом данных. Язык Smalltalk не имеет операторов объявления типа, и переменные могут принимать произвольные значения. Мы поговорим о принципиальной разнице между языками с типами данных и языками без таковых позднее, при обсуждении пересылки сообщений. В нашем классе нет переменных класса и переменных-словарей (pool variables). Эти понятия относятся к следующему уровню сложности. Переменные класса будут обсуждаться в последующих главах. Переменные-словари выходят за рамки настоящей книги.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Знак # перед словом Card идентифицирует его как символ. Наиболее важное свойство символов — однозначное соответствие между именем и значением. То есть именованные символы могут иметь различные значения, но все символы с одним именем соответствуют одному и тому же значению. Тем самым символы обычно используются как ключи или заменители категорий.

Закончив с определением характеристик нового класса, пользователь выбирает команду accept из меню операций. Теперь новый класс введен в систему, и третье окно высвечивает разрешенные операции (первоначально список пуст).

Выбор окна с группами операций активизирует последнее окно в верхней группе — в нем указаны конкретные методы. Как и со списком категорий, при выборе имени группы в четвертом окне высвечиваются существующие методы, принадлежащие группе; одновременно в нижнем окне выводится шаблон, который может редактироваться для генерирования новых методов. Подобный шаблон редактирования показан на предыдущем рисунке.

Чтобы создать новый метод, пользователь редактирует шаблон и выбирает команду accept из меню операций, когда редактирование закончено. Ниже показан метод, инициализирующий масть и ранг игральной карты.

```
setSuit: s rank: r
   " устанавливает значения переменных экземпляра suit и rank "
   suit := s.
   rank := r
```

В языке Smalltalk аргументы разделяются ключевыми словами, которые легко распознаются, поскольку оканчиваются двоеточием (идентификаторы не могут оканчиваться двоеточием). Тем самым именем метода, определенного выше, является setSuit:rank:. Метод имеет два аргумента, которые известны ему как идентификаторы s и r. В некоторых версиях языка Smalltalk оператор присваивания записывается как стрелка , в большинстве других версий используется более традиционное обозначение :=. Наконец, можно заметить, что точка применяется в качестве разделителя операторов, и для последнего оператора ее можно опустить.

Доступ к переменным экземпляра не из методов класса в языке Smalltalk запрещен. Следовательно, мы должны определить явные функции доступа (accessor functions). Метод suit, показанный ниже, возвращает текущее значение переменной экземпляра с тем же именем:

```
   suit
" вернуть значение масти для данной карты "
suit
```

Стрелка, направленная вверх, — это то же самое, что ключевое слово return в других языках программирования. Она показывает, что следующее за ней выражение возвращается в качестве результата при выходе из метода. Заметим, что методам разрешено иметь те же имена, что и переменным экземпляра, и никакой путаницы не возникает (по крайней мере для системы — мы ничего не можем сказать о программистах).

Целые числа от 1 до 13 — это значения, представляющие ранг карты (то есть значение поля rank). Мы будем использовать символы (в смысле SmallTalk) для представления

PDF created with pdfFactory Pro trial version www.pdffactory.com

масти карты. Соответственно метод color (цвет карты) тоже будет возвращать символ в качестве результата. Следующий код описывает этот метод:

```
color
  " вернуть цвет данной карты "
 (suit = #diamond)            ifTrue:        [  #red ]
 (suit = #club)               ifTrue:        [  #black ]
 (suit = #spade)              ifTrue:        [  #black ]
 (suit = #heart)              ifTrue:        [  #red ]
```

Обратите внимание, что условные операторы в языке Smalltalk записываются так, как если бы они были сообщениями, пересылаемыми условной части (на самом деле так оно и есть). Квадратные скобки образуют то, что в Smalltalk называется blocks, их можно рассматривать как конструкцию, аналогичную блокам в языке Pascal (пара команд begin, end). (В действительности все сложнее. Фактически блок сам по себе является объектом, который пересылается в качестве аргумента вместе с сообщением ifTrue к булевскому объекту. Начинающим Smalltalk-программистам лучше проигнорировать подробности.)

### 3.5.3. Классы и методы в языке Objective-C

Язык программирования Objective-C — это объектно-ориентированное расширение директивного языка C. В качестве такового он наследует большую часть структур и методов использования C. В частности, реализация модулей основана на стандартном соглашении языка C о разделении файлов на две категории: интерфейсные файлы (обычно с расширением ".h") и файлы реализации (в языке C они обычно имеют расширение ".c", а в Objective-C —".m"). Предполагается, что пользователю класса (первая категория людей, перечисляемая в дихотомии Парнаса) требуется просмотреть только интерфейсные файлы.

Интерфейсный файл, подобный тому, что используется для нашей абстракции игральных карт (листинг 3.4), служит двум целям. Для программиста он является удобным средством документирования назначения и функционирования класса. Для системы он передает информацию о типах данных и требованиях к оперативной памяти. Иногда эти два применения оказываются в конфликте друг с другом. Например, в языке Objective-C, как и в языке Smalltalk,

**Листинг 3.4.** Интерфейсный файл класса Card на языке Objective-C

```
/*
описание интерфейса класса Card
язык программирования: Objective-C
автор: Тимоти Бадд, 1995
*/
# import <objc/Object.h>
/*  определить символьные константы для мастей */
# define     Heart                    0
# define     Club      1
# define     Diamond   2
# define     Spade     3
/* определить символьные константы для цветов */
# define     Red       0
# define     Black     1
/* интерфейс класса Card */
@ interface Card : Object
{
        int  suit;
        int  rank;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        int  faceup;
}
/* методы класса Card */
- (void)    suit: (int) s rank: (int) c;
- (int)     color;
- (int)     rank;
- (int)     suit;
- (void)    flip;
- (void)    drawAt: (int) and: (int);
@ end
```

пользователям класса не разрешается доступ к информации внутри экземпляров (то есть к внутреннему состоянию). Только связанные с классом методы могут иметь доступ или модифицировать данные экземпляра. Однако чтобы определить требуемую оперативную память, система должна знать размер каждого объекта. Тем самым переменные экземпляра описываются в интерфейсном файле не для пользователя (хотя они и обеспечивают пользователя информацией, но являются недоступными), но для компилятора.

Несколько первых строк интерфейсного файла содержат код, общий для языков C и Objective-C. Директива import аналогична директиве include из языка C, за исключением того, что она гарантирует, что файл подключается только один раз. В данном случае импортируемый файл — это описание интерфейса класса Object. Директива define задает некоторые символьные константы, которые мы будем использовать для обозначения мастей и цветов.

Символ @ обозначает начало кода, специфического для Objective-C. В данном случае код описывает интерфейс класса Card. Разрешается записывать интерфейсы для нескольких классов в одном интерфейсном файле, хотя обычно каждый класс имеет отдельный файл. В языке Objective-C, так же как и в языках Smalltalk и Delphi Pascal, каждый класс должен являться подклассом уже существующего класса; класс Object является наиболее общим порождающим классом.

Список переменных, заключенный в фигурные скобки, который следует за признаком начала класса, содержит описания переменных (данных) объекта класса. Каждый экземпляр класса имеет отдельную область данных. Язык Objective-C делает различие между традиционными значениями языка C (целыми, вещественными, структурами и т. д.) и объектами. Объекты объявляются с типом данных id.

Как и для указателей в языке C, переменная, объявленная с типом данных id, может содержать либо допустимое значение, либо специальное значение Null.

Строки, следующие за описанием данных, описывают методы, которые связаны с данным классом. Описание каждого метода начинается с символа "-" (дефис) в первой колонке, за которым может следовать выражение, аналогичное приведению типов данных в C. Оно показывает тип значения, возвращаемого методом. Тип объекта (id) предполагается по умолчанию, если не указано ничего другого. Тем самым метод suit (заметьте, что методы могут иметь те же имена, что и поля данных) возвращает значение типа integer. Метод flip описан как имеющий тип void. В языке C это является индикатором того, что возвращаемое значение отсутствует (то есть метод является процедурой, а не функцией). Точнее будет сказать, что возвращаемое значение игнорируется. Как и раньше, табуляция, комментарии и алфавитное упорядочивание делают описание более понятным.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Методы, которым требуются аргументы (вроде функции перемещения карты или метода, проверяющего попадание точки внутрь области, ограниченной полем карты), записываются в стиле языка Smalltalk с ключевыми словами, разделяющими список аргументов. Однако в отличие от языка Smalltalk каждый аргумент должен сопровождаться описанием типа данных, причем при отсутствии такого описания подразумевается тип id. Указание типа дается такой же синтаксической конструкцией, какая используется для описания типа данных результата функции.

Файл реализации (листинг 3.5) начинается с импорта интерфейсного файла для нашей абстракции игральной карты. Код языка Objective-C может свободно смешиваться с кодом обычного C. Например, в листинге 3.5 две строчки, определяющие символьные константы для длины и ширины игральной карты, используют синтаксис C.

Директива implementation определяет фактический код для методов, связанных с классом. Как имя родительского класса, так и определения переменных экземпляра в области implementation иногда опускают — они могут быть взяты из описания интерфейса.

**Листинг 3.5.** Файл реализации класса Card на языке Objective-C

```
/*
файл реализации для класса Card
язык программирования: Objective-C
автор: Тимоти Бадд, 1995
*/
# import "card.h"
# define     cardWidth   68
# define     cardHeight  75
@ implementation Card
- (int)     color
{
    return suit % 2;
}
- (int)     rank
{
    return rank;
}
- (void)    suit: (int) s rank: (int) c
{
    suit = s;
    rank = c;
    faceup = 0;
}
    // ... кое-что опущено
@ end
```

Нет необходимости, чтобы методы в области implementation следовали в том же порядке, как они были заданы в интерфейсной части. Чтобы упростить поиск конкретного метода, их часто перечисляют в алфавитном порядке. Заголовки методов повторяют интерфейсный файл, но только теперь за ними следует тело метода. Как и в языке C, тело функции заключено в фигурные скобки.

### 3.5.4. Классы и методы в языке C++

Язык C++, подобно Objecive-C, является объектно-ориентированным расширением директивного языка программирования C. Как и в C, полезно различать интерфейсные файлы (имеющие расширение ".h") и файлы реализации (расширение зависит от системы).

PDF created with pdfFactory Pro trial version www.pdffactory.com

Как и в языке Objective-C, интерфейсный файл (описывающий, например, абстракцию игральной карты) может содержать описания более чем одного класса, хотя обычно это происходит, только если классы тесно связаны. Поскольку языки C и C++ не поддерживают ключевого слова import (в отличии от Objective-C), для достижения того же эффекта используется условное подключение файла. Когда файл card.h считывается впервые, символ CARDH (предполагается, что он не встречается в других местах) является неопределенным, и тем самым срабатывает условный оператор ifndef (если не определено), так как значение CARDH действительно не определено. Значит, файл card.h будет считан. При всех последующих попытках считать этот файл символ будет известен, и загрузка файла будет пропущена.

```
# ifndef  CARDH  // файл должен читаться лишь один раз
# define CARDH
 . . .
# endif
```

Описание класса начинается с ключевого слова class (листинг 3.6). В C++ описание класса во многом напоминает структуру в языке C, за исключением того, что вместе с полями данных стоят заголовки процедур. Ключевое слово private: предшествует фрагментам кода, доступ к котором разрешен только из самого класса. Ключевое слово public: обозначает область интерфейса — то есть то, что видно извне класса. Как и в языке Objective-C, описание переменных экземпляра в области private дается в интерфейсном файле только ради компилятора (чтобы он мог определить размер памяти, требуемой для объекта), а для пользователя данного класса эти поля остаются недоступными (что является нарушением первого принципа Парнаса).

Поскольку пользователи часто интересуются открытой областью интерфейса public, эта часть должна идти первой. Как и выше, следует использовать комментарии, табуляцию, группирование и упорядочивание по алфавиту, чтобы сделать описание более читаемым.

Функция card(suit, int) в описании класса является уникальной во многих отношениях — не только потому, что ее имя совпадает с именем класса, но и потому, что у нее нет возвращаемого значения. Эта функция называется конструктором, она используется при инициализации создаваемых экземпляров класса. Мы обсудим конструкторы более подробно в главе 4.

Ключевое слово void, как и в языке Objective-C, показывает отсутствие типа. Когда оно используется как тип возвращаемого значения, это означает, что метод применяется как процедура ради побочного эффекта, а не для вычисления результата.

Методы draw и halfdraw иллюстрируют описание типов параметров как составной части объявления функции. Этот стиль декларирования называется прототипом функции. Теперь он является частью ANSI стандарта языков C и C++. Заметьте, что прототип аналогичен списку аргументов, хотя аргументы представлены как типы данных и их имена являются необязательными.

Аргумент с типом данных window, обрабатываемый функцией draw, передается через ссылку. На это указывает & в списке аргументов. Большие структуры, вроде описания окон (тип данных window в нашем примере), часто передаются через ссылку.

**Листинг 3.6.** Описание класса card на языке C++

```
enum suits {diamond, club, heart, spade};
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
enum colors {red, black};
        //              абстракция игральной карты
        //              используется в пасьянсе
        //              язык программирования: C++
        //              автор: Тимоти Бадд, 1995
class card{
    public:
    // конструктор
        card (suits, int);
    // доступ к атрибутам карты
        colors                  color                   ();
        bool                    faceUp                  ();
        int                     rank                    ();
        suits                   suit                    ();
    // выполняемые действия
        void   draw                     (window &);
void   halfdraw(window &, int x, int y);
        void   flip                     ();
    private:
        bool            faceup;
        int    r;                       // ранг
suits  s;                                       // масть
};
```

Поскольку методы рассматриваются просто как поля специального вида, принадлежащие объекту и неразличимые от полей данных, метод и поле данных не могут иметь общего имени. Тем самым переменная s хранит значение, представляющее собой масть карты, в то время как метод suit возвращает это значение. Аналогично идентификаторы r и rank нужны для хранения и для возврата ранга карты.

Файл реализации для данного класса должен обеспечить работу методов, описанных в интерфейсном файле. Начало файла реализации для нашей абстракции игральной карты показано ниже.

```
//
//  файл реализации
//  для абстракции игральной карты
//
# include "card.h"
card::card (suits sv, int rv)
{
    s = sv;             // инициализировать масть
    r = rv;             // инициализировать ранг
    faceup = true; // начальное положение — картинкой вверх
}
int card::rank()
{
    return r;
}
```

Тело метода записывается как стандартная функция языка C, но имени метода предшествует имя класса и два двоеточия. На переменные экземпляра (поля данных класса) можно ссылаться внутри метода как на обычные переменные. Комбинация имени класса и имени метода образует полное имя, она может рассматриваться как аналог имени и фамилии при идентификации личности.

Чтобы вдохновить программистов использовать такие принципы разработки программ как абстрагирование и инкапсуляция, язык программирования C++ предоставляет им возможность определять встраиваемые функции. Для того кто к ней обращается,

PDF created with pdfFactory Pro trial version www.pdffactory.com

встраиваемая функция выглядит точно так же, как и обычная, с теми же самыми синтаксическими правилами для задания аргументов. Единственная разница состоит в том, что компилятор имеет право преобразовать вызов встраиваемой функции непосредственно в код в точке ее вызова, сокращая тем самым расходы на обращение к функции и возврат управления. (Как и в случае директивы register, inline-реализация является пожеланием, компилятор имеет право его проигнорировать.)

```
inline int Card::rank()
{
   return r;
}
```

Абстрагирование и инкапсуляция часто способствуют появлению большого количества функций, которые выполняют незначительную часть работы и, следовательно, имеют небольшой размер. Определяя их как встраиваемые функции, программист может сохранить выгоды инкапсуляции и избежать затрат на вызов функций во время выполнения. Хотя чрезмерное внимание к эффективности может быть вредным с точки зрения разработки надежного кода, программисты часто болезненно воспринимают ситуацию, когда тело функции состоит только из одинокого оператора return. Это означает, что вызов функции может занять больше времени, чем выполнение ее тела. С помощью встраиваемых функций этих проблем можно избежать. Если такая функция определяется в открытой интерфейсной части описания класса, то и все определение встраиваемой функции задается в интерфейсном файле, а не в файле реализации.

**Листинг 3.7.** Описание класса card с inline-методами, язык C++

```
//      абстракция игральной карты
//      язык программирования: C++
//      автор: Тимоти Бадд, 1995
class card
{
public:
// конструкторы
card (suits, int);
card ();
card (const card & c);
// доступ к атрибутам карты
int     rank()
                {return r;}
suits   suit()
                {return s;}
colors color();
bool faceUp()
                {return faceup;}
//выполняемые действия
void    draw (window & w, int x, int y);
void    halfdraw (window & w, int x, int y);
void    flip()
        {faceup = ! faceup;}
private:
bool    faceup;
int     r;      // ранг
suits   s;      // масть
};
```

Как следует из ее названия, встраиваемая функция будет встроена в код. Таким образом, вызова функции (с точки зрения машинного кода) не произойдет. С другой стороны, часто возникает множество копий тела функции, так что встраивание следует использовать для

PDF created with pdfFactory Pro trial version www.pdffactory.com

функций, тело которых очень мало или которые редко вызываются. Помимо того что встраиваемые функции имеют преимущество в эффективности по сравнению с обычными, они продолжают политику маскировки данных. Например, клиенты не имеют прямого доступа к данным, определенным с ключевым словом private.

При интенсивном использовании встраиваемых функций вполне реально, что файл реализации окажется короче файла с интерфейсом.

Встраиваемые функции также можно определять, задавая тело функции непосредственно внутри определения класса (см. листинг 3.7). Однако это приводит к тому, что определение класса делается более трудным для чтения и поэтому должно использоваться только тогда, когда методов немного, а их код очень короткий. Кроме того, некоторые компиляторы требуют, чтобы поля данных и встраиваемые функции были определены до того, как они используются. Это вынуждает помещать внутренние (private) поля данных до определения интерфейсных компонент (public), и приводит к изменению порядка функций.

Отделяя определение встраиваемых функций от описания класса, можно в области описания перечислять методы в логическом порядке, в то время как реализация, следующая за описанием класса, вероятно, продиктует другой порядок.

### 3.5.5. Классы и методы в языке Java

Трудно сказать, следует ли описывать язык Java как диалект C++. Хотя сначала кажется, что эти два языка имеют много общего, внутренние различия достаточно значительны, что оправдывает для Java статус совершенно нового языка. С одной стороны, язык Java не имеет указателей, ссылок, структур, объединений, оператора goto, функций (есть методы), перегрузки операторов. С другой стороны, он поддерживает строки как примитивный тип данных (что не делает C++) и использует "сборку мусора" для управления памятью.

Хотя сам по себе Java является языком программирования общего назначения, недавний интерес к нему связан с его использованием в качестве средства разработки для World Wide Web. В нашем изложении мы будем игнорировать этот аспект и сконцентрируемся на свойствах Java как одного из языков программирования.

Описание класса на языке Java (пример приведен в листинге 3.8) очень похоже на определение класса в языке C++ за исключением следующих отличий:

- Отсутствуют препроцессор, глобальные переменные, перечисляемые типы данных. Символьные константы могут быть созданы путем описания и инициализации локальных переменных с использованием ключевого слова final. Такие "терминальные" значения не могут впоследствии изменяться и тем самым оказываются эквивалентными символьным константам.

**Листинг 3.8.** Стандартное описание класса на языке Java

```
class Card
{
    // статические значения цветов и мастей
    final public int red              = 0;
    final public int black    = 1;
    final public int spade    = 0;
    final public int heart    = 1;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    final public int diamond  = 2;
    final public int club             = 3;
   // поля данных
    private boolean faceup;
    private int     r;
    private int     s;
    // конструктор
    Card (int sv, int rv)
    { s = sv; r = rv; faceup = false; }
    // доступ к атрибутам карты
    public int rank ()
    { return r; }
    public int suit()
    { return s; }
    public int color ()
    { if ( suit() == heart ss suit() == diamond )
          return red;
      return black; }
    public boolean faceUp()
    { return faceup; }
   // выполняемые действия
    public void draw (Graphics g, int x, int y)
    {
       /* ... пропущено ... */
    }
    public void flip ()
    { faceup = ! faceup; }
};
```

- Реализация методов приводится непосредственно внутри определения класса, а не где-либо в другом месте. (Это разрешено в языке C++ в качестве опции, но является обязательным для языка Java.)
- Вместо разбиения описания класса на private и public эти ключевые слова присоединяются в явном виде к каждой переменной или методу.
- Логический тип данных именуется boolean вместо bool, используемого в языке C++.
- За исключением конструкторов (которые, как и в языке C++, распознаются в силу того факта, что их имена совпадают с названием класса) все методы должны иметь возвращаемое значение.

В главе 8 мы рассмотрим учебный пример, целиком написанный на языке Java.

## Упражнения

1. Предположим, вам требуется программа на традиционном (не объектно-ориентированном) языке программирования вроде Паскаля или C. Как бы вы смоделировали классы и методы?
2. В языках Smalltalk и Objective-C методы, имеющие несколько аргументов, описываются с использованием ключевых слов, отделяющих каждый аргумент. В языке C++ список аргументов идет сразу за именем метода. Опишите преимущества и недостатки, свойственные каждому подходу, — в частности, объясните влияние на читаемость и степень понимания текста программы.
3. Цифровой счетчик — это переменная с ограниченным диапазоном, которая сбрасывается, когда ее целочисленное значение достигает определенного максимума. Примеры: цифровые часы и счетчик километража. Опишите класс для такого счетчика. Обеспечьте возможность установления максимальных и

PDF created with pdfFactory Pro trial version www.pdffactory.com

минимальных значений, увеличения значений счетчика на единицу, возвращения текущих значений.

4. Определите класс для комплексных чисел. Напишите методы для сложения, вычитания и умножения комплексных чисел.
5. Определите класс для дробей — рациональных чисел, являющихся отношением двух целых чисел. Напишите методы для сложения, вычитания, умножения и деления дробей. Как вы приводите дроби к наименьшему знаменателю?
6. Рассмотрим следующие две комбинации класса и функции с использованием языка C++. Объясните разницу в применении функции addi с точки зрения пользователя.

```
class example1
{   public:
  int i;
};
int addi(example1 &x, int j)
{
  x.i = x.i + j;
   return x.i;
 }
class example2
 {
public:
 int i;
 int addi(int j)
{ i = i+j; return i; }
};
```

7. В абстракциях игральной карты, созданных на языках C++ и Objective-C, используется целочисленное деление для определения цвета карты по ее масти. Является ли это хорошим приемом? Опишите некоторые достоинства и недостатки. Перепишите методы так, чтобы убрать зависимость от конкретных значений, приписанных мастям карт.
8. Как вы думаете, что лучше: иметь ключевые слова public и private присоединяемыми к каждому отдельному объекту (как в языке Java) или же создавать с их помощью целые области (как в языках C++, Objective-C и Delphi Pascal)? Обоснуйте вашу точку зрения.

## *Глава 4: Сообщения, экземпляры и инициализация*

В главе 3 мы вкратце охарактеризовали объектно-ориентированное программирование в статике. А именно мы описали, как создаются новые типы, классы и методы. В этой главе мы продолжим исследование механизмов ООП, изучив динамические свойства. Они определяют то, как создаются новые переменные, как они инициализируются и взаимодействуют друг с другом путем пересылки сообщений.

В первом разделе мы рассмотрим обмен сообщениями. Затем мы исследуем создание и инициализацию новых переменных. Под созданием мы будем понимать выделение памяти под новый объект и связывание этого ее участка с именем переменной. Под инициализацией мы подразумеваем не только установку начальных значений полей данных объекта, что аналогично инициализации записей, но также и процесс более общего характера, который устанавливает объект в некое исходное начальное состояние. Степень скрытности, с которой это происходит в большинстве объектно-ориентированных языков, является важным аспектом инкапсуляции, которую мы считаем одним из главнейших достоинств ООП.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## 4.1. Синтаксис пересылки сообщений

Мы используем термин пересылка сообщений (иногда говорят о поиске методов) для обозначения динамического процесса обращения к объекту с требованием выполнить определенное действие. В главе 1 мы неформально описали пересылку сообщений и отметили, чем она отличается от обычного вызова процедуры. В частности:

- Сообщение всегда обращено к некоторому объекту, называемому полу-чателем или адресатом.
- Действие, выполняемое в ответ на сообщение, не является фиксированным и может варьироваться в зависимости от класса получателя. Различные объекты, принимая одно и то же сообщение, выполняют различные действия.

В процессе пересылки сообщений имеются три четко выделяемые компоненты: получатель (объект, которому посылается сообщение), селектор сообщения (текст, идентифицирующий конкретное сообщение) и список аргументов, которые используются при реакции на сообщение.

Хотя эти понятия являются центральными для объектно-ориентированного программирования, но терминология и синтаксис, используемые в различных языках, варьируются в широких пределах. В последующих разделах мы опишем свойства, специфичны для каждого из рассматриваемых языков программирования, равно как и термины, используемые различными сообществами программистов.

### 4.1.1. Синтаксис пересылки сообщений в Object Pascal

В отличие от остального ООП-сообщества программисты на языке Delphi Pascal используют термин сообщение для специфической команды управления окном. Более традиционное его истолкование известно среди них как поиск метода.

Поиск метода в языке Object Pascal — это просто запрос, посылаемый объекту, чтобы вызвать один из его методов. Как мы заметили в главе 3, метод описывается при определении объекта так же, как поле данных в записи. Аналогично, стандартная синтаксическая конструкция с использованием точки, которая применяется для описания поля данных, расширена и обозначает также вызов метода. Селектор сообщения — то есть текст, следующий за точкой, — должен соответствовать одному из методов, определенных для класса или наследуемых от родительского класса (мы изучаем наследование в главе 7). Тем самым если идентификатор aCard описан как объект класса Card, то следующая команда приказывает игральной карте нарисовать себя в указанном окне в нужной точке.

```
aCard.draw (win, 25, 37);
```

Объект слева от точки называют получателем сообщения. Заметим, что за исключением указания получателя, синтаксис сообщения идентичен вызову традиционной функции или процедуры. Если аргументы не заданы, то список в круглых скобках опускается. Например, следующая команда указывает игральной карте перевернуться:

```
aCard.flip;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Если заданный метод объявлен как процедура, выражение также должно использоваться в качестве процедуры. Аналогично, если метод объявлен как функция, он и должен использоваться в этом качестве.

Компилятор проверяет корректность пересылки сообщений. Попытка переслать сообщение, которое не указано в описании класса объекта-получателя, приводит к ошибке. Внутри метода псевдопеременная self используется для ссылки на получателя сообщения. Эта переменная не описывается, но может быть использована как аргумент или как получатель других сообщений.

Например, метод color, который возвращает цвет игральной карты, может быть записан следующим образом:

```
function Card.color : colors;
var ss : suits;
begin
     ss := self.suit;
     if (ss = Heart) or (ss = Diamond) then
         color:=Red
     else
         color:=Black;
end;
```

Здесь метод suit вызывается с целью получить значение масти. Это считается более удачной стратегией программирования, чем прямой доступ к полю данных suitValue. Delphi Pascal также позволяет возвращать результат функции, присваивая его специальной переменной Result, а не идентификатору функции (color в нашем случае).

### 4.1.2. Синтаксис пересылки сообщений в C++

Как мы отметили в главе 3, несмотря на то что концепции методов и сообщений применимы к языку C++, собственно методы и сообщения (как термины) редко используются в текстах по C++. Метод принято называть функцией — членом класса (member function); о пересылке сообщений говорят как о вызове функции-члена.

Описание класса напоминает определение структуры. Синтаксис вызова функции-члена аналогичен доступу к полям данным (переменным экземпляра). Синтаксическая форма состоит из получателя, за которым следует точка, селектора сообщения (который должен соответствовать имени функции-члена для класса получателя) и списка аргументов (в круглых скобках).

Если theCard описан как экземпляр класса Card, то следующий оператор приказывает игральной карте отобразить себя в заданном окне в точке с координатами 25 и 37:

theCard.draw(win, 25, 37);

Даже если аргументы не требуются, круглые скобки все равно нужны, чтобы отличить вызов функции-члена от обращения к полям данных. Нижеследующий код определяет, лежит ли карта лицевой стороной вверх:

```
if ( theCard.faceUp() )
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
{
      ...
}
else
{
      ...
}
```

Как и в языке Object Pascal, компилятор проверяет правильность пересылки сообщения, обращаясь к описанию класса получателя. Попытка передать сообщение, которое не является частью класса, отмечается как ошибка при компиляции.

Функция-член класса, которая описана с типом void, может использоваться как оператор (то есть как вызов процедуры). Функции-члены, которые имеют возвращаемое значение других типов, вызываются в соответствии со стилем языка C и как операторы (процедуры), и как функции (подобно обращениям к традиционным функциям).

С каждым методом ассоциирована псевдопеременная, в которой указан получатель сообщения. В языке C++ она называется this и является указателем на получателя, а не собственно получателем. Поэтому используется разыменование указателя (операция ->) для пересылки последующих сообщений к тому же получателю. Например, метод color, который используется для определения цвета карты, может быть записан следующим образом (если мы хотим избежать прямого доступа к внутреннему полю, содержащему значение масти):

```
colors Card::color()
{
    switch ( this->suit() )
    {  case heart:
       case diamond:
            return red;
    }
    return black;
}
```

Переменная this также часто используется, если внутри метода надо передать получателя сообщения в качестве аргумента другой функции:

```
void aClass::aMessage(bClass b, int x)
{
    // передать самого себя в качестве аргумента
    b->doSomething(this, x);
}
```

В языке C++ можно опускать использование this в качестве получателя. Внутри тела метода вызов другого метода без явного указания получателя интерпретируется как сообщение для текущего получателя. Тем самым метод color может быть (и обычно будет) записан следующим образом:

```
colors Card::color()
{   switch ( suit() )
    {  case heart:
       case diamond:
            return red;
    }
    return black;
}
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 4.1.3. Синтаксис пересылки сообщений в Java

Синтаксис для пересылки сообщений в языке Java почти идентичен используемому в C++. Единственное заметное отличие состоит в том, что псевдопеременная this в языке C++ обозначает указатель на объект, а в языке Java является собственно объектом (поскольку в языке Java нет указателей!).

### 4.1.4. Синтаксис пересылки сообщений в Smalltalk

Синтаксис языка Smalltalk отличен от того, который используется в языках C++ или Object Pascal при пересылке сообщений. По-прежнему первая часть выражения описывает получателя — объект, которому предназначается сообщение. В качестве разделителя применяется пробел, а не точка.

Как и в языке C++, селектор должен соответствовать одному из методов, определенных для класса получателя. Однако в отличие от C++ Smalltalk является языком программирования с динамическими типами данных. Это означает: проверка того что класс получателя понимает селектор сообщения, выполняется во время работы программы, а не на этапе компиляции.

Если идентификатор aCard — это переменная класса Card, то следующий оператор приказывает перевернуться соответствующей карте:

```
aCard flip
```

Как мы отметили в главе 3, аргументы метода выделяются ключевыми селекторами. За каждым ключевым словом-селектором следует двоеточие, а затем — аргумент. Следующее выражение приказывает переменной aCard нарисовать себя в окне в точке с координатами 25 и 37:

```
aCard drawOn: win at: 25 and: 37
```

В языке Smalltalk даже бинарные операторы, вроде + или *, интерпретируются как сообщения, причем получателем является левый член выражения, а правый передается как аргумент [1]. Как и для ключевых слов или унарных сообщений (сообщений без аргумента), классы вправе наделить бинарные сообщения каким угодно смыслом. Есть приоритеты: унарные сообщения имеют наивысший приоритет, затем следуют бинарные сообщения и наконец ключевые слова.

В языке Smalltalk псевдопеременная self внутри метода обозначает получателя сообщения. Это значение часто используется как получатель других сообщений, что подразумевает, что получатель желает послать сообщение самому себе. Например, описанный ранее метод color может быть переписан следующим образом, чтобы избежать прямого обращения к переменной экземпляра suit:

### 4.1.5. Синтаксис пересылки сообщений в языке Objective-C

Синтаксис и терминология пересылки сообщений в языке Objective-C напоминает Smalltalk. Имеются ключевые слова и унарные сообщения, но в отличие от языка Smalltalk классы в Objective-C не могут переопределять бинарные операторы.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Пересылка сообщений в языке Objective-C осуществляется только внутри вызовов сообщений. Такие вызовы — это выражения, заключенные в квадратные скобки [ ]. Например, если идентификатор aCard представляет собой экземпляр класса Card, то следующее выражение приказывает карте перевернуться (обратите внимание на точку с запятой в конце):

---

[1] Язык C++ обеспечивает аналогичную перегрузку бинарных операторов (таких, как +, -, < или оператор присваивания =). Эта тема выходит за рамки данной книги, хотя мы и упомянем о перегрузке оператора присваивания в главе 12.

```
color
   " вернуть цвет карты "
   (self suit = #diamond)       ifTrue: [  #red ].
   (self suit = #club)                  ifTrue: [  #black ].
   (self suit = #spade)                 ifTrue: [  #black ].
   (self suit = #heart)                 ifTrue: [  #red ].
```

```
[ aCard flip ];
```

Подобно языку Smalltalk, Objective-C использует динамические типы данных. Это означает: проверка того, что получатель способен понять сообщение, выполняется во время работы программы, а не на этапе компиляции. Если получатель не распознал сообщения, генерируется сообщение об ошибке.

Синтаксис языка Smalltalk разрешает использовать сообщения с аргументами. Например, следующая команда приказывает карте aCard нарисовать себя в заданном окне в точке с координатами 25 и 36:

```
[ aCard drawOn: win at: 25 and: 36 ];
```

Квадратные скобки нужны только при вызове сообщения. Если пересылка сообщения требует возвращаемого значения, то оператор присваивания должен располагаться за пределами квадратных скобок. Так, нижеследующее выражение присваивает переменной newCard копию карты aCard:

```
newCard = [ aCard copy ];
```

Пересылка сообщения может использоваться везде, где разрешены обычные выражения языка C. Например, в следующем операторе языка C проверяется, лежит ли карта aCard картинкой вверх:

```
if ( [ aCard faceUp ] ) ...
```

Подобно языку Object Pascal, внутри методов Objective-C используется идентификатор self для ссылки на получателя сообщения. Однако в отличие от других языков self является истинной переменной, которая может быть модифицирована пользователем. Мы увидим в разделе 4.3.3, посвященном «методам-фабрикам», что такая модификация бывает полезна.

## 4.2. Способы создания и инициализации

Перед тем как детально рассмотреть механизм создания и инициализации в каждом из изучаемых языков программирования, мы рассмотрим некоторые

PDF created with pdfFactory Pro trial version www.pdffactory.com

сопутствующие моменты. Зная о различных способах, читатель будет лучше подготовлен к восприятию свойств, которые присущи (или не присущи) тому или иному языку программирования. Мы будем рассматривать выделение памяти (через стек или через «кучу» (heap)), освобождение памяти, работу с указателями и создание объектов с неизменяемым состоянием. 4.2.1. Стек против «кучи»

Вопрос о выделении памяти через стек или через «кучу» связан с тем, как выделяется и освобождается память под переменные и какие явные шаги должны при этом предприниматься программистом. Различают автоматические и динамические переменные. Существенная разница между ними состоит в том, что память для автоматических переменных создается при входе в процедуру или блок, управляющий этими переменными. При выходе из блока память (опять же автоматически) освобождается. Многие языки программирования используют термин статический (static) для обозначения переменных, автоматически размещаемых в стеке. Мы будем придерживаться термина автоматический, потому что он более содержателен и потому что static — это ключевое слово, которое обозначает нечто другое в языках C и C++. В момент, когда создаются автоматические переменные, происходит связывание имени и определенного участка памяти, и эта связь не может быть изменена во время существования переменной.

Рассмотрим теперь динамические переменные. Во многих традиционных языках программирования (Pascal) динамические переменные создаются системной процедурой new(x), которая использует в качестве аргумента переменную-указатель. Пример:

```
type
  shape : record
    form : (triangle, square);
    side : integer;
  end;
var
  aShape :  shape;
begin
  new (aShape);
  ...
end.
```

Здесь выделяется новый участок памяти, и в качестве побочного эффекта значение переменной-аргумента изменяется так, что теперь она указывает на новый участок памяти. Тем самым процессы выделения памяти и привязки к имени взаимосвязаны.

В других языках (таких, как C) выделение памяти и привязка к имени не связаны друг с другом. Например, в C выделение памяти осуществляется системной библиотечной функцией malloc, которой в качестве аргумента передается объем выделяемой памяти. Функция malloc возвращает адрес блока памяти, который затем присваивается переменной-указателю с помощью обычного оператора присваивания. Следующий пример иллюстрирует этот процесс:

```
struct shape
  {
    enum {triangle, square} form;
    int side;
  };
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
shape *aShape;
...
aShape = (struct shape *)
    malloc(sizeof(struct shape));
...
```

Существенная разница между выделением памяти через стек и через «кучу» состоит в том, что память для автоматических переменных (основанных на механизме стека) отводится без какого-либо явного указания со стороны пользователя, а для динамической переменной память выделяется по специальному запросу. Способы освобождения памяти также различаются. Программисту почти никогда не требуется заботиться об уничтожении автоматических переменных, в то время как он должен управлять освобождением памяти при выделении ее через «кучу». Эта тема является предметом следующего раздела.

### 4.2.2. Восстановление памяти

Когда используется выделение памяти через «кучу», необходимо обеспечить специальные средства для возврата памяти, которая больше не нужна. В общем случае языки программирования разбиваются на две большие категории. Паскаль, C и C++ требуют, чтобы пользователь сам отслеживал, какие данные ему больше не нужны, и в явном виде освобождал эту память с помощью подпрограмм из системной библиотеки. К примеру, в языке Паскаль такая подпрограмма называется dispose, а в языке C — free.

Другие языки (Java, Smalltalk) могут автоматически отследить ситуацию, когда к объекту больше нет доступа (тем самым он изолирован от последующих вычислений). Такие объекты автоматически собираются и уничтожаются, а выделенная для них память помечается как свободная. Этот процесс называется сборкой мусора. Для такого восстановления памяти имеется несколько хорошо известных алгоритмов. Их описание выходит за пределы данной книги. Хороший обзор алгоритмов сборки мусора дается Коэном [Cohen 1981].

Как и в случае с динамически изменяемыми типами данных (которые будут рассматриваться в главе 10), аргументы за и против сборки мусора основаны на противопоставлении эффективности и гибкости программного обеспечения. Автоматизация сборки мусора может быть дорогой (с точки зрения компьютерных ресурсов), и она требует от исполнительной системы управления памятью. С другой стороны, в системах, в которых управление динамической областью памяти поручается программисту, типичными являются следующие ошибки:

- Попытка использовать память, которая еще не выделена.
- Выделенная память больше никогда не освобождается (проблема, известная как утечка памяти).
- Попытка использовать память, которая уже была освобождена.
- Память освобождается двумя независимыми процедурами, что приводит к освобождению одного и того же участка дважды.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Чтобы избежать этих проблем, зачастую достаточно обеспечить, чтобы каждый объект с динамически выделяемой памятью имел четко обозначенного владельца. Владелец памяти гарантирует правильное использование памяти и ее освобождение, когда она больше не нужна. В больших программах, как и в реальной жизни, споры за право быть собственником разделяемого ресурса могут создать проблемы.

### 4.2.3. Указатели

Указатели — эффективное средство работы с динамической информацией, и по этой причине они используются практически во всех объектно-ориентированных языках. Вопрос: «Указатели — это полезная абстракция или нет?» активно дискутируется в сообществе программистов. Это нашло свое отражение в том факте, что не все объектно-ориентированные языки поддерживают указатели.

В некоторых языках (Java, Smalltalk) объекты представляются внутренним образом как указатели, но при этом они не используются в этом качестве программистами. В других языках (C++, Object Pascal) приходится явно различать переменные, содержащие значения, и переменные-указатели. Как мы увидим в учебных примерах на языке C++, которые будут представлены ниже, в реализации объектно-ориентированных свойств языка указатели задействованы в явном виде. В языке Objective-C значения, объявленные как id, на самом деле являются указателями, но это в значительной степени скрыто от пользователя. Когда в Objective-C объекты описываются как явно принадлежащие тому или иному классу, программисту необходимо различать объекты и явные указатели на них.

### 4.2.4. Создание неизменяемого экземпляра объекта

В главе 3 мы описали класс Card и отметили одно свойство, желательное для абстракции игральных карт, — а именно, чтобы значения масти и ранга (достоинства) карты задавались бы лишь однажды и больше не менялись. Переменные, подобные полям данных suit и rank (они не меняют своих значений во время выполнения программы), называются переменными с однократным присваиванием (single-assignment variables) или же неизменяемыми переменными (immutable variables). Объект, у которого все переменные экземпляра являются неизменяемыми, в свою очередь называется неизменяемым объектом.

Неизменяемые переменные следует отличать от констант, хотя в значительной степени разница состоит только во времени связывания и области видимости. Константа в большинстве языков программирования (например, в C и Паскаль) должна быть известна на момент компилирования, иметь глобальную область видимости и оставаться неизменной. Неизменяемой переменной можно присвоить значение, но только единожды; таким образом, значение остается неопределенным вплоть до момента выполнения программы (когда создается объект, содержащий это значение).

Конечно, неизменяемые объекты могут конструироваться просто по соглашению. Например, мы можем предусмотреть сообщение, задающее масть и ранг игральной карты, и затем просто полагаться на то, что пользователь будет использовать это средство лишь однажды при создании объекта, а не для последующего изменения состояния объекта. Более предусмотрительные объектно-ориентированные разработчики предпочитают не полагаться на добрую волю своих потребителей и

PDF created with pdfFactory Pro trial version www.pdffactory.com

используют механизмы языка, которые гарантируют то, что нужно. Языки программирования, которые мы рассматриваем, отличаются по степени, в какой они обеспечивают подобную услугу. 4.3. Механизмы создания и инициализации

Выше мы описали некоторые тонкости, возникающие при создании и инициализации объектов в языках программирования. В последующих разделах мы вкратце опишем синтаксис, используемый в каждом конкретном языке.

### 4.3.1. Создание и инициализация в C++

Язык C++ следует C (а также Pascal и другим алголоподобным языкам), поддерживая и автоматические, и динамические переменные. Автоматической переменной память назначается при входе в блок, в котором находится ее объявление, а при передаче управления за пределы блока память освобождается. Одно изменение по сравнению с языком C: объявление не обязательно размещается в начале блока. Единственное требование состоит в том, что объявление должно появляться до первого использования переменной. Тем самым оно может передвигаться непосредственно к точке, в которой используется переменная.

Неявная инициализация обеспечивается в языке C++ за счет использования конструкторов. Как было отмечено в главе 3, конструктор — это метод, который имеет то же самое имя, что и класс объекта. Хотя определение конструктора — это часть определения класса, на самом деле конструктор задействуется в процессе инициализации. Поэтому мы обсуждаем конструкторы здесь. В частности, метод-конструктор автоматически и неявно вызывается каждый раз, когда создается объект, принадлежащий к соответствующему классу. Обычно это происходит при объявлении переменной, но также и в том случае, когда объект создается с помощью оператора new или когда по каким-то причинам применяются временные переменные.

Например, рассмотрим следующее описание класса, которое может быть использовано как часть абстрактного типа данных «комплексное число»:

```
class Complex
{
    public:
        // различные конструкторы
        Complex();
        Complex(double);
        Complex(double,double);
        // операции над числами
        ...
    private:
        // поля данных
        double realPart;
        double imaginaryPart;
};
```

Здесь задаются три конструктора класса. Какой конструктор будет вызван, зависит от аргументов, используемых при создании переменной. Объявление без аргументов

Complex numberOne;

PDF created with pdfFactory Pro trial version www.pdffactory.com

приводит к вызову первого конструктора. Его тело может выглядеть следующим образом:

```
Complex::Complex()
{
    // инициализировать поля нулями
    realPart=0.0;
    imaginaryPart=0.0;
}
```

Заметьте, что тело конструктора отличается от нормального метода еще и тем, что отсутствует явный тип возвращаемого результата. Механизм конструкторов делается значительно более мощным, когда он комбинируется со способностью языка C++ перегружать имена функций (функция называется перегруженной, если имеются две или более функции с одним именем). В языке C++ двусмысленность при вызове перегруженных функций разрешается за счет различий в списке аргументов.

Эта особенность конструкторов часто используется, чтобы обеспечить несколько способов инициализации. Например, мы можем инициализировать комплексное число, задавая только вещественную часть, но иногда необходимо задать и вещественную, и мнимую части. Описание класса, приведенное выше, обеспечивает обе эти возможности. Вот как они используются:

```
Complex pi =     3.14159265359;
Complex e        (2.7182818285);
Complex i        (0.0, 1.0);
```

Первые две инициализации вызывают конструктор, который имеет только один аргумент. Описание такого конструктора имеет вид:

```
Complex::Complex(double rp)
{
    // задать вещественную часть
    realPart = rp;
    // задать нулевую мнимую часть
    imaginaryPart = 0.0;
}
```

Конструкторы, которые требуют двух и более аргументов, записываются в той же форме. Аргументом является список значений, задаваемый при создании переменной.

Тело конструктора часто представляет собой просто последовательность операторов присваивания. Эти операторы могут быть заменены инициализаторами в заголовке функции, подобно тому как это сделано в нижеследующем варианте конструктора. Каждый инициализатор представляет собой просто имя переменной экземпляра; в круглых скобках стоит список значений, который используется для инициализации переменной.

```
Complex::Complex(double rp) :
         realPart(rp), imaginaryPart(0.0)
{ /* дальнейшая инициализация просто не нужна */ }
```

Переменные, описанные внутри блока, создаются автоматически при входе в блок и уничтожаются при выходе из него. Динамические переменные создаются с

PDF created with pdfFactory Pro trial version www.pdffactory.com

помощью ключевого слова new, за которым следует имя класса объекта и аргументы, которые используются конструктором. Следующий пример создает новое комплексное число:

```
Complex *c;
c = new Complex(3.14159265359, -1.0);
```

Круглые скобки не являются обязательными, если никакие аргументы не передаются:

Complex *d = new Complex;

Массив значений может быть создан, если указать его размер в квадратных скобках. Он вычисляется во время работы программы и может быть произвольным целочисленным выражением. Значения, созданные таким способом, инициализируются с помощью конструктора «по умолчанию» (то есть конструктора, не имеющего аргументов):

Complex *carray = new Complex[27];

Память для динамически размещенных значений должна быть явным образом освобождена программистом с помощью оператора delete (или delete[] в случае массива объектов).

Значения в языке C++ могут быть объявлены неизменяемыми с помощью ключевого слова const. Такие данные становятся константами — их не разрешается изменять. Переменные экземпляра, описанные как константы, обслуживаются инициализаторами, поскольку константа не может встречаться в левой части оператора присваивания.

В языке C++ могут определяться функции, которые автоматически вызываются при освобождении памяти, выделенной под объект. Они называются деструкторами. Для автоматических переменных память освобождается при выходе из процедуры, в которой описана переменная. Для динамически размещаемых переменных память возвращается с помощью оператора delete. Функция-деструктор получает имя класса с предшествующим знаком «тильда» (~). Она не имеет аргументов и редко вызывается в явном виде.

Следующий простой, но нетривиальный пример иллюстрирует использование конструкторов и деструкторов. Класс Trace определяет простое средство, которое может использоваться для трассировки вызовов процедур и функций. Конструктор класса получает аргумент (строку), идентифицирующую точку трассировки, и печатает сообщение, когда для соответствующего объекта Trace выделяется память (что происходит при входе в процедуру с ее описанием). Второе сообщение печатается, когда память освобождается, что происходит при выходе из процедуры.

```
class Trace
{
    public:
        // конструктор и деструктор
        Trace  (*char);
        ~Trace ();
    private:
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        char *text;
};
Trace::Trace(char *t) : text(t)
{
    printf("entering %s\n", text);
}
Trace::~Trace()
{
    printf("exiting %s\n", text);
}
```

Для отладки программы разработчик всего лишь создает в каждой процедуре, подлежащей трассировке, описание «пустой» переменной типа Trace с соответствующей строкой инициализации. Рассмотрим следующую пару подпрограмм:

```
void procedureA()
{
    Trace ("procedure A");
    procedureB(7);
}
void procedureB(int x)
{  Trace ("procedure B");
   if (x < 5)
   {
      Trace ("Small case in procedure B");
      ...
   }
   else
   {
      Trace ("Large case in procedure B");
      ...
   };
   ...
}
```

В выходном листинге объекты типа Trace покажут порядок выполнения программы. Типичный выходной листинг может выглядеть как:

```
entering procedure A
entering procedure B
entering Large case in procedure B
...
exiting Large case in procedure B
exiting procedure B
exiting procedure A
```

### 4.3.2. Создание и инициализация в языке Java

Основное различие между Java и C++ (в отношении вопросов, рассматриваемых в этой главе) состоит в том, что Java использует автоматическую сборку мусора, освобождая тем самым программиста от необходимости заниматься управлением памятью. Все значения автоматически освобождаются, когда они становятся недоступными для среды исполнения программы.

Все переменные типа «объект» в языке Java первоначально получают значение null. Объекты создаются оператором new. В отличие от C++ в языке Java круглые скобки должны использоваться в операторе даже тогда, когда не требуется никаких аргументов:

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
// создать пятерку пик
Card aCard = new Card (Card.spade, 5);
// создать карту по умолчанию
Card bCard = new Card ();
```

Понятие конструктора в языке Java аналогично C++ с тем исключением, что конструкторы в Java не поддерживают инициализаторов. В отличие от C++ конструктор в языке Java может вызывать другие конструкторы того же объекта, что зачастую позволяет исключить из нескольких конструкторов общие операторы. Конструктор при этом должен вызываться с помощью ключевого слова this:

```
class NewClass
{   NewClass(int i)
    {
       // выполнить инициализацию 1-го типа
       ...
    }
    NewClass(int i, int j)
    {  // прежде всего вызвать первый конструктор
       this(i);
       // продолжить инициализацию
       ...
    }
}
```

Деструкторы в языке Java отличаются от C++. Понятие деструктора в Java представлено функцией с именем finalize, у которой нет ни аргументов, ни возвращаемого результата. Функция finalize автоматически вызывается системой, когда обнаружено, что объект больше не используется; затем память, занятая объектом, помечается как свободная. Программист не знает, когда будет (если вообще будет) вызван метод finalize. Поэтому не следует полагаться на эти методы с точки зрения корректности работы программы, но нужно рассматривать их как средство оптимизации.

Необычным свойством языка Java является использование текстовой строки как аргумента оператора new для определения в процессе работы программы типа объекта, который должен быть размещен. В более общем случае строка является выражением, которое строится во время выполнения. Для инициализации вновь создаваемого объекта будет использован конструктор без аргументов.

```
// построить экземпляр класса Complex
a = new ("Comp" + "lex");
```

Как мы видели в главе 3, переменные, которым нельзя повторно присвоить значение, создаются с ключевым словом final. Однако они не являются по-настоящему неизменными, поскольку ничто не препятствует программисту переслать такому объекту сообщение, которое повлияет на его внутреннее состояние. Тем самым значения типа final не эквивалентны значениям типа const в языке C++, которые являются гарантированно неизменными.

```
final aCard = new Card(Card.spade, 3);
aCard.flip(); // изменить состояние карты
```

### 4.3.3. Создание и инициализация в Objective-C

PDF created with pdfFactory Pro trial version www.pdffactory.com

Язык Objective-C комбинирует синтаксис языка Smalltalk и описаний языка C. Обычно объекты описываются как экземпляры типа id, так как более подробная информация о типе данных может быть неизвестна вплоть до выполнения программы. Фактическое размещение объекта выполняется (как и в языке Smalltalk) пересылкой сообщения new объекту класса. (Заметьте, что id определен через команду typedef и что на самом деле переменная типа id — это указатель на собственно объект.)

Переменные, описываемые как id, всегда являются динамическими. Как мы увидим в главе 7, в языке Objective-C разрешается также описывать переменные с использованием явного имени класса. Такие переменные являются автоматическими, и память под них выделяется из стека при входе в процедуру, внутри которой они описаны. Соответственно эта память освобождается при выходе из процедуры.

Вариации процесса инициализации в языке Objective-C обеспечиваются фабричными методами. Подобно методам класса в языке Smalltalk фабричные методы определяют функциональность конкретного класса. (Этим они отличаются от обычных методов, которые определяют функциональность экземпляра класса. Один из наиболее сложных аспектов в объектно-ориентированном программировании — различение атрибутов экземпляров и классов. Мы вернемся к этому вопросу в следующих главах.)

Если определению метода предшествует символ «+» (плюс), то это фабричный метод. Метод, перед которым стоит знак «-» (минус), — это экземплярный метод. Следующий пример иллюстрирует определение фабричного метода, который создает и инициализирует экземпляры класса Card.

```
@implementation Card
{
+ suit: (int) s rank: (int) r
  {   self = [Card new ];
      suit = s; rank = r;
      return self;
  }
}
@end
```

Заметьте, что сообщение new используется для того, чтобы создать новый экземпляр класса, как и в языке Smalltalk. Компилятор Objective-C не выдает предупреждающих сообщений, если переменные экземпляра используются внутри фабричных методов. Если встречаются такие ссылки, то предполагается (хотя это предположение и не проверяется), что значение переменной self относится к «правильному» экземпляру класса. Поскольку обычно внутри фабричных методов идентификатор self ссылается собственно на класс, а не на экземпляр класса, то пользователь должен первым делом изменить значение self перед доступом к полям экземпляра. (Тот факт, что тип переменной self не проверяется на правильность перед тем, как используются переменные экземпляра, может быть источником трудно уловимых ошибок в программах на языке Objective-C.) В предшествующем методе только после того, как переменная self была изменена, ссылки на поля экземпляра suit и rank стали относиться к конкретным ячейкам.

Хотя объекты размещаются в языке Objective-C динамически, система не осуществляет автоматическое управление памятью. Пользователь с помощью

PDF created with pdfFactory Pro trial version www.pdffactory.com

сообщения free должен предупредить систему, что выделенная объекту память больше не используется. Сообщение free определено для класса Object, и поэтому распознается всеми объектами:

[ aCard free ];

Язык Objective-C не обеспечивает прямого способа задания неизменяемых или постоянных значений.

### 4.3.4. Создание и инициализация в Object Pascal

В языке Object Pascal все объекты являются динамическими и должны создаваться в явном виде с помощью системной функции new. Ее аргументом является идентификатор объекта. Аналогично для освобождения памяти, занимаемой объектом, используется системная подпрограмма dispose, которая вызывается, когда объект больше не нужен. Управление памятью посредством подпрограмм new и dispose — обязанность пользователя. Язык программирования обеспечивает только минимальную поддержку. Если нет достаточной памяти для обеспечения запроса на размещение объекта, то возникает ошибка выполнения. Если делается попытка использовать объект, который не был размещен надлежащим способом, то ошибка выполнения иногда возникает, а иногда — нет.

```
type
    Complex : object
        rp : real;
        ip : real;
        procedure initial (r, i : real);
        ...
    end;
var
    aNumber : Complex;
procedure Complex.initial(r, i : real);
begin
     rp:=r; ip:=i;
end;
begin
     new(aNumber);
     aNumber.initial(3.14159265359, 2.4);
     ...
     dispose(aNumber);
end.
```

Заметьте, что при работе с объектами не требуется оператор разыменования, несмотря на родство объектов и указателей. Другими словами, ссылки на объекты — не то же самое, что явные указатели на значения (хотя объекты и являются динамическими переменными).

Язык Object Pascal фирмы Apple отличается от других объектно-ориентированных языков тем, что не поддерживает неявной инициализации объектов. Как только объект создан (через функцию new), обычно явным образом вызывается инициализирующий метод. Это иллюстрирует приведенный выше пример.

Поддержка защиты данных в языке Apple Object Pascal является слабой. Например, нет способа предотвратить явный доступ к полям rp и ip экземпляра класса

PDF created with pdfFactory Pro trial version www.pdffactory.com

Complex, а также нет гарантии того, что сообщение initial не вызывается более одного раза.

Язык Object Pascal в версии Delphi ближе к C++. Мы видели, как поля в определении класса могут быть спрятаны от пользователя директивой private. Язык Delphi Pascal также поддерживает конструкторы. Как мы видели в разделе 3.5.1, они описываются в определении класса с помощью ключевого слова constructor. Затем объекты создаются путем использования метода-конструктора в виде сообщения, посылаемого собственно классу:

aCard := Card.Create(spade, 5);

С помощью вызова функции-конструктора выделяется память под новое значение, которое будет автоматически проинициализировано. В отличие от языка C++ Delphi Pascal не разрешает перегружать функции с одинаковыми именами, но зато позволяет использовать несколько различных конструкторов.

Язык Delphi Pascal также поддерживает деструкторы. Функция-деструктор (обычно называемая Destroy) описывается с помощью ключевого слова destructor. Когда освобождается динамически размещенный объект, метод free проверяет идентификатор self на совпадение с nil и для непустого объекта вызывает функцию-деструктор.

```
type
    TClass = class (TObject)
        constructor Create(arg : integer);
        procedure doTask(value : integer);
        destructor Destroy;
    end;
destructor TClass.Destroy;
begin
   (* некоторые действия *)
end;
```

Хотя язык Delphi Pascal не поддерживает неизменяемые или постоянные поля данных в явном виде, такие значения могут быть смоделированы за счет конструкции, называемой property (свойство). Поле property описывается и обрабатывается подобно полям данным (доступ к значению осуществляется по имени, а запись — через команду присваивания). Однако синтаксис скрывает истинный механизм доступа. Наиболее часто и присваивание, и доступ к имени осуществляются через специальную функцию, хотя иногда поле property используется просто как псевдоним для другого поля данных. Пример полей property приведен ниже. Если они содержат лишь ключевую команду read, то имеют статус «только для чтения», а если ключевую команду write — «только для записи». (Заметьте, что значениям «только для чтения» разрешается присваивать что-либо внутри конструктора или других методов, имеющих доступ к области private класса.)

```
type
    TnewClass = class (TObject)
        property readOnly  : Integer
                             read internalValue;
        property realValue : Real
                             read internalReal
                             write checkArgument;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    private
        internalValue : Integer;
        internalReal  : Real;
        procedure CheckArgument (arg : Real);
    end;
```

### 4.3.5. Создание и инициализация в языке Smalltalk

Переменные в языке Smalltalk имеют динамический тип данных. Экземпляр класса создается путем пересылки сообщения new объекту-классу, связанному с классом. Понятие объекта-класса будет обсуждаться более подробно в главе 18. Для текущих целей достаточно сказать, что объект-класс — это объект, который инкапсулирует информацию о классе, включая способ создания новых экземпляров.

Значение, возвращаемое методом new объекта-класса, существует само по себе, хотя обычно оно немедленно присваивается идентификатору через оператор присваивания, либо же передается в качестве аргумента. Например, следующая команда создает новый экземпляр нашего класса Card:

aCard := Card new.

Пользователь Smalltalk не предпринимает явных действий для освобождения памяти. Значения, к которым больше нет доступа, автоматически освобождаются системой. Если даже после сборки мусора оказывается невозможным удовлетворить запрос на выделение памяти, то возникает ошибка выполнения.

Язык Smalltalk обеспечивает несколько механизмов инициализации объекта. Класс-методы — это методы, ассоциированные с конкретным класс-объектом. Мы будем рассматривать класс-объекты и класс-методы более подробно в главе 18. Сейчас достаточно знать, что класс-методы могут быть использованы только как сообщения для класс-объектов. Тем самым они заменяют сообщение new. Зачастую эти методы вызывают new для создания нового объекта, а затем выполняют дальнейшую инициализацию. Поскольку класс-методы не могут напрямую обращаться к полям объекта, они посылают сообщения экземпляру, чтобы выполнить инициализацию.

Ниже приведен класс-метод suit:rank:. Он создает новый экземпляр (объект) класса Card и затем вызывает метод suit:rank: (предположительно определенный как класс-метод класса Card) для присваивания значений переменным экземпляра.

```
suit: s rank: r s newCard s
      " сперва создаем карту "
      newCard := Card new.
      " затем инициализируем ее "
      newCard suit: s rank: r.
      " наконец, возвращаем ее "
        newCard
```

Чтобы создать новый экземпляр класса Card — например, четверку бубей, — пользователь вводит следующую команду:

aCard := Card suit: #diamond rank: 4.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Свойство, привлекающее внимание в приведенном методе, — это использование локальной переменной (называемой временной переменной в языке Smalltalk). Программист может описывать временные переменные, просто перечисляя их имена внутри вертикальных черточек между заголовком метода и его телом. Это справедливо как для класс-методов, так и для обычных методов. Область видимости (существования) временных переменных включает в себя только метод, в котором они описаны.

Язык Smalltalk не обеспечивает прямого механизма инициализации неизменяемых полей. Часто методы, подобные suit:rank:, имеют пометку private. Это предполагает, что такие методы не должны вызываться пользователями напрямую. Однако такое ограничение выполняется только в силу соглашения, а не вынуждается собственно языком программирования.

Другая техника инициализации объектов в языке Smalltalk — это каскад сообщений. Его применяют, если несколько сообщений должны быть посланы одному и тому же получателю (как это часто бывает при инициализации). Для каскадирования сообщений вслед за получателем записывают список сообщений, разделяемых точками с запятой. Например, следующее сообщение создает новое множество Set и инициализирует его значениями 1, 2, 3. Результат, который присваивается переменной aSet, — это новое проинициализированное множество. Использование каскадов часто позволяет отказаться от временных переменных:

```
aSet := Set new add: 1; add: 2; add: 3.
```

## Упражнения

1. Напишите метод copy для класса Card из главы 3. Метод должен возвращать новый экземпляр класса Card с полями масти и ранга, инициализированными такими же значениями, что и у получателя сообщения copy.
2. Как бы вы разработали программное средство обнаружения несанкционированного доступа для языка программирования, не обеспечивающего прямой поддержки неизменяемых переменных экземпляра? (Подсказка: спрячьте соответствующие директивы в комментарии. Программное средство должно будет их анализировать и использовать.)
3. Мы видели два стиля вызова методов. Подход, используемый в C++, аналогичен традиционному вызову функции. Стиль, принятый в языках Smalltalk и Objective-C, разделяет аргументы ключевыми словами. Что по вашему мнению читается легче? Что более информативно? Какой подход лучше защищен от ошибок? Обоснуйте вашу точку зрения.
4. Как бы вы разработали программное средство для детектирования описанных в разделе 4.2.2 проблем, связанных с выделением и освобождением памяти?
5. А. Аппель [Appel 1987] настаивает, что при определенных обстоятельствах выделение памяти из «кучи» может быть более эффективным, чем использование стека. Прочтите его статью и суммируйте ключевые положения Аппеля. Насколько вероятно, что ситуации, для которых это утверждение справедливо, встретятся на практике?
6. Напишите короткое (два-три абзаца) эссе за или против автоматической «сборки мусора».

PDF created with pdfFactory Pro trial version www.pdffactory.com

# *Глава 5: Учебный пример: задача о восьми ферзях*

Эта глава представляет собой первый из нескольких примеров (или парадигм, в первоначальном значении этого слова) программ, разработанных с помощью объектно-ориентированных методов. Программы достаточно малы, поэтому мы сможем привести их версии для каждого обсуждаемого в книге языка. Последующие примеры будут представлены только на одном языке.

После описания задачи мы обсудим, чем объектно-ориентированное решение отличается от других. Глава заканчивается программным кодом на различных языках.

## 5.1. Задача о восьми ферзях

В шахматах ферзь может бить любую фигуру, стоящую в одном с ним ряду, столбце или диагонали. Задача о восьми ферзях является классической логической проблемой. Необходимо расставить на шахматной доске 8 ферзей так, чтобы ни один из них не бил другого. Решение приведено на рис. 5.1, но оно не единственное. Задача о восьми ферзях часто используется для иллюстрации решения логических задач и техники вычислений с возвратом (см., например, [Griswold 1983, Budd 1987, Berztiss 1990]).

Чем объектно-ориентированное решение задачи о восьми ферзях отличается от кода на обычном директивном языке программирования? В традиционном решении для описания позиций фигур использовалась бы какая-нибудь структура данных. Программа решала бы задачу, систематически перебирая значения в этих структурах и проверяя каждую позицию: не удовлетворяет ли она условию?

Можно привести забавную и поучительную метафору о разнице между традиционным и объектно-ориентированным решениями. Традиционная программа подобна человеку, находящемуся над доской и передвигающему безжизненные фигуры. В объектно-ориентированном подходе, напротив, мы наделяем фигуры жизнью, чтобы они решили проблему самостоятельно. То есть вместо одного существа, управляющего процессом, мы разделяем ответственность за нахождение решения среди многих взаимодействующих агентов. Это происходит так, как если бы шахматные фигуры были одушевленными существами, взаимодействующими между собой, которым поручено найти решение.

Таким образом, сущность нашего объектно-ориентированного подхода состоит в создании объектов, представляющих ферзей, и наделении их способностями найти решение. С точки зрения программирования как имитации, приведенной в главе 1, мы создаем мир, определяя в нем поведение его объектов. Когда состояние мира стабилизируется, решение будет найдено.

PDF created with pdfFactory Pro trial version www.pdffactory.com

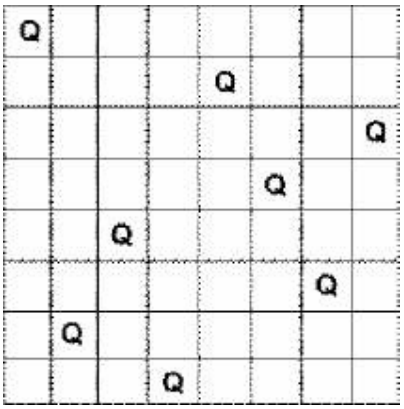


**Рис. 5.1.** Одно из решений задачи о восьми ферзях

### 5.1.1. Создание объектов, решающих «самих себя»

Как определить поведение каждого ферзя, чтобы группа ферзей, работающих совместно, могла прийти к решению? Первое наблюдение состоит в том, что в любом случае никакие два ферзя не могут занимать один столбец и, следовательно, все столбцы заняты. Поэтому мы можем сразу присвоить каждому ферзю определенный столбец и свести задачу к более простой — найти соответствующие строки.

Чтобы прийти к решению, ферзи должны взаимодействовать друг с другом. Поняв это, мы можем сделать следующее важное наблюдение, которое очень упростит нашу задачу, а именно: каждый ферзь должен знать только о своем соседе слева. Таким образом, данные о ферзе будут состоять из трех значений: столбец, который остается неизменным; строка, меняющаяся в ходе решения; и сосед слева.

Определим приемлемое решение для столбца N как такую конфигурацию столбцов с 1 по N, в которой ни один ферзь из этих столбцов не бьет другого. Каждому ферзю будет поручено найти приемлемое решение для себя и своих соседей слева. Мы получим решение всей задачи, когда самый правый ферзь отыщет приемлемое решение. Описание класса Queen (Ферзь) на CRC-карточке, включая данные о каждом представителе (напомним, что эта информация содержится на обороте), приведено на рис. 5.2.

PDF created with pdfFactory Pro trial version www.pdffactory.com

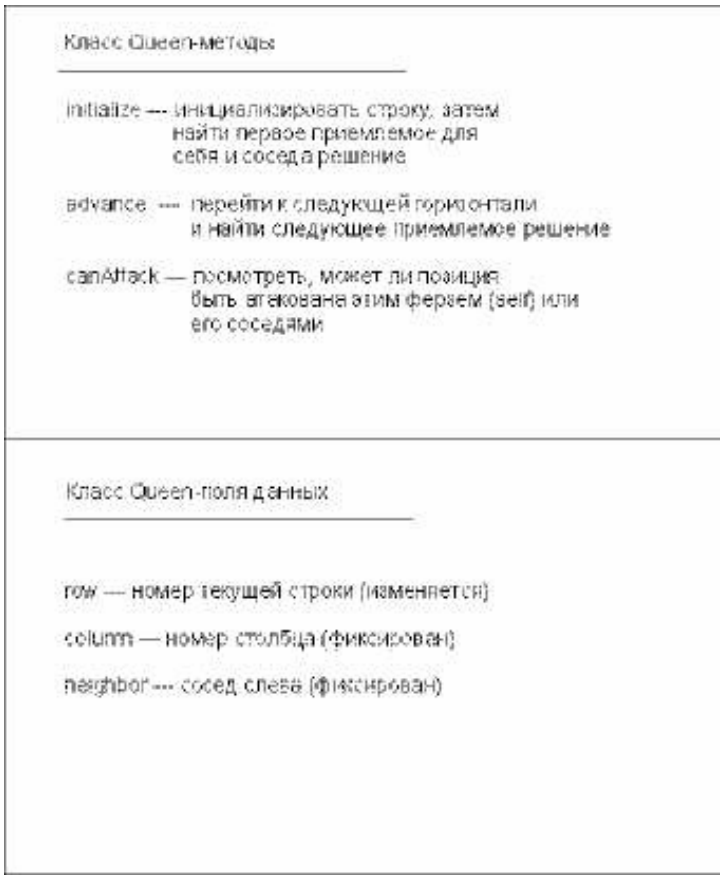



**Рис. 5.2.** Две стороны CRC-карточки для класса Queen

## 5.2. Использование генераторов

Как и в других подобных задачах, решение проблемы восьми ферзей состоит из двух взаимосвязанных шагов: генерация возможных частичных решений и отсеивание решений, не удовлетворяющих дальнейшим тестам. Такой стиль анализа проблем иногда называют «генерация и тестирование» [Hanson 1981], [Bertiss 1990].

Сначала рассмотрим шаг отсеивания как более простой. Для проверки потенциального решения системе достаточно, чтобы ферзь взял координатную пару (строка-столбец) и выдал логическое значение, указывающее, будет ли этот ферзь (или какой-нибудь другой слева) бить это поле (см. приведенный ниже алгоритм на псевдокоде). Процедура canAttack использует тот факт, что при движении по диагонали смещение по строкам равно смещению по столбцам.

```
function queen.canAttack(testRow, testColumn) -> boolean
  /* проверка на ту же строку */
  if row = testRow then
    return true
  /* проверка диагонали */
  columnDifference := testColumn – column
  if (row + columnDifference = testRow) or
    (row – columnDifference = testRow)
      then return true
  /* мы не можем бить, а соседи? */
  return neighbor.canAttack(testRow, testColumn)
end
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 5.2.1. Инициализация

Мы разделим задачу на части. Метод initialize устанавливает необходимые начальные условия, в данном случае — просто задает данные. Далее обычно сразу же следует вызов метода findSolution, находящего решение для данного столбца. Поскольку решение часто будет неудовлетворительным для последующих ферзей, сообщение advance (продвинуться) используется для перехода к следующему решению.

Ферзь из столбца N инициализируется присваиванием номера столбца и определением соседнего ферзя (из столбца N–1). На этом уровне анализа мы оставим неопределенными действия самого левого ферзя, не имеющего соседа. Предполагаем, что ферзи-соседи уже инициализированы, это значит, что они нашли взаимно удовлетворяющее решение. Ферзь из текущего столбца просто ставит себя в первую строку. Ниже приведен алгоритм на псевдокоде.

```
function queen.initialize(col, neigh) -> boolean
  /* инициализация значений столбца и соседа */
  column := col
  neighbor := neigh
  /* начинаем со строки 1 */
  row := 1
end
```

### 5.2.2. Нахождение решения

Чтобы найти решение, ферзь просто спрашивает своих соседей, могут ли они его атаковать. Если да, то ферзь продвигается. Если дальнейшее передвижение запрещено, возвращается значение «неудача». Если соседи сообщают, что они атаковать не могут, то (локальное) решение найдено.

```
function queen.findSolution -> boolean
  /* проверка позиции */
  while neighbor.canAttack (row, column) do
    if not self.advance then
     return false
  /* нашли решение */
  return true
end
```

Как мы указывали в главе 4, псевдопеременная self обозначает получателя последнего сообщения. В данном случае мы хотим, чтобы ферзь, которому поручено найти решение, послал сообщение advance самому себе.

### 5.2.3. Продвижение на следующую позицию

Процедура advance распадается на два случая. Если мы еще не достигли конца, ферзь просто увеличивает номер строки на 1. Если ферзь уже попробовал все позиции и не нашел решения, то не остается ничего, кроме как попросить у соседа нового решения и начать со строки 1.

```
function queen.advance -> boolean
  /* пробуем следующую строку */
  if row < 8 then
  begin
    row := row + 1
    return self.findSolution
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  end
  /* не можем двигаться дальше */
  /* сдвинем соседа к следующему решению */
  if not neighbor.advance then
    return false
  /* начнем снова с 1-й строки */
  row := 1
  return self.findSolution
end
```

Теперь осталось только напечатать решение. Это делается очень просто с помощью метода print, который рекурсивно повторяется для всех соседей.

```
procedure print
  neighbor.print
  write row, column
end
```

## 5.3. Задача о восьми ферзях на различных языках программирования

В этом параграфе мы приведем решение задачи о восьми ферзях для каждого рассматриваемого языка программирования. Сравните тексты программ. Обратите внимание на то, как свойственные языку черты привносят хитроумные изменения в решение задачи. В частности, проанализируйте решения на языках Smalltalk и Objective-C, использующие специальный «караульный» класс, и сравните их с кодом на Object Pascal, C++ или Java, каждый из которых использует нулевой указатель для самого левого ферзя и постоянно вынужден проверять значение указателей.

### 5.3.1. Задача о восьми ферзях: Object Pascal

Определение класса для задачи о восьми ферзях на языке Object Pascal версии Apple приведено ниже. Тонкой и важной чертой является то, что это определение рекурсивно — объекты типа Queen содержат поле данных типа Queen. Это достаточно ясно показывает, что объявление и выделение памяти не обязательно связаны. В противном случае для каждого объекта Queen потребовалось бы бесконечно много памяти. Мы сравним это с кодом для C++ и детально рассмотрим связь между объявлением и выделением памяти в главе 12.

```
type
  Queen = object
    (* поля данных *)
    row  : integer;
    column : integer;
    neighbor : Queen;
    (* инициализация *)
    procedure initialize (col : integer; ngh : Queen);
    (* операции *)
    function canAttack (testRow, testColumn : integer) :
             boolean;
    function findSolution : boolean;
    function advance : boolean;
    procedure print;
end;
```

Для языка Delphi определение класса будет отличаться совсем немного (оно показано ниже). Вариант языка для фирмы Borland позволяет разбить определение класса на

PDF created with pdfFactory Pro trial version www.pdffactory.com

открытую (public) и закрытую (private) части, а также добавить функцию-конструктор, используемую вместо программы initialize.

```
TQueen = class (TObject)
public
  constructor Create (initialColumn : integer;
    nbr : TQueen);
  function findSolution : boolean;
  function advance : boolean;
  procedure print;
private
  function canAttack (testRow, testColumn : integer) :
    boolean;
  row  : integer;
  column : integer;
  neighbor : TQueen;
end;
```

Псевдокод, приведенный в предыдущих параграфах, достаточно близок к варианту на языке Pascal с двумя основными отличиями. Первое: отсутствие в языке Pascal оператора return. Второе: необходимость сначала проверить, есть ли у ферзя сосед, перед посылкой сообщения этому соседу. Функции findSolution и advance, приведенные ниже, демонстрируют эти различия. (Заметим, что язык Delphi Pascal отличается от стандартного Pascal тем, что допускает «быстрое» выполнение директив and (и) и or (или), как и C++. Таким образом, код на языке Delphi может в одном выражении объединить проверку существования ненулевого соседа и посылку сообщения этому соседу.)

```
function Queen.findSolution : boolean;
var
  done : boolean;
begin
  done := false;
  findsolution := true;
  (* проверка позиции *)
  if neighbor <> nil then
  while not done and neighbor.canAttack(row, column) do
    if not self.advance then
    begin
      findSolution := false;
      done := true;
    end;
end;
function Queen.advance : boolean:
begin
advance := false;
  (* пробуем следующую строку *)
  if row < 8 then
  begin
    row := row + 1;
    advance := self.findSolution;
  end
  else begin
    (* не можем двигаться дальше *)
    (* сдвинуть соседа к следующему решению *)
    if neighbor <> nil then
     if not neighbor.advance then
       advance := false
     else begin
       row := 1;
       advance := self.findSolution;
     end;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  end;
end;
```

Основная программа отводит память для каждого из восьми ферзей и инициализирует их, определяя столбец и соседа. Поскольку во время инициализации будет найдено первое (локальное) решение, ферзи должны будут напечатать это решение. Код на языке Apple Object Pascal, выполняющий эту задачу, показан ниже. Здесь neighbor и i — временные переменные, используемые во время инициализации, а lastQueen — последний созданный ферзь.

```
begin
  neighbor := nil;
  for i := 1 to 8 do
  begin
    (*   создать и инициализировать нового ферзя *)
    new (lastQueen);
    lastQueen.initial (i, neighbor);
    if not lastQueen.findSolution then
      writeln('no solution');
    (* самый новый ферзь — сосед предыдущего *)
    neighbor := lastQueen;
  end;
  (* печать решения *)
  lastQueen.print;
  end;
end.
```

Предоставляя явные конструкторы, объединяющие создание новых объектов и их инициализацию, язык Delphi позволяет ликвидировать одну из временных переменных. Основная программа на Delphi будет такова:

```
begin
  lastQueen := nil;
  for i := 1 to 8 do
  begin
    (* создать и инициализировать нового ферзя *)
    lastQueen := Queen.create(i, lastQueen);
    lastQueen.findSolution;
  end;
  // печать решения
  lastQueen.print;
end;
```

### 5.3.2. Задача о восьми ферзях: C++

Наиболее существенной разницей между приведенным ранее описанием алгоритма на псевдокоде и типичным кодом для C++ является непосредственное использование указателей. Ниже приведено описание класса Queen. Каждый представитель класса содержит в себе указатель на другого ферзя. Заметьте, что в отличие от Object Pascal в языке C++ это значение должно быть объявлено как указатель на объект, а не как сам объект.

```
class Queen
{
  public:
   // конструктор
    Queen (int, Queen *);
   // поиск и печать решения
    bool findSolution();
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    bool advance();
    void print();
  private:
  // поля данных
    int row;
    const int column;
    Queen *neighbor;
  // внутренний метод
    bool canAttack (int, int);
};
```

Как и в случае программы на языке Delphi Pascal, мы реализовали метод initialize в конструкторе. Коротко опишем его.

Есть три поля данных. Целочисленное поле column объявлено как const. Это определяет поле как неизменяемое. Третье поле данных имеет тип указателя; оно либо содержит нулевое значение (то есть ни на что не указывает), либо ссылается на другого ферзя.

Поскольку инициализация осуществляется конструктором, основная программа может просто создать восемь объектов-ферзей и потом напечатать решение. Переменная lastQueen указывает на последнего созданного ферзя. Вначале это значение равно null-указателю — ни на что не указывает. Затем в цикле конструируются восемь ферзей, инициализируемых значениями столбца и предыдущего ферзя. Когда цикл закончится, самый левый ферзь будет содержать нулевой указатель в поле neighbor, а все остальные — указатели на своих соседей; lastQueen указывает на самого правого ферзя.

```
void main()
{
  Queen *lastQueen = 0;
  for (int i = 1; i <= 8; i++)
  {
    lastQueen = new Queen(i, lastQueen);
    if (! lastQueen -> findSolution())
      cout << "no solution\n";
  }
  lastQueen -> print();
}
```

Мы опишем только методы, иллюстрирующие важные моменты. Полное решение можно посмотреть в Приложении А.

Конструктор должен использовать в заголовке инициализирующую конструкцию для присваивания значения константе column, так как запрещается присваивать что-либо полям, объявленным как const. Инициализирующая конструкция используется и для поля neighbor, хотя мы не объявляли это поле как константу.

```
Queen::Queen(int col, Queen *ngh)
  : column(col), neighbor(ngh)
{
  row = 1;
}
```

Поскольку значение переменной neighbor может быть либо ссылкой на ферзя, либо нулевым, необходима проверка, прежде чем сообщение будет послано соседу. Это показано в методе findSolution. Использование укороченного вычисления логических выражений и возможность выхода из середины процедуры упрощают код по сравнению с версией на Object Pascal — в остальном же они очень похожи.

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
bool Queen::findSolution()
{
  while (neighbor && neighbor->canAttack(row, column))
    if (! advance())
      return false;
  return true;
}
```

Метод advance подобным же образом должен проверять наличие соседа перед тем, как продвигать его к следующему решению. Посылая сообщение себе, как это делается в рекурсивном findSolution, не обязательно указывать получателя.

```
bool Queen::advance()
{
  if (row < 8)
  {
    row++;
    return findSolution();
  }
  if (neighbor && ! neighbor->advance())
    return false;
  row = 1;
  return findSolution();
}
```

### 5.3.3. Задача о восьми ферзях: Java

Решение на языке Java во многом напоминает код на C++. Однако в Java тело метода пишется прямо «на месте», и указания public и private помещаются в определение класса. Ниже приводится определение класса Queen; некоторые методы опущены.

```
class Queen
{
  // поля данных
  private int row;
  private int column;
  private Queen neighbor;
   // конструктор
  Queen (int c, Queen n)
  {
     // инициализировать поля данных
    row = 1;
    column = c;
    neighbor = n;
  }
  public boolean findSolution()
  {
    while (neighbor != null &&
       neighbor.canAttack(row, column))
     if (! advance ())
       return false;
    return true;
  }
  public boolean advance()
  {
     . . .
  }
  private boolean canAttack(int testRow, int testColumn)
  {
     . . .
  }
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

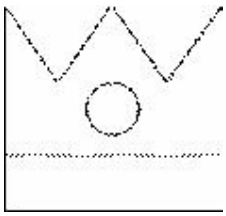
```
  public void paint (Graphics g)
  {
     . . .
  }
}
```

В отличие от языка C++ в Java связь со следующим ферзем определяется как объект типа Queen, а не как указатель на Queen. Перед посылкой сообщения ферзю, определяемому переменной neighbor, выполняется явная проверка на ненулевое значение.

Поскольку язык Java предоставляет богатое множество графических подпрограмм, решение будет отличаться от прочих тем, что окончательная расстановка ферзей будет нарисована на экране. Метод paint рисует ферзя, потом изображает соседей.

```
class Queen
{
  . . .
  public void paint (Graphics g)
  {
    // x, y — левый верхний угол
    int x = (row — 1) * 50;
    int y = (column — 1) * 50;
    g.drawLine(x+5, y+45, x+45, y+45);
    g.drawLine(x+5, y+45, x+5, y+5);
    g.drawLine(x+45, y+45, x+45, y+5);
    g.drawLine(x+5, y+35, x+45, y+35);
    g.drawLine(x+5, y+5, x+15, y+20);
    g.drawLine(x+15, y+20, x+25, y+5);
    g.drawLine(x+25, y+5, x+35, y+20);
    g.drawLine(x+35, y+20, x+45, y+5);
    g.drawLine(x+20, y+20, 10, 10);
    // затем рисуем соседа
    if (neighbor != null)
      neighbor.paint(g);
  }
}
```

Графические программы рисуют маленькую корону, выглядящую вот так:



В языке Java нет ни глобальных переменных, ни «безклассовых» функций. Как мы опишем более детально в главе 8, программа начинается с определения подкласса системного класса Applet и переопределения некоторых методов. В частности, метод init используется для инициализации приложения, а метод paint — для перерисовки экрана. Мы также определим метод mouseDown, вызываемый при нажатии кнопки мыши, чтобы заставить программу переходить к следующему решению. Назовем класс нашего приложения QueenSolver и определим его так:

```
public class QueenSolver extends Applet
{ private Queen lastQueen;
  public void init()
  { lastQueen = null;
    for (int i = 1; i <= 8; i++)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    {  lastQueen = new Queen(i, lastQueen);
       lastQueen.findSolution();
    }
  }
  public void paint(Graphics g)
  { // рисуем доску
    for (int i = 0; i <= 8; i++)
    {
       g.drawLine(50 * i, 0, 50 * i, 400);
       g.drawLine(0, 50 * i, 400, 50 * i);
    }
    // рисуем ферзей
    lastQueen.paint(g);
  }
  public boolean mouseDown(java.awt.Event evt, int x,
    int y)
  {
     // найти и напечатать следующее решение
    lastQueen.advance();
    repaint();
    return true;
  }
}
```

Заметьте, что класс приложения должен быть объявлен как public, чтобы быть доступным в основной программе.

### 5.3.4. Задача о восьми ферзях: Objective-C

Описание интерфейса нашего класса Queen таково:

```
@interface Queen : Object
{  /* поля данных */
   int row;
   int column;
   id neighbor;
}
 /* методы */
- (void) initialize: (int) c neighbor: ngh;
- (int) advance;
- (void) print;
- (int) canAttack: (int) testRow column:
    (int) testColumn;
- (int) findSolution;
@end
```

Каждый ферзь имеет три поля данных: строку, столбец и ферзя-соседа. Последнее объявлено с типом id. Это указывает, что значением переменной будет объект, хотя и не обязательно ферзь.

В действительности мы можем использовать бестиповую сущность переменных в языке Objective-C себе во благо. Применим технику, которая невозможна или, по крайней мере, не так проста в языках со строгими ограничениями на тип (такими, как C++ или Object Pascal). Вспомним, что самый левый ферзь не имеет соседа. В решении на языке C++ признаком этого служило нулевое значение переменной, содержащей указатель на соседа, для самого левого ферзя. В данном решении мы вместо этого создаем новый класс — караульное (sentinel) значение. Самый левый ферзь будет указывать на это караульное значение, тем самым обеспечивая действительного соседа каждому ферзю.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Караульные величины часто используются как маркеры конца; их можно обнаружить в алгоритмах, работающих со списками, такими как наш список ферзей. Различие между объектно-ориентированным «караулом» и более традиционными проверками состоит в том, что первый может быть активным — у него есть поведение, то есть он может отвечать на запросы.

Каково должно быть поведение караульных величин? Вспомним, что ссылки на соседа в нашем алгоритме используются двояко. Во-первых, чтобы убедиться, что интересующая нас позиция не атакована; караульная величина на такой запрос всегда должна отвечать отрицательно, поскольку она не может ничего бить. Второе использование ссылок на соседа — в рекурсивном вызове процедуры print. В этом случае караульная величина просто ничего не делает, так как не имеет никакой информации о решении.

Объединяя вместе все эти рассуждения, приходим к следующей реализации нашего «караульного ферзя»:

```
@implementation SentinelQueen : Object
- (int) advance
{
  /* ничего не делаем */
  return 1;
}
- (int) findSolution
{
  /* ничего не делаем */
  return 1;
}
- (void) print
{
  /* ничего не делаем */
}
- (int) canAttack: (int) testRow column:
    (int) testColumn;
{
  /* не можем атаковать */
  return 0;
}
@end
```

В полном решении есть часть, реализующая SentinelQueen, но к ней нет интерфейса. Это допустимо, хотя компилятор выдаст предупреждение, поскольку такая ситуация несколько необычна.

Использование караульного объекта позволяет методам класса Queen просто посылать сообщения соседу без предварительного изучения, является ли данный ферзь самым левым. Например, в методе canAttack:

```
- (int) canAttack: (int) testRow column:
    (int) testColumn
{  int columnDifference;
    /* можем бить ту же строку */
   if (row == testRow)
     return 1;
   columnDifference = testColumn – column;
   if ((row + columnDifference == testRow) ||
       (row – columnDifference == testRow))
     return 1;
   return [ neighbor canAttack:testRow column:
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    testColumn ];}
```

Внутри метода посылка сообщения самому себе осуществляется с помощью псевдопеременной self:

```
- (void) initialize: (int) c neighbor: ngh
{
   /* установить поля — константы */
  column = c;
  neighbor = ngh;
  row = 1;
}
- (int) findSolution
{
  /* цикл, пока не найдем решение */
  while ([neighbor canAttack: row and: column ])
    if (! [self advance])
      return 0; /* возвращаем false */
  return 1;    /* возвращаем true */
}
```

Остальные методы аналогичны и здесь не рассматриваются.

### 5.3.5. Задача о восьми ферзях: Smalltalk

Решение задачи о восьми ферзях на языке Smalltalk во многих отношениях похоже на Objective-C. Как и последний, язык Smalltalk учитывает тот факт, что самый левый ферзь не имеет соседа, вводя специальный караульный (sentinel) класс. Его единственная цель в том, чтобы предоставить получателя сообщений, посланных самым левым ферзем.

Караульная величина является единственным представителем класса SentinelQueen (подкласс класса Queen), который реализует следующие три метода:

```
advance
  " караульный ферзь не атакует "
   false
canAttack: row column: column
  " караульный ферзь не может атаковать "
   false
result
  " возвращаем пустой список в качестве результата "
   List new
```

Единственное различие между версиями на Objective-C и Smalltalk в том, что программа на языке Smalltalk возвращает результат как список величин, а не выводит их на печать. Техника вывода на печать довольно хитро устроена в Smalltalk и различается от реализации к реализации. Возвращая список, мы можем отделить эти различия от основных методов.

Класс Queen является подклассом класса Object. Представители класса Queen содержат три внутренние переменные: row (строка), column (столбец) и neighbor (сосед). Инициализация осуществляется в методе setColumn : neighbor:.

```
setColumn: aNumber neighbor: aQueen
  " инициализация полей данных "
  column := aNumber.
  neighbor := aQueen.
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  row := 1.
```

Метод canAttack отличается от аналогичного метода для языка Objective-C только синтаксисом:

```
canAttack: testRow column: testColumn |
    columnDifference |
  columnDifference := testColumn – column.
  ((row = testRow) or:
    [ row + columnDifference = testRow]) or:
    [ row – columnDifference = testRow])
     ifTrue: [  true ].
 neighbor canAttack: testRow column: testColumn
```

Вместо того чтобы проверять условие на отрицание, язык Smalltalk предоставляет явный оператор ifFalse, используемый в методе advance:

```
advance
  " сначала попробуем следующую строку "
  (row < 8)
    ifTrue: [ row := row + 1.  self findSolution].
  " не можем двигаться дальше, сдвигаем соседа "
  (neighbor advance)
    ifFalse: [  false ].
  " начнем со строки 1 "
  row := 1.
   self findSolution
```

Цикл while в языке Smalltalk должен использовать блок при проверке условия, как в следующем примере:

```
findSolution
  [ neighbor canAttack: row column: column ]
     whileTrue: [ self advance ifFalse: [  false ] ].
   true
```

Для получения списка окончательных позиций используется рекурсивный метод. Вспомним, что караульная величина создает пустой список в ответ на сообщение result:

```
result
   neighbor result; addLast: row
```

Решение будет получено с помощью вызова следующего метода, не являющегося частью класса Queen и относящегося к некоторому другому классу, например, такому как Object:

```
solvePuzzle | lastQueen |
  lastQueen := SentinelQueen new.
   1 to: 8 do: [:i | lastQueen := (Queen new)
    setColumn: i neighbor: lastQueen.
    lastQueen findSolution ].
   lastQueen result
```

Решение задачи о восьми ферзях, построенное без применения караульной величины, описано в моей более ранней книге по языку Smalltalk [Budd 1987].

## Упражнения

PDF created with pdfFactory Pro trial version www.pdffactory.com

1. Измените программы так, чтобы они выдавали все возможные решения, а не только одно. Сколько существует решений задачи о восьми ферзях? Сколько из них являются поворотами других? Как можно отбросить повороты?
2. Как вы можете объяснить, что караульный класс в языках Objective–C и Smalltalk не предоставляет свою версию метода findSolution, несмотря на то что сообщение findSolution посылается соседу в методе advance?
3. Предположим, мы обобщим задачу о восьми ферзях как проблему N ферзей. Задача: как расположить N ферзей на шахматной доске N r N? Как изменятся программы? Ясно, что существуют N, для которых нет решений (например, N=2 или N=3). Что будет в этом случае с нашими программами? Как можно организовать более осмысленный вывод ответа?
4. Используя графические возможности вашей системы, измените одну из программ так, чтобы она динамически изображала на шахматной доске позиции каждого ферзя по мере своей работы. Какие части кода должны знать об устройстве вывода?

## *Глава 6: Учебный пример: игра «Бильярд»*

Во втором примере мы построим простую имитацию бильярдного стола. Программа написана на языке Object Pascal для Macintosh [1] . Как и в случае с восемью ферзями, разработка делает упор на создание автономных агентов, взаимодействующих между собой для достижения желаемого результата.

### 6.1. Элементы бильярда

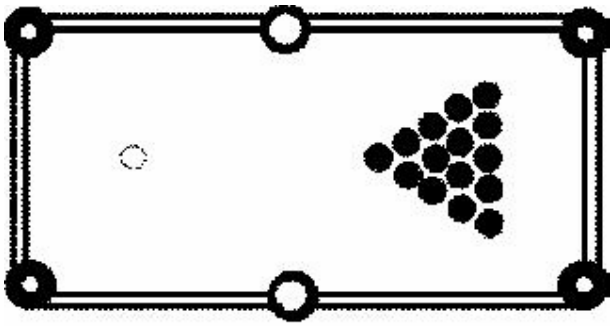
Для пользователя бильярдный стол представляет собой окно, содержащее прямоугольник с лузами по углам, 15 черных шаров и 1 белый шар. Нажатием кнопки мыши пользователь имитирует удар кием по шару, сообщая ему некоторую энергию. Шар движется в сторону, противоположную указателю мыши. Получив энергию, шар начинает катиться, отскакивая от стенок, ударяя другие шары, и, наконец, попадает в лузу.

---

[1] Программа «Бильярд» была разработана на компьютере PowerPC Macintosh с использованием компилятора CodeWarrior Pascal версии 1.1. В версии для PowerPC движение столь быстро, что я решил вставить в процедуру Ball.update замедляющий цикл, несколько раз вызывающий метод draw. Игра, реализованная программой из этой главы, не соответствует никакой настоящей игре. Это не пул, это не бильярд, это просто движение шаров по столу со стенками и лузами.

Когда один шар сталкивается с другим, часть энергии первого передается второму и в результате направление движения обоих шаров меняется.

PDF created with pdfFactory Pro trial version www.pdffactory.com



## 6.2. Графические объекты

Основу имитации составляют три списка графических объектов, представляющих стенки, лузы и шары. Каждый графический объект включает в себя поле ссылки и поле, описывающее местоположение объекта на экране [1].

Мы ввели упрощающее предположение, что все графические объекты занимают прямоугольную область. Это, конечно, совершенно неверно для круглых объектов наподобие шара. Более реалистичной альтернативой было бы написать процедуру, определяющую, пересеклись ли два шара, на основе их действительной геометрии. Но сложность процедуры только отвлекла бы нас от того главного, чего мы хотим добиться, приводя этот пример, а именно понять способ наделения объектов ответственностью за их поведение. Каждый графический объект знает не только как изображать себя, но и как взаимодействовать с другими объектами нашей модели мира.

### 6.2.1. Графический объект Wall (стенка)

Первым из наших трех графических объектов является стенка Wall. Она определяется следующим описанием класса:

---

[1] Неясно, куда поместить изучение этого примера. С одной стороны, читателю важно как можно быстрее увидеть применение объектных принципов; поэтому желательно, чтобы этот пример встретился в книге пораньше. С другой стороны, эта программа только выиграла бы от более изощренной техники, которая обсуждается ниже. В частности, графические объекты было бы лучше представлять в виде иерархии наследования, как описано в главе 7. Кроме того, считается плохим стилем программирования размещение ссылочных полей в области данных объектов, объединенных в список; лучше отделить контейнер и элементы списка. Решение этих проблем нетривиально и содержит сложности. Мы обсудим классы контейнеров в главе 15.

```
Wall = object
  (* поля данных *)
  link : Wall;
  region : Rect;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  (* угол отскока шаров *)
  convertFactor : real;
  (* инициализирующая функция *)
  procedure initialize
    (left, top, right, bottom : integer; cf : real);
  (* изображение стенки *)
  procedure draw;
  (* сообщение стенке, что о нее ударился шар *)
  procedure hitBy (aBall : Ball);
end;
```

Поле link (ссылка) служит для поддержания списка объектов Wall. Инициализирующий метод просто задает местоположение (region) стенки и параметр отскока (convert factor):

```
procedure Wall.initialize
  (left, top, right, bottom : integer; cf : real);
begin
  (* инициализация convertFactor *)
  convertFactor := cf;
  (* установить область для стены *)
  SetRect (region, left, top, right, bottom);
end;
```

Стенка может быть нарисована просто как сплошной прямоугольник. Это выполняется стандартной процедурой для Macintosh:

```
procedure Wall.draw;
begin
  PaintRect (region);
end;
```

Самое интересное происходит со стенкой, когда о нее ударяется шар. Направление его движения изменяется, основываясь на значении параметра convertFactor для стенки. (Переменная convertFactor равна или нулю, или pi, в зависимости от того, горизонтальная стенка или вертикальная.) В результате столкновения шар будет двигаться в новом направлении.

```
procedure Wall.hitBy (aBall : Ball);
begin
  (* оттолкнем шар от стенки *)
  aBall.setDirection(convertFactor – aBall.direction);
end;
```

### 6.2.2. Графический объект Hole (луза)

Hole (луза) определяется следующим описанием класса:

```
Hole = object
  (* поля данных *)
  link : Hole;
  region : Rect;
  (* инициализирующая функция *)
  procedure initialize (x, y : integer);
  (* изображение лунки *)
  procedure draw;
  (* сообщение лузе, что в нее попал шар *)
  procedure hitBy (aBall : Ball);
end;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Как и в случае стенок, инициализация и изображение лузы в основном состоят из вызова соответствующих стандартных подпрограмм:

```
procedure Hole.initialize (x, y : integer);
begin
  (* определить область с центром в x, y *)
  SetRect(region, x-5, y-5, x+5, y+5);
end;
procedure Hole.draw;
begin
  PaintOval (region);
end;
```

Больший интерес представляет происходящее при «ударе» шара о лузу. Есть два случая. Если шар оказался белым (он идентифицируется глобальной переменной CueBall), то он возвращается назад в игру на определенную позицию. В остальных случаях шар лишается энергии и убирается со стола в специальную область.

```
procedure Hole.hitBy (aBall : Ball);
begin
  (* остановить шар; убрать его со стола *)
  aBall.energy := 0.0;
  aBall.erase;
  (* передвинуть шар *)
  if aBall = CueBall then
   aBall.setCenter(50.100);
  else begin
   saveRack := saveRack + 1;
   aBall.setCenter (10 + saveRack * 15, 250);
  end;
  (* перерисовать шар *)
  aBall.draw;
end;
```

### 6.2.3. Графический объект Ball (шар)

Последним графическим объектом является шар, определяемый следующим описанием класса:

```
Ball = object
  (* поля данных для шара *)
  link  : Ball;
  region : Rect;
  direction : real; (* направление в радианах *)
  energy : real;
  (* инициализирующая функция *)
  procedure initialize (x, y : integer);
  (* методы *)
  procedure draw;
  procedure erase;
  procedure update;
  procedure hitBy (aBall : Ball);
  procedure setDirection (newDirection : real);
  (* возвращают x, y — координаты центра шара *)
  function x : integer;
  function y : integer;
end;
```

В дополнение к полям ссылки (link) и местоположения (region), общими с остальными объектами, шар имеет два новых поля данных: direction (направление), вычисленное в

PDF created with pdfFactory Pro trial version www.pdffactory.com

радианах, и energy (энергия), представляющее собой вещественное значение. Как и в случае лузы, шар инициализируется аргументами, описывающими координаты его центра. Первоначально шар не имеет энергии и его направление нулевое.

```
procedure Ball.initialize (x, y : integer);
begin
  SetRect (region, x-5, y-5, x+5, y+5);
  setDirection (0.0);
  energy := 0.0;
end;
```

Шар изображается либо окружностью, либо сплошным кругом, в зависимости от того, является ли он белым или нет.

```
procedure Ball.draw;
begin
  if self = CueBall then
   (* рисуем окружность *)
   FrameOval (region);
  else
   (* рисуем круг *)
   PaintOval (region);
end;
procedure Ball.erase;
begin
  EraseRect (region);
end;
```

Метод update используется для изменения позиции шара. Если он имеет заметную энергию, то слегка сдвигается, а затем проверяет, не задел ли он другой объект. Глобальная переменная ballMoved устанавливается в true, если какой-либо шар на столе сдвинулся. Если шар задел другой объект, шар сообщает об этом объекту. Сообщения бывают трех видов; они соответствуют ударам по лузе, стенке и другим шарам. Наследование, которое мы изучаем в главе 7, предоставляет методы объединения этих трех тестов в один цикл.

```
procedure Ball.update;
var
  hptr : Hole;
  wptr : Wall;
  bptr : Ball;
  dx, dy : integer;
  theIntersection : Rect;
begin
  if energy > 0.5 then
  begin
    ballMoved := true;
    (* удалить шар *)
    erase;
    (* уменьшить энергию *)
    energy := energy – 0.05;
    (* сдвинуть шар *)
    dx := trunc(5.0 * cos(direction));
    dy := trunc(5.0 * sin(direction));
    offsetRect(region, dx, dy);
    (* перерисовать шар *)
    draw;
    (* проверить, не попали ли в лузу *)
    hptr := listOfHoles;
    while (hptr <> nil) do
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
      if SectRect (region, hptr.region, theIntersection)
      then
      begin
        hptr.hitBy(self);
        hptr := nil;
      end
      else
        hptr := hptr.link;
    (* проверить, не ударились ли в стенку *)
    wptr := listOfWalls;
    while (wptr <> nil) do
      if SectRect (region, wptr.region, theIntersection)
      then
      begin
        wptr.hitBy(self);
        wptr := nil;
      end
      else
        wptr := wptr.link;
   (* проверить, не ударили ли шар *)
   bptr := listOfBalls;
   while (bptr <> nil) do
     if SectRect (region, bptr.region, theIntersection)
     then
     begin
       bptr.hitBy(self);
       bptr := nil;
     end
     else
       bptr := bptr.link;
  end;
end;
```

Когда один шар ударяет другой, энергия первого делится пополам между ними. Также меняются направления движения обоих шаров.

```
procedure Ball.hitBy (aBall : Ball);
var
  da : real;
begin
  (* уменьшить энергию ударяющего шара наполовину *)
  aBall.energy := aBall.energy / 2;
  (* и добавить ее к нашему шару *)
  energy := energy + aBall.energy;
  (* установить наше новое направление *)
  setDirection(hitAngle(self.x aBall.x,
    self.y – aBall.y);
  (* и направление ударяющего шара *)
  da := aBall.direction – direction;
  aBall.setDirection (aBall.direction + da);
end;
function hitAngle (dx, dy : real) : real;
const
  PI = 3.14159;
var
  na : real;
begin
  if (abs(dx) < 0.05) then
    na := PI / 2;
  else
    na := arctan (abs(dy / dx));
 if (dx < 0) then
    na := PI – na;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  if (dy < 0) then
   na := -na;
  hitAngle := na;
end;
```

### 6.3. Основная программа

В предыдущем параграфе описывались статические характеристики программы. Динамика начинается при нажатии кнопки мыши. При этом вызывается следующая процедура:

```
procedure mouseButtonDown (x, y : integer);
var
  bptr : Ball;
begin
  (* присвоим белому шару некоторую энергию *)
  CueBall.energy := 20.0;
  (* и направление *)
  CueBall.setDirection(hitAngle (CueBall.x – x,
    CueBall.y – y));
  (* изменения происходят, пока движется хотя бы один шар *)
  ballMoved := true;
  while ballMoved do
  begin
    ballMoved := false;
    bptr := listOfBalls;
    while bptr <> nil do
    begin
      bptr.update;
      bptr := bptr.link;
    end;
  end;
end;
```

Оставшаяся часть программы относительно прямолинейна и не представлена здесь. Весь текст находится в Приложении Б. Основная часть кода связана с инициализацией новых объектов и организацией цикла ожидания события, то есть действия пользователя.

Главное — понять то, как было децентрализовано управление и как сами объекты были наделены возможностями влиять на ход выполнения программы. Все, что происходит при нажатии кнопки мыши, — это наделение белого шара некоторой энергией. В дальнейшем модель руководствуется исключительно взаимодействием шаров.

### 6.4. Использование наследования

В главе 1 мы описали наследование неформально, а в главе 7 обсудим, как оно работает в каждом из рассматриваемых нами языков. Здесь мы поясним, как наследование используется для упрощения имитации бильярда. Думается, что читателю лучше вернуться к этому параграфу после ознакомления с общими положениями о наследовании в следующей главе.

Первым шагом в использовании наследования в нашей имитации бильярда является описание общего класса «графический объект». Он породит трех потомков: шары, стенки и лузы. Родительский класс определяется следующим образом:

```
GraphicalObject = object
  (* поля данных *)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  link : GraphicalObject;
  region : Rect;
  (* инициализирующая функция *)
  procedure setRegion (left, top, right, bottom :
    integer);
  (* операции, выполняемые графическими объектами *)
  procedure draw;
  procedure erase;
  procedure update;
  function intersect (anObj : GraphicalObject) :
    boolean;
  procedure hitBy (anObj : GraphicalObject);
end;
```

Инициализирующая функция setRegion просто устанавливает область, занимаемую объектом. Методы draw и update ничего не делают, так как их фактическое поведение определено в дочерних классах. Программа erase очищает область, занимаемую объектом. intersect возвращает значение true, если объект-аргумент пересекается с рассматриваемым объектом. И наконец, метод hitBy также переопределяется в дочерних классах. Хотя двигаются только шары и, следовательно, аргументом этой функции всегда будет шар, тот факт, что класс Ball еще не определен, означает, что мы должны объявить аргумент как имеющий более общий тип GraphicalObject:

```
procedure GraphicalObject.setRegion
  (left, top, right, bottom : integer);
begin
  SetRect(region, left, top, right, bottom);
end;
procedure GraphicalObject.draw;
begin
  (* переопределяется в дочернем классе *)
end;
procedure GraphicalObject.erase;
begin
  EraseRect (region);
end;
procedure GraphicalObject.update;
begin  (* переопределяется в дочернем классе *)
end;
procedure GraphicalObject.hitBy(anObject :
  GraphicalObject);
begin  (* переопределяется в дочернем классе *)
end;
function GraphicalObject.intersect
   (anObject : GraphicalObject) : boolean;
var
  theIntersection : Rect;
begin
  intersect := SectRect
   (region, anObject.region, theIntersection);
end;
```

Теперь Ball, Wall и Hole объявляются как подклассы общего класса GraphicalObject, и внутри них ни к чему объявлять данные или функции, если только они не переопределяются:

```
Hole = object (GraphicalObject)
  (* инициализация местоположения лузы *)
  procedure initialize (x, y : integer);
  (* изображение лузы *)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  procedure draw; override;
  (* сообщить лузе, что в нее попал шар *)
  procedure hitBy (anObject : GraphicalObject);
    override;
end;
```

Процедура hitBy должна преобразовать тип аргумента в Ball. Благоразумно проверить тип до приведения:

```
procedure Wall.hitBy (anObj : GraphicalObject);
var
  aBall : Ball;
begin
  if Member (anObj, Ball) then
  begin
   aBall := Ball(anObj);
   aBall.setDirection(convertFactor – aBall.direction);
  end;
end;
```

Делая класс CueBall подклассом Ball, мы ликвидируем условный оператор в программе изображения шара.

```
CueBall = Object (Ball)
  procedure draw; override;
end;
procedure Ball.draw;
begin
  (* рисуем круг *)
  PaintOval (region);
end;
procedure CueBall.draw;
begin
  (* рисуем окружность *)
  FrameOval (region);
end;
```

Наибольшее упрощение достигается тем, что теперь можно держать все графические объекты в одном списке. Программа, рисующая весь экран, записывается так:

```
procedure drawBoard;
var
  gptr : GraphicalObject;
begin
  SetPort (theWindow);
  gptr := listOfObjects;
  while gptr <> nil do begin
    gptr.draw;
    gptr := gptr.link;
  end;
end;
```

Наиболее важным местом этого кода является вызов функции draw внутри цикла. Несмотря на то что вызов написан один, иногда будет вызываться функция класса Ball, а в других случаях — класса Wall или Hole. Тот факт, что одно обращение к функции может привести к вызовам различных функций, относится к понятию полиморфизма. Мы обсудим его в главе 14.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Часть подпрограммы Ball.update, проверяющая, ударился ли движущийся шар обо что-нибудь, также упрощается аналогичным образом. Это можно увидеть в полном исходном тексте в Приложении Б.

### Упражнения

1. Предположим, вы хотите производить определенное действие каждый раз, когда программа «Бильярд» выполняет цикл обработки события. В каком месте лучше всего поместить этот код?
2. Предположим, вы хотите сделать шары цветными. Какие части программы вам придется изменить?
3. Предположим, вы хотите добавить лузы на боковых стенках, как на обычном бильярдном столе. Какие части программы вам придется изменить?
4. Программа «Бильярд» использует метод, при котором в цикле просматривается список шаров и каждый шар, имеющий энергию, немного сдвигается. Альтернативный и более объектно-ориентированный подход заключается в том, чтобы позволить каждому шару, пока он имеет энергию, изменять свое состояние и состояние шаров, которые он задевает. Тогда для запуска модели бильярда необходимо только придать движение белому шару. Измените программу, чтобы использовать этот подход. Что дает более реальную модель? Почему?

## *Глава 7: Наследование*

Первым шагом при изучении объектно-ориентированного программирования было осознание задачи как взаимодействия программных компонент. Этот организационный подход был главным при разборе примеров в главах 5 и 6.

Следующим шагом в изучении объектно-ориентированного программирования станет организация классов в виде иерархической структуры, основанной на принципе наследования. Под наследованием мы понимаем возможность доступа представителей дочернего класса (подкласса) к данным и методам родительского класса (надкласса).

### 7.1. Интуитивное описание наследования

Давайте вернемся к Фло — хозяйке цветочного магазина из первой главы. Мы вправе ожидать от нее вполне определенного поведения не потому, что она хозяйка именно цветочного магазина, а потому, что она хозяйка магазина. Например, Фло попросит вас оплатить заказ, а затем даст вам квитанцию. Эти действия не являются уникальными для владельца цветочного магазина; они общие для булочников, бакалейщиков, продавцов канцелярских товаров и автомобилей и т. д. Таким образом, мы как бы связали определенное поведение с общей категорией «хозяева магазинов» Shopkeeper, и поскольку хозяева (и хозяйки) цветочных магазинов (Florist) являются частным случаем категории Shopkeeper, поведение для данного подкласса определяется автоматически.

В языках программирования наследование означает, что поведение и данные, связанные с дочерним классом, всегда являются расширением (то есть большим множеством) свойств, связанных с родительскими классами. Подкласс имеет все свойства родительского класса и, кроме того, дополнительные свойства. С другой стороны, поскольку дочерний класс является более специализированной (или ограниченной) формой родительского класса, он также, в определенном смысле, будет сужением родительского класса. Это диалектическое противоречие между наследованием как расширением и наследованием как сужением является источником большой силы, присущей данной технике, и в то же

PDF created with pdfFactory Pro trial version www.pdffactory.com

время вызывает некоторую путаницу. Мы это увидим в следующих разделах при практическом изучении наследования.

Наследование всегда транзитивно, так что класс может наследовать черты надклассов, отстоящих от него на несколько уровней. Например, если собаки Dog являются подклассом класса млекопитающих Mammal, а млекопитающие Mammal являются подклассом класса животных Animal, то класс собак Dog наследует свойства и млекопитающих Mammal, и всех животных Animal.

Усложняющим обстоятельством в нашем интуитивном описании наследования является тот факт, что подклассы могут переопределять поведение, унаследованное от родительского класса. Например, класс утконосов Platypus переопределяет процедуру размножения, унаследованную от класса млекопитающих Mammal, поскольку утконосы откладывают яйца. В этой главе мы коротко коснемся механизма переопределения. К более детальному обсуждению семантики наследования мы вернемся в главе 11.

## 7.2. Подкласс, подтип и принцип подстановки

Рассмотрим связь между типом данных, связанным с родительским классом, и типом данных, связанным с дочерним классом. Можно утверждать следующее:

- Представители подкласса должны владеть всеми областями данных родительского класса.
- Представители подкласса должны обеспечивать выполнение, по крайней мере через наследование (если нет явного переопределения), всех функциональных обязанностей родительского класса. Это не противоречит тому, что у нового класса могут появиться дополнительные обязанности.
- Представитель дочернего класса может имитировать поведение родительского класса и должен быть неотличим от представителя родительского класса в сходных ситуациях.

Ниже в этой главе мы, разбирая различные варианты наследования, увидим что эти утверждения не всегда верны. Тем не менее они дают хорошее описание идеализированного подхода к наследованию. Поэтому формализуем такой подход в виде принципа подстановки.

Принцип подстановки утверждает, что если есть два класса A и B такие, что класс B является подклассом класса A (возможно, отстоя в иерархии на несколько ступеней), то мы должны иметь возможность подставить представителя класса B вместо представителя класса A в любой ситуации, причем без видимого эффекта.

Как мы увидим в главе 10, термин подтип часто применяется к такой связи «класс—подкласс», для которой выполнен принцип подстановки, в отличие от общего случая, в котором этот принцип не всегда удовлетворяется.

Мы видели применение принципа подстановки в главе 6. В разделе 6.4 описывалась следующая процедура:

```
procedure drawBoard;
var
   gptr : GraphicalObject;
begin
   SetPort (theWindow);
   (* нарисовать все графические объекты *)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
   gptr := listOfObjects;
   while gptr <> nil do
   begin
      gprt.draw;
      gptr := gptr.link;
   end;
end;
```

Глобальная переменная listOfObjects относится к списку графических объектов одного из трех типов. Переменная gptr объявлена просто как графический объект, хотя во время выполнения тела цикла она принимает значения, которые фактически представляют собой объекты порожденных подклассов. В одном случае gptr содержит шар, в другом — лузу, а в третьем — стенку. При обращении к функции draw всегда вызывается подходящий метод для текущего значения gptr, отличный от метода объявленного класса GraphicalObject. Для того чтобы эта процедура работала верно, необходимо, чтобы функциональные возможности каждого из подклассов соответствовали ожидаемым функциональным обязанностям, определенным для родительского класса; то есть подклассы должны быть также и подтипами.

### 7.2.1. Подтипы и строгий контроль типов данных

Языки программирования со статическими типами данных (такие, как C++ и Object Pascal) делают более сильный упор на принцип подстановки, чем это имеет место в языках с динамическими типами данных (Smalltalk и Objective-C). Причина этого в том, что языки со статическими типами данных склонны характеризовать объекты через приписанные им классы, тогда как языки с динамическими типами данных — через их поведение. Например, полиморфная функция (функция, которая может принимать в качестве аргументов объекты различных классов) в языке со статическими типами данных может обеспечить должный уровень функциональных возможностей, только потребовав, чтобы все аргументы были подклассами нужного класса. Поскольку в языке с динамическими типами данных аргументы вовсе не имеют типа, подобное требование просто означало бы, что аргумент должен уметь отвечать на определенный набор сообщений. Дальнейшее обсуждение статических и динамических типов данных мы проведем в главе 10, а детальное изучение полиморфизма — в главе 14.

## 7.3. Формы наследования

Наследование применяется на удивление по-разному. В этом разделе мы опишем наиболее часто используемые варианты. Заметим, что в следующем списке приведены абстрактные категории общего характера, и он не претендует на полноту. Более того, иногда происходит так, что различные методы класса реализуют наследование по-разному.

В следующих разделах [1] обратите особое внимание на то, когда наследование

---

[1] Описанные здесь категории взяты из [Halbert 1987], хотя я добавил несколько собственных. Пример «редактируемого окна» взят из [Meyer 1988a]. обеспечивает порождение подтипов, а когда — нет.

### 7.3.1. Порождение подклассов для специализации (порождение подтипов)

Наверное наиболее часто порождение подклассов и наследование используются для специализации. При порождении подкласса для специализации новый класс является специализированной формой родительского класса, но удовлетворяет спецификациям родителя во всех существенных моментах. Таким образом, для этой формы полностью

PDF created with pdfFactory Pro trial version www.pdffactory.com

выполняется принцип подстановки. Вместе со следующей категорией (наследование для спецификации) специализация является наиболее идеальной формой наследования, к которой должна стремиться хорошая программа.

Вот пример порождения подклассов со специализацией. Класс окон Window предоставляет общие операции с окнами (сдвиг, изменение размеров, свертывание и т. д.). Специализированный подкласс текстовых окон TextEditWindow наследует операции с окнами и дополнительно обеспечивает средства, позволяющие окну отображать текстовую информацию, а пользователю — ее редактировать. Поскольку класс TextEditWindow удовлетворяет всем свойствам, ожидаемым от окон в общем виде (и, следовательно, является подтипом класса Window в дополнение к тому, что он является его подклассом), мы распознаем такую ситуацию как порождение подклассов для специализации.

### 7.3.2. Порождение подкласса для спецификации

Следующий вариант использования наследования состоит в том, чтобы гарантировать поддержку классами определенного общего интерфейса, то есть реализацию ими одних и тех же методов. Родительский класс может быть комбинацией реализованных операций и операций, осуществление которых доверено дочерним классам. Часто нет никаких отличий в интерфейсе между родительским и дочерним классами — последний просто обеспечивает выполнение описанного, но не реализованного в родительском классе поведения.

Фактически это специальный случай порождения специализирующего подкласса за исключением того, что подклассы являются не усовершенствованием

существующего типа, а, скорее, реализацией неполной, абстрактной спецификации. В таких случаях родительский класс иногда называют абстрактно специфицированным классом.

В примере, имитирующем игру бильярд, в главе 6 класс графических объектов GraphicalObject является абстрактным классом, поскольку он не реализует методы отображения объектов и их реакцию на соприкосновение с шаром. Последующие классы Ball, Wall и Hole используют порождение подклассов для спецификации, обеспечивая реальное воплощение этих методов.

В общем случае порождение подклассов для спецификации распознается по тому, что фактическое поведение не определено в родительском классе — оно только описано и будет реализовано в дочернем классе.

### 7.3.3. Порождение подкласса с целью конструирования

Часто класс наследует почти все функциональное поведение родительского класса, изменяя только имена методов или определенным образом модифицируя аргументы. Это может происходить даже в том случае, когда новому и родительскому классам не удается сохранить между собой отношение «быть экземпляром» (is-a relationship).

Например, иерархическая структура классов в языке Smalltalk реализует обобщение массива, называемое Dictionary (словарь). Словарь представляет собой набор пар «ключ–значение»; ключи могут быть произвольными. Таблицу символов, которую можно было бы использовать в компиляторе, разумно представить словарем, в котором индексами

PDF created with pdfFactory Pro trial version www.pdffactory.com

служат символические имена, а значения имеют фиксированный формат, определенный для отдельных записей этой таблицы. Следовательно, класс SymbolTable (таблица символов) должен быть порожден из класса Dictionary введением новых методов, специфичных для таблицы символов. Другим примером может служить построение совокупности абстрактных данных на основе базового класса, обеспечивающего методы работы со списками. В обоих случаях дочерний класс не является более специализированной формой родительского класса, так как у нас и в мыслях не будет подставлять представителей дочернего класса туда, где используются представители родительского класса.

Типичный пример порождения с целью конструирования наблюдается при создании класса, записывающего значения в двоичный файл, например, в системах хранения информации. Родительский класс, как правило, обеспечивает запись только неструктурированных двоичных данных. Подкласс строится для каждой структуры, требующей сохранения. Он реализует процедуру хранения для определенного типа данных, которая использует методы родительского класса для непосредственной записи [1].

В языках со статическими типами данных косо смотрят на порождение подклассов для конструирования, поскольку оно часто напрямую нарушает принцип подстановки (появляются подклассы, не являющиеся подтипами). С другой стороны, будучи быстрым и легким способом построения новых абстракций, оно широко используется в языках с динамическими типами данных. В библиотеке стандартных программ языка Smalltalk можно найти множество примеров порождения подклассов с целью конструирования.

Мы изучим конкретный вариант порождения с целью конструирования в главе 8. Там же мы узнаем, что язык программирования C++ предоставляет интересный механизм: закрытое наследование, который позволяет порождать подклассы для конструирования без нарушения принципа подстановки.

### 7.3.4. Порождение подкласса для обобщения

Использование наследования при порождении подклассов для обобщения в определенном смысле является противоположностью порождению для специализации. В этом случае

---

[1] Этот пример иллюстрирует расплывчатость категорий. Если дочерний класс реализует процедуру хранения, используя другое имя метода, мы говорим, что это наследование для конструирования. Если же дочерний класс пользуется тем же именем, что и родительский класс, мы могли бы сказать, что это наследование для спецификации.

```
class Storable
{
   void writeByte(unsigned char);
}
class StoreMyStruct : public Storable
{
   void writeStruct (MyStruct &aStruct);
}
```

подкласс расширяет родительский класс для создания объекта более общего типа. Порождение подкласса для обобщения часто применяется, когда построение происходит на основе существующих классов, которые мы не хотим или не можем изменять.

Рассмотрим систему графического отображения, в которой был определен класс окон Window для черно-белого фона. Мы хотим создать тип цветных графических окон

PDF created with pdfFactory Pro trial version www.pdffactory.com

ColoredWindow. Цвет фона будет отличаться от белого за счет добавления нового поля, содержащего цвет. Придется также переопределить наследуемую процедуру изображения окна, в которой происходит фоновая заливка.

Порождение подкласса для обобщения часто используется в случае, когда общий проект основывается в первую очередь на значениях данных и меньше — на функциональном поведении. Это видно на примере с цветным окном, так как оно содержит поля данных, необязательные в случае черно-белого окна.

Как правило, следует избегать порождения подкласса для обобщения, пользуясь перевернутой иерархией типов и порождением для специализации. Однако это не всегда возможно.

### 7.3.5. Порождение подкласса для расширения

В то время как порождение подкласса для обобщения модифицирует или расширяет существующие функциональные возможности объекта, порождение для расширения добавляет совершенно новые свойства. Его можно отличить по тому, что порождение для обобщения обязано переопределить по крайней мере один метод родителя, а функциональные возможности подкласса привязаны к родительским. Расширение просто добавляет новые методы к родительским, и функциональные возможности подкласса менее крепко привязаны к существующим методам родителя.

Примером порождения подкласса для расширения является множество текстовых строк StringSet, наследующее свойства общего класса множеств Set и предназначенное для хранения строковых величин. Такой класс мог бы предоставлять дополнительные методы для строковых операций, например «найти по префиксу», который возвращал бы подмножество всех элементов множества, начинающихся с определенной подстроки. Такие операции имеют смысл для подкласса, но не для родительского класса.

Поскольку функциональные возможности родителя остаются нетронутыми и доступными, порождение подкласса для расширения не противоречит принципу подстановки и, следовательно, такие подклассы всегда будут подтипами.

### 7.3.6. Порождение подкласса для ограничения

Порождение для ограничения происходит в случае, когда возможности подкласса более ограничены, чем в родительском классе. Так же, как и при обобщении, порождение для ограничения чаще всего возникает, когда программист строит класс на основе существующей иерархии, которая не должна или не может быть изменена.

Допустим, существующая библиотека классов предоставляет структуру данных Deque (double-ended-queue, очередь с двумя концами). Элементы могут добавляться или удаляться с любого конца структуры типа Deque, но программист желает создать класс Stack, вводя требование добавления или удаления элементов только с одного конца стека.

Таким же образом, как и при порождении для конструирования, программист может построить класс Stack как подкласс класса Deque и модифицировать нежелательные методы так, чтобы они выдавали сообщение об ошибке в случае применения. Такие методы переопределяют существующие и ограничивают их возможности, что характеризует порождение подкласса для ограничения. (Переопределение, с помощью

PDF created with pdfFactory Pro trial version www.pdffactory.com

которого подкласс изменяет смысл метода родительского класса, обсуждается в следующей главе.)

Поскольку порождение подкласса для ограничения является явным нарушением принципа подстановки и поскольку оно строит подклассы, не являющиеся подтипами, его следует избегать, где только возможно.

### 7.3.7. Порождение подкласса для варьирования

Порождение подкласса для варьирования применяется, когда два класса имеют сходную реализацию, но не имеют никакой видимой иерархической связи между абстрактными понятиями, ими представляемыми. Например, программный код для управления мышкой может быть почти идентичным тому, что требуется для управления графическим планшетом. Теоретически, однако, нет никаких причин, для того чтобы класс Mouse, управляющий манипулятором «мышь», был подклассом класса Tablet, контролирующего графический планшет, или наоборот. В таком случае в качестве родителя произвольно выбирается один из них, при этом другой наследует общую программную часть кода и переопределяет код, зависящий от устройства.

Обычно, однако, лучшей альтернативой является выделение общего кода в абстрактный класс, скажем PointingDevice, и порождение обоих классов от этого общего предка. Как и в случае порождения подкласса для обобщения, такой путь может быть недоступен при доработке уже существующего класса.

### 7.3.8. Порождение подкласса для комбинирования

Обычным является желание иметь подкласс, представляющий собой комбинацию двух или более родительских классов. Ассистент учителя, например, имеет характерные особенности как учителя, так и учащегося и, следовательно, может вести себя двояко. Способность наследовать от двух или более родительских классов известна как множественное наследование; оно достаточно сложно и коварно, и мы посвятим этому вопросу целую главу.

### 7.3.9. Краткое перечисление форм наследования

Мы можем подвести итог изучению различных форм наследования в виде следующей таблицы:

**Специализация**. Дочерний класс является более конкретным, частным или специализированным случаем родительского класса. Другими словами, дочерний класс является подтипом родительского класса.

**Спецификация**. Родительский класс описывает поведение, которое реализуется в дочернем классе, но оставлено нереализованным в родительском.

**Конструирование**. Дочерний класс использует методы, предоставляемые родительским классом, но не является подтипом родительского класса (реализация методов нарушает принцип подстановки).

**Обобщение**. Дочерний класс модифицирует или переопределяет некоторые методы родительского класса с целью получения объекта более общей категории.

PDF created with pdfFactory Pro trial version www.pdffactory.com

**Расширение**. Дочерний класс добавляет новые функциональные возможности к родительскому классу, но не меняет наследуемое поведение.

**Ограничение**. Дочерний класс ограничивает использование некоторых методов родительского класса.

**Варьирование**. Дочерний и родительский классы являются вариациями на одну тему, и связь «класс—подкласс» произвольна.

**Комбинирование**. Дочерний класс наследует черты более чем одного родительского класса. Это — множественное наследование; оно будет рассмотрено в одной из следующих глав.

## 7.4. Наследование в различных языках программирования

В следующих разделах мы подробно опишем синтаксис, используемый для описания наследования в каждом из рассматриваемых языков программирования. Обратите внимание на разницу между теми языками, которые требуют, чтобы все классы были порождениями общего родительского класса (обычно называемого Object, как в языках Smalltalk и Objective-C), и теми, которые допускают различные независимые иерархии классов.

Преимущество наследования с единым предком в том, что функциональные возможности последнего (класса Object) наследуются всеми объектами. Таким образом гарантируется, что каждый объект обладает общим минимальным уровнем функциональности. Минус в том, что единая иерархия «зацепляет» все классы друг с другом.

В случае нескольких независимых иерархий наследования приложению не придется тащить за собой большую библиотеку классов, из которой лишь немногие будут использоваться в каждой конкретной программе. Конечно, это означает, что нет функциональности, которой гарантированно обладают все объекты.

Различные взгляды на объекты отчасти являются еще одним различием между языками с динамическими и статическими типами данных (мы вернемся к этой проблеме в главе 12). В языках программирования с динамическими типами данных объекты в основном характеризуются теми сообщениями, которые они понимают. Если два объекта понимают одно и то же множество сообщений и реагируют на них сходным образом, они с практической точки зрения неразличимы, невзирая на их родственные связи. В таких случаях разумно, чтобы все объекты наследовали большую часть своего поведения от общего базового класса.

### 7.4.1. Наследование в языке Object Pascal

В языке программирования Object Pascal фирмы Apple наследование от родительского класса указывается помещением его имени в круглые скобки после ключевого слова object. Предположим, например, что в нашей имитации бильярда мы решили породить классы Ball, Wall и Hole от общего класса GraphicalObject. Мы сделаем это так, как показано в листинге 7.1.

**Листинг 7.1**. Пример наследования в языке Object Pascal фирмы Apple

```
type
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  GraphicalObject = object
    (* поля данных *)
    region : Rect;
    link  : GraphicalObject;
    (* операции *)
    procedure draw;
    procedure update;
    procedure hitBy(aBall : Ball);
  end;
  Ball = object(GraphicalObject)
    (* поля данных *)
    direction : real;
    energy  : real;
    (* инициализация *)
    procedure initialize (x, y : integer);
    (* переопределяемые методы *)
    procedure draw;  override;
    procedure update; override;
    procedure hitBy(aBall : Ball); override;
    (* методы, специфичные для класса *)
    procedure erase;
    procedure setCenter(newx, newy : integer);
    function x : integer;
    function y : integer;
end;
```

Как показано в этом примере, дочерние классы могут добавлять как новые поля данных, так и новые методы. Кроме того, они модифицируют существующее поведение, переопределяя методы с помощью ключевого слова override, как сделано для метода draw. Аргументы в переопределенном методе должны совпадать по типу и числу с аргументами метода родительского класса.

Версия языка Object Pascal фирмы Borland (система Delphi) отличается в двух важных отношениях. Во-первых, как мы видели в предыдущих главах, вместо ключевого слова object используется слово class, и классы всегда должны выводиться один из другого. Класс TObject — общий предок всех объектов. Во-вторых, в дополнение к ключевому слову override директива virtual помещается после описания тех методов родительского класса, которые могут быть переопределены — почти как в языке C++. Пример, иллюстрирующий эти изменения, приведен в листинге 7.2. Пропуск ключевого слова override является частым источником ошибок, поскольку описание остается синтаксически законным, а его интерпретация неверной. Это мы обсудим подробнее в главе 10.

**Листинг 7.2**. Пример наследования в языке Delphi Pascal

```
type
  GraphicalObject = class(TObject)
    (* поля данных *)
    region : Rect;
    link  : GraphicalObject;
    (* операции *)
    procedure draw; virtual;
    procedure update; virtual;
    procedure hitBy(aBall : Ball); virtual;
  end;
Ball = class(GraphicalObject)
    (* поля данных *)
    direction : real;
    energy  : real;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    (* инициализация *)
    procedure initialize (x, y : integer);
    (* переопределяемые методы *)
    procedure draw;  override;
    procedure update; override;
    procedure hitBy(aBall : Ball); override;
    (* методы, специфичные для класса *)
    procedure erase;
    procedure setCenter(newx, newy : integer);
    function x : integer;
    function y : integer;
end;
```

Более существенным различием между языками программирования Delphi Pascal и Apple Object Pascal является введение динамических методов. Они используют другие механизмы поиска во время выполнения программы (больше напоминающие Objective-C, чем C++; см. главу 21). Это делает динамические методы более медленными, чем виртуальные, но они требуют меньше памяти. Ключевое слово dynamic вместо virtual показывает, что объявляется динамический метод. Многие методы, связанные с действиями операционной системы по управлению окнами, реализованы как динамические. Значение термина сообщение часто ограничивается только действиями, связанными с управлением окнами.

### 7.4.2. Наследование в языке Smalltalk

Как отмечено нами в главе 3, наследование как средство создания новых классов обязательно в языке Smalltalk. Новый класс не может быть определен без предварительного описания существующего класса, которому он наследует. Фактически новый класс создается с помощью сообщения родительскому классу.

```
List subclass: #Set
   instanceVariables: #()
   classVariables: #()
```

Имеется единый предок, называемый Object, от которого в конце концов происходят все остальные классы. Object обеспечивает все объекты общими и совместимыми функциональными возможностями. Примерами методов этого класса служат функции сравнения одного объекта с другим, печати строковых представлений объекта и т. д.

Язык Smalltalk обеспечивает только одиночное наследование, то есть каждый класс наследует только одному родительскому классу. Новый метод может заместить метод родительского класса, просто будучи так же названным.

### 7.4.3. Наследование в языке Objective-C

Как и в языке Smalltalk, для Objective-C наследование является неотъемлемой частью формирования нового класса. Описание интерфейса каждого нового класса должно определить предка, от которого происходит наследование. Следующий пример показывает, что класс игральных карт Card происходит от универсального класса Object:

```
@interface Card : Object
{
 . . .
}
 . . .
@end
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Как и в языке Smalltalk, существует единый предок Object, от которого в конечном счете происходят все остальные классы. Object обеспечивает все объекты общей и полной функциональностью. Он часто используется как родитель для нового класса.

Разрешается только одиночное наследование, то есть класс не может наследовать от двух или более родителей. Как и в языке Smalltalk, метод, имеющий то же имя, что и метод родительского класса, переопределяет его.

### 7.4.4. Наследование в языке C++

В отличие от Smalltalk и Objective-C новый класс в языке программирования C++ не обязан происходить от уже существующего класса. Наследование указывается в заголовке описания класса с помощью ключевого слова public, за которым следует имя родительского класса. Новый класс TablePile происходит от более общего класса CardPile, представляющего собой колоду карт:

```
class TablePile : public CardPile
{
 . . .
};
```

Ключевое слово public может быть заменено на слово private, указывающее на порождение класса для конструирования, то есть на форму наследования, не создающую подтипа. Такой пример будет рассмотрен в главе 11.

Как было отмечено в предыдущей главе, преимущество объектно-ориентированных языков в том, что они стремятся объединить создание новой переменной и ее инициализацию. Наследование несколько усложняет этот процесс, поскольку родительский и новый классы могут иметь разный инициализирующий код. Для обеспечения наследования конструктор дочернего класса должен явно вызвать конструктора родительского класса. Делается это с помощью инициализирующего предложения в конструкторе дочернего класса:

```
TablePile::TablePile (int x, int y, int c)
 : CardPile(x,y) // инициализация родителя
{
   // теперь инициализируем дочерний класс
   . . .
}
```

Язык программирования C++ поддерживает множественное наследование, то есть новый класс может быть определен как потомок двух или более родителей. Мы исследуем смысл этого более подробно в следующей главе.

В предыдущей главе мы описали ключевые слова public (открытый) и private (закрытый), указав, что одно описывает интерфейс класса, а другое — детали реализации. В описании классов, получаемых наследованием, может быть использовано третье ключевое слово protected (защищенный). Защищенные поля являются частью реализации, но доступны подклассам так же, как и самому классу.

Когда ключевое слово virtual предшествует описанию входящей в класс функции, оно означает, что эта функция, вероятно, будет переопределена в подклассе или что эта функция сама переопределяет функцию надкласса. (Это ключевое слово является необязательным в дочернем классе, но его желательно оставлять.) Однако семантика

PDF created with pdfFactory Pro trial version www.pdffactory.com

переопределения в языке C++ является тонким моментом и зависит от того, как был описан получатель, которому присваивается объект. Мы отложим ее обсуждение до главы 11.

### 7.4.5. Наследование в языке Java

Из всех обсуждаемых в этой книге языков Java идет дальше всех в разделении понятий подкласса и подтипа. Подклассы объявляются с помощью ключевого слова extends, как в следующем примере:

```
class window
{
  // . . .
}
class textEditWindow extends window
{
  // . . .
}
```

Предполагается, что подклассы являются подтипами (хотя, как и в случае языка C++, это предположение не всегда верно). Это означает, что представитель подкласса может быть присвоен переменной, объявленной с типом родительского класса. Методы дочернего класса, имеющие те же имена, что и методы родителя, переопределяют наследуемое поведение. Как и в языке C++, ключевое слово protected может быть использовано для указания методов и данных, доступных только внутри класса или подкласса, но не входящих в более общий интерфейс.

Все классы происходят от единого предка Object. Если родительский класс не указан явно, то предполагается класс Object. Таким образом, определение класса window, приведенное выше, эквивалентно следующей записи:

```
class window extends Object
{
  // . . .
}
```

Альтернативная форма порождения подтипов данных основана на интерфейсе. Интерфейс определяет протокол определенного поведения, а не реализацию. В этом отношении он подобен абстрактному родительскому классу. В следующем примере приведен интерфейс, описывающий объекты, которые могут читать и писать во входной/выходной поток.

```
public interface Storing
{  void writeOut(Stream s);
   void readFrom(Stream s);
}
```

Интерфейс определяет новый тип. Это означает, что переменные могут быть объявлены просто с именем интерфейса. А класс может указать, что он реализует протокол, определенный интерфейсом. Представители класса могут присваиваться переменным, объявленным с типом интерфейса, точно так же, как представители дочернего класса могут присваиваться переменным, объявленным с типом родительского класса:

```
public class BitImage implements Storing
{  void writeOut (Stream s)
   {
      // . . .
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
   };
   void readFrom (Stream s)
   {
      // . . .
   };
};
```

Хотя язык Java поддерживает только одиночное наследование (наследование исключительно от одного родительского класса), класс может указывать, что он поддерживает несколько интерфейсов (реализует множественный интерфейс). Многие проблемы, для которых в языке C++ пришлось бы использовать множественное наследование, в языке программирования Java разрешаются через множественные интерфейсы. Интерфейсам позволено расширять другие интерфейсы, в том числе и множественные, через указание ключевого слова extend.

В языке Java идея порождения подкласса для спецификации формализована через модификатор abstract. Если класс объявлен как abstract, то из него должны порождаться подклассы. Не разрешается создавать представителей абстрактного класса, можно только порождать подклассы. Методы тоже могут быть объявлены как abstract, и в таком случае они не обязаны иметь реализацию. Таким образом, объявление класса как abstract обеспечивает, что он будет использоваться только как спецификация поведения, а не в виде конкретных объектов:

```
abstract class storable
{
   public abstract writeOut();
}
```

Наоборот, модификатор final указывает, что из класса не могут порождаться подклассы или что метод, к которому относится этот модификатор, не может быть изменен. Тем самым пользователю гарантируется, что поведение класса будет таким, каким определено, и не будет модифицировано при порождении последующих подклассов:

```
final class newClass extends oldClass
{
   . . .
}
```

## 7.5. Преимущества наследования

В этом разделе мы опишем некоторые из многих важных преимуществ правильного использования механизма наследования.

### 7.5.1. Повторное использование программ

При наследовании поведения от другого класса программный код не нуждается в переписывании. Это кажется очевидным, но на самом деле имеет важное значение. Многие программисты тратят массу времени на переработку кода, уже написанного не один раз до этого, — например, при поиске по образцу в строке или при включении нового элемента в таблицу. С введением объектно-ориентированной техники, эти функции могут быть написаны однажды и потом повторно использоваться.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Другое преимущество повторно используемого кода — в его повышенной надежности (чем в большем числе ситуаций используется код, тем больше возможностей обнаружения ошибок) и низкой стоимости, так как она делится на всех пользователей кода.

### 7.5.2. Использование общего кода

При применении объектно-ориентированной техники использование общего кода происходит на нескольких уровнях. Во-первых, клиенты могут пользоваться одними и теми же классами (Бред Кокс [Cox 1986] называет их software-IC, то есть программными интегральными схемами, по аналогии с аппаратными интегральными схемами). Иная форма использования общего кода возникает в случае, когда два или более класса, разработанных тем же самым программистом для некоторого проекта, наследуют от единого родительского класса. Например, множество Set и массив Array могут рассматриваться как разновидности совокупности данных Collection. В этом случае два или более типов объектов совместно используют наследуемый код. Он пишется единожды и входит в программу только в одном месте.

### 7.5.3. Согласование интерфейса

Когда два класса наследуют одному и тому же предку, мы можем быть уверены, что наследуемое поведение будет одинаковым во всех случаях. Таким образом, легко гарантировать, что объекты, схожие по интерфейсу, будут и фактически сходными. В противном случае пользователь получит почти одинаковые объекты, поведение которых совершенно разное.

### 7.5.4. Программные компоненты

В главе 1 мы отмечали, что наследование предоставляет программистам возможность создавать повторно (многократно) используемые программные компоненты. Цель: обеспечить развитие новых приложений с минимальным написанием нового кода. Уже сейчас доступны несколько коммерческих библиотек такого типа, и в будущем мы можем ожидать появления многих новых специализированных систем.

### 7.5.5. Быстрое макетирование

Когда программное обеспечение конструируется в основном из повторно используемых компонент, большая часть времени, требуемого на разработку, может быть посвящена пониманию новых необычных частей системы. Таким образом программные комплексы могут создаваться быстрее и проще, приводя к стилю программирования, известному как быстрое макетирование или исследовательское программирование. Создается система-прототип (макет), пользователи экспериментируют с ней, потом на основе этих опытов создается вторая система, с ней проводятся эксперименты и т. д. Такое программирование особенно выгодно в ситуации, когда цели и требования к системе в начале разработки представлены весьма расплывчато.

### 7.5.6. Полиморфизм и структура

Программное обеспечение традиционно создавалось снизу вверх, хотя и могло разрабатываться сверху вниз. То есть сначала писались программы нижнего уровня, на их основе строились более абстрактные элементы, а затем — еще более абстрактные. Такой процесс похож на строительство дома.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Обычно мобильность кода уменьшается с увеличением абстракции, то есть программы нижнего уровня могут быть использованы в нескольких различных проектах, и даже, возможно, абстракции следующего уровня могут повторно использоваться, но программы верхнего уровня тесно связаны с определенным приложением. Компоненты нижнего уровня могут быть перенесены в новую систему, и часто есть смысл в их самостоятельном существовании. Компоненты верхнего уровня обычно имеют смысл (из-за их функциональности или зависимости от данных), только когда они построены на определенных элементах нижнего уровня.
Полиморфизм в языках программирования позволяет программисту создавать многократно используемые компоненты высокого уровня, которые можно перекраивать под различные приложения за счет изменения нижнего уровня. Мы еще поговорим об этом в последующих главах.

### 7.5.7. Маскировка информации

Программист, использующий программную компоненту, должен понимать только ее назначение и интерфейс. Программисту совсем не обязательно иметь подробную информацию о технических средствах, использованных при реализации компоненты. Таким образом уменьшается необходимость внутренних связей между программными системами. Мы ранее указывали на «зацепляющийся» характер традиционных программных продуктов как на одну из причин их сложности.

## 7.6. Издержки наследования

Хотя преимущества наследования в объектно-ориентированном программировании несомненны, ничего не дается даром. По этой причине мы должны рассмотреть издержки технических средств объектно-ориентированного программирования — в частности, наследования.

### 7.6.1. Скорость выполнения

Редко программные инструменты общего назначения являются столь же быстрыми, как и аккуратно, «вручную» разработанные специальные системы. Таким образом, унаследованные методы, способные иметь дело с произвольными подклассами, часто медленнее специального кода.

И все же заботы об эффективности часто бывают не к месту [1]. Во-первых, разница не слишком велика. Во-вторых, снижение скорости выполнения может компенсироваться повышением скорости разработки программного обеспечения. И наконец, большинство программистов на самом деле мало знают о том, как распределены временные затраты в их программах. Гораздо лучше создать работающую систему, произвести замеры времени, чтобы обнаружить, на что же, собственно, оно тратится, и улучшить эти части, чем затратить уйму времени, заботясь об эффективности на ранних стадиях проекта.

### 7.6.2. Размер программ

Использование любой программной библиотеки часто приводит к увеличению размера программ. Этого не происходит в специально разработанных системах. Хотя такие затраты могут быть существенными, по мере уменьшения стоимости памяти размер программы перестает быть критичным. Снизить затраты на разработку и быстро выдать высококачественный и свободный от ошибок программный код значит сейчас гораздо больше, чем малый размер приложения [2].

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 7.6.3. Накладные расходы на посылку сообщений

Много внимания обращалось на тот факт, что посылка сообщений по своей сути более дорогая операция, чем просто вызов процедуры. Однако, как и с суммарной скоростью выполнения, слишком большие заботы о цене посылки сообщений часто бывают экономией на спичках. Между прочим, замедление

часто минимально — два или три дополнительных ассемблерных оператора и общее увеличение времени на 10 процентов. Результаты замеров скорости различны для разных языков. Накладные расходы на посылку сообщений больше в языках программирования с динамическими типами данных (например, Smalltalk) и гораздо меньше в языках со статическими типами (C++, в частности). Эти затраты, как и другие, должны рассматриваться на фоне многих преимуществ объектно-ориентированной техники.

Некоторые языки программирования, и особенно C++, предоставляют программистам некоторое количество опций, дающих возможность уменьшить накладные расходы на посылку сообщений. Они включают в себя исключение полиморфизма из сообщений (при указании имени класса в вызовах функций) и поддержку встраиваемых (inline) процедур. Подобным образом программист на языке Delphi Pascal может выбрать методы, описанные с помощью ключевого слова dynamic, которые будут использовать механизм поиска во время выполнения или использовать методы с ключевым словом virtual, которые применяют несколько более быструю технику. Динамические методы более медленны при наследовании, но требуют меньше памяти.

### 7.6.4. Сложность программ

Хотя объектно-ориентированное программирование часто выдвигается как способ разрешения проблемы сложности программного обеспечения, необдуманное использование наследования может часто просто заменить одну форму сложности на

---

[1] Следующая цитата из статьи Билла Вульфа предлагает удачное замечание по поводу важности эффективности: «Во имя эффективности (как правило, эфемерной) совершается больше программных ошибок, чем по какой-либо другой причине, включая полную тупость» [Wulf 1972].

[2] Стоит, однако, привести и другую точку зрения. Следующие цитаты принадлежат Алану Голубу — программисту, консультанту и преподавателю, специализирующемуся в области ООП: «Разбухание программ является огромной проблемой. Жесткий диск в 350 Мб на моей машине может вместить операционную систему, усеченную версию компилятора и редактор, и больше ничего. В стародавние времена я мог разместить версии CP/M для тех же программ на одной-единственной дискете в 1,2 Мб... Я убежден, что большая часть этого разбухания памяти является результатом небрежного программирования»; «Если только вы не проникнетесь сознанием необходимости дисциплинировать себя, то можете закончить гигантским модулем из неподдающейся сопровождению тарабарщины, только притворяющейся компьютерной программой». — Примеч. перев.

другую. Для понимания программы, использующей наследование, может потребоваться несколько сложных переходов вверх и вниз в иерархическом дереве. Эта проблема известна под именем «вверх-вниз», или «йо-йо». Мы обсудим ее в следующей главе.

### Упражнения

1. Предположим, вам требуется написать программный проект на языке программирования, который не является объектно-ориентированным (например, Pascal или C). Как бы вы имитировали классы и методы? Как бы вы имитировали

PDF created with pdfFactory Pro trial version www.pdffactory.com

наследование? Сможете ли вы обеспечить множественное наследование? Обоснуйте свой ответ.

2. Мы указывали, что накладные расходы, связанные с посылкой сообщений, обычно больше, чем при традиционном вызове процедур. Как вы могли бы их измерить? Для языка программирования, поддерживающего и классы, и процедуры (C++ или Object Pascal), придумайте эксперимент для определения фактических затрат на посылку сообщений.
3. Рассмотрите три геометрических понятия: линия (бесконечна в обоих направлениях), луч (начало в фиксированной точке, бесконечен в одном направлении), сегмент (отрезок прямой с фиксированными концами). Как бы вы построили классы, представляющие эти три понятия, в виде иерархии наследования? Будет ли ваше решение другим, если вы обратите особое внимание на представление данных (на поведение)? Охарактеризуйте тип наследования, который вы использовали. Объясните ваше решение.
4. Почему использованный в следующем рассуждении пример не является верной иллюстрацией наследования?

Видимо, наиболее важным понятием в объектно-ориентированном программировании является наследование. Объекты могут наследовать свойства других объектов, тем самым ликвидируется необходимость написания какого-либо кода! Предположим, например, что программа должна обрабатывать комплексные числа, состоящие из вещественной и мнимой частей. Для комплексных чисел вещественная и мнимая части ведут себя как вещественные величины, поэтому все операции (+, –, /, *, sqrt, sin, cos и т. д.) могут быть наследованы от класса Real вместо того, чтобы писать новый код. Это, несомненно, окажет большое влияние на продуктивность работы программиста.

# *Глава 8: Учебный пример: Пасьянс*

Программа для раскладывания карточного пасьянса проиллюстрирует всю мощь наследования и переопределения. В главах 3 и 4 встречались фрагменты этой программы, в частности абстракция игральной карты, представленная классом Card. Языком программирования этого учебного примера будет Java.

Основное внимание будет уделено классу CardPile, абстрагирующему стопку игральных карт. Так как перекладывание карт из одной стопки в другую — это основное действие пасьянса, то подклассы CardPile будут базовыми структурами данных при реализации пасьянса. Имеется множество стопок карт, и наследование вкупе с переопределением интенсивно используется для упрощения разработки этих компонент и обеспечения их единообразия.

## *8.1. Класс игральных карт Card*

В предыдущих главах мы обсуждали абстрактный класс Card. Повторим некоторые важные моменты.

Каждый экземпляр класса Card (**листинг 8.1**) наделен мастью и рангом. Чтобы предотвратить их изменение, поля данных (переменные экземпляра) объявлены закрытыми, и сделать что-либо с ними можно только посредством функций доступа.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Значения полей масти и ранга устанавливаются конструктором класса. Кроме того, отдельная функция позволяет пользователям определять цвет карты. Значения целочисленных констант (определяемых в языке Java с помощью спецификаторов final static) заданы для черного и красного цветов, а также для мастей. Еще одна пара целочисленных констант определяет высоту и ширину карты.

Есть важные причины для того, чтобы обращаться к масти и рангу только через функции доступа. Прямой доступ к этим полям следует запретить. Тогда поля масти и ранга могут быть прочитаны, но не модифицированы. (Соответствующая функция, используемая главным образом для изменения значений полей данных, часто называется мутатором (mutator).)

**Листинг 8.1.** Описание класса card

```
class Card
{
  // конструктор
  Card (int sv, int rv)
  {
    s = sv; r = rv; faceup = false;
  }
  // доступ к атрибутам карты
  public int rank ()
  {  return r;  }
  public int suit ()
  {  return s;  }
  public boolean faceUp()
  {  return faceup;  }
  public void flip()
  {  faceup = ! faceup;  }
  public int color()
  {
    if (suit() == heart || suit == diamond)
      return red;
    return black;
  }
  public void draw (Graphics g, int x, int y)
  {
    ...
  }
  // статические поля данных для цвета и масти
  final static int width              = 50;
  final static int heigth             = 70;
  final static int red                        = 0;
  final static int black              = 1;
  final static int heart              = 0;
  final static int spade              = 1;
  final static int diamond    = 2;
  final static int club                       = 3;

  // поля данных
  private boolean faceup;
  private int r;
  private int s;
}
```

Итак, все действия, которые может выполнить карта (кроме установки и возврата состояния), — это переворачивание и показ себя. Функция **flip()** состоит из одной строчки, которая просто обращает значение, содержащееся в переменной экземпляра faceup, на противоположное. Функция рисования **draw()** сложнее: она использует

PDF created with pdfFactory Pro trial version www.pdffactory.com

графические средства, предоставляемые стандартной библиотекой приложений Java. Библиотека приложений поставляет тип данных, называемый **Graphics**, который обеспечивает множество методов рисования линий и фигур, а также раскрашивание. В качестве аргумента функции рисования передается значение типа **Graphics**, а также целочисленные координаты, соответствующие верхнему левому углу карты.

Графические изображения карт — рисунки из простых линий, как показано ниже. Черви и бубны нарисованы красным, а пики и кресты — черным. Штриховка рубашки выполнена желтым цветом. Фрагмент процедуры рисования игральной карты показан в **листинге 8.2**.

Наиболее важная особенность абстракции игральной карты — это стиль, при котором каждая карта ответственна за хранение в себе всей информации и поведения, к ней относящихся. Карта знает и свое значение, и то, как себя нарисовать. Таким образом, информация инкапсулирована и изолирована от приложения, использующего игральные карты. Если, например, программа перенесена на новую платформу, использующую другие графические средства, то изменить нужно будет только метод **draw** внутри самого класса.

**Листинг 8.2.** Процедура рисования игральной карты

```
class Card
{ ...
  public void draw (Graphics g, int x, int y)
  {  String names[] = {"A", "2", "3", "4", "5", "6",
        "7", "8", "9", "10", "J", "Q", "K"};
    // Очистить прямоугольник, нарисовать границу
    g.clearRect(x, y, width, height);
    g.setColor(Color.black);
    g.drawRect(x, y, width, height);
    // нарисовать тело карты
    if (faceUp) // лицевой стороной вверх
    { if (color() == red) g.setColor(Color.red);
        else g.setColor(Color.blue);
      g.drawString(names[rank()], x+3, y+15);
      if (suit() == heart)
      { g.drawLine(x+25, y+30, x+35, y+20);
        g.drawLine(x+35, y+20, x+45, y+30);
        g.drawLine(x+45, y+30, x+25, y+60);
        g.drawLine(x+25, y+60, x+5, y+30);
        g.drawLine(x+5, y+30, x+15, y+20);
        g.drawLine(x+15, y+20, x+25, y+30);
      }
      else if (suit() == spade )
      {  ...  }
      else if (suit() == diamond )
      {  ...  }
      else if (suit() == club )
      { g.drawOval(x+20, y+25, 10, 10);
        g.drawOval(x+25, y+35, 10, 10);
        g.drawOval(x+15, y+35, 10, 10);
        g.drawOval(x+23, y+45, x+20, y+55);
        g.drawOval(x+20, y+55, x+30, y+55);
        g.drawOval(x+30, y+55, x+27, y+45);
      }
    }
    else // картинкой вниз
    {   g.setColor(Color.yellow);
        g.drawLine(x+15, y+5, x+15, y+65);
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        g.drawLine(x+35, y+5, x+35, y+65);
        g.drawLine(x+5, y+20, x+45, y+20);
        g.drawLine(x+5, y+35, x+45, y+35);
        g.drawLine(x+5, y+50, x+45, y+50);
    }
  }

}
```

## *8.2. Связные списки*

Контейнер *стопка* карт использует для их хранения модель *связного списка*. Отделяя класс контейнера данных от его конкретных представителей (стопок игральных карт), мы позволяем каждому классу сконцентрироваться на ограниченном множестве задач.

Это является продвижением по сравнению с главой 6 (где, как вы помните, каждый графический объект содержал указатель на следующий графический объект). В подходе, изложенном в главе 6, плохо не только то, что поле указателя-связки не особенно важно для объекта, содержащегося в контейнере, но и то, что при таком способе объект не может быть включен в два (или более) списка одновременно. Создавая отдельные классы для абстракции связных списков, мы получаем гораздо большую гибкость в использовании контейнеров.

В абстракции связного списка задействованы два класса. Класс **LinkedList** — это «фасад» списка, то есть класс, с которым взаимодействует пользователь. В действительности значения хранятся в экземплярах класса **List**. Обычно пользователь даже не догадывается о существовании класса List. Оба класса показаны в **лист. 8.3**.

Так как контейнер данных на основе связного списка является абстракцией общего назначения и ничего не знает о типе объекта, который он будет содержать, то тип данных, приписываемый объекту-значению, — это класс всех объектов **Object**. Переменная, объявленная с типом данных **Object** (в частности, поле данных *value* в классе *Link*), является полиморфной — она может содержать значение любого типа.

Класс **LinkedList** обеспечивает: добавление элемента в список, проверку списка на наличие в нем элементов, доступ к первому элементу списка, удаление первого элемента списка.

**Листинг 8.3.** Классы Link и LinkedList

```
class Link
{
  public Link (Object newValue, Link next)
  {
    valueField = newValue; nextLink = next;
  }
  public Object value ()
  {  return valueField;  }
  public Link next ()
  {  return nextLink;  }
  private Object valueField;
  private Link nextLink;
}
class LinkedList
{
  public LinkedList ()
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  {  firstLink = null;  }
  public void add (Object newValue)
  {  firstLink = new Link(newValue, firstLink);  }
  public boolean empty ()
  {  return firstLink == null;  }
  public Object front ()
  {
    if (firstLink == null)
      return null;
    return firstLink.value();
  }
  public void pop ()
  {
    if (firstLink != null)
      firstLink = firstLink.next();
  }
  public ListIterator iterator()
  {  return new ListIterator (firstLink);  }
  private Link firstLink;

}
```

В более общем случае мы хотели бы предоставить пользователю нашей абстракции связного списка способ для перебора величин, содержащихся в списке, без необходимости их удаления и без знания детальной информации о внутренней структуре списка (в данном случае без сведений о классе *Link*). Как мы увидим в главе 16, такие возможности часто обеспечиваются разработчиками класса «список» через доступ к специальной разновидности объектов, называемых итераторами. Итератор скрывает детали представления контейнера данных и обеспечивает простой интерфейс для доступа к значениям в порядке очереди. Итератор для связного списка показан в **листинг 8.4**. С его помощью цикл записывается следующим образом:

```
ListIterator itr = aList.iterator();
while (! Itr.atEnd() )
{
   ... do something list itr.current() ...
   itr.next();
}
```

Обратите внимание на то, как сам список возвращает итератор в результате вызова метода и как использование итератора позволяет избежать упоминания о связных полях списка.

**Листинг 8.4.** Класс ListIterator

```
class ListIterator
{
  public ListIterator (Link firstLink)
  {
    currentLink = firstLink;
  }
  public boolean atEnd ()
  {
    return currentLink == null;
  }
  public void next ()
  {
    if (currentLink != null)
      currentLink = currentLink.next();
  }
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  public Object current ()
  {
    if (currentLink == null)
      return null;
    return currentLink.value();
  }
  private Link currentLink:
}
```

### *8.3. Правила пасьянса*

Версия пасьянса, которую мы будем описывать, известна под названием «Косынка» (или *Klondike*). Бесчисленные вариации этой игры делают ее, возможно, наиболее распространенной версией пасьянса, так что когда вы говорите слово «пасьянс», многие люди думают о «косынке». Версия, которую мы будем использовать здесь, описана в книге [*Morehead 1949*]. В упражнениях мы рассмотрим некоторые распространенные разновидности этого пасьянса.

Расположение карт показано на **рис. 8.1**. Используется одна стандартная колода из 52 карт. Расклад пасьянса (*tableau*) состоит из 28 карт в 7 стопках. Первая стопка состоит из 1 карты, вторая — из 2 и т. д. до 7. Верхняя карта в каждой стопке изначально лежит картинкой вверх; все остальные — картинкой вниз.

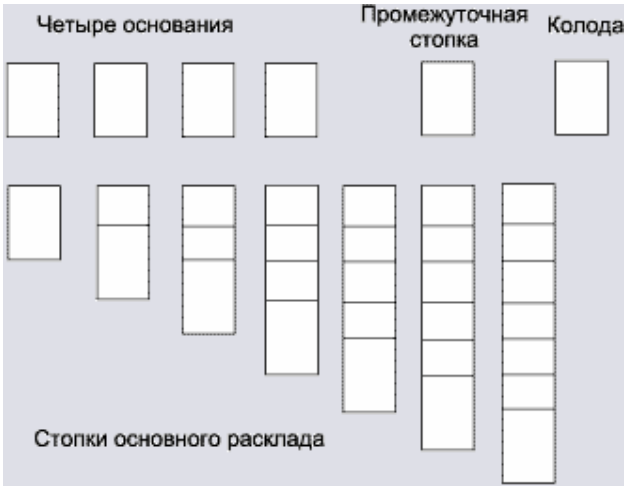



**Рис. 8.1.** Начальный расклад пасьянса

Стопки мастей (иногда называемые основаниями (*foundations*)) строятся от тузов до королей по мастям. Они создаются сверху расклада по мере того, как нужные карты становятся доступными. Цель игры — сложить все 52 карты в основания по мастям.

Те карты, которые не выложены в стопки, изначально находятся в колоде (*deck*). Карты там лежат картинкой вниз, они достаются из колоды по одной и кладутся картинкой вверх в промежуточную стопку (*discard pile*). Оттуда они перемещаются на расклад или в основания. Карты достаются из колоды, пока она не опустеет. Игра заканчивается, если дальнейшие перемещения карт невозможны.

Карты кладутся в стопки расклада только на карту следующего по старшинству ранга и противоположного цвета. Карта переносится в основание, если она той же масти и следует по старшинству за верхней картой одного из оснований (или если основание пустое и карта является тузом). Пустые промежутки, возникающие в раскладе во время игры, заполняются только королями.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Самая верхняя карта промежуточной стопки всегда доступна. Существует только одна возможность переместить более одной карты — положить целый набор открытых карт расклада (*называемый последовательностью* (build)) в другую стопку расклада. Это можно сделать, если самая нижняя карта последовательности может быть по правилам положена на самую верхнюю карту в стопке назначения. Наша первоначальная игра не будет поддерживать перемещение последовательностей, но мы обсудим это в качестве возможного расширения. Самая верхняя карта расклада всегда лежит картинкой вверх. Если карта удаляется из расклада, оставляя на вершине закрытую карту, то последнюю можно открыть — перевернуть ее картинкой вверх.

Из этого короткого описания ясно, что пасьянс в основном заключается в манипулировании стопками карт. Каждый тип стопки, имея ряд общих свойств с другими стопками, обладает своей спецификой.

В следующем разделе мы детально проанализируем, как в таком случае может быть использовано наследование для упрощения реализации различных стопок карт. Идея ясна уже сейчас: создать класс стопки с основными действиями и для каждой конкретной стопки переопределить его.

## *8.4. Стопки карт — наследование в действии*

Значительная часть поведения, которое мы связываем со стопкой карт, является общим для всех типов стопок в игре. Например, каждая стопка содержит связный список карт; операции добавления и удаления элементов из этого связного списка тоже похожи. Другие операции, которым приписано поведение «по умолчанию» от класса CardPile, иногда переопределяются для разных подклассов. Класс CardPile показан в **листинге 8.5**.

Каждая стопка карт содержит координаты своего верхнего левого угла, а также связный список карт в стопке. Все эти значения устанавливаются конструктором класса. Поля данных объявлены как protected и таким образом доступны только методам класса (или его подкласса).

Три функции top(), pop() и empty(), манипулирующие списком карт, используют интерфейс, предоставляемый классом LinkedList. Новая карта добавляется в список путем вызова addCard(Card). Она модифицируется внутри подклассов. Обратите внимание: метод класса front() связного списка возвращает значение типа Object. Оно должно быть преобразовано к типу данных Card в функциях top() и pop().

**Листинг 8.5.** Описание класса CardPile

```
class CardPile
{ CardPile (int x1,int y1)
  {
    x = x1; y = y1; cardFile = new LinkedList();
  }
  public Card top()
  {
    return (Card) cardList.front();
  }
  public boolean empty()
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  {
    return cardList.empty();
  }
  public Card pop()
  {
    Card result = (Card) cardList.front();
    cardList.pop();
    return result;
  }
  // нижеследующие иногда переопределяются
  public boolean includes (int tx, int ty)
  {
    return x <= tx && tx <= x + Card.width &&
      y <= ty && ty <= y + Card.height;
  }
  public void select (int tx, ty)  { }
  public void display (Graphics G)
  {
    g.setColor(Color.black);
    if (cardList.empty())
      g.drawRect(x, y, Card.width, Card.height);
    else
      top().draw(g, x, y);
  }
  public boolean canTake (Card aCard)
  {  return false;  }
  // координаты стопки карт
  protected int x;
  protected int y;
  protected LinkedList cardList;
}
```

Оставшиеся пять операций являются типичными с точки зрения нашей абстракции стопки игральных карт. Однако они различаются в деталях в каждом отдельном случае. Например, функция canTake(Card) запрашивает, можно ли положить карту в данную стопку. Карта может быть добавлена к основанию, только если она следует по старшинству и имеет ту же масть, что и верхняя карта основания (или если карта — туз, а стопка пуста). С другой стороны, карта может быть добавлена в стопку расклада, только если 1) цвет карты противоположен цвету текущей верхней карты в стопке и 2) карта имеет следующее по рангу младшее значение, чем верхняя карта в стопке или 3) стопка пуста, а карта является королем.

Действия пяти виртуальных функций, определенных в классе CardPile, могут быть охарактеризованы так:

**includes**
: определяет, содержатся ли координаты, переданные в качестве аргументов, внутри границ стопки. Действие по умолчанию просто проверяет самую верхнюю карту стопки. Для стопки DeckPile это действие переопределено как проверка всех карт, содержащихся в стопке.

**canTake**
: сообщает, можно ли положить данную карту в стопку. Только стопка DeckPile и основания SuitPile могут принимать карты, поэтому действие по умолчанию — вернуть «нет». В двух вышеупомянутых классах стопок карт это действие переопределяется.

**addCard**

PDF created with pdfFactory Pro trial version www.pdffactory.com

добавляет карту к списку карт (к стопке). Для промежуточной стопки карт DiscardPile это действие переопределяется так, чтобы гарантировать, что карта лежит картинкой вверх.

**display**

отображает на экране стопку карт. По умолчанию этот метод просто показывает самую верхнюю карту стопки, но для класса стопок расклада TablePile он заменяется на показ колонки карт. При этом отображается верхняя половина каждой скрытой карты. Так что из всех карт такой стопки наиболее далеко отстоящими оказываются самая первая и самая последняя карты. Это позволяет определить границы, занимаемые стопкой карт.

**select**

выполняет действие в ответ на щелчок мыши. Функция вызывается, когда пользователь выбирает стопку карт щелчком мышью в области стопки. По умолчанию не делается ничего, но для стопок расклада TablePile, колоды DeckPile и промежуточной стопки DiscardPile оно переопределяется на операцию розыгрыша верхней карты, если это возможно.

Следующая таблица иллюстрирует пользу наследования. Даны пять операторов и пять классов, так что имеется 25 потенциальных методов, которые мы должны были бы определить. Используя наследование, мы должны реализовать только 13 методов. Более того, нам гарантировано, что каждая стопка будет реагировать одинаковым образом на похожие запросы.

| | CardPile | SuitPile | DeckPile | DiscardPile | TablePile |
|---|---|---|---|---|---|
| includes | * | | | | * |
| canTake | * | * | | | * |
| addCard | * | | | * | |
| display | * | | | | * |
| select | * | | * | * | * |

### *8.4.1. Основание SuitPile*

Мы детально рассмотрим каждый из подклассов CardPile, заостряя внимание на различных свойствах объектно-ориентированного программирования по мере их проявления. Самый простой подкласс — это основания SuitPile. Он показан в **листинге 8.6**. Стопка лежит в верхнем углу стола, в ней находятся карты одной масти от туза до короля.

**Листинг 8.6.** Класс SuitPile

```
class SuitPile extends CardPile
{
  SuitPile (int x, int y)
  {
    super(x, y);
  }
  public boolean canTake (Card aCard)
  {
    if (empty())
      return aCard.rank() == 0;
    Card topCard = top();
    return (aCard.suit() == topCard.suit()) &&
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
        (aCard.rank() == 1 + topCard.rank());
  }
}
```

Класс SuitPile определяет только два метода. Его конструктор берет два целочисленных аргумента и не делает ничего, кроме вызова конструктора надкласса *CardPile*. Обратите внимание на ключевое слово super, указывающее родительский класс. Метод canTake определяет, можно или нет поместить карту в стопку. Перемещение карты законно, если стопка пуста и эта карта — туз или если эта карта той же масти, что и верхняя карта в стопке, и ее ранг — следующий по старшинству (например, тройка пик может быть положена только на двойку пик).

Все остальное поведение стопки *SuitPile* такое же, как и у общей стопки карт. При выборе мышью основание не выполняет никаких действий. Когда карта добавляется, она просто вставляется в связный список. Для отображения стопки на экране рисуется только верхняя карта.

### *8.4.2. Колода DeckPile*

Класс DeskPile (**листинг 8.7**) обслуживает исходную колоду карт. Она отличается от стопки карт общего типа двумя моментами. При конструировании экземпляра вместо пустой стопки класс создает полную колоду из 52 карт, вставляя их в случайном порядке в связный список. Подпрограмма random библиотеки языка Java генерирует случайную величину с двойной точностью в диапазоне от 0 до 1. Она преобразуется в случайное целое число во время процесса тасования колоды.

Метод select вызывается, когда щелчок мыши производится над колодой DeskPile. Если она пуста, то ничего не происходит. В противном случае верхняя карта удаляется из колоды и добавляется в промежуточную стопку.

В языке Java нет глобальных переменных. Когда значение используется несколькими объектами классов (такими, как разные стопки карт в нашем пасьянсе), переменная объявляется с ключевым словом static. Как мы увидим в главе 20, при этом создается одна копия статической переменной, которая доступна всем экземплярам. В данной программе статические переменные применяются для хранения различных стопок карт. Они будут содержаться в экземпляре класса Solitare, который мы опишем впоследствии. Для доступа к ним мы используем полностью специфицированное имя, которое кроме имени переменной включает название класса. Это показано в методе select (**листинг 8.8**), который обращается к переменной *Solitare.discardPile* для доступа к промежуточной стопке.

**Листинг 8.7.** Класс DeckPile

```
class DeckPile extends CardPile
{
  DeckPile (int x, int y)
  {
    // сначала инициализируется надкласс
    super(x, y);
    // затем создается новая колода
    // сначала она кладется в локальную стопку
    CardPile pileOne = new CardPile(0, 0);
    CardPile pileTwo = new CardPile(0, 0);
    int count = 0;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    for (int i = 0; i < 4; i++)
    {
      pileOne.addCard(new CArd(i, j));
      count++;
    }
// затем случайно вытаскивается карта
    for (; count > 0; count--)
    { int limit = ((int)(Math.random() * 1000))
        % count;
      // перемещается вниз в случайное место
      for (int i = 0; i < limit; i++)
        pileTwo.addCard(pileOne.pop());
      // потом добавляется карта отсюда
      addCard(pileOne.pop());
      // затем колоды складываются обратно
      while (! pileTwo.empty())
        pileOne.addCard(pileTwo.pop());
    }
  }
  public void select(int tx, int ty)
  {
    if (empty())
      return;
    Solitaire.discardPile.addCard(pop());
  }

}
```

### *8.4.3. Промежуточная стопка DiscardPile*

Класс *DiscardPile* (см. листинг 8.8) интересен тем, что он демонстрирует две совершенно разные формы наследования. Метод ***select*** замещает или переопределяет поведение, по умолчанию обеспечиваемое классом *CardPile*. Новый код при вызове (то есть при нажатии кнопки мыши в области стопки) проверяет, может ли верхняя карта быть перемещена на какое-нибудь основание или на одну из стопок расклада. Если карта не может быть перемещена, она остается в промежуточной стопке.

Метод ***addCard*** демонстрирует другой тип переопределения. Здесь поведение уточняет функциональность надкласса. То есть полностью отрабатывается поведение надкласса и, кроме того, добавляется новое поведение. В данном случае новый код гарантирует, что когда карта лежит в промежуточной стопке, она всегда будет смотреть картинкой вверх. После того как это условие удовлетворено, путем посылки сообщения для псевдопеременной super вызывается код надкласса, который добавляет карту в стопку.

Другая форма уточнения возникает для конструкторов различных подклассов. До того как конструктор выполнит свои собственные действия, каждый из них должен вызвать конструктор надкласса, дабы гарантировать, что предок инициализировался должным образом. Конструктор предка вызывается через псевдо-переменную super; он вызывается как функция внутри конструктора дочернего класса. В главе 11 мы поговорим подробнее о различии между замещением и уточнением при переопределении методов.

**Листинг 8.8.** Класс DiscardPile

```
class discardPile extends CardPile
{
  DiscardPile (int x, int y)
  {
    super (x, y);
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  }
  public void addCard (Card aCard)
  {
    if (! aCard.faceUp())
      aCard.flip();
    super.addCard(aCard);
  }
  public void select (int tx, int ty)
  {
    if (empty())
      return;
    Card topCard = pop();
    for (int i = 0; i < 4; i++)
    {
      if (Solitaire.suitPile[i].canTake(topCard))
      {
        Solitaire.suitPale[i].addCard(topCard);
        return;
      }
    }
    for (int i = 0; i < 7; i++)
    {
      if (Solitaire.tableau[i].canTake(topCard))
      {
        Solitaire.tableau[i].addCard(topCard);
        return;
      }
    }
    // никто не может ее использовать,
   // положим ее назад
    addCard(topCard);
  }
}
```

### *8.4.4. Стопка расклада TablePile*

Наиболее сложный из подклассов класса *CardPile* — это тот, который используется для хранения стопок расклада TablePile. Он показан в **листингах 8.9** и **8.10**. Стопки расклада отличаются от стопок карт общего назначения следующими моментами:

- При инициализации (с помощью конструктора) стопки расклада забирают определенное количество карт из колоды, перемещая их к себе. Количество карт, удаленных таким образом, определяется дополнительным аргументом, передаваемым конструктору. Верхняя карта стопки открывается.
- Карта может быть добавлена в стопку (проверяется методом *canTake*), только если стопка пуста и эта карта — король или если карта оказывается противоположного цвета по сравнению с текущей верхней картой стопки и ее ранг на единицу меньше, чем ранг верхней карты.
- Для проверки попадания щелчка мыши в пределы области стопки (метод *includes*) учитываются только левая, правая и верхняя границы. Нижняя граница игнорируется, так как стопка расклада *TablePile* может быть переменной длины.
- При щелчке мышью в пределах стопки карт расклада верхняя карта открывается, если она была закрыта. Если карта открыта, то делается попытка переместить ее сперва в какое-нибудь основание, а затем — в какую-нибудь стопку расклада. Только если ни одна из стопок не может принять карту, она остается на месте.
- Для отображения стопки на экране карты в ней рисуются друг за другом, так что каждая следующая карта понемногу сдвигается вниз. Для того чтобы сделать это с нижней до верхней карты включительно, небольшая рекурсивная подпрограмма

PDF created with pdfFactory Pro trial version www.pdffactory.com

(объявленная как *private*) просматривает весь связный список, отображая карты в тот момент, когда управление возвращается назад после рекурсивного вызова. Эта функция использует итератор для цикла по элементам списка.

## 8.5. Полиморфная игра

Как мы уже видели в «Задаче о восьми ферзях» (глава 5), среда для всех приложений на языке Java обеспечивается классом Applet. Для создания нового приложения программист определяет подклассы Applet, переопределяя при этом различные методы. Класс Solitare, который является центральным классом нашего приложения, показан в **листинге 8.11**.

Мы ранее уже отмечали, что переменные, которые хранят общие для всех объектов данные, объявляются с ключевым словом static. Такие поля инициализируются в методе init класса1.

Массивы в языке Java — это нечто, отличное от массивов в большинстве других языков программирования. Java различает для массивов три действия: объявление, распределение и присваивание. Заметьте, что объявление

**Листинг 8.9.** Класс TablePile, часть I

```
class TablePile extends CardPile
{
  TablePile (int x, int y, int c)
  {
   // инициализация надкласса
   super(x, y);
    // затем инициализируется наша стопка карт
   for (int i = 0; i < c; i++)
   {
      addCard(Solitaire.deckPile.pop());
   }
    // верхняя карта открывается
   top.flip();
  }
  public boolean cantake (Card aCard)
  {
    if (empty())
      return aCard.rank() == 12;
    Card topCard = top();
    return (aCard.color() != topCard.color()) &&
        (aCard.rank() == topCard.rank() – 1);
  }
  public boolean includes (int tx, int ty)
  {
    // не проверяет нижнюю границу
    return x < = tx && tx<= x + Card.width &&
        y <= ty;
  }

  private int stackDisplay
    (Graphics g, ListIterator itr)
  {
    int localy;
    if (itr.atEnd())
      return y;
    Card aCard = (Card) itr.current;
    itr.next;
    localy = stackDisplay(g, itr);
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    aCard.draw(g, x, localy);
    return localy + 35;
  }
...
```

**Листинг 8.10.** Класс TablePile, часть II

```
class TablePile extends CardPile
{
  ...
  public void select (int tx, int ty)
  {
    if (empty())
      return;
     // если карта закрыта, перевернуть
    Card topCard = top();
    if (! topCard.faceUp())
    {
       topCard.flip();
       return;
    }
    // иначе смотрим, можно ли ее положить в основание
    topCard = pop();
    for (int i = 0; i < 4; i++)
    {
      if (Solitaire.suitPile[i].canTake(topCard))
      {
        Solitaire.suitPile[i].addCard(topCard);
        return;
      }
    }
    // нельзя ли положить в другую стопку расклада
    for (int i = 0; i < 7; i++)
    {
      if (Solitaire.tableau[i].canTake(topCard))
      {
        Solitaire.tableau[i].addCard(topCard);
        return;
      }
    }
    // иначе кладем обратно
    addCard(topCard);
  }
  public void display (Graphics g)
  {
    stackDisplay(g, cardList.iterator());
  }
}
```

**Листинг. 8.11.** Класс Solitaire

```
public class Solitaire extends Applet
{
  static DeckPile deckPile;
  static DisacrdPile discardPile;
  static TablePile tableau [ ];
  static SuitPile suitPile [ ];
  static CardPile allPiles [ ];
  public void init()
  {
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    // сначала отводим место под массивы
    allPiles = new CardPile[13];
    suitPile = new SuitPile[4];
    tableau = new TablePile[7];
    // затем заполняем их данными
    allPiles[0] = deckPile = new DeckPile(335, 5);
    allPiles[1] = discardPile =
      new DiscardPile(268, 5);
    for (int i = 0; i < 4; i++)
    {  allPiles[2+i] = suitPile[i] =
         new SuitPile(15 + 60 * i, 5);
    }
    for (int i = 0; i < 7; i++)
    {  allPiles[6+i] = tableau[i] =
         new TablePile(5 + 55 * i, 80, i+1);
    }
  }
  public void paint(Graphics g)
  {
    for (int i = 0; i < 13; i++)
    {  allPiles[i].display(g);  }
  }
  public boolean mouseDown(Event evt, int x, int y)
  {
    for (int i = 0; i < 13; i++)
    {
      if (allPiles[i].includes(x, y))
      {  allPiles[i].select(x, y);
         repaint();
         return true;
      }
    }
    return true;
  }
}
```

показывает только то, что объекты являются массивами; про их границы ничего не говорится. Один из первых шагов процедуры инициализации — выделение места под три массива (основания, стопки расклада и массив allPiles, который мы рассмотрим ниже). Команда new отводит память для этих массивов, но не присваивает никаких значений их элементам.

Следующий шаг — создание колоды DeskPile. Вспомните, что конструктор этого класса генерирует и перетасовывает полную колоду из 52 карт. Промежуточная стопка DiscardPile создается аналогичным образом. Затем в цикле порождаются и инициализируются четыре основания SuitPile, а второй цикл создает и инициализирует стопки расклада TablePile. Вспомните, что при инициализации стопок расклада карты берутся из колоды и вставляются в стопку расклада.

Массив allPiles используется для представления всех 13 стопок карт. Заметьте, что как только создается очередная стопка, ей тут же присваивается ячейка в этом массиве, равно как и соответствующая статическая переменная. Мы воспользуемся этим массивом для иллюстрации еще одного аспекта наследования. Следуя принципу подстановки, allPiles объявлен как массив из элементов с типом данных CardPile, но на самом деле он содержит стопки карт разнообразного вида.

Данный массив используется в ситуациях, когда различия между типами стопок карт не важны. Например, в процедуре перерисовки экрана каждую стопку просто просят самостоятельно перерисовать себя. Похожим образом при щелчке мышью опрашивается

PDF created with pdfFactory Pro trial version www.pdffactory.com

каждая стопка, не содержит ли она указанную точку экрана. Если да, то стопка выделяется. Среди них есть семь стопок расклада, четыре основания, промежуточная стопка и колода. Более того, фактический код, исполняемый в ответ на вызов методов select и includes, может различаться в зависимости от типа обрабатываемой стопки.

Использование переменных, объявленных как экземпляры родительского класса, но содержащих значения, относящиеся к подклассам, — это один из аспектов *полиморфизма* (тема, к которой мы вернемся в следующей главе).

## *8.6. Создание более сложной игры*

Пасьянс, описанный здесь, обладает минимальными возможностями и в нем чрезвычайно трудно выиграть. Более реалистичная игра включала бы по крайней мере некоторые из следующих вариаций:

- Метод select в классе TablePile следует расширить до распознавания последовательностей (напомним: последовательность — это часть стопки, состоящая из открытых карт расклада, которая перемещается в другую стопку расклада как единое целое). В этом случае если верхняя карта стопки не может быть перемещена, то нужно сделать проверку для самой нижней открытой карты. Если ее можно перенести, то весь набор открытых карт должен быть перемещен в другую стопку.
- Наша игра останавливается после одного просмотра колоды. Альтернатива: когда пользователь выбирает пустую стопку колоды (щелкая мышкой там, где была колода), то промежуточная стопка складывается обратно в колоду, позволяя продолжить пасьянс.
-

Другие альтернативные правила описаны в упражнениях.

### *Упражнения*

1. Этот пасьянс был преднамеренно создан как можно более простым. Небогатый набор возможностей слегка раздражает, не правда ли? Его легко расширить за счет добавления кода. Возможные новые свойства:
    a. верхняя карта стопки расклада не должна перемещаться в другую стопку расклада, если под ней имеется открытая карта;
    b. целая последовательность не может перемещаться, если самая нижняя карта — король и не осталось закрытых карт.

    Для каждого случая опишите, какие процедуры требуют изменений, и составьте для них исправленный код.

2. Ниже следуют общеизвестные вариации «косынки». Опишите для каждой из них, какие части пасьянса должны быть изменены.
    a. Если пользователь щелкнул мышью по пустой стопке колоды, то промежуточная стопка перемещается (возможно, с перемешиванием) назад в колоду. Таким образом пользователь может перебирать колоду неоднократно.
    b. Карты могут быть передвинуты из оснований назад в стопку расклада.

PDF created with pdfFactory Pro trial version www.pdffactory.com

c. Карты вытягиваются из колоды по три сразу и располагаются в промежуточной стопке в обратном порядке. Как и прежде, в игре участвует только самая верхняя карта в промежуточной стопке. Если в колоде остается меньше трех карт, то все оставшиеся карты перемещаются в промежуточную стопку. (На практике эта разновидность часто сочетается с вариантом а, обеспечивая многоразовый проход колоды.)
d. То же, что вариант в, но в игре принимает участие любая из трех карт промежуточной стопки. (Это требует небольшого изменения во внешнем виде карточного стола, и больших исправлений в классе промежуточной стопки.)
e. На пустую стопку расклада может быть положена любая фигурная карта (король, дама, валет), а не только король.

3. Пасьянс «thumb and pouch» похож на «косынку» за исключением того, что карта может быть положена на другую карту следующего по старшинству ранга и любой масти за исключением ее собственной. Так, девятку пик разрешается класть на десятку крестей, но не на десятку пик. Эта разновидность значительно увеличивает шансы на победу. (Согласно Морехеду [Morehead 1949], шансы на победу в «Klondike» составляют 1 к 30, тогда как в «thumb and pouch» —1 к 4.) Опишите, какие фрагменты программы требуют приспособления к новому варианту.

# *Глава 9: Повторное использование кода*

Объектно-ориентированное программирование было объявлено как технология, которая позволит наконец конструировать программы из многократно используемых компонент общего назначения. Такие авторы, как Брэд Кокс, зашли так далеко, что уже говорили об объектно-ориентированном подходе как о предвестнике «промышленной революции» в разработке программного обеспечения [*Cox 1986*]. Пока действительность не вполне соответствует ожиданиям пионеров ООП (тема, к которой мы еще обратимся в конце этой главы). Что действительно справедливо — так это то, что ООП позволяет встраивать многократно используемые программные компоненты гораздо интенсивнее, чем раньше. В этой главе мы рассмотрим два наиболее общих механизма многократного использования программного обеспечения, которые известны как *наследование и композиция*.

Механизмы многократного использования — это только первый шаг. Наследование и композиция обеспечивают средства многократного использования, но чтобы быть эффективными, они должны, вообще говоря, применяться в единой среде разработки, которая располагает поддержкой многократного использования. Схемы и среды разработки, которые предоставляют такое окружение, будут рассмотрены в главе 18.

## *9.1. Наследование и принцип подстановки*

Наследованиеикомпозициювкачестветехникимногократного использования кода, возможно, легче понять в их связи с принципом подстановки. Вспомните главу 8, в которой мы ссылались на этот принцип в связи с переменной, объявленной с одним классом, которая получает значение из другого класса. Принцип подстановки утверждает, что допустимо присваивать значение переменной, если класс значения является классом переменной или его подклассом.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Мы видели примеры переопределения при моделировании игры в бильярд в главе 6. В процедуре, которая рисует образ экрана, переменная была объявлена как принадлежащая классу GraphicalObject, но на самом деле она последовательно содержала в качестве значений различные объекты, каждый из которых являлся экземпляром подкласса класса GraphicalObject.

В этом разделе мы обсудим не настоящие классы, а абстрактные концепции, программной реализацией которых выступают классы. При каких условиях одно абстрактное понятие можно подставить вместо другого? То есть при каких условиях экземпляр некоторого абстрактного понятия перестановочен с экземпляром другого абстрактного понятия? Одно из классических правил применяемых здесь, ставшее основным для объектно-ориентированного проектирования, известно как «быть экземпляром» (правило *«is-a»*).

### *9.1.1. «Быть экземпляром» и «включать как часть»*

Знание двух различных форм отношений — основа понимания того, как и когда применять приемы многократного использования кода. Имеются два типа отношений, известных как *быть экземпляром* и *включать как часть* (is-a и has-a).

Отношение *быть экземпляром* имеет место между двумя понятиями, если первое является уточнением второго. То есть для всех практических целей поведение и данные, связанные с более конкретным понятием, составляют подмножество поведения и данных, связанных с более абстрактным понятием. Например, все примеры наследования, описанные нами в предыдущих главах, удовлетворяют отношению *быть экземпляром* (хозяйка цветочного магазина Florist является экземпляром класса владельцев магазина Shopkeeper, собака Dog является экземпляром класса млекопитающих Mammal, бильярдный шар Ball является экземпляром класса графических объектов GraphicalObject, и т. д.).

Название этого отношения происходит из простого правила проверки. Чтобы определить, является ли понятие X уточненным вариантом Y, просто составьте предложение «X *является экземпляром* Y». Если утверждение звучит корректно, то есть оно соответствует вашему жизненному опыту, то вы можете заключить, что X и Y связаны отношением *быть экземпляром*.

Напротив, отношение *включать как часть* имеет место, когда второе понятие является компонентой первого, но оба эти понятия не совпадают ни в каком смысле независимо от уровня общности абстракции. Например, автомобиль Car *имеет* двигатель Engine, хотя ясно, что это не тот случай, когда Car *является экземпляром* Engine или Engine *является экземпляром* Car. Car тем не менее *является экземпляром* класса автомобилей Vehicle, который в свою очередь *является экземпляром* класса средств передвижения MeansOtTransportation 1.

1. Условие «X является экземпляром Y» не должно трактоваться ни как утверждение «X является подмножеством Y», рассматриваемое в теоретико-множественном смысле, ни как утверждение «X является наращиванием Y» в смысле расширения структуры данных. Условие «X является экземпляром Y» справедливо, если сущность X — это уточнение, детализация, конкретизация, специализированная форма, и т. д. сущности Y, и ни в каком другом случае. В частности, приводимый далее пример с автомобилем и двигателем выглядит следующим образом: двигатель входит в автомобиль в смысле теории множеств (двигатель — более

PDF created with pdfFactory Pro trial version www.pdffactory.com

мелкая «подробность» и фрагмент устройства автомобиля), автомобиль усложняет двигатель (автомобиль — это двигатель целиком плюс еще много другого), но ни в том, ни в другом случае не справедливо условие «X является экземпляром Y». — Примеч. перев.

Еще раз, чтобы проверить отношение включать как часть, просто составьте предложение «X включает Y как часть» и предоставьте решать здравому смыслу.

В большинстве случаев различие ясно. Но иногда оно может быть сомнительно или зависеть от обстоятельств. В следующем разделе мы анализируем один такой случай, чтобы проиллюстрировать два метода разработки программного обеспечения, которые естественно основываются на этих двух отношениях.

### *9.2. Композиция и наследование: описание*

Чтобы проиллюстрировать композицию и наследование, мы построим тип данных set — абстракцию множества — на основе существующего класса List. Экземпляры класса List содержат списки целочисленных величин. Допустим, что мы уже создали класс List со следующим интерфейсом:

```
class List
{
  public:
     // конструктор
     List  ();
     // методы
     void addToFront   (int);
     int firstElement  ();
     int length                (();
     int includes              (int);
     int remove                (int);
   ...
};
```

Наша абстракция списка позволяет нам добавлять новый элемент в начало списка, выдавать его первый элемент, находить количество элементов, проверять, содержится ли значение в списке, и удалять элемент из списка.

Мы хотим создать абстракцию множества, чтобы выполнять такие операции, как добавление значения к множеству, определение количества элементов, выяснение принадлежности к множеству.

#### *9.2.1. Использование композиции*

Сначала мы исследуем, может ли абстракция множества быть создана с помощью композиции. Напомним, что объект — это просто инкапсуляция данных и поведения. Когда для многократного использования существующей абстракции данных при создании нового типа используется композиция, то часть новой структуры данных является просто экземпляром существующей структуры. Это показано ниже, где тип данных Set содержит поле, названное theData, которое объявлено с типом List.

```
Class Set
{
  public:
    Set (); // конструктор
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
   // операции
   void  add                  (int);
   int   size          ();
   int   includes      (int)
  private:  // область данных для значений
    List  theData;
};
```

Поскольку абстракция List хранится как часть области данных нашего множества, она должна быть инициализирована в конструкторе. Будучи аналогичными командам инициализации полей данных для классов (глава 4), команды инициализатора в начале конструктора задают аргументы для инициализации полей данных. В данном случае конструктор, который мы вызываем для класса List, — безаргументный:

```
// список инициализации
Set::Set() : theData()
{
  // никакой дальнейшей инициализации
}
```

Операции в новой структуре данных реализованы с использованием уже существующих действий, предоставляемых старым типом данных. Например, операция includes для множества просто вызывает функцию с аналогичным названием, уже определенную для списков:

```
int Set::size ()
{
    return theData.length();
}
int Set::includes  (int newValue)
{
     return theData.includes(newValue);
}
```

Только одна операция оказывается чуть более сложной. Это — добавление нового элемента, так как нужно сначала убедиться, что данная величина не содержится в множестве (величины не могут появляться в множестве более одного раза):

```
void Set::add (int newValue)
{
   // если не в множестве
   if (! Includes (newValue))
{
      // тогда добавить
     theData.addToFront(newValue);
   };
   // иначе ничего не делать
}
```

Важным является тот факт, что такая композиция помогает повторному использования кода в новых приложениях. За счет существующего класса List большая часть трудной работы по управлению значениями данных для нашей новой компоненты была уже проделана.

Однако композиция ничего не говорит о соблюдении принципа подстановки. При создании нового типа указанным способом абстракции List и Set будут абсолютно различны, и ни одна их них не может быть подставлена вместо другой.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### *Композиция в других языках*

Композиция может быть применена в любом объектно-ориентированном языке программирования, рассматриваемом в этой книге. Но она встречается и в языках, не являющихся объектно-ориентированными. Единственная существенная разница — в способе инициализации инкапсулированных данных. В языке Smalltalk в общем случае это выполняется через *класс-методы*, в языке Objective-C — с помощью *методов-фабрик*, в языках Java и Object Pascal — с использованием конструкторов.

## *9.2.2. Применение наследования*

Абсолютно другим механизмом многократного использования кода в ООП является наследование. С его помощью новый класс может быть объявлен как подкласс, или дочерний класс, существующего класса. В этом случае все области данных и функции, связанные с исходным классом, автоматически переносятся на новую абстракцию данных. Новый класс может определять дополнительные значения или функции. Он переопределяет некоторые функции исходного класса, просто объявив новые с такими же именами, как и в исходном классе.

Все это проиллюстрировано ниже в классе, который реализует другую версию абстракции Set. Упоминая класс List в заголовке класса, мы показываем, что наша абстракция Set является расширением или уточнением существующего класса List. Таким образом, операции, связанные со списками, применимы и к множествам:

```
Class Set : public List
{ public:
   // конструктор
   Set();
   // операции
   void add (int);
   int  size ();
};
```

Заметьте, что новый класс не определяет никаких новых полей данных. Вместо этого поля данных класса List будут использоваться для хранения элементов множества. Эти поля должны быть по-прежнему проинициализированы. Данная операция выполняется вызовом конструктора надкласса в конструкторе нового класса:

```
Set::Set() : List()
{
  // никакой дальнейшей инициализации
}
```

Аналогично функции, определенные в родительском классе, могут быть использованы без каких-либо дальнейших усилий, и, следовательно, нам не нужно беспокоиться по поводу метода includes, так как наследованный метод из List имеет такое же имя и служит тем же целям. Добавление в множество нового элемента требует немного больше работы, чем в классе List:

```
void Set::add   (int newValue)
{
     // добавить, если нет в множестве
    if (! Includes(newValue))
        addToFront (newValue);
}
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Сравните эту функцию с предыдущей версией. Обе техники — мощные механизмы для многократного использования кода, но в отличие от композиции наследование поддерживает неявное предположение, что подклассы на самом деле являются подтипами. Это значит, что экземпляры новой абстракции должны вести себя так же, как и экземпляры родительского класса.

### *Наследование в других языках*

В главе 7 мы вкратце описали синтаксис, используемый для наследования в каждом из рассматриваемых нами языков программирования. Как и в случае композиции, главное — гарантировать, что абстракция надкласса проинициализирована должным образом.

### *9.2.3. Закрытое наследование в языке C++*

C++ предоставляет интересный компромисс между композицией и наследованием как механизмами многократного использования кода. Это происходит путем использования ключевого слова private вместо ключевого слова public в заголовке определения класса. В этом случае программист сигнализирует, что наследование следует использовать при конструировании новой абстракцииданных, но такая абстракция не должна рассматриваться как уточненная форма родительского класса:

```
Class Set : private List
{
  public:
     // конструктор
    Set  () : List () { }
    // операторы
    void add                    (int);
    int includes       (int x);
    {
      return List::includes(x);
    }
    int size                    ()
    {
       return List::length();
    }
};
```

Применяя термины, которые будут определены более строго в главе 10, можно сказать, что закрытое наследование создает подкласс, который не является подтипом. Тем самым закрытое наследование использует механизм наследования, но в явном виде нарушает принцип подстановки. Операции и области данных, наследуемые из родительского класса, задействуются в методах новой абстракции, но они не «просматриваются насквозь» и недоступны ее пользователям. По этой причине любой метод, который программист хочет экспортировать (такой, как includes в абстракции множества), должен быть переопределен заново для нового класса, даже если все, что он делает, — это вызов метода класса-предка. (Как было уже проиллюстрировано, чтобы избежать накладных расходов при вызовах процедур в подобных простых случаях, часто используются встраиваемые методы.)

Закрытое наследование является интересной идеей и наиболее полезно, когда (как в данном случае) объект в основном составляется из абстракции данных другого типа и работа при создании нового объекта выполняется в основном инкапсулированной абстракцией, однако новое понятие не удовлетворяет отношению *быть экземпляром*, необходимому для открытого наследования.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### *9.3. Противопоставление композиции и наследования*

Проиллюстрировав два механизма многократного использования программного обеспечения и увидев, что они оба применимы для реализации множеств, мы можем прокомментировать некоторые недостатки и преимущества двух подходов:

- Композиция более проста. Ее преимущество заключается в том, что она ясно показывает, какие точно операции будут выполняться над конкретной структурой данных. При взгляде на описание абстракции данных Set становится очевидно, что для типа данных предусмотрены только операции добавления элемента, проверки на наличие элемента и определение числа элементов в наборе. Это справедливо независимо от того, какие операции определены для списков.
- При наследовании операции новой абстракции данных являются надмножеством операций исходной структуры данных. Таким образом, чтобы точно знать, какие операции разрешены для новой структуры, программист должен рассмотреть объявление исходной структуры. Например, изучение описания класса Set не показывает сразу же, что проверка на наличие элемента (метод includes) разрешена для множеств. Только из рассмотрения описанной ранее абстракции данных List видно, что имеется еще целый набор допустимых операций. Трудность состоит в следующем: чтобы понять класс, сконструированный с помощью наследования, программист должен постоянно переключаться «взад-вперед» между двумя (или более) описаниями классов. Она известна как проблема «вверх-вниз» («йо-йо») [Taenzer 1989].
- С другой стороны, лаконичность абстракции данных, созданной с помощью наследования, является преимуществом. Используя наследование, не обязательно писать весь код для доступа к функциям базового класса. По этой причине реализации с использованием наследования (как это было в нашем случае) значительно меньше по объему, если сравнить их с композицией. Наследование также часто обеспечивает большую функциональность. Например, применение наследования в нашем случае делает доступным для множеств не только проверку include, но и функцию remove.
- Наследование не запрещает пользователям манипулировать новыми структурами через вызовы методов родительского класса, даже если эти методы не вполне подходят под идеологию потомка. Например, когда мы использовали наследование для получения множеств Set из списков List, то ничто не мешало пользователям добавлять новые элементы к множеству, вызывая унаследованный от класса List метод addToFront.
- При композиции тот факт, что класс List используется для хранения наших множеств, — просто деталь реализации. С этой техникой было бы легко заново реализовать класс, чтобы извлечь пользу из применения других методов (например, таких, как хэш-таблицы) с минимальным воздействием на пользователей абстракции данных Set. Если пользователь рассчитывает на тот факт, что абстракция Set — это просто уточненная форма абстракции List, то такие изменения было бы трудно реализовать.
- Наследование позволяет нам использовать новую абстракцию как аргумент существующей полиморфной функции. Мы будем исследовать эту возможность более детально в главе 14. Так как композиция не подразумевает соблюдение принципа подстановки, она обычно устраняет *полиморфизм*.
- О степени понятности кода судить трудно. Наследование имеет преимущество в краткости кода, но не протокола. При композиции код класса, хотя он и оказывается длиннее, — это все, что должен понять другой программист, чтобы использовать абстракцию. Человек, столкнувшийся с необходимостью разобраться

PDF created with pdfFactory Pro trial version www.pdffactory.com

с версией кода с наследованием, вынужден проверять, является ли поведение, наследуемое из родительского класса, необходимым для должного использования нового класса, и таким образом он анализирует оба класса.
- Структуры данных, реализованные с помощью наследования, имеют (незначительное) превосходство в смысле скорости выполнения над структурами с композицией. Причина состоит в том, что в первом случае исключается один дополнительный вызов функции (хотя техника inline в языке C++ может ликвидировать практически все накладные расходы по вызову функций).

Имея два различных механизма реализации, можем ли мы сказать, который из них лучше в нашем конкретном случае? Обратимся к принципу подстановки. Спросите себя, корректно ли в приложении, которое предполагает использование абстракции данных List, подставлять вместо нее множество Set? Хотя чисто техническим ответом может быть «да» (абстракция Set действительно реализует все операции List), здравый смысл говорит, скорее, «нет». Поэтому в *данном случае* композиция подходит лучше.

Последний штрих: обе техники очень полезны, и объектно-ориентированный программист должен быть знаком с обеими.

### *9.4. Повторное использование кода: реальность?*

На заре объектно-ориентированного программирования утверждалось, что композиция и наследование обеспечили возможность создания программного обеспечения из взаимозаменяемых компонент общего назначения. Но вопреки очевидному прогрессу (сейчас работает огромное количество коммерческих поставщиков, предлагающих объектно-ориентированные библиотеки общего назначения для различных приложений — пользовательские интерфейсы, контейнеры данных и т. д.), общий процесс не оправдал всех ожиданий. На то имеется ряд причин:

- Наследование и композиция предоставляют средства для создания многократно используемых компонент, но сами собой они не дают указаний, как такая цель может быть достигнута. Оказывается, что создание хороших и полезных программных компонент почти всегда труднее разработки конкретно требуемой программы.
- Так как создание многократно используемых компонент затруднено, то окупаемость не достигается в пределах одного проекта. На самом деле стремление к модульности может даже снизить скорость реализации проекта. Немедленная выгода не достигается, так что приходится рассчитывать на погашение затрат за счет последующих программных проектов. Но так как последние зачастую имеют свой собственный бюджет и планирование, то административные механизмы подобной амортизации отсутствуют.
- Так как применение многократно используемых компонент непосредственно не улучшает проект, программисты не имеют стимула бороться за многократное использование.
- Так как каждая новая задача обычно требует специфической функциональности, часто трудно с самого начала спроектировать действительно полезные программные компоненты общего назначения. Скорее всего, подходящие компоненты будут постепенно эволюционировать во многих проектах, пока наконец не достигнут стабильного состояния.
- Многие программисты и администраторы подозрительно относятся к программам, которые не были разработаны в родных стенах. Такая осторожность называется синдромом «сделано не здесь». Так как сами администраторы гордятся своими

PDF created with pdfFactory Pro trial version www.pdffactory.com

командами, они, естественно, верят, что их программисты могут все сделать лучше, чем кто-либо.
- Так как многие программисты недостаточно формально обучены или не держат руку на пульсе последних программных нововведений (таких, как ООП), они могут не знать о механизмах для разработки многократно используемого кода.

Короче говоря, развитие программных механизмов для многократного использования само по себе не гарантирует технологическую и управленческую *культуру*, которая бы поддерживала и поощряла повторное использование программных компонент. Человеческие организации прогрессируют медленнее, чем технологии, поэтому, возможно, пройдет еще много лет до того, как мы увидим действительные выгоды, обещанные объектно-ориентированным подходом. Тем не менее многократное использование объектов применяется, возможно, не везде и далеко не так часто, как было заявлено, но все же применяется, и при правильном обращении оно тысячу раз доказало свою полезность и способность сокращать затраты. По этой причине многократное использование неизбежно станет нормой разработки программного обеспечения.

Вот список недавно изданных книг, посвященных разработке многократно используемых компонент [*Carroll 1995, McGregor 1992, Meyer 1994, Goldberg 1995*].

### *Упражнения*

- Есть различные способы реализации структуры данных стек — например, с помощью списков или в виде массива. Предположим, мы имеем и класс списков List, и класс массивов Array. Для каждого из них проиллюстрируйте, как можно построить стек, используя и наследование, и композицию. Вам разрешается ввести какие угодно методы для базовых классов. Какая из техник реализации кажется вам более подходящей в данном случае?
- Снова предположим, что мы имеем готовую структуру данных List и хотим построить абстракцию, представляющую собой упорядоченный список, в котором элементы вставляются в нужной последовательности, а не просто в начало списка. Наследование или композицию вы будете использовать в данном случае? Обоснуйте ваш ответ.

## *Глава 10: Подклассы и подтипы*

Одной из наиболее интересных особенностей объектно-ориентированных языков программирования является тот факт, что фактический тип переменной может не совпадать с типом, заявленным при ее описании. Мы видели это в главе 6, в конце программы, моделирующей бильярд, где переменная типа GraphicalObject в действительности принимала значения типа Ball, Wall или Hole. Это один из аспектов полиморфизма, который мы будем детально обсуждать в главе 14. В традиционных языках программирования, таких как Pascal или C, если переменная описана как integer, то, что бы ни случилось, содержимое части памяти, отведенной под эту переменную, гарантированно будет интерпретироваться как целая величина. В объектно-ориентированном языке переменная, описанная как Window, в действительности может принимать значения GraphicWindow, TextEditWindow или какого-либо другого оконного типа.

Несколько терминов помогут нам понять, что же подразумевается под этими изменениями. Будем использовать понятие статический тип для обозначения типа,

PDF created with pdfFactory Pro trial version www.pdffactory.com

присвоенного переменной при ее описании. Термин динамический тип характеризует тип фактического значения. Тот факт, что динамический и статический типы переменной не обязаны совпадать, является одним из главнейших достоинств объектно-ориентированного программирования. Переменная, для которой динамический тип не совпадает (точнее, может не совпадать) со статическим, называется полиморфной.

Еще два термина имеют отношение к обсуждаемому вопросу. Понятие подкласс уже было введено в главе 7. Благодаря подклассам новые компоненты программ разрабатываются на основе уже существующих. Понятие подтипа является более абстрактным. Подтип определяется в терминах поведения, а не структуры. Мы говорим, что тип B есть подтип типа A, если мы можем в любой ситуации подставить экземпляр класса B вместо экземпляра класса A без каких-либо видимых изменений в поведении.

Отметим, что понятие подтипа соответствует нашему идеализированному принципу подстановки, введенному в главе 9. Однако, абстрактно рассуждая, не существует никаких видимых причин для того, чтобы понятия подкласса и подтипа имели какую-то взаимосвязь. Действительно, в языках программирования с динамическими типами данных вроде Smalltalk они и не связаны. Два класса могут иметь общее поведение — например, по отношению к одному и тому же набору сообщений, — но без общей реализации или единого предка (за исключением класса всех объектов Object). Если их реакции на общие сообщения в достаточной степени аналогичны, то один класс может быть с легкостью подставлен вместо другого. Представьте себе класс разреженных массивов, который поддерживает операции класса массивов Array. Тогда в любой алгоритм, предназначенный для работы с массивами, можно подставить разреженный массив.

Большинство объектно-ориентированных языков программирования со строгим контролем типов данных (такие, как Object Pascal и C++) делают размытым различие между подтипами и подклассами в двух отношениях. Во-первых, они разрешают переменным принимать значения другого типа, только когда динамический тип значения является подклассом статического типа. Во-вторых, они предполагают, что фактически все подклассы являются подтипами. Это предположение не всегда справедливо. Поскольку подклассы могут переопределять методы родителей на произвольное новое поведение, то в общем случае не существует никаких гарантий того, что подкласс будет также и подтипом.

## 10.1. Связывание методов и сообщения

В следующем разделе мы обсудим, как выглядит различие между подклассами и подтипами в обсуждаемых нами языках программирования. Прежде всего, однако, мы должны затронуть две проблемы. Первая из них связана с вопросом о способе связывания. Должны ли мы связывать сообщение и метод, основываясь на статическом типе переменной, или следует принять во внимание ее динамический тип? Второй проблемой является вопрос обращения полиморфизма — можно ли присвоить значение переменной, основываясь на ее динамическом, а не на статическом типе? Третий, тесно связанный с данной темой вопрос — а именно о точном значении переопределения методов, — достаточно сложен, поэтому мы отложим его обсуждение до следующей главы.

### 10.1.1. Связывание методов

Существование полиморфной переменной естественным образом

PDF created with pdfFactory Pro trial version www.pdffactory.com

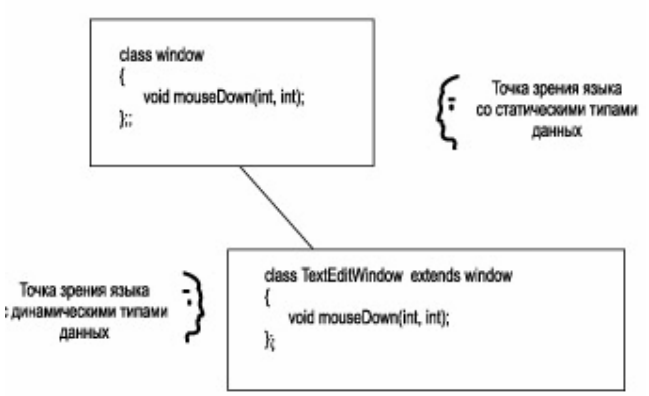



**Рис. 10.1**. Две точки зрения на полиморфную переменную

подразумевает наличие двух различных представлений о ней (рис. 10.1). Мы можем рассматривать переменную с точки зрения описания (статически) или с точки зрения ее текущего значения (динамически). Это различие становится важным, когда мы встречаем метод, определенный в родительском классе и переопределенный в дочернем. На рис. 10.1, например, показан метод mouseDown, который имеется как у родительского класса (Window), так и у дочернего (TextEditWindow). Когда полиморфная переменная получает сообщение о щелчке мышью, с каким из методов должно связываться это сообщение?

Однозначного правильного ответа на этот вопрос не существует. Чаще всего мы ожидаем, что связывание сообщения должно происходить исходя из динамического типа данных. Однако существуют ситуации, где противоположный подход также полезен. Ниже мы обсудим, как разные языки программирования решают эту проблему.

### 10.1.2. Проблема обращения полиморфизма

Принцип подстановки гласит, что переменной, описанной как экземпляр родительского класса, мы можем присвоить значение дочернего типа. Можем ли мы двигаться в другом направлении? То есть, присвоив переменной Y типа Window значение переменной X типа TextEditWindow, сможем ли мы затем присвоить новой переменной Z типа TextEditWindow нашу переменную Y?

Этот вопрос в действительности содержит в себе две тесно связанные проблемы. Чтобы проиллюстрировать это, предположим, что мы определяем класс бильярдных шаров Ball и два его подкласса — черные шары BlackBall и белые шары WhiteBall. Далее мы создаем программный эквивалент коробки, в которую мы можем положить двух представителей класса Ball, один из которых

PDF created with pdfFactory Pro trial version www.pdffactory.com

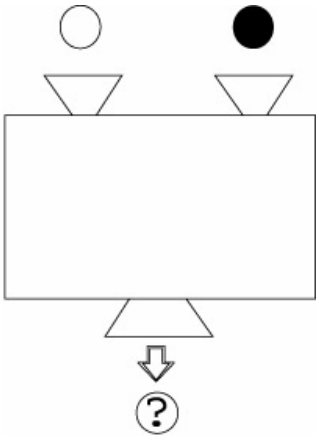


**Рис. 10.2**. Бильярдный шар, теряющий свою однозначность

(выбранный случайно) затем вынимается обратно (рис. 10.2). Мы опускаем BlackBall и WhiteBall в коробку и смотрим, что получается на выходе.

Вытащенный из коробки объект определенно может рассматриваться как шар Ball и поэтому может быть присвоен переменной, описанной с этим типом. Но будет ли он черным шаром BlackBall? Мы можем задать здесь два вопроса: 1) могу ли я определить, что этот объект принадлежит к типу BlackBall, или нет? 2) какие механизмы необходимы для того, чтобы присвоить это значение переменной дочернего класса BlackBall?

Хотя пример с шарами BlackBall и WhiteBall может показаться надуманным, скрывающаяся за ним проблема является весьма общей. Рассмотрим проектирование классов для часто используемых структур данных: множества, стеки, очереди, списки и т. п., которые используются, чтобы хранить совокупность объектов. Рекламируемая выгода объектно-ориентированного программирования состоит в производстве многократно используемых компонент программ, и контейнеры для совокупностей данных являются неплохими кандидатами на эту роль. Однако контейнеры при определенных обстоятельствах похожи на машину с выбором шаров. Если программист помещает два различных объекта в набор и позже вынимает оттуда один, как он узнает, что за объект он получил?

Эти проблемы нетривиальны, но разрешимы. В разделах этой главы, посвященных конкретным языкам программирования, мы опишем механизмы обращения полиморфизма (если они имеются). В следующей главе мы рассмотрим проблему контейнерных классов подробнее.

## 10.2. Связывание в языках программирования

В следующих разделах мы опишем связывание методов в рассматриваемых языках программирования. В частности, будут противопоставляться между собой способы связывания в языках со статическими типами данных (Object Pascal, C++ и Java) и в языках с динамическими типами (Smalltalk и Objective-C).

### 10.2.1. Связывание в языке Object Pascal

PDF created with pdfFactory Pro trial version www.pdffactory.com

Язык ObjectPascal является языком программирования со статическими типами данных. Каждый идентификатор в программе должен быть описан. Кроме того, понятия подкласса и подтипа объединены. Неявно предполагается, что подклассы являются подтипами и что идентификатор, описанный как объект, может иметь значение этого типа или любого, полученного из него путем наследования.

Хотя язык базируется на статических типах данных, объекты тем не менее несут с собой знания об их собственном динамическом типе. В Object Pascal версии Apple может использоваться встроенная в систему логическая функция Member, которая определяет, является ли идентификатор экземпляром определенного класса. Функция Member использует в качестве параметров ссылку на объект (обычно это просто идентификатор) и имя класса (в терминологии языка Object Pascal — имя типа данных). Она возвращает значение true, если параметр-ссылка является объектом данного типа, и false в противном случае. Например, задан класс животных Animal. Мы хотим определить, является ли переменная fido, относящаяся к типу Animal, экземпляром более узкого подкласса млекопитающих Mammal с помощью следующего теста:

```
if Member (fido, Mammal) then
  writeln ('fido is a mammal')
else
  writeln ('fido is not a mammal');
```

Отметим, что функция Member возвращает значение true, даже если fido является представителем еще более узкого класса (такого, как класс собак Dog). Вкупе с приведением типа данных функция Member может быть использована для частичного решения проблемы обращения полиморфизма. Когда шар вынут из ящика, программист может использовать функцию Member для того, чтобы определить, какой это шар — BlackBall или WhiteBall, и соответственно привести его к правильному типу.

В языке программирования Delphi Pascal класс объекта может быть протестирован с помощью оператора is. Он возвращает значение true, когда класс левого аргумента или совпадает с именем класса справа, или же является его подклассом. Оператор as осуществляет безопасное приведение динамического типа данных. Если левый аргумент не является экземпляром правого класса, возникает исключительная ситуация. В противном случае он приводится к типу правого аргумента. Эти механизмы могут использоваться при работе с обращением полиморфизма.

```
if aBall is BlackBall then
  bBall := aBall as BlackBall
else
  writeln('cannot convert ball to black ball');
```

Если мы хотим узнать истинный класс объекта (то есть действительное имя класса, а не с точностью до наследования), мы можем использовать поле ClassInfo объекта (каждый объект в языке Delphi Pascal имеет поле ClassInfo).

```
if aBall.ClassInfo = BlackBall then
  ...
```

Хотя язык Object Pascal является языком программирования со статическими типами данных, он всегда использует динамическое связывание для того, чтобы сопоставить пересылаемое сообщение и используемый метод. Таким образом, если метод hasLiveYoung (имеет живое потомство?) определен в классе млекопитающих Mammal и

PDF created with pdfFactory Pro trial version www.pdffactory.com

переопределен для класса утконосов Platypus, поиск метода будет производиться в подклассе Platypus, даже если его получатель объявлен как класс (тип) Mammal.

Методы в языке Object Pascal связываются динамически, но законность пересылки сообщения определяется статическим классом получателя. Только если статический класс понимает пересылаемое сообщение, компилятор будет генерировать код для обработки сообщения. Чтобы проиллюстрировать это, рассмотрим пример. Пусть переменная phyl объявлена как экземпляр класса Animal, но содержит значение типа Mammal. Тогда компилятор будет возражать против сообщения hasLiveYoung, которое, по предположению, определено для млекопитающих Mammal, но не определено для класса всех животных Animal.

```
var
  phyl : Animal;
  newPlat : Platypus;
begin
  new (newPlat);
  phyl := newPlat;
  (*  компилятор обнаружит ошибку в этой команде  *)
  if phyl.hasLiveYoung then
    ...
```

### 10.2.2. Связывание в языке Smalltalk

Smalltalk — это язык с динамическими типами данных. Переменные-экземпляры, как мы видели в предыдущих главах, описываются только именем, а не типом данных. Как и во всех языках с динамическими типами данных, переопределение и обращение полиморфизма не являются проблемой, поскольку любой переменной может быть присвоено любое значение.

Можно запросить класс любого объекта. Все объекты реагируют на сообщение class, возвращая объект, представляющий класс объекта. Таким образом, если fido — переменная, для которой ожидается, что она содержит значение типа Dog, то тест

```
( fido class == Dog ) ifTrue: [ ... ]
```

покажет нам, оправдались ли наши ожидания. Но тест не сработает, если мы подставим класс Mammal вместо класса Dog. Таким образом, в языке Smalltalk (как и в Objective-C) для объектов могут использоваться два теста. Первый, с именем isMemberOf:, берет имя класса в качестве аргумента и эквивалентен только что рассмотренному тесту. Второй, с именем isKindOf:, подобен функции Member в языке Object Pascal — он сообщает, является ли получатель, напрямую или путем наследования, экземпляром класса, передаваемого в качестве аргумента. Таким образом, если переменная fido содержит значение типа Dog, то показанный ниже тест пройдет, а тест с использованием метода isMemberOf: — нет:

```
( fido isKindOf: Mammal ) ifTrue: [ ... ]
```

Использование метода isKindOf считается плохой практикой программирования, так как он жестко связывает код со значением определенного типа. Обычно класс объекта интересует нас меньше, чем вопрос о том, будет ли он понимать определенное сообщение. Другими словами, мы больше интересуемся подтипами, чем подклассами. Таким образом, и класс тюленей Seal, и класс собак Dog могут реализовывать метод bark (лаять). Взяв определенный объект, скажем fido, мы

PDF created with pdfFactory Pro trial version www.pdffactory.com

можем использовать для определения того, ответит ли объект на сообщение bark, следующий метод:

( fido respondsTo: #bark ) ifTrue: [ ... ]

Отметим, что селектор метода представлен как символ (начинается с #).

В языке Smalltalk понятия подкласса и подтипа четко разделены. Так как статические типы недоступны, для подбора методов к сообщениям всегда используется динамическое связывание. Если получатель не понимает конкретного сообщения, выдается диагностика ошибки на этапе выполнения.

Стандартная библиотека языка Smalltalk (называемая standard image) обеспечивает богатый набор контейнерных классов, которые широко используют свойство позднего связывания для этого языка программирования.

### 10.2.3. Связывание в языке Objective-C

Objective-C является в своей основе языком с динамическими типами данных. Это неудивительно, учитывая мнение его создателя Бреда Кокса, приводимое в конце этой главы. Большей частью все, сказанное для языка Smalltalk о связывании и поиске методов, подходящих под пересылаемое сообщение, справедливо также и для Objective-C. Это включает и тот факт, что все объекты понимают сообщения class, isKindOf: и isMemberOf:. Когда получатель является объектом с динамическим типом данных, всегда используется динамическое связывание.

Для того чтобы определить, понимает ли данный объект конкретное сообщение, мы можем использовать системную процедуру @selector для преобразования текстового формата селектора сообщения во внутреннее представление, как это показано в следующем примере:

if ( [ fido respondsTo: @selector(bark) ] ) { ... }

Интересной особенностью языка Objective-C является возможность использовать переменные статического типа в комбинации с величинами, обладающими динамическим типом данных. В противоположность тому что было сказано в

главе 3, объекты могут быть описаны с указанием статического типа, причем двумя способами. При наличии определенного класса (например, Mammal) объявление переменной

Mammal anAnimal;

создает новый идентификатор с именем anAnimal и отводит для него место в памяти. Как и в языке C++, последующие пересылки сообщений будут основываться на статическом типе данных (Mammal), и такие значения не поддерживают полиморфное присваивание. То есть если вы попытаетесь присвоить такой переменной в качестве значение экземпляр подкласса (например, Dog), то оно по-прежнему будет рассматриваться как Mammal.

Другой путь — это описание, которое задается в терминах явных указателей. Оно не отводит пространство для объекта, и полиморфное присваивание в этом случае возможно. Память выделяется для таких значений с помощью сообщения new, как

PDF created with pdfFactory Pro trial version www.pdffactory.com

это делается для обычных динамических переменных. Однако когда в качестве получателя используются статические имена классов, то связывание методов, хотя оно по-прежнему является динамическим, выполняется чуть быстрее.

```
Mammal *fido;
fido = [ Dog new ];
```

Для языка Objective-C относительно легко писать контейнерные классы благодаря позднему связыванию. На самом деле большой набор таких структур данных имеется в стандартной библиотеке Objective-C.

### 10.2.4. Связывание в языке C++

Двумя основными целями при разработке языка программирования C++ были эффективное использование памяти и скорость выполнения [Ellis 1990, Stroustrup 1994]. Он был задуман как усовершенствование языка C, в частности, для объектно-ориентированных приложений. Основной принцип C++: никакое свойство языка не должно приводить к возникновению дополнительных издержек (как по памяти, так и по скорости), если данное свойство программистом не используется. Например, если вся объектная ориентированность C++ игнорируется, то оставшаяся часть должна работать так же быстро, как и традиционный C. Поэтому неудивительно что большинство методов в C++ связываются статически (во время компиляции), а не динамически (во время выполнения).

Как было отмечено в главе 7, язык C++ не поддерживает принцип подстановки за исключением использования указателей и значений-ссылок. Мы увидим причину этого в главе 12, где будем исследовать управление памятью в C++.

Связывание методов в этом языке является довольно сложным. Для обычных переменных (не указателей или ссылок) оно осуществляется статически. Но когда объекты обозначаются с помощью указателей или ссылок, используется динамическое связывание. В последнем случае решение о выборе метода статического или динамического типа диктуется тем, описан ли соответствующий метод с помощью ключевого слова virtual. Если он объявлен именно так, то метод поиска сообщения базируется на динамическом классе, если нет — на статическом. Даже в тех случаях, когда используется динамическое связывание, законность любого запроса определяется компилятором на основе статического класса получателя.

Рассмотрим, например, следующее описание классов и глобальных переменных:

```
class Mammal
{
  public:
    void speak()
    {
       printf("can't speak");
    }
};
class Dog : public Mammal
{
  public:
    void speak()
    {
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
      printf("wouf wouf");
    }
    void bark()
    {
      printf("wouf wouf, as well");
    }
};
Mammal fred;
Dog  lassie;
Mammal * fido = new Dog;
```

Выражение fred.speak() печатает «can't speak». lassie.speak() выдаст нам собачий лай. Однако вызов fido->speak() также напечатает «can't speak», поскольку соответствующий метод в классе Mammal не объявлен как виртуальный. Выражение fido->bark() не допускается компилятором, даже если мы знаем, что динамический тип для fido — класс Dog. Тем не менее статический тип переменной — всего лишь класс Mammal, а млекопитающие обычно не лают.

Если мы добавим слово virtual:

```
class Mammal
{
  public:
    virtual void speak()
    {
      printf("can't speak");
    }
};
```

то получим на выходе для выражения fido->speak() ожидаемый результат (а именно, fido будет лаять).

Относительно недавнее изменение в языке C++ — добавление средств для распознавания динамического класса объекта. Они образуют систему RTTI (Run-Time Type Identification — идентификация типа во время выполнения).

В системе RTTI каждый класс имеет связанную с ним структуру типа typeinfo, которая кодирует различную информацию о классе. Поле данных name — одно из полей данных этой структуры — содержит имя класса в виде текстовой строки. Функция typeid может использоваться для анализа информации о типе данных. Следовательно, следующая ниже команда будет печатать строку «Dog» — динамический тип данных для fido. В этом примере необходимо разыменовывать переменную-указатель fido, чтобы аргумент был значением, на которое ссылается указатель, а не самим указателем:

```
cout << "fido is a " << typeid(*fido).name() << endl;
```

Можно также спросить, используя функцию-член before, соответствует ли одна структура с информацией о типе данных подклассу класса, соотносящегося с другой структурой. Например, следующие два оператора выдают true и false:

```
if (typeid(*fido).before(typeid(fred))) ...
if (typeid(fred).before(typeid(lassie))) ...
```

До появления системы RTTI стандартный программистский трюк состоял в том, чтобы явным образом закодировать в иерархии класса методы «быть

PDF created with pdfFactory Pro trial version www.pdffactory.com

экземпляром». Например, для проверки значения переменных типа Animal на принадлежность к типу Cat или к типу Dog можно было бы определить следующую систему методов:

```
class Mammal
{
  public:
    virtual int isaDog()
    {
      return 0;
    }
    virtual int isaCat()
    {
      return 0;
    }
};
class Dog : public Mammal
{
  public:
    virtual int isaDog()
    {
      return 1;
    }
};
class Cat : public Mammal
{
  public:
    virtual int isaCat()
    {
      return 1;
    }
};
Mammal *fido;
```

Теперь для определения того, является ли текущим значением переменной fido величина типа Dog, можно использовать команду fido->isaDog(). Если возвращается ненулевое значение, то можно привести тип переменной к нужному типу данных.

Возвращая указатель, а не целое число, мы объединяем проверку на принадлежность к подклассу и приведение типа. Это аналогично другой части системы RTTI, называемой dynamic_cast, которую мы вкратце опишем. Если некая функция в классе Mammal возвращает указатель на Dog, то класс Dog должен быть предварительно описан. Результатом присваивания является либо нулевой указатель, либо правильная ссылка на класс Dog. Итак, проверка результата все еще должна осуществляться, но мы исключаем необходимость приведения типа. Это показано в следующем примере:

```
class Dog;   // предварительное описание
class Cat;
class Mammal
{
  public:
    virtual Dog* isaDog()
    {
      return 0;
    }
    virtual Cat* isaCat()
    {
      return 0;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    }
};
class Dog : public Mammal
{
  public:
    virtual Dog* isaDog()
    {
      return this;
    }
};
class Cat : public Mammal
{
  public:
    virtual Cat* isaCat()
    {
      return this;
    }
};
Mammal *fido;
Dog  *lassie;
```

Оператор lassie = fido->isaDog(); теперь выполним всегда. В результате переменная lassie получает ненулевое значение, только если fido имеет динамический класс Dog. Если fido не принадлежит Dog, то переменной lassie будет присвоен нулевой указатель.

```
lassie = fido->isaDog();
if (lassie)
{
  ... // fido и в самом деле относится к типу Dog
};
else
{
  ... // присваивание не сработало
  ... // fido не принадлежит к типу Dog
};
```

Хотя программист и может использовать этот метод для обращения полиморфизма, недостаток такого способа состоит в том, что требуется добавление методов как в родительский, так и в дочерний классы. Если из одного общего родительского класса проистекает много дочерних, метод становится громоздким. Если изменения в родительском классе не допускаются, такая техника вообще невозможна.

Поскольку подобные проблемы встречаются часто, было найдено их общее решение. Функция шаблона dynamic_cast берет тип в качестве аргумента шаблона и, в точности как функция, определенная выше, возвращает либо значение аргумента (если приведение типа законно), либо нулевое значение (если приведение типа неразрешено). Присваивание, эквивалентное сделанному в предыдущем примере, может быть записано так:

```
// конвертировать только в том случае, если fido является собакой
lassie = dynamic_cast (fido);
// затем проверить, выполнено ли приведение
if (lassie) . . .
```

В язык C++ были добавлены еще три типа приведения (static_cast, const_cast и reinterpret_cast), но они используются в особых случаях и поэтому здесь не

PDF created with pdfFactory Pro trial version www.pdffactory.com

описываются. Программистам рекомендуется применять их как более безопасные средства вместо прежнего механизма приведения типов.

### 10.2.5. Связывание в языке Java

Хотя внешне язык Java похож на C++, его способы связывания сообщений и поиска подходящих методов существенно проще, чем в C++. В Java все переменные знают свой динамический тип данных. Предполагается, что подклассы являются подтипами, и поэтому значение может быть присвоено типу родительского класса без явного преобразования. Обратное присваивание (обращение полиморфизма) допускается с явным приведением типа. Для того чтобы определить допустимость присваивания, во время выполнения программы производится проверка и, если присваивание недопустимо, индуцируется исключительная ситуация.

```
Ball aBallValue;
  // ... пропущенный код
  // выполнить присваивание через обращение полиморфизма
BlackBall bball = (BlackBall) aBallValue;
```

Также можно проверять динамический тип значения, используя оператор instanceOf.

```
if (aBallValue instanceOf BlackBall)
  ...
else
  ...
```

Сообщения всегда связываются с методами на основе динамического типа данных получателя. В отличие от C++ ключевые слова virtual отсутствуют. Интересная особенность языка Java состоит в том, что поля данных также могут переопределяться, но в этом случае доступ к переменной основывается на статическом типе данных, а не на динамическом. Оба случая переопределения иллюстрируются следующим примером:

```
class A
{
  String name = "class A";
  public void print()
  {
    println("class A");
  }
}
class B extends A
{
  String name = "class B";
  public void print()
  {
    println("class B");
  }
}
class test
{
  public void test()
  {
         Class B b = new B();
  Class A a = b;
         println(a);    // печатает "класс A"
         println(b);    // печатает "класс B"
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  a.print();      // печатает "класс B", а не "класс A"
  b.print();      // печатает "класс B"
    }
  }
```

Хотя язык Java смешивает понятия подкласса и подтипа, предполагая, что все подклассы являются подтипами, но из всех рассматриваемых нами строго типизированных языков Java наиболее четко разделяет эти концепции, предлагая понятие интерфейса.

Интерфейсы обеспечивают иерархическую организацию, подобную классам, но не зависят от последних. Как отмечалось в главе 7, интерфейсы определяют только протокол выполнения операций, но не их реализацию. Используя ключевое слово extends, можно строить новые интерфейсы поверх существующих, как это делается для классов. Таким образом создается иерархия подтипов, полностью независимая от иерархии подклассов.

Интерфейсы могут использоваться для определения переменных, которые будут принимать значения любых типов, заявляющих, что они реализуют интерфейс. Тем самым статический тип — это тип интерфейса, тогда как динамический тип — это тип класса. Связывание основывается на динамическом типе.

Например, при проектировании на языке Java системы итератора для структур данных (см. главу 16, где обсуждаются итераторы) можно представить себе иерархию итераторов: однонаправленные, двунаправленные, с произвольным доступом. Они определены следующими протоколами:

```
interface forwardIterator
{
  void advance();
  int currentValue();
}
interface bidirectionalIterator extends forwardIterator
{
  void moveBack();
}
interface randomAccessIterator extends
  bidirectionalIterator
{
  int at(int);
}
```

Структуры данных могут затем определить нужный итератор. Это задается реализацией.

```
class listIterator implements forwardIterator
{
  // ...
}
```

## 10.3. Как связывать: статически или динамически?

Споры по поводу относительных достоинств статических и динамических типов данных, а также статического и динамического связывания неизменно сводятся к противопоставлению эффективности и гибкости. Формулируя вкратце, статические

PDF created with pdfFactory Pro trial version www.pdffactory.com

типы данных и статическое связывание более эффективны, а динамические типы данных и динамическое связывание более гибки.

Как мы видели, динамические типы данных подразумевают, что каждый объект должен отслеживать свой собственный тип данных. Если мы посмотрим на объекты с точки зрения «непреклонного индивидуалиста» (каждый объект должен сам заботиться о себе и не зависеть от других), то динамические типы данных кажутся более объектно-ориентированной методикой. Их применение сильно упрощает, например, разработку структур данных общего назначения. Но не все так просто — во время выполнения происходит постоянный поиск подходящего кода. Хотя имеются способы уменьшения этих затрат, они не могут быть устранены полностью. В основном именно потому, что такие накладные расходы могут быть весьма значительными, большинство языков программирования использует статические типы данных.

Статические типы данных упрощают реализацию языка, даже если (как в Java, Object Pascal и иногда в C++) используется динамическое связывание метода с сообщением. Когда компилятору известны статические типы данных, выделение памяти под переменные происходит более эффективно, а для простых операций (например, сложение чисел) генерируется почти совершенный код. Но имеются затраты, связанные и со статическими типами данных. К примеру, часто объекты могут терять знание о себе (хотя это не обязательно, как мы видели в случае языка Object Pascal). Как мы отмечали при обсуждении проблемы контейнеров данных, этот факт затрудняет написание абстракций структур данных общего назначения.

Если статические типы данных упрощают реализацию языка программирования, то статическое связывание делает его почти элементарным. Если соответствие между методом и сообщением может быть обнаружено компилятором, то пересылка сообщений (независимо от используемого синтаксиса) представляется просто как вызов процедуры: во время выполнения уже не требуется поиск подходящего метода. Это, по-видимому, максимально эффективно (хотя механизм inline-функций в C++ может устранить даже и код вызова процедуры).

Напротив, динамическое связывание всегда требует некоего run-time-механизма (хотя бы и примитивного) для сопоставления метода и сообщения. В языках программирования, которые используют динамические типы данных (и, как следствие, динамическое связывание), вообще говоря, нельзя определить заранее, будет ли пересылаемое сообщение восприниматься получателем. Когда сообщение не распознается, то не остается другой альтернативы, кроме как сгенерировать сообщение об ошибке этапа выполнения. Защитники статических языков программирования настаивают, что большинство таких ошибок может быть отловлено на этапе компиляции для языка со статическим связыванием методов. Эта точка зрения оспаривается (как и следовало ожидать) сторонниками динамического программирования.

Тем самым нам приходится решать, что более важно: эффективность или гибкость, правильность или легкость использования. Брэд Кокс [Cox 1986] настаивает, что решение зависит как от уровня абстракции программы, так и от того, являемся ли мы производителями или потребителями программной системы. Кокс утверждает, что объектно-ориентированное программирование будет основным (хотя и не единственным) средством в «революции в области программной индустрии». Точно так же как в девятнадцатом веке индустриальная революция стала возможна

PDF created with pdfFactory Pro trial version www.pdffactory.com

только после разработки взаимозаменяемых деталей, так и целью программной революции является конструирование многократно используемых и надежных абстрактных компонент высокого уровня для программного обеспечения, из которых собираются реальные приложения.

Эффективность — главное, о чем следует думать на низком уровне (Кокс называет этот уровень «абстракцией уровня транзисторов» («gate-level abstractions») по аналогии с электроникой). По мере повышения уровня абстракции (до интегральных микросхем и готовых плат) гибкость становится более существенной.

Эффективность программы является первостепенной заботой разработчика. Для потребителя, заинтересованного в комбинировании программных систем, гибкость может быть более важной.

Итак, нет единственно правильного ответа на вопрос, какая из техник связывания является более подходящей. Все зависит от конкретной ситуации.

### Упражнения

- Язык Object Pascal использует и статические типы данных, и динамическое связывание. Объясните, почему обратная ситуация (динамические типы данных и статическое связывание сообщений с методами) невозможна.
- Проиллюстрируйте, почему в объектно-ориентированных языках программирования со статическими типами данных (например, C+ или Object Pascal) компилятор проверяет, чтобы значение переменной родительского класса не присваивалось переменной, описанной как экземпляр подкласса.
- Подумайте о том, стоит ли проверка ошибок, которая становится возможной при статических типах данных, проигрыша в гибкости. Насколько важны контейнерные классы?
- Где не срабатывает аналогия Брэда Кокса между аппаратным и программным обеспечением? Что препятствует разработке переносимых надежных программных компонент?

## *Глава 11: Замещение и уточнение*

До сих пор в нашей интуитивной модели наследования данные и методы дочерних классов оказывались шире, чем в родительском классе. То есть подкласс просто добавлял новые данные или методы к тем, что были унаследованы от надкласса. Однако ясно, что это не всегда имеет место. Помните утконосов Фила и Филлис, описанных в главе 1? В соответствии с нашими рассуждениями млекопитающие — это живородящие. Тем не менее Филлис, стойко требующая отнести ее к законным членам семейства млекопитающих, дает жизнь детенышам, откладывая яйца.

В этой ситуации подкласс утконосов Platypus не просто добавляет что-то к информации о родительском классе млекопитающих Mammal. На самом деле дочерний класс изменяет некоторые свойства родительского класса. Чтобы понять, как подкласс модифицирует смысл метода родительского класса, мы должны исследовать понятие наследования более подробно.

### 11.1. Добавление, замещение и уточнение

PDF created with pdfFactory Pro trial version www.pdffactory.com

Мы предполагали до сих пор, что данные и методы, которые добавляют подклассы, отличаются от унаследованных от родительского класса. Другими словами, методы и значения данных, доопределенные дочерним классом, отличаются от значений и методов родительских (и прародительских) классов. Мы говорим о такой ситуации, как о добавлении к протоколу родительского класса.

Все меняется, когда дочерний класс определяет некоторый метод под тем же самым именем, что и в родительском классе. В результате метод дочернего класса переопределяет метод родительского класса. Например, мои специфические знания о классе утконосов изменяют мои взгляды на класс млекопитающих.

Как описано в главе 1, если сообщение посылается некоторому объекту, то поиск подходящего метода всегда начинается с проверки методов, связанных с классом этого объекта. Если ни один метод не найден, проверяются методы, связанные с ближайшим родительским классом. Если опять-таки ни один метод не найден, проверяется ближайший родительский класс родительского класса и т. д. В конце концов либо больше уже не остается классов (и тогда выдается сообщение об ошибке), либо находится подходящий метод.

Говорят, что метод в классе, имеющий то же имя, что и в надклассе, переопределяет (override) метод надкласса. Во время поиска метода, который вызывается в качестве реакции на сообщение, метод дочернего класса, естественно, будет найден раньше, чем одноименный метод в родительском классе. (По большей части переопределение связано только с методами. Java — единственный из обсуждаемых нами языков программирования (и не только среди них), который допускает ограниченную форму переопределения полей данных.)

### 11.1.1. Американская и скандинавская семантики

Метод дочернего класса может переопределять наследуемый метод одним из двух способов. В случае замещения метод родительского класса замещается целиком во время работы программы. То есть код родительского класса никогда не исполняется при обработке экземпляров дочернего класса. Уточняющий метод включает в себя как часть своих действий вызов метода родительского класса. Таким образом, родительское поведение сохраняется и присоединяется.

Первый тип переопределения часто называют американской семантикой, поскольку он обычно ассоциируется с языками программирования американского происхождения (Smalltatk или C++). Второй известен как скандинавская семантика, так как он чаще всего ассоциируется с языком Simula [Dahl 1966, Birtwistle 1979, Kirkerud 1989], первым объектно-ориентированным языком программирования, и с более поздним языком Beta [Madsen 1993]. Оба языка имеют скандинавское происхождение.

В главе 7 мы описали механизм добавления нового метода в дочерний класс. Остальная часть данной главы посвящена механизму замещения и уточнения в каждом из рассматриваемых нами языков. Мы поговорим также о некоторых достоинствах и недостатках этих двух подходов.

## 11.2. Замещение методов

PDF created with pdfFactory Pro trial version www.pdffactory.com

В языке Smalltalk целые числа и числа с плавающей точкой являются объектами — экземплярами классов Integer и Float соответственно. В свою очередь оба эти класса — подклассы более общего класса Number. Теперь предположим, что у нас есть переменная aNumber, которая содержит в текущий момент значение, относящееся к классу целых чисел, и мы посылаем этой переменной сообщение sqrt, вызывающее вычисление квадратного корня. Метода, соответствующего этому имени, в классе Integer нет, поэтому исследуется класс Number и находится следующий метод:

```
sqrt
        " преобразовать к числам с плавающей точкой "
        " затем вычислить квадратный корень  "
    self asFloat sqrt
```

Этот метод посылает сообщение asFloat переменной self, которая получила сообщение sqrt. Сообщение asFloat возвращает число с плавающей точкой той же величины, что и исходное целое. Затем этому значению посылается сообщение sqrt. Поиск метода начинается с класса Float. Обнаруживается, что класс Float содержит метод с тем же именем sqrt, который для чисел с плавающей точкой переопределяет собой метод в классе Number. Метод класса Float (который здесь не приведен) вычисляет и возвращает квадратный корень.

Переопределение и полное замещение метода sqrt означает, что многие типы чисел могут обладать единственной процедурой по умолчанию, расположенной в классе всех чисел Number. Такое совместное использование позволяет избежать необходимости повторять этот код для каждого подкласса класса Number (который содержит не только целые числа и числа с плавающей точкой, но и целые числа с неограниченной разрядностью, а также рациональные числа). Классы типа Float, которые требуют отличного от установленного по умолчанию поведения, легко могут переопределять метод и подставлять альтернативный код.

### 11.2.1. Замещение методов и принцип подстановки

Основная трудность при использовании замещения методов в качестве фундаментальной модели наследования — предположение, что подклассы являются подтипами. Вспомните, что основой принципа подстановки является то, что представители подкласса ведут себя во всех существенных отношениях подобно представителям родительского класса. Но если подклассы могут переопределять методы, то где гарантии того, что поведение дочернего класса будет иметь хоть какое-то отношение к родительскому классу?

В только что упомянутом примере методы в обоих классах — Float и Number — были связаны только их концептуальными отношениями к абстрактному понятию «квадратный корень». Может возникнуть большой беспорядок, если подкласс поменяет поведение унаследованного метода слишком радикально — например, изменяя sqrt так, чтобы он вычислял логарифм.

В языках программирования можно встретить несколько возможных путей решения конфликта между замещением методов и принципом подстановки:

1. Просто игнорировать эту проблему и предоставить программисту заботиться о том, чтобы подклассы делали правильные вещи во всех важных ситуациях. Этот подход принят на вооружение во всех рассматриваемых нами языках программирования: Object Pascal, C++, Objective-C, Java и Smalltalk.

PDF created with pdfFactory Pro trial version www.pdffactory.com

2. В языке Eiffel [Meyer 1988a, Rist 1995], еще одном хорошо известном объектно-ориентированном языке, программист может присоединить к методу утверждения. Они являются логическими выражениями, которые проверяют состояние объекта во время исполнения, обеспечивая тем самым выполнение определенных условий. Утверждения автоматически наследуются подклассами в прежней форме, даже когда текущие методы переопределяются методами подкласса. Таким образом, утверждения могут использоваться для того, чтобы удостовериться, что дочерний класс ведет себя допустимым образом.
3. Разделить понятия подкласса и подтипа, как это частично сделано в Java. Подклассы затем могут использовать семантику замещения в качестве техники реализации, при этом не обязательно подразумевая, что результирующий класс будет подтипом первоначального класса.
4. Совершенно отбросить семантику замещения, а использовать уточнение. Эта возможность рассматривается в следующем разделе.

### 11.2.2. Уведомление о замещении

Языки программирования используют различные подходы к указанию на то, что некоторый метод переопределяется (неважно как: с замещением или уточнением). Smalltalk, Java и Objective-C вообще не требуют указания на переопределение. В C++ базовый (родительский) класс должен иметь специальные указания о возможности переопределения. В языке Object Pascal версии Apple это указание происходит не в родительском, а в дочернем классе. В языке Delphi Pascal ключевое слово должно быть помещено в оба класса.

Расположение какого-либо указания в родительском классе обычно облегчает реализацию, поскольку если нет возможности переопределить метод, то посылка сообщения реализуется через более эффективный вызов процедуры (то есть динамический поиск во время выполнения не нужен). С другой стороны, снятие необходимости уведомления делает язык более гибким, так как позволяет из любого класса порождать подклассы, даже если автор родительского класса не предусматривал такой возможности.

Например, один программист создает класс (скажем, List) для конкретного приложения. Позднее другой программист захочет построить специализированную форму этого класса (например, OrderedList), переопределяя многие его методы. В таких языках, как C++, это потребует текстовых изменений в исходном классе, чтобы объявить методы как виртуальные (virtual). Напротив, в Java или Objective-C не потребуется никаких изменений в описании родительского класса.

## 11.3. Замещение в разных языках

Ниже кратко характеризуется замещение в различных языках программирования.

### 11.3.1. Замещение в C++

Переопределение методов в языке C++ усложняется взаимным перекрытием переопределения, перегрузки, виртуальных (или полиморфных) функций и конструкторов. Перегрузка и виртуальные функции рассмотрены подробнее в следующих главах. В этом же разделе мы ограничимся простым замещением (листинг 11.1).

**Листинг 11.1**. Описания класса и подкласса в C++

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
class CardPile
{
  public:
    CardPile (int, int);
    card & top();
    void pop();
    bool empty();
    virtual bool includes(int, int);
    virtual void select(int, int);
    virtual void addCard(card &);
    virtual void display(window &);
    virtual void canTake(card &);
  protected:
    Card * firstCard;
    int x;
    int y;
};
class SuitPile : public CardPile
{
  public:
    SuitPile(int, int);
    virtual bool canTake(card &);
};
```

Простое замещение встречается в том случае, когда 1) аргументы метода дочернего класса идентичны по типу и числу аргументов методу родительского класса, 2) для описания метода в родительском классе используется модификатор virtual [1] . Пример: класс CardPile, используемый в пасьянсе из главы 8. Если мы должны перевести пасьянс на C++, то объявления могут выглядеть примерно так, как в листинге 11.1, где представлено описание класса CardPile и его подкласса (в данном случае SuitPile).

Метод canTake описан таким образом, что он переопределяет метод родительского класса. Последний «говорит только нет» — он всегда возвращает значение «ложь» в ответ на запрос, можно ли положить в стопку новую карту:

```
bool CardPile::canTake (card & c) {
  // всегда «нет»
  return false;
}
```

Напротив, этот метод в классе SuitPile дает значение «истина», если колода пуста и карта является тузом или если карта подходит по масти верхней карте колоды и на единицу больше ее по рангу:

```
bool SuitPile::canTake (card & c)
{       // можно добавить к стопке туза
        if (empty ())
          return c.rank() == 0;
        card & topcard = top()
        // иначе должно быть совпадение по масти
        // и карта должна быть следующей по старшинству
 if ((c.suit() == topcard.suit()) &&
```

[1] За одним исключением: когда тип возвращаемого значения для функции, описанной в дочернем классе, не совпадает с типом возвращаемого значения метода родительского класса. См. раздел 12.3.1 следующей главы.

```
    (c.rank() == 1 + topcard.rank()))
      return true;
  return false;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
}
```

Для этой методики совершенно необходимо, чтобы метод дочернего класса полностью перекрывал метод родителя.

Для компилятора языка C++ есть тонкий смысловой нюанс в том, объявлен ли метод canTake виртуальным и в классе CardPile тоже. Оба варианта имеют право на существование. Чтобы метод работал в соответствии с тем, что мы называем объектно-ориентированным стилем, он должен быть объявлен как виртуальный. Модификатор virtual необязателен при описании в дочернем классе. Как только метод объявляется виртуальным, он остается виртуальным и во всех подклассах. Однако для целей документирования этот модификатор обычно повторяется во всех производных классах.

Если модификатор virtual не задан, метод по-прежнему будет замещать одноименный метод родительского класса. Однако процесс связывания метода и сообщения будет происходить по-другому. Невиртуальные методы являются статическими в смысле, описанном в главе 10. То есть связывание вызова невиртуального метода будет выполняться во время компиляции, исходя из объявленного (статического) типа получателя, а не во время выполнения программы, исходя из динамического типа получателя. Если ключевое слово virtual удалено из описания метода canTake, то переменные, объявленные как SuitPile, будут выполнять метод из класса SuitPile, а переменные, объявленные как CardPile, будут выполнять метод по умолчанию независимо от действительного значения переменной.

Еще один усложняющий фактор при переопределении функций в языке C++ — это взаимодействие между перегрузкой и переопределением. Мы будем обсуждать это в главе 17. Сложная семантика замещения методов в языке C++ находится в согласии с его философией, которая состоит в том, что свойства языка должны приводить к дополнительным накладным расходам во время выполнения программы только в том случае, когда эти свойства используются. Если виртуальные функции не применяются, то наследование не навязывает

абсолютно никаких издержек времени выполнения. Это не так для других рассматриваемых нами языков программирования. Однако тот факт, что синтаксисом языка дозволены как виртуальные, так и невиртуальные формы записи, причем это приводит к различиям в интерпретации, часто является источником трудноуловимых ошибок в программах на языке C++.

Конструкторы в C++ всегда используют семантику уточнения, а не замещения. Мы обсудим это после того, как введем понятие уточнения методов.

### 11.3.2. Замещение методов в Object Pascal

В языке Object Pascal версии фирмы Apple метод может замещать метод надкласса, только если:

- имя метода пишется точно так же, как в родительском классе;
- порядок, типы, имена параметров и тип результата функций точно совпадают;
- описание метода в дочернем классе следует за ключевым словом override.

Листинг 11.2 иллюстрирует замещение метода. Здесь класс Employee — общее описание служащих фирмы, а классы SalaryEmployee и HourlyEmployee — два его подкласса.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Функция computePay в классе Employee вычисляет зарплату за данный период. Этот метод переопределяется для подклассов, поскольку вычисления, используемые для двух типов служащих, различны.

Рассмотрим переменную emp, объявленную как экземпляр класса Employee. Как мы уже отмечали, она может иметь значение либо класса SalaryEmployee, либо класса HourlyEmployee (или любого другого типа служащих). Независимо от ее значения обращение к процедуре computePay приведет к вызову нужного метода для соответствующего типа служащих.

Синтаксис, используемый языком Delphi Pascal фирмы Borland, гораздо ближе к C++. В языке фирмы Borland метод, переопределяемый в подклассах, должен следовать за ключевым словом virtual в родительском классе, как это показано в листинге 11.3.

**Листинг 11.2.** Замещение метода в языке Object Pascal фирмы Apple

```
type
  Employee = objet
    name : alpha;
    function computePay : integer;
    function hourlyWorker : boolean;
    procedure create;
  end;
  SalaryEmployee = object (Employee)
    salary : integer;
function computePay : integer; override;
    function hourlyWorker : boolean; override;
    procedure create; override;
  end;
  HourlyEmployee = object (Employee)
    wage : integer;
    hourworked : integer;
    function computePay : integer; override;
    function hourlyWorker : boolean; override;
    procedure create; override;
  end;
function Employee.computePay : integer;
begin
  return 0; (* будет переопределяться подклассами *)
end;
function HourlyEmployee.computePay : integer;
begin
  return hoursworked * wage;
      (* зарплата равна числу часов, умноженному на ставку *)
end;
function SalaryEmployee.computePay : integer;
begin
  return salary div 12;
    (* делим установленную зарплату в год на число месяцев *)
end;
```

**Листинг 11.3**. Уведомление о замещении в Delphi Pascal

```
type
  Employee = class (TObject)
    name : string;
    function computePay : integer; virtual;
    function hourlyWorker : boolean; virtual;
    constructor create; virtual;
  end;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
SalaryEmployee = class (Employee)
  salary : integer;
  function computePay : integer; override;
  function hourlyWorker : boolean; override;
  constructor create; override;
end;
```

### 11.3.3. Замещение в Smalltalk и Objective-C

В языках Smalltalk и Objective-C метод, имеющий то же самое имя, что и метод родительского класса, всегда переопределяет и полностью заменяет собой метод класса-предка. Пользователю нет необходимости явно указывать, что какой-то метод переопределяется. Однако при хорошем документировании программисту полезно отметить этот факт в комментарии.

### 11.3.4. Замещение в Java

В языке программирования Java, как и в Smalltalk и Objective-C, нет необходимости уведомлять о переопределении метода. Достаточно того, что новый метод имеет то же самое имя, список аргументов и тип возвращаемого значения, что и метод родительского класса.

Как отмечалось в главе 10, в языке Java возможно даже замещение полей данных (чего нет ни в одном другом из рассматриваемых нами языков). Однако такие замещения не являются динамическими, и при использовании поля данных его тип будет определяться объявленным (статическим) классом переменной, а не ее динамическим значением. Эта ситуация иногда описывается как маскировка в дочернем классе переменной родителя. Пример был приведен в главе 10.

Интересное свойство Java — ключевое слово final. Если оно применяется к имени метода, то определяет, что метод является «листом» («терминатором») в иерархическом дереве класса и не может быть в дальнейшем переопределен каким бы то ни было способом. Если это ключевое слово встречается в определении класса, то это означает, что из класса не могут порождаться подклассы. Компилятору языка Java позволено оптимизировать методы, описанные с помощью ключевого слова final, превращая их в inline-функцию и подставляя ее явный код в точку вызова (что напоминает язык Beta, см. раздел 11.4.1).

## 11.4. Уточнение методов

Ранее мы отмечали дилемму между переопределением с замещением и сохранением свойств класса в подтипе. Один из способов смягчения этой проблемы — изменение семантики переопределения. Вместо того чтобы полностью заменять код родительского класса, действия дочернего класса комбинируются с действиями родителя. Тем самым гарантируется, что действия родительского класса будут отрабатываться во всех случаях (этим обеспечивается минимальный уровень функциональности). Желательность такого поведения наиболее часто проявляется при инициализации нового объекта. В этом случае мы хотим осуществить действия по инициализации, определенные для родительского класса, и затем уже любые другие инициализирующие действия, которые могут понадобиться для дочернего класса.

Так как в большинстве объектно-ориентированных языков дочерним классом наследуется доступ и к данным, и к методам, добавление новых функций может быть достигнуто

PDF created with pdfFactory Pro trial version www.pdffactory.com

просто копированием кода из родительского класса. Но этот подход нарушает некоторые важные принципы хорошего стиля программирования. Например, он снижает уровень совместного использования кода, препятствует маскировке информации о родительском классе и уменьшает надежность, так как в процессе копирования могут появляться ошибки, и, кроме того, исправление ошибок в родительском классе может не распространиться на потомков.

По этой причине полезно иметь некоторый механизм внутри переопределяемого метода, который бы вызывал метод-предшественник родительского класса и таким образом «повторно» использовал бы код переопределяемого метода. Когда метод вызывает таким способом переопределяемый метод родительского класса, мы будем говорить, что новый метод уточняет поведение родительского класса.

### 11.4.1. Уточнение в языках Simula и Beta

Семантика уточнения появилась в самом первом объектно-ориентированном языке — Simula, который был разработан в начале 1960 года [Dahl 1966]. В языке Simula инициализация вновь создаваемого объекта определялась блоком команд, присоединенным к определению класса, как показано ниже:

```
class Employee
begin
  integer identificationNumber;
               comment описание класса опущено;
               comment операторы, представленные здесь,
               comment выполняют инициализацию
               comment каждого вновь создаваемого объекта;
  identificationNumber :=
    prompt_for_integer("Enter idNumber: ");
  inner;
end;
```

Этот блок инициализации выполняется каждый раз при создании экземпляра класса Employee. Ключевое слово inner в блоке инициализации определяет точку вставки, где выполняются дополнительные действия подкласса. Например, представьте себе, что мы строим подкласс класса Employee, который представляет работающих с почасовой оплатой. Для этого класса также может быть задан блок инициализации. Когда создается какой-либо экземпляр класса HourlyEmployee, сначала выполняется блок инициализации из родительского класса Employee. При достижении команды inner вызывается блок инициализации класса HourlyEmployee. Этот блок может в свою очередь содержать собственный оператор inner, который запускает блоки инициализации подклассов следующего уровня вложенности и т. д. Если нет никаких подклассов, оператор inner не делает ничего:

```
Employee class HourlyEmployee
begin
  integer hourlyWage;
    ... comment описание класса опущено;
  hourlyWage :=
    prompt_for_integer("Enter wage: ");
  inner;
end;
```

Язык программирования Simula использует уточнение и ключевое слово inner только во время инициализации новых объектов. Переопределение обычных методов производится

PDF created with pdfFactory Pro trial version www.pdffactory.com

с помощью семантики замещения. Языку программирования Beta [Madsen 1993] осталось только систематически применить семантику уточнения ко всем методам посредством унификации описаний классов, функций и методов в единую конструкцию, называемую схемой (pattern) (не путайте со схемами разработки классов (design patterns), которые мы будем обсуждать в главе 18).

Чтобы проиллюстрировать стиль языка Beta в отношении схем и уточнения методов, рассмотрим сначала пример, который использует простые функции, а не классы и методы. Предположим, мы хотим задать действия для печати HTML-тэгов для WWW-адресов. Метка-тэг состоит из некоторого начального текста, за которым следует поле URL (адрес машины и полное имя файла), а за ним идет некоторый заключительный текст. Полный HTML-тэг может выглядеть так:

```
<A HREF="http://www.cs.orst.edu/~budd/oop.html">
```

В языке Beta мы можем достигнуть этого, используя сначала функцию, которая печатает только начальный и конечный текст, а не реальный WWW-адрес:

```
printAnchor:
  (#
    do
       '<A HREF="http:'->puttext; INNER
       '">'->puttext
  #);
```

Команда puttext осуществляет текстовый вывод. Три оператора приведенной выше функции создают начальный текст, за которым следует уточнение (если оно есть), а потом — конечный текст.

Вторая функция может специализировать действие первой путем ограничения Web-адресов до конкретной области — например, до адресов, соответствующих Web-серверу в университете штата Орегон. Это достигается определением новой функции, уточняющей первую. Новая функция задает действия, которые должны быть выполнены вместо оператора INNER в исходной функции:

```
printOSUAnchor : printAnchor
  (#
    do
      '//www.cs.orst.edu/'->puttext;
      INNER
  #);
```

Когда мы вызываем эту функцию, код в родительской функции (printAnchor) выполняется первым, так как мы определили, что новая функция является уточнением первой. Во время выполнения оператора INNER в родительской функции происходит вызов кода новой функции. В данном случае этот код печатает конкретный Web-адрес, а затем выполняет любые дальнейшие уточняющие действия.

Уточнение может быть расширено на любую глубину. Например, третья функция может конструировать Web-адрес для отдельного индивидуума:

```
printBuddAnchor : printOSUAnchor
  (#
    do
      '~budd/'->puttext;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    INNER
#);
```

В результате выполнения функции printBuddAnchor получаем текст:

```
<A HREF="http://www.cs.orst.edu/~budd/">
```

В языке Beta уточнения могут быть даже встраиваемыми. В результате выполнения оператора

printBuddAnchor(# do 'oop.html/'->puttext #)

получаем на выходе текст:

&ltA HREF="http://www.cs.orst.edu/~budd/oop.html/">

При описании класса эффект, который мы описываем как уточнение, достигается комбинированием уточнения функции и виртуальных методов (называемых в языке Beta virtual pattern declarations). Как и в предыдущем нашем примере, мы создаем класс Employee, содержащий среди прочих элементов уникальный идентификационный номер служащего и функцию для отображения информации о служащем:

```
Employee :
  (#
    identificationNumber : @integer;
    display:<
      (#
        do 'Employee Number: '->puttext;
        identificationNumber->printInteger;
        INNER
      #);
  #);
```

Подкласс, подобный приведенному ниже подклассу HourlyEmployee, расширяет родительский класс. Определенные внутри подкласса виртуальные функции расширяют методы родительского класса с помощью уже рассмотренного нами механизма. В данном случае вызов метода display приводит сперва к выполнению функции родительского класса, а затем, в точке вхождения оператора INNER, происходит обращение к коду дочернего класса:

```
HourlyEmployee : Employee
  (#
    wage : @integer;
    display::<
      (#
        do
          ' wage: '->puttext;
          wage->printInteger;
          INNER
      #)
  #);
```

Подчиняясь принципу подстановки, экземпляр подкласса HourlyEmployee можно подставить вместо экземпляра класса Employee. Если над первым выполнить

PDF created with pdfFactory Pro trial version www.pdffactory.com

действие display, то сперва вызовется метод родительского класса. При достижении команды INNER будут вставлены команды из функции display дочернего класса.

Систематическое использование уточнения для переопределения методов является концептуально элегантным и делает почти невозможным написание подкласса, не являющегося подтипом. С другой стороны, многие трюки, полезные при семантике замены (вроде описанной ранее реализации функции sqrt в языке Smalltalk), трудно смоделировать в таких языках программирования, как Beta. Скорее всего, только историческая случайность (возможно, вкупе с тем фактом, что уточнение несколько сложнее в реализации, чем замещение) объясняет доминирование языков, использующих замещение методов.

### 11.4.2. Методы-оболочки в языке CLOS

Еще одна интересная вариация на тему семантики уточнения встречается в языке программирования CLOS [Keen 1989] — диалекте Lisp. В языке CLOS подкласс может переопределять метод в родительском классе и вводить метод-оболочку (wrapping method). Метод-оболочка может быть методом-до, методом-после или методом-внутри. В соответствии со своим типом, до метода, после метода или внутри метода, выполняется вызов метода родительского класса. В последнем случае специальный оператор дочернего метода call-next-method вызывает метод родительского класса. Это напоминает способ, которым моделируется уточнение в таких языках программирования, как C++ и Object Pascal.

## 11.5. Уточнение в разных языках

Среди рассматриваемых нами языков программирования только C++ использует семантику уточнения, да и то в основном для конструкторов. (C++ следует в этом отношении языку Simula, где семантика уточнения используется только во время инициализации.) Однако во всех языках эффект уточнения может быть смоделирован с помощью других механизмов. Как достичь этого, будет описано в следующих разделах.

### 11.5.1. Уточнение в Object Pascal

Уточнение в языке программирования Object Pascal осуществляется методом дочернего класса, который явным образом вызывает переопределяемый метод родительского класса. Это подход более или менее противоположен принятому в языках Simula или Beta, где метод родительского класса сам вызывает метод из дочернего класса. Однако в большинстве случаев эффект одинаков — оба метода будут выполнены.

Ключевое слово inherited используется внутри дочернего класса, чтобы отметить точку внутри дочернего метода, в которой должен быть вызван метод родительского класса. Предположим, например, что мы хотим написать метод с именем initialize, который будет запрашивать у пользователя значения, инициализирующие поля данных объекта. Этот метод для родительского класса Employee мог бы выглядеть следующим образом:

```
procedure Employee.initialize;
begin
  writeln("enter employee name: ");
  readln(name);
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
end;
```

Подкласс SalaryEmployee может присоединять инициализацию полей данных родительского класса к своему коду инициализации следующим образом:

```
procedure SalaryEmployee.initialize;
begin
  inherited initialize;
  writeln("enter salary: ");
  readln(salary);
end;
```

В языке Delphi Pascal ключевое слово inherited используется даже в конструкторах. Иногда это полезно, поскольку точка внутри конструктора дочернего класса, где вызывается конструктор родителя, может задаваться программистом.

### 11.5.2. Уточнение в C++

В языке C++ вызов метода может иметь расширенный синтаксис составного имени, при котором вместо используемой по умолчанию процедуры поиска подходящего метода точно указывается, из какого класса должен браться вызываемый метод. Это составное имя записывается как имя класса, за которым следуют два двоеточия и затем имя метода. Использование такого составного имени делает ненужным виртуальный механизм посылки сообщений и поиска адресата и гарантирует, что метод будет вызываться из названного класса.

Механизм составного имени применяется в языке C++ для моделирования уточнения при переопределении. Замещаемый метод явным образом вызывает метод родителя, тем самым гарантируя, что оба метода будут выполнены.

Мы еще раз задействуем пример из программы пасьянса, переписанной на C++. Метод addCard в классе CardPile выполняет основные действия по помещению новой карты наверх стопки. Класс DiscardPile должен дополнительно обеспечивать, чтобы добавляемая карта была открыта. Это делается с помощью следующего метода:

```
void  DiscardPile::addCard (card & newCard)
{
  // убедиться, что новая карта лежит картинкой вверх
  if (! newCard.faceUp())
    newCard.flip();
  // затем добавить ее к колоде
  CardPile::addCard (newCard);
}
```

Ранее мы отмечали, что один из аспектов, в которых конструктор отличается от других методов языка C++, состоит в том, что в дочернем классе он всегда использует уточнение, а не замещение. То есть конструктор в дочернем классе всегда вызывает конструктор родительского класса.

В том случае, если в конструкторе дочернего класса не дается никаких указаний и если родительский класс содержит конструктор, принимаемый по умолчанию (конструктор без аргументов), то он и будет автоматически вызываться. В противном случае должны быть явно заданы аргументы, которые используются конструктором родительского класса. Конструктор родительского класса может

PDF created with pdfFactory Pro trial version www.pdffactory.com

вызываться явным образом в заголовке дочернего конструктора путем указания имени родительского класса и аргументов, которые использует конструктор родителя. Пример: конструктор класса TablePile, который берет два аргумента и использует их для вызова конструктора родительского класса:

```
TablePile::TablePile (int c, int x, int y)
  : CardPile(x, y)
{
  column = c;
}
```

Деструкторы в языке C++ используют как раз противоположный подход. За вызовом деструктора дочернего класса следуют вызовы всех других деструкторов для элементов данных и для родительских классов.

### 11.5.3. Уточнение в Smalltalk, Java и Objective-C

В главе 6 мы столкнулись с псевдопеременной super. Единственная (практически) роль этой переменной в языках Smalltalk, Java и Objective-C — это разрешить уточнение при переопределении метода. Передача сообщения переменной super указывает на то, что поиск соответствующего метода должен начинаться с родителя текущего класса:

```
class A
{
  private int a;
  public initialize()
  {
    a = 3;
  }
}
class B extends A
{
  private int b;
  public initialize()
  {
     b = 7;
     // выполнить метод родительского класса
    super.initialize();
  }
}
```

Конструктор в языке Java вызывает конструктор родительского класса с помощью ключевого слова super:

```
class newClass extends oldClass
{
  newClass (int a, int b, int c)
  {
    // вызвать конструктор родительского класса
   super (a, b);
   // . . .
  }
}
```

С той же самой целью подобные конструкции используются в языках программирования Smalltalk и Objective-C.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### Упражнения

1. Обоснуйте утверждение: если не используется ни замещение, ни уточнение, то подкласс всегда должен быть подтипом.
2. Приведите пример, иллюстрирующий, что если имеет место семантика замещения, то подкласс не обязан быть подтипом.
3. Хотя систематическое использование семантики уточнения делает более сложным создание подклассов, не являющихся подтипами, такое все же возможно. Проиллюстрируйте это, приведя пример уточняющего подкласса, не являющегося тем не менее подтипом базового класса.
4. Часто во время инициализации экземпляра подкласса должны быть выполнены по очереди: некоторый код родительского класса, код дочернего класса, затем снова код родительского класса. Например, для оконных систем родительский класс выделяет память для важных структур данных, затем дочерний класс модифицирует некоторые поля этих структур (такие, как имя и размер окна), и наконец родительский класс отображает окно на экране дисплея. Как данная последовательность вызовов может быть выполнена в объектно-ориентированном языке программирования? Подсказка: вероятно, вам придется разбить процесс инициализации на два сообщения.
5. Не всегда семантика уточнения легко моделируется семантикой замещения. Чтобы продемонстрировать это, напишите набор классов, обеспечивающих выполнение действий, напоминающих подпрограммы создания тэгов WWW-адресов, описанные в разделе 11.4.1. Как и в случае упражнения 4, вам, возможно, понадобится ввести ряд «скрытых» методов.

## *Глава 12: Следствия наследования*

Наследование оказывает большое влияние практически на все аспекты языка программирования. В этой главе мы исследуем некоторые следствия, вытекающие из реализации наследования, рассматривая систему типов данных, смысл операторов (типа оператора присваивания), проверку равенства объектов, выделение памяти.

Мы уже описали отношение «быть экземпляром» как фундаментальное свойство наследования. Одна из точек зрения на отношение «быть экземпляром» — это рассматривать его как средство, связывающее тип данных (в смысле типа переменной) и набор значений (а именно значения, которые могут законным образом содержаться в переменной). Если переменная win описана как экземпляр конкретного класса, скажем Window, то конечно же она может содержать значения типа Window. Если мы имеем подкласс класса Window, например TextWindow, то, поскольку TextWindow «является экземпляром» Window, имеет смысл присвоить переменной win значение типа TextWindow. Это называется принципом подстановки, который мы встречали в предыдущих главах.

В то время как сам принцип имеет интуитивно понятный смысл, с практической точки зрения наличествуют трудности реализации объектно-ориентированных языков таким образом, чтобы это интуитивное поведение могло быть реализовано. Такие трудности не являются непреодолимыми, но способ их решения различается в разных языках. Исследование этих проблем, а также то, как они влияют на язык программирования, выявляет скрытые свойства языка, которые с большой вероятностью доставляют неприятности неосторожным программистам.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## 12.1. Выделение памяти

Начнем с рассмотрения внешне простого вопроса, ответы на который весьма разнятся, а именно: сколько памяти надо выделить переменной, которая описана как принадлежащая конкретному классу? Сколько памяти должно быть выделено переменной win, которая описана как экземпляр класса Window?

Всеми признано, что размещение переменных в стеке при вызове процедуры более эффективно, чем использование «кучи» (см., однако, [Appel 1987]).

Соответственно разработчики языков программирования прилагают максимум усилий, чтобы сделать возможным размещение переменных в стеке. Но при выделении памяти через стек имеется проблема — ее размер должен определяться статически, то есть во время компиляции или по крайней мере во время входа в процедуру. Эти действия осуществляются до того, как становится известно значение, которое будет содержаться в переменной.

Трудность состоит в том, что подклассы могут добавлять данные, не присутствующие в надклассе. Например, класс TextWindow, вероятно, привносит с собой области данных для буфера текста, положения текущей точки редактирования и т. д. Следующее описание может быть типичным примером:

```
class Window
{
        int height;
        int width;
        ...
        public
    virtual void oops();
};
class TextWindow : public Window
{
        char *contents;
        int cursorLocation;
        ...
        public:
    virtual void oops();
};
Window win; // объявлена переменная класса Window
```

Следует ли принимать во внимание дополнительные значения данных (поля contents и cursorLocation) при размещении переменной win? Имеется по крайней мере три правдоподобных способа действий:

1. Выделить память, достаточную только для базового класса. То есть разместить для переменной win исключительно данные, описанные как часть класса Window, игнорируя требования памяти для подкласса.
2. Разместить максимум памяти, достаточной для любого законного значения, независимо от того, принадлежит ли оно базовому классу или одному из подклассов.
3. Разместить память под указатель. Выделять память, необходимую для реального значения, из «кучи» во время выполнения программы (при этом указатель устанавливается надлежащим образом).

PDF created with pdfFactory Pro trial version www.pdffactory.com

Возможны все три решения, и два из них встречаются в языках программирования, которые мы рассматриваем. В последующих разделах мы будем исследовать некоторые последствия этих решений.

### 12.1.1. Размещение минимальной статической памяти

Язык программирования C был разработан в соответствии с требованиями максимальной эффективности при выполнении программы. Тем самым, учитывая распространенное убеждение, что выделение памяти через стек приводит к более быстрому выполнению программы, чем при динамическом размещении, неудивительно, что и его преемник C++ сохраняет концепции нединамических и динамических (размещаемых во время работы программы) переменных.

В языке C++ отслеживается, как именно описывается переменная и соответственно используются ли указатели для доступа к ее полям данных. Ниже, например, переменная win размещается через стек. Пространство для нее будет выделено в стеке при входе в процедуру, где описывается переменная. Память выделяется под размер переменной базового класса. Переменная tWinPtr, с другой стороны, содержит только указатель. Память под значение, на которое указывает tWinPtr, будет выделяться динамически при выполнении оператора new. Поскольку к этому времени размер объектов типа TextWindow уже известен, при выделении из «кучи» памяти, нужной для объекта TextWindow, проблем не возникнет.

```
Window win;
Window *tWinPtr;
...
tWinPtr = new TextWindow;
```

Что происходит, когда значение, на которое указывает переменная tWinPtr, присваивается переменной win? Другими словами, как выполняется оператор

```
win = *tWinPtr;
```

Память, выделенная под переменную win, вмещает только объекты типа Window, в то время как значение, на которое указывает переменная tWinPtr, больше по размеру. Очевидно, что не все значения, на которые указывает tWinPtr, могут быть скопированы. Поведение по умолчанию состоит в том, что копируются только совпадающие поля (рис. 12.1). (В языке C++ пользователь может переопределить смысл оператора присваивания и обеспечить любое желаемое функционирование. Так что здесь мы имеем в виду только стандартное поведение, которое имеет место при отсутствии определяемых пользователем альтернатив.)

Очевидно, что некоторая информация (содержащаяся в дополнительных полях tWinPtr) теряется. Некоторые авторы используют термин срезка (slicing) для этого процесса, поскольку поля данных объекта справа «срезаются» перед присваиванием левому объекту.

Насколько опасна потеря информации? Только в том случае, если пользователь что-то заподозрит. Вопрос: как пользователь сможет заметить отсутствие «лишних» полей?

Семантика языка гарантирует, что для переменной win вызываются только методы, определенные для класса Window, но не методы класса TextWindow.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Методы, определенные и реализованные в классе Window, не могут иметь доступа к полям данных подклассов. Но как насчет методов, определенных в классе Window и переопределяемых в подклассах?

Рассмотрим, к примеру, две процедуры oops, показанные выше. Если пользователь выполняет команду win.oops() и при этом выбирается метод класса TextWindow, то может произойти попытка вывести данные из поля win.cursorLocation, которого не существует в блоке памяти переменной win. Это вызовет либо нарушение доступа к памяти, либо (что более вероятно) приведет к выводу мусора.

```
void Window::oops()
{
  printf("Window oops");
}
void TextWindow::oops()
{
  printf("TextWindow oops %d", cursorLocation);
}
```

Решение этой дилеммы, выбранное разработчиком языка C++, — изменить правила привязки процедуры к вызову виртуального метода. Новые принципы могут быть сформулированы следующим образом:

- Для указателей и ссылок, когда сообщение вызывает функцию-член, которая может в принципе быть переопределенной, выбираемая функция-член определяется динамическим значением получателя.

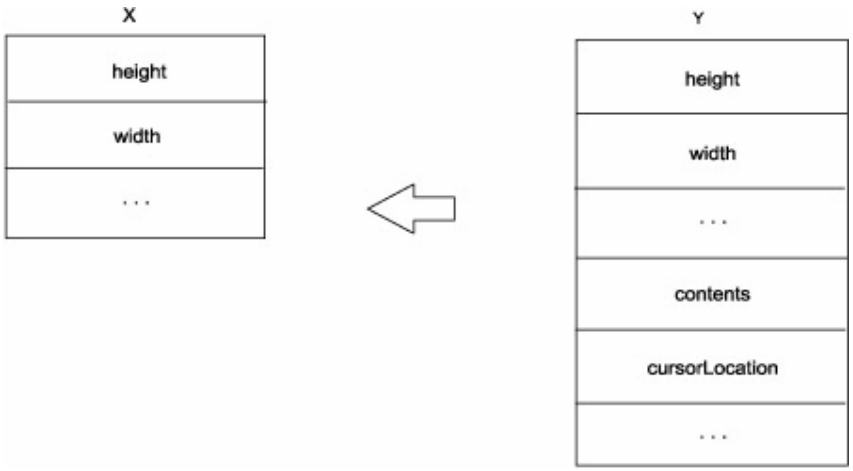



**Рис. 12.1**. Присваивание большого значения маленькому

- Для других переменных связывание вызова виртуальной функции определяется статическим (заданным при описании), а не динамическим классом (то есть фактическим значением).

Более точно, во время процесса присваивания значение меняет тип с подкласса на тип данных родителя. Это аналогично тому, как целочисленное значение может быть изменено при присваивании вещественной переменной. При такой интерпретации можно гарантировать, что для переменных, размещаемых через стек, динамический класс всегда совпадает со статическим. При соблюдении этого правила процедура никогда не получит доступа к полям данных, которые физически отсутствуют в объекте. Метод, выбираемый

PDF created with pdfFactory Pro trial version www.pdffactory.com

при вызове win.oops(), будет принадлежать классу Window, и пользователь не заметит, что часть памяти была потеряна при операции присваивания.

Тем не менее это решение получено за счет некоторой непоследовательности. В выражениях с указателями виртуальные методы связываются так, как мы описывали в предыдущих главах. Поэтому эти значения будут вести себя иным образом, чем выражения, использующие нединамические значения. Рассмотрим следующий пример:

```
Window win;
TextWindow *tWinPtr, *tWin;
...
tWinPtr = new TextWindow;
win = * tWinPtr;
tWin = tWinPtr;
...
win.oops();
(*tWin).oops();
```

Хотя пользователь, вероятно, думает, что переменная win и значение, на которое указывает указатель tWin, — это одно и то же, важно помнить, что присваивание переменной win изменило тип значения. Из-за этого первое обращение к процедуре oops() будет вызывать метод класса Window, в то время как второе — метод класса TextWindow.

### 12.1.2. Размещение максимальной статической памяти

Другое решение проблемы: при описании объекта надо выделить максимальный объем памяти, достаточный для любого значения, которое может содержаться в объекте (независимо от того, относится оно к объявленному классу или к его подклассам). Этот подход аналогичен тому, что используется при размещении перекрывающихся типов данных в традиционных языках программирования (записи с вариантами в языке Паскаль, объединения (union) в C). При присваивании будет невозможно присвоить значение, не умещающееся в памяти, отведенной под переменную в правой части оператора присвоения. Поэтому рис. 12.1 более не актуален и проблемы, вытекающие из него, отныне не возникают.

Это, по-видимому, было бы идеальным решением, если бы не одна маленькая проблема: размер любого объекта не известен, пока не скомпилирована вся программа целиком.

Не просто модуль (unit в языке Object Pascal, файл в C++), но вся программа должна быть отсканирована прежде, чем мы сможем определить максимальный размер подкласса данного класса. Это требование является столь ограничивающим, что ни один из основных объектно-ориентированных языков не использует данный подход.

### 12.1.3. Динамическое выделение памяти

Третий подход вообще не хранит значение объекта в стеке. При входе в процедуру в стеке выделяется память для указателя. Значения содержатся в другой области данных («куче»), которая не поддерживает протокол выделения памяти FIFO («первый вошел — последний ушел»), типичный для стека. Поскольку все указатели имеют постоянный и фиксированный размер, то не возникает проблем при присваивании значения подкласса переменной, объявленной как надкласс.

Этот подход используется в языках Object Pascal, Smalltalk, Java и Objective-C, о чем пользователь уже мог догадаться по сходству объектов и указателей в Object Pascal. Как

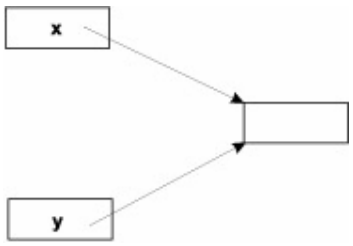
PDF created with pdfFactory Pro trial version www.pdffactory.com

для указателей, так и для объектов необходимо вызывать стандартную процедуру new для размещения памяти перед обращением к объектам. Аналогично пользователь явно вызывает процедуру free для освобождения памяти, выделенной объекту.

Кроме требования явного выделения памяти, при таком подходе имеется еще одна проблема — оператор присваивания тесно связан с семантикой указателей. При использовании указателей при присваивании пересылается указатель на значение, а не собственно значение, обозначаемое указателем. Рассмотрим приведенную ниже программу, моделирующую буфер под одно слово, который устанавливается и опрашивается пользователем:

```
type
  intBuffer = object
    value : integer;
  end;
var
  x, y : intBuffer;
begin
       new(x);  {создать буфер}
       x.value:=5; writeln(x.value);
       y:=x;   {y — тот же буфер, что и x}
       y.value:=7; writeln(x.value);
end;
```

Заметим, что экземплярами этого класса объявлены две переменные. При выполнении программы пользователь, вероятно, удивится, когда последний оператор напечатает значение 7, а не 5. Причина: при присваивании x и y не просто получили одно значение, они стали указывать на одно значение. Эта ситуация показана на рис. 12.2. Семантика указателей для объектов в языке Object Pascal отчасти смущает, поскольку альтернативный подход — семантика копирования — используется для всех других типов данных. Если бы x и y были структурами, то присваивание y:=x привело бы к копированию информации из переменной x в переменную y. Поскольку при этом создаются две различные копии, то дальнейшие изменения в переменной y не влияют на x.




**12.2.** Присваивание

Как в C++, так и в Object Pascal используемые механизмы выделения памяти влияют на смысл операции присваивания. Поэтому здесь мы определим точные значения этого оператора в рассматриваемых языках. Как было отмечено в предыдущем разделе, имеются две интерпретации операции присваивания.

**Семантика копирования**. В операции присваивания полностью копируется значение справа, затем оно присваивается левой части. Следовательно, два значения являются независимыми, и изменение одного из них не влияет на другое.

PDF created with pdfFactory Pro trial version www.pdffactory.com

**Семантика указателей**. Операция присваивания изменяет стоящую слева ссылку так, что она указывает на то же, что и правая часть. (Этот подход иногда называется присваиванием указателей.) Тем самым две переменные не только имеют одно значение, но и указывают на один и тот же объект. Изменения в этом объекте отразятся на значении, получаемом разыменованием любого из двух указателей.

В некоторых языках программирования наблюдается компромисс между семантикой копирования и семантикой указателей, хотя это не относится к языкам, которые мы рассматриваем в этой книге. Идея состоит в использовании семантики указателей при присваивании с последующим преобразованием значения в новую структуру, если оно модифицируется. При таком подходе операция присваивания выполняется очень эффективно и значение не может быть изменено в результате грубой ошибки при введении переменной-синонима. Этот метод часто называется копированием при записи.

В общем случае, если используется семантика указателей, то языки программирования предусматривают некоторые средства создания истинной копии. Опять же, семантика указателей, вообще говоря, используется чаще при размещении объектов через «кучу» (то есть динамически), а не через стек (автоматически). Когда применяется семантика указателей, значения довольно часто «переживают» свой контекст, внутри которого они были созданы.

Отличие объектно-ориентированных языков состоит в использовании разных семантик (первой, второй или их комбинации).

### 12.2.1. Присваивание в C++

Алгоритм, по умолчанию используемый в языке C++ для присваивания переменной значения какого-либо класса, состоит в рекурсивном копировании соответствующих полей данных. Однако разрешается переназначить оператор присваивания с тем, чтобы получить любое желаемое действие. Эта техника является настолько стандартной, что некоторые компиляторы C++ выдают предупреждающее сообщение, если используется правило присваивания по умолчанию.

При переопределении присваивания его интерпретация состоит в следующем. Оператор присваивания — это метод, определенный для класса в левой части, вызываемый с аргументом, стоящим в правой части. Результат может быть равен void, если вложенное присваивание недопустимо, хотя, как правило, в качестве результата передается ссылка на объект в левой части. Следующий пример демонстрирует присваивание строк (здесь оно переопределяется с тем, чтобы две копии одной строки имели общие символы):

```
String & String::operator = (String& right)
{
        len = right.len;              // копировать длину строки
        buffer = right.buffer;        // копировать указатель на строку
        return (*this);
}
```

Типичный источник ошибок начинающего программиста на языке C++ — это использование одного и того же символа равенства (=) для операции присваивания и для операции инициализации. В стандартном C присваивание при объявлении — это просто удобное синтаксическое сокращение. Так что эффект от

```
int limit = 300;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
такой же, что и от
int limit;
limit = 300;
```

В языке C++ присваивание при объявлении может вызывать произвольные конструкторы и не использовать присваивание вообще. Тем самым оператор типа

```
Complex x = 4;
интерпретируется по смыслу как
Complex x(4);
```

При инициализации часто используются ссылки; тем самым ситуация напоминает семантику указателей. Например, если идентификатор s — это объект типа String, то следующая команда делает идентификатор t синонимом идентификатора s (так что изменение в одной переменной приводит к изменению в другой).

```
... // использование переменной s
String &t = s;
... // переменные t и s теперь ссылаются на одно значение
```

Переменные-ссылки наиболее часто применяются для реализации передачи параметров «по ссылке» при вызове процедуры. Это может рассматриваться как разновидность присваивания указателей, где параметру присваивается значение аргумента. Конечно же, семантика указателей в C++ может быть осуществлена через переменные-указатели.

Пересылка параметров может приводить к присваиванию (при присваивании аргументам значений параметров). Поэтому неудивительно, что здесь возникают те же явления, что и при присваивании. Например, рассмотрим описания вида

```
class Base
{
  public:
    virtual void see();
};
class Derived
{
  public:
    virtual void see();
};
void f(Base);
void g(Base &);
Derived z;
f(z); g(z);
```

Обе функции f() и g() используют в качестве аргумента значение класса Base, но функция g() описывает аргумент как переменную-ссылку. Если вызывается функция f() с аргументом, принадлежащим к классу Derived, то как часть вызова процедуры аргумент преобразуется (с использованием срезки), чтобы создать значение, принадлежащее к классу Base. Тем самым если внутри функции f() вызывается метод see, то будет использована виртуальная функция из класса Base. С другой стороны, это преобразование не происходит при передаче параметров в функцию g(). Поэтому если метод see вызывается из функции g(), то будет использована виртуальная функция из класса Derived. Эта разница в интерпретации, которая зависит только от одного символа в заголовке функции, иногда называется проблемой срезки.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Язык C++ позволяет переопределять символ присваивания и выбирать механизм пересылки параметров (по ссылке или по значению). Это мощные средства, но они могут привести к неожиданным последствиям. Например, символ присваивания, используемый при инициализации (хотя это точно такой же знак =), не подвергается изменению при перегрузке оператора присваивания. Хорошее объяснение правильного использования мощного потенциала операции присваивания, заложенного в языке C++, дано в работах [Koenig 1989a, Koenig 1989b].

Копии значений часто создаются исполняющей системой языка C++ в качестве временных значений (при возврате значений функций, при вычислении сложных выражений и т. д.) или аргументов при вызове процедур. Пользователь может управлять этой деятельностью, определяя копирующий конструктор. Аргументом копирующего конструктора является ссылка на параметр того же типа, что и собственно класс. Считается хорошей практикой программирования всегда определять копирующий конструктор.

```
class Complex
{
  ...
  Complex (const Complex &source)
  {
    // просто дублирует поля источника
    rl = source.rl;
    im = source.im;
  }
  ...
  private:
    double rl;
    double im;
}
```

### 12.2.2. Присваивание в Object Pascal и Java

Как Object Pascal, так и Java используют семантику указателей для присваивания объектов. В языке Object Pascal нет предусмотренных системой механизмов для создания копии объекта, поэтому принято определять безаргументный метод copy, который создает копию получателя, если такое функционирование является желательным. В языке Java класс всех объектов Object определяет метод clone, который создает побитную копию получателя сообщения clone. Подклассы могут переопределять этот метод. Тип возвращаемого результата в этом методе определен как Object, так что должно использоваться приведение типа для получения значения нужного типа:

```
newBall = (Ball) aBall.clone();
```

Заметьте, что в языке Object Pascal семантика указателей используется только для объектов. Все другие типы данных (массивы, записи) осуществляют семантику копирования при присваивании. Это часто приводит в смущение начинающего программиста.

### 12.2.3. Присваивание в Smalltalk

Язык Smalltalk использует семантику указателей при присваивании. Класс Object, который является надклассом всех классов, определяет два метода копирования объектов.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Тем самым копирующее присваивание должно использовать комбинацию присваивания и пересылки копирующего сообщения. Оператор

x := y copy

создает новый экземпляр, в точности похожий на объект y, в котором поля (переменные экземпляра) указывают на объекты, являющиеся общими с аналогичными полями объекта y. Напротив, оператор

x := y deepCopy

создает новый объект, аналогичный y, в котором поля (переменные экземпляра) инициализированы копиями полей y.

Другими словами, метод copy (называемый также shallowCopy) делает переменные экземпляра общими с переменными оригинала, в то время как метод deepCopy копирует другие переменные экземпляра
С другой стороны, метод deepCopy создает новые копии переменных экземпляра. Собственно переменные экземпляра создаются с использованием метода copy. Классы имеют право переопределять любой из методов копирования copy, shallowCopy и deepCopy, так что для экземпляров некоторых классов можно получить нестандартное поведение.

### 12.2.4. Присваивание в Objective-C

Язык Objective-C использует семантику указателей для присваивания объектов. Копия объекта может быть получена одним из трех методов: copy, shallowCopy и deepCopy, которые аналогичны одноименным методам, используемым в языке Smalltalk.

```
id x, y, z;
// ... определение объекта y
x = [ y copy ];
z = [ y deepCopy ];
```

## 12.3. Проверка на равенство

Подобно операции присваивания, вопрос о том, является ли один объект эквивалентным другому объекту, является сложнее, чем это может показаться на первый взгляд. Отчасти трудность в понимании того, что в точности значит эквивалентность (идентичность), аналогична проблеме разговорного языка. Если кто-то спрашивает: «Является ли утренняя звезда вечерней звездой?» (Is the morning star the evening star?), ответ с полным основанием может быть как «да», так и «нет». Если вопрос стоит о сравнении физических объектов, обозначаемых этими двумя терминами (а в обоих случаях речь идет о планете Венера), то ответом, безусловно, будет «да». С другой стороны, если спрашивающий хочет узнать, обозначает ли в данном языке термин «утренняя звезда» объекты, появляющиеся на вечернем небе, то ответом столь же безусловно будет «нет».

Изучение ссылок, значений и эквивалентности в естественном языке — дело трудное, и мы не будем углубляться в этот вопрос дальше. Заинтересованный читатель может обратиться к эпизоду с Белым Рыцарем в книге «Алиса в Зазеркалье» или к избранным работам, цитируемым в [Rosenberg 1971] и [Whorf 1956]. К счастью, равенство в языках программирования обычно хорошо формализовано, хотя сделано это по-разному в разных языках.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Наиболее общее разделение отражает различие между семантикой указателей и семантикой копирования. Многие языки используют эквивалентность указателей, когда две ссылки на объект считаются эквивалентными, если они указывают на один и тот же объект. Если мы рассмотрим слова «утренняя звезда» как указатель на Венеру, то при такой интерпретации «утренняя звезда» эквивалентна «вечерней звезде». Эта форма эквивалентности иногда называется эквивалентностью объектов.

Зачастую программист интересуется не столько тем, указывают ли две переменные на идентичный объект, сколько тем, обладают ли два объекта одинаковым значением. Последнее обычно требуется, например, при сравнении текстовых строк (рис. 12.3). Но как должно определяться равенство значений? Для чисел и текстовых строк обычно под равенством понимается побитное совпадение. При такой интерпретации два объекта являются эквивалентными, если их битовое представление в памяти одинаково.

Для составных объектов вроде записей в языках Pascal и C побитное сравнение может оказаться недостаточным. Часто блок памяти для таких типов данных может включать пустые участки, которые не имеют отношения к значениям, хранимым в объекте. Поскольку эти пропуски не должны учитываться при определении равенства, используется второй механизм, а именно поэлементное равенство. При поэлементном сравнении мы проверяем сопоставляемые элементы на совпадение, применяя это правило рекурсивно, пока не встретится элемент, отличный от записи. В последнем случае применяется побитное сравнение. Если все элементы удовлетворяют проверке, две записи рассматриваются как равные друг другу. Если какие-либо два элемента не совпадают, то записи не равны друг другу. Такое отношение равенства иногда называется структурной эквивалентностью.

Техника объектно-ориентированного программирования привносит свои особенности в проверку на равенство. Например, если при сравнении двух значений как статических типов они оказываются равными, то при сравнении их в качестве динамических типов это не обязательно так. Следует ли при проверке на равенство принимать это во внимание? Что если один из типов определяет поля, которые отсутствуют в другом? Проблема состоит также в том, что выбор интерпретации для вызываемого сообщения определяется получателем. Нет гарантии, что такое фундаментальное свойство, как коммутативность, будет сохраняться. Если идентификаторы x и y принадлежат к различным типам данных, то вполне может быть, что отношение x=y справедливо, а отношение y=x — нет!

В одном аспекте, однако, проблему равенства легче решить, чем разобраться с аналогичными трудностями при присваивании. Хотя последнее обычно рассматривается как часть синтаксиса и семантики языка и в силу этого может быть неизменяемым, программист всегда волен создать свои методы проверки на совпадение (возможно, с несколько другим синтаксисом). Тем самым нет единого логического смысла отношения равенства; оно может означать разные вещи для объектов разных классов.

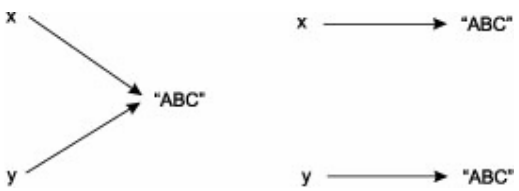



**Рис. 12.3**. Идентичность и равенство текстовых строк

Техника объектно-ориентированного программирования привносит свои особенности в проверку на равенство. Например, если при сравнении двух значений как статических

PDF created with pdfFactory Pro trial version www.pdffactory.com

типов они оказываются равными, то при сравнении их в качестве динамических типов это не обязательно так. Следует ли при проверке на равенство принимать это во внимание? Что если один из типов определяет поля, которые отсутствуют в другом? Проблема состоит также в том, что выбор интерпретации для вызываемого сообщения определяется получателем. Нет гарантии, что такое фундаментальное свойство, как коммутативность, будет сохраняться. Если идентификаторы x и y принадлежат к различным типам данных, то вполне может быть, что отношение x=y справедливо, а отношение y=x — нет!

В одном аспекте, однако, проблему равенства легче решить, чем разобраться с аналогичными трудностями при присваивании. Хотя последнее обычно рассматривается как часть синтаксиса и семантики языка и в силу этого может быть неизменяемым, программист всегда волен создать свои методы проверки на совпадение (возможно, с несколько другим синтаксисом). Тем самым нет единого логического смысла отношения равенства; оно может означать разные вещи для объектов разных классов.

### 12.3.1. Ковариация и контрвариация

При проверке на равенство часто полезно изменить сигнатуру типа в методах подкласса. Рассмотрим класс Shape (фигура) и два его подкласса Triangle (треугольник) и Square (квадрат). Кажется разумным, чтобы треугольники могли сравниваться только с треугольниками, а квадраты — исключительно с квадратами. Соответственно программист, возможно, напишет определение класса следующим образом:

```
class Shape
{
  public:
    boolean equals (Shape)
    {
      return false;
    }
    ...
}
class Triangle : public Shape
{
  public:
    boolean equals (Triangle);
    ...
}
class Square : public Shape
{
  public:
    boolean equals (Square);
    ...
}
```

Заметьте, что аргумент функции, проверяющей равенство, отличается для каждого из классов. Для родительского класса аргумент — это просто любая фигура, в то время как для подклассов он ограничен более специализированными типами. При таком переопределении метода используются два понятия:

- Аргумент, который для дочернего класса расширяется до более общего типа данных, чем в соответствующем методе родителя, называется ковариантом.
- Аргумент, который для дочернего класса ограничен более узким типом данных, чем в соответствующем методе родителя, называется контрвариантом.

В этом примере аргумент функции проверки на равенство является контрвариантом.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Ковариантные и контрвариантные аргументы часто оказываются естественным решением задачи, но очень незначительное число языков программирования поддерживает их. Так, ни в одном из рассматриваемых нами языков они не применяются. Далее мы рассмотрим некоторые причины такого ограничения.

Прежде всего это касается обсуждавшегося в предыдущих главах принципа подстановки. Он предписывает, что мы можем подставлять экземпляры дочернего класса везде, где допустимы экземпляры родительского класса. Но этот принцип подразумевает, что дочерний класс должен воспринимать по крайней мере все те сообщения, которые допустимы для родительского класса. Это значит, что разрешено ковариантное переопределение методов (поскольку дочерний класс сможет обрабатывать более широкий набор аргументов, чем родительский), но контрвариантное переопределение должно быть запрещено правилами (поскольку в таком случае дочерний класс будет более ограничивающим, чем родительский). Однако, как показывает данный пример, именно контрвариантное переопределение более типично в реальных задачах. (Аналогичное рассуждение приводит к выводу, что для типа возвращаемого значения чаще должно встречаться ковариантное переопределение.)

Для воображаемого языка программирования попробуем решить, какой смысл можно придать сравнению треугольника и квадрата. Имеются два варианта:

- Поиск подходящего метода основан исключительно на получателе (треугольнике), что дает «треугольный» метод, требующий треугольник в качестве аргумента. Следовательно, использование аргумента-квадрата приведет к ошибке компиляции.
- Поиск подходящего метода основан на сигнатурах типа как получателя, так и аргумента. Поскольку аргумент не соответствует определению метода в классе Triangle, то будет вызван метод класса Shape.

Большинство программистов согласятся, что вторая интерпретация кажется более естественной (хотя она усложняет задачу компилятора). Это, однако, может привести к некоторой путанице. Представим себе полиморфную переменную типа Shape, которая может содержать как значение типа Triangle, так и величину типа Square. Предположим сначала, что эта полиморфная переменная используется как аргумент в следующей программе:

```
Triangle aTriangle;
Shape  aShape;
aShape := aTriangle;
if aTriangle.equals(aTriangle) ... // возвращает true
if aTriangle.equals(aShape)  ... // возвращает false!
```

Первый вызов функции equals в данном примере связывается с методом класса Triangle и, как и ожидается, возвращает значение true. Второй оператор вызовет метод класса Shape, поскольку аргумент — это не треугольник в явном виде. Возвращаемый результат равен false, несмотря на то что фактическое значение аргумента равно (в действительности, идентично) тому же треугольнику.

Аналогичная неконгруэнтность возникает, если полиморфная переменная используется как получатель. Поскольку единственный разумный метод, который здесь можно вызвать, — это метод родителя, то следующий тест неожиданно дает нам значение false:

```
if aShape.equals(aTriangle) ... // возвращает false
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Поскольку реализация как ковариантного, так и контрвариантного переопределения методов сложна, а семантика туманна, то почти все объектно-ориентированные языки запрещают изменение типа аргументов для переопределяемых методов. Такая политика называется безвариантной (novariance). Чтобы обойти указанное ограничение, программисты часто используют явную проверку и приведение типа, как это сделано в следующем примере:

```
boolean Triangle.equals(Shape & aShape)
{
  Triangle & right = dynamic_cast(aShape);
  if (right)
  {
    ... // сравнить два треугольника
  }
  else
    return false;
}
```

В слабой форме язык C++ разрешает контрвариантное переопределение. Это случай, когда возвращаемое значение новой функции может быть подклассом родительского класса. То есть функция-член класса Shape описывается как возвращающая значение типа Shape, в то время как подкласс Triangle переопределяет ее и задает возвращаемое значение типа Triangle. Такое ослабление безвариантного правила устраняет необходимость многочисленных приведений типов и не приводит к ошибкам, связанным с типом данных. Например, переменные, описанные с типом данных Shape, будут и в самом деле возвращать Shape, даже если они являются полиморфными переменными со значениями Triangle (в последнем случае возвращаемое значение окажется подтипа Triangle).

К языкам программирования, которые допускают ковариацию и контрвариацию, относятся Eiffel [Rist 1995] и Sather.

### 12.3.2. Равенство в Objective-C, Java и Object Pascal

В языках Objective-C, Java и Object Pascal объекты всегда (для Objective-C — почти всегда) представлены внутренним образом как указатели. Неудивительно, что смысл по умолчанию оператора равенства (= в языке Object Pascal, == в языках Objective-C и Java) — это идентичность, то есть равенство указателей. Две объектные переменные при тестировании равны, только если они указывают на один и тот же объект.

Хотя ни в одном из этих языков нельзя переопределить встроенный оператор, стандартной практикой является введение методов, которые обеспечивают альтернативное понятие равенства. Следующий пример иллюстрирует проверку равенства через метод класса Card. Две игральные карты рассматриваются как равные, если они обладают одинаковыми мастью и рангом, даже если это не идентичные карты:

```
function Card.equal(aCard : Card) : boolean;
begin
   if (suitValue = aCard.suit) and
    (rankValue = aCard.rank)
   then equal := true
   else equal := false;
end;
```

Ни один из этих языков не поддерживает ковариантное или контрвариантное переопределение, о чем сообщает компилятор. В ситуациях, когда может быть полезным

PDF created with pdfFactory Pro trial version www.pdffactory.com

ковариантное переопределение (например, при проверке на равенство), должна проводиться явная проверка динамического типа данных. В главе 10 мы рассматривали механизм такой проверки.

```
class Triangle extends Shape
{
  boolean equals (Shape aShape)
  {
   if (aShape instanceOf Triangle)
   {
     ... // сравнение треугольников
   }
   else return false;
  }
}
```

### 12.3.3. Равенство в Smalltalk

Язык Smalltalk различает идентичность объектов и их равенство. Идентичность объектов проверяется с помощью удвоенного символа равенства (==). Равенство объектов анализируется с помощью однократного символа равенства (=) и рассматривается как сообщение, пересылаемое левому объекту. По умолчанию смысл этого сообщения тот же самый, что и для проверки на идентичность. Однако каждый класс может переопределить этот символ произвольным образом. Например, класс Array определяет, что равенство имеет место тогда, когда объект справа является массивом той же длины, а соответствующие элементы массивов равны.

То, что проверка равенства может быть переопределена произвольным образом, означает: нет гарантии, что равенство симметрично. Между сравнением x=y и сравнением y=x нет связи.

Поскольку Smalltalk — это язык с динамическими типами данных, то понятия ковариантного и контрвариантного переопределения имеют меньшую значимость для программистов. Там, где необходимо, могут использоваться явные проверки фактического типа значения.

### 12.3.4. Равенство в C++

В языке C++ не обеспечивается смысл по умолчанию для проверки на равенство. Отдельные классы могут переопределять оператор ==. Те же самые правила, которые используются для снятия двусмысленности переопределяемых функций, применяются и для перегружаемых операторов. Это дает иллюзию ковариации и контрвариации. В действительности они не разрешены. Например, рассмотрим такое описание класса:

```
class A
{
  public:
    int i;
    A(int x) {i = x;}
    int operator== (A& x)
    {
      return i == x.i;
    }
};
class B : public A
{
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  public:
    int j;
    B (int x, int y) : A(x) { j = y;}
    int operator== (B& x)
    {
      return (i == x.i;) && (j == x.j);
    }
};
```

Если переменные a и b — это экземпляры классов A и B, то сравнения a==a и a==b используют метод класса A, а сравнение b==b — метод класса B. Выражение b==a вызовет сообщение об ошибке компиляции, поскольку аргумент (a) не соответствует определению оператора равенства для класса B. (Это было первым и наименее интуитивным вариантом выбора в предыдущем обсуждении, когда мы пытались приписать смысл контрвариантному методу.)

Более важно то, что если полиморфная переменная (которая в C++ должна быть либо указателем, либо ссылкой) типа A на самом деле содержит значение типа B, то оператор сравнения все равно связывается с классом A. Это происходит потому, что два определения, показанные выше, — совершенно разные функции, так что никакого переопределения на самом деле не происходит. Это утверждение справедливо, даже если в первом определении используется ключевое слово virtual.

## 12.4. Преобразование типов

Для языков программирования со статическими типами данных (вроде C++ и Object Pascal) нельзя присваивать значение типа надкласса переменной, объявленной как экземпляр подкласса. Значение, о котором компилятору известно, что оно класса Window, не может быть присвоено переменной, описанной с классом TextWindow. Причины такого ограничения почти очевидны. Если это не так, то они рассматриваются в упражнениях 1 и 2 этой главы.

Тем не менее в некоторых редких случаях желательно нарушить это правило. Чаще всего такая ситуация возникает, когда программист дополнительно знает, что значение, хотя и содержится в переменной типа надкласса, на самом деле является экземпляром более специализированного класса. Тогда можно (хотя это и не приветствуется) перехитрить систему проверки типов данных.

В C++ и Objective-C мы проделываем этот маневр с использованием конструкции языка C, называемой приведением типа данных. Приведение типа заставляет компилятор преобразовать значение из одного типа данных в другой. Наиболее часто этот подход используется в случае указателей, когда осуществляется только логическая замена, а не физическое преобразование.

Мы проиллюстрируем приведение типа в варианте игры в бильярд, которая предполагается переписанной на язык C++. Вместо того чтобы заставлять каждый экземпляр шара Ball содержать указатель, используемый в связном списке, мы определим обобщенный класс Link следующим образом:

```
class Link
{
  protected:
    Link *link;
  public:
    Link * next();
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    void setLink(Link * elem);
};
void Link::setLink(Link *elem)
{
  link = elem;
}
Link * Link::next()
{
  return link;
}
```

Класс Ball сделаем подклассом класса Link. Поскольку метод setLink наследуется от надкласса, его нет необходимости повторять. Однако имеется проблема с наследуемым методом next. Он утверждает, что возвращает экземпляр класса Link, а не Ball. Тем не менее мы знаем, что объект на самом деле принадлежит классу Ball, поскольку это единственные объекты, которые мы помещаем в список. Поэтому мы переписываем класс Ball с тем, чтобы переопределить метод next, и используем приведение типа для того, чтобы изменить тип возвращаемого значения:

```
class Ball : public Link
{
  ...
  public:
    ...
    Ball * next();
    ...
};
Ball * Ball::next()
{
  return dynamic_cast(Link::next());
}
```

Заметьте, что метод next не описан как виртуальный: нельзя изменить тип возвращаемого значения виртуальной функции. Важно помнить, что в языке C++ это приведение типа будет законным только для указателей, а не для собственно объектов (см. упражнение 2 этой главы). Функция dynamic_cast является частью системы RTTI (Run-Time Typing Information — идентификация типа во время выполнения), которая была описана в главе 10. Приведение типов через RTTI должно использоваться вместо более ранних синтаксических форм, поскольку неправильное приведение типов данных является стандартным источником ошибок в программах.

В языке Object Pascal задействована аналогичная идея. Поскольку объекты рассматриваются внутренним образом как указатели, то это преобразование может применяться к объектам любого типа, а не только к указателям. Язык Object Pascal обеспечивает проверку класса объекта во время выполнения, так что все «сомнительные» приведения типа должны быть выявлены с помощью явной проверки до присваивания.

```
var
  x : TextWindow;
  y : Window;
begin
  ...
  if Member(y, TextWindow)
    then x := TextWindow(y)
    else writeln('illegal window assignment');
  ...
end;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

В языке Java переменная, которая содержит значение, принадлежащее подклассу приписанного ей класса, может быть приведена к типу подкласса. Однако такие преобразования проверяются во время выполнения программы, и если результат неправилен, то возбуждается исключительная ситуация. Если программист ее не желает, тип значения должен проверяться с помощью оператора instanceOf, как это делается в следующем примере:

```
Ball aBall;
WhiteBall wBall;
if (aBall instanceOf WhiteBall)
  wBall = (WhiteBall) aBall;
else
  ...
```

### Упражнения

1. Объясните, почему в языках со статическими типами данных (вроде C++ и Object Pascal) нельзя присваивать значение типа надкласса переменной, описанной как экземпляр подкласса. То есть команды вроде нижеследующих приводят к сообщению компилятора об ошибке:

   ```
     TextWindow X;
   Window Y;
   ...
   X := Y;
   ```

2. Предположим, что в C++ метод распределения памяти работает так, как описано в разделе 12.1. Объясните, какие могут возникнуть проблемы, если пользователь пытается обойти ограничение, описанное в упражнении 1, путем приведения типов данных, используя команду присваивания вроде

   ```
   x = (TextWindow) Y;
   ```

3. Приведите пример на Object Pascal или C++, который иллюстрирует, почему для определения размера объекта с помощью метода из раздела 12.1 должна просматриваться вся программа целиком, а не только отдельный файл.
4. Объясните, почему при условии соблюдения принципа подстановки возвращаемое значение для метода дочернего класса не может быть более общим, чем у родительского класса.
5. Покажите, что можно определить язык, аналогичный Object Pascal, который не использует семантику указателей при присваивании. Другими словами, опишите алгоритм для операции присваивания в языке программирования, который реализует управление памятью, описанное в разделе 12.1, но не приводит к тому, что две переменные указывают на одно место в памяти после операции присваивания. Как вы думаете, почему разработчики языка Object Pascal не использовали ваш алгоритм присваивания?

## *Глава 13: Множественное наследование*

Выше во время наших рассуждений мы предполагали, что класс наследует только от одного родительского класса. Хотя эта ситуация является, безусловно, типичной, тем не менее бывают случаи, когда некоторая абстракция логически вытекает из двух (или более) независимых источников. Если вы представляете себе классы как аналоги категорий, как мы делали в главе 1, и пытаетесь описать себя в терминах групп, к которым

PDF created with pdfFactory Pro trial version www.pdffactory.com

принадлежите, то весьма вероятно, что вы построите много непересекающихся классификаций. Например, я — отец ребенка, профессор, гражданин США. Ни одна из этих категорий не является собственным подмножеством другой.
Еще пример. Бет занимается художественной лепкой. Ее мы относим к классу Potter. Ее соседка Маргарет рисует портреты, она — PortraitPainter. Тот тип живописи, которым она занимается, отличен от ремесла Пола: он — маляр (HousePainter). Обычно мы рассматриваем однонаправленное наследование как способ специализации (в данном примере Potter — это частный случай художника Artist). Однако множественное наследование следует рассматривать как комбинирование (портретист PortraitPainter — это творческий человек Artist и художник Painter, как это показано на рис. 13.1).

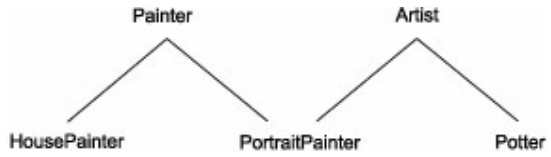


**Рис. 13.1**. Наследование как комбинирование

## 13.1. Комплексные числа

Мы проиллюстрируем трудности, возникающие при одиночном наследовании, на более конкретном примере. В языке Smalltalk класс Magnitude определяет некий протокол для объектов с определенной мерой: они могут сравниваться друг с другом по величине [1].

Например, отдельные символы (экземпляры класса Char) сравниваются по своей внутренней кодировке (скажем, ASCII). Более традиционный класс сравнимых объектов — числа, то есть экземпляры класса Number в терминах Smalltalk. Помимо сравнения, экземпляры класса Number поддерживают выполнение арифметических операций (сложение, умножение и т. д.). Эти операции не имеют смысла для объектов класса Char. В Smalltalk имеется несколько типов чисел: целые (Integer), дробные (Fraction), вещественные (Float). Часть порождаемой иерархии классов показана на рис. 13.2.

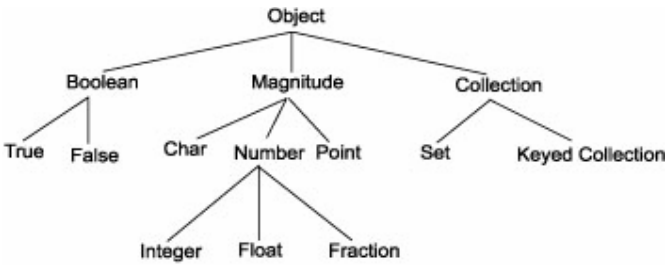


**Рис. 13.2**. Часть иерархии классов Smalltalk

Предположим теперь, что мы добавляем класс Complex, который представляет собой абстракцию комплексного числа. Арифметические операции, несомненно, определены для комплексных чисел. Разумно сделать Complex подклассом класса Number, так что арифметика наследуется и переопределяется. Проблема состоит в том, что сравнение двух комплексных чисел — это нечто двусмысленное. Комплексные числа просто не сравнимы между собой.

---

[1] Мера — это нечто большее, чем просто способность сравниваться друг с другом, а зачастую и просто нечто отличное от отношения «больше–меньше» (например, для рассматриваемых автором далее комплексных чисел характеристика меры определена, а отношение сравнения — нет). — Примеч. перев.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Итак, мы имеем следующие ограничения:

- Класс Char должен быть подклассом класса Magnitude, но не подклассом класса Number.
- Класс Integer должен быть подклассом как класса Magnitude, так и класса Number.
- Класс Complex должен быть подклассом класса Number, но не подклассом класса Magnitude.

Невозможно удовлетворить всем этим условиям с помощью иерархии одиночного наследования. Имеется несколько альтернативных решений данной проблемы:

1. Сделать класс Complex подклассом класса Number, который в свою очередь является подклассом класса Magnitude. Затем переопределить методы, имеющие отношение к сравнению экземпляров в классе Complex, с тем, чтобы при их вызове генерировалось сообщение об ошибке. Это — создание подкласса с ограничением, как описано в главе 7. Хотя такое решение и не является элегантным, оно иногда наиболее целесообразно, если ваш язык программирования не поддерживает множественное наследование.
2. Не использовать наследование вообще. Определить заново каждый метод во всех классах Char, Integer, Complex и т. д. Это решение иногда называется декомпозицией иерархического дерева. Естественно, оно устраняет все преимущества наследования, описанные в главе 7, — например, многократное использование кода и гарантированность интерфейсов. Для языков программирования со статическими типами данных (таких, как C++ или Object Pascal) этот способ запрещает полиморфные объекты: например, нельзя создать переменную, которая содержала бы произвольный измеримый объект или число произвольного типа.
3. Использовать часть иерархии наследования и имитировать оставшуюся ветвь. Например, можно поместить все числа в класс Number, а для каждого измеримого объекта (символа или числа) задать операцию сравнения.
4. Сделать два класса Magnitude и Number независимыми друг от друга и в силу этого потребовать, чтобы класс Integer наследовал от них обоих при задании свойств (рис. 13.3). Класс Float будет аналогичным образом наследовать как от Number, так и от Magnitude.

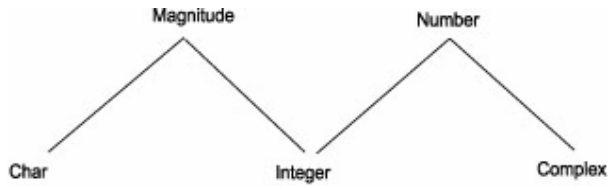



**Рис. 13.3**. Иерархия множественного наследования для комплексных чисел

Важный момент в альтернативах 2 и 3 — это то, что они в гораздо большей степени привлекательны для языков программирования с динамическими типами данных (Objective-C, Smalltalk). В языках C++ или Object Pascal определение того, какие именно типы являются «измеримыми» или «сравнимыми», выражается в терминах классов. А именно объект «измерим», если он может быть присвоен переменной, объявленной с классом Magnitude. С другой стороны, в языках Smalltalk и Objective-C объект является «измеримым», если он понимает сообщения, относящиеся к сравнению объектов, независимо от того, в каком месте иерархии классов он находится. Тем самым, чтобы заставить комплексные числа взаимодействовать с другими объектами, даже если они не

PDF created with pdfFactory Pro trial version www.pdffactory.com

имеют общих классов-предков, может использоваться техника двойной диспетчеризации (см. работу [Ingalls 1986] или раздел 18.2.3).

О классе, который наследует от двух или более родительских классов, говорят, что он порожден множественным наследованием. Множественное наследование — это мощное и полезное свойство языка программирования, но оно создает много изощренных и сложных проблем при реализации. Из рассматриваемых нами языков только C++ поддерживает множественное наследование, хотя некоторые исследовательские версии Smalltalk также обладают этим свойством. В данной главе мы будем изучать некоторые из преимуществ и проблем, связанных с множественным наследованием.

## 13.2. Всплывающие меню

Второй пример проиллюстрирует многие моменты, которые необходимо иметь в виду, когда вы рассматриваете множественное наследование. Он вдохновлен библиотекой для создания графических пользовательских интерфейсов, которая связана с объектно-ориентированным языком Eiffel [Meyer 1988a, Meyer 1988b]. В этой системе меню описываются как класс Menu. Экземпляры класса Menu поддерживают такие свойства, как число пунктов меню, список команд и т. д. Функционирование, связанное с меню, подразумевает способность отображать меню (то есть себя) на графическом экране и выбирать один из его пунктов (рис. 13.4).

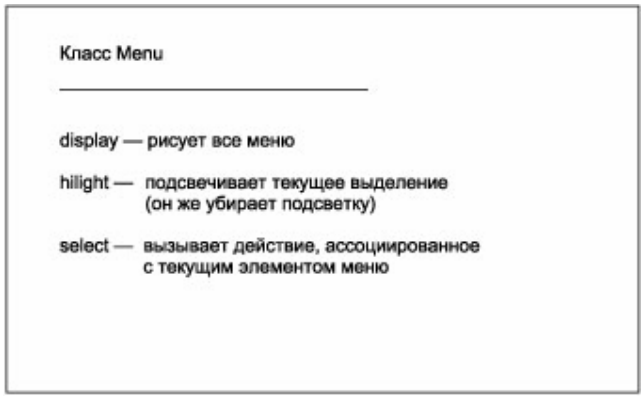



**Рис. 13.4**. CRC-карточка для класса Menu

Каждый элемент (пункт) меню представляет собой экземпляр класса MenuItem. Экземпляры содержат текст элемента, ссылку на родительское меню и описание команды, которую надо выполнить при выборе этого пункта меню (рис. 13.5).

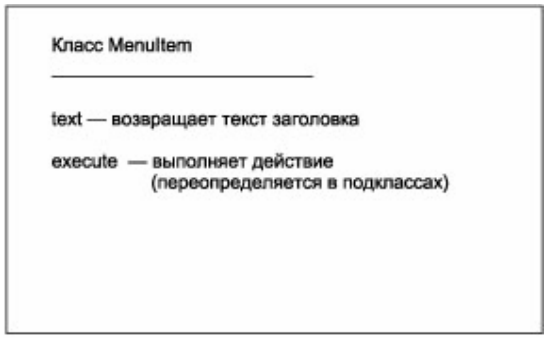



**Рис. 13.5**. CRC-карточка для класса MenuItem

PDF created with pdfFactory Pro trial version www.pdffactory.com

Типичным средством пользовательского графического интерфейса являются многоуровневые меню (в некоторых системах они называются каскадными меню), которые требуются, когда элемент меню содержит несколько альтернативных команд. Например, пункт главного меню для эмулятора терминала может иметь имя «задать опции». Когда этот пункт (подменю) выбран пользователем, высвечивается второе меню, которое позволяет выбирать из набора имеющихся опций (темный/светлый фон и т. д.). Многоуровневое меню определенно принадлежит классу Menu. Оно содержит ту же информацию, что и класс Menu, и должно вести себя подобным образом. С другой стороны, оно так же, несомненно, является элементом MenuItem, так как содержит имя и способно выполнить команду (вывести свой образ на экран), когда соответствующий пункт выбран в родительском меню. Требуемое поведение может быть достигнуто с минимальными усилиями, если мы разрешим классу WalkingMenu наследовать от обоих родителей. Например, когда всплывающему меню требуется выполнить действие по щелчку мыши (унаследованное от класса MenuItem), оно выводит на экран свое содержимое (вызывая графический метод, унаследованный от класса Menu).
Как и в случае с одиночным наследованием, при использовании множественного наследования важно иметь в виду условие «быть экземпляром». В нашем примере множественное наследование оправдано, поскольку имеет смысл каждое из утверждений: «подменю есть меню» и «подменю есть пункт меню». Когда отношение «быть экземпляром» не выполнено, множественное наследование может использоваться неправильно. Например, неверно описывать «автомобиль» как подкласс двух классов: «мотор» и «корпус». Точно так же класс «яблочный пирог» не следует выводить из классов «пирог» и «яблоко». Очевидно, что яблочный пирог есть пирог, но он не есть яблоко.
Когда множественное наследование используется правильным образом, происходит тонкое, но тем не менее важное изменение во взгляде на наследование. Интерпретация условия «быть экземпляром» при одиночном наследовании рассматривает подкласс как специализированную форму другой категории (родительского класса). При множественном наследовании класс является комбинацией нескольких различных характеристик со своими интерфейсами, состояниями и базовым поведением, специализированным для рассматриваемого случая. Операции записи и поиска объектов в неком хранилище (скажем, на жестком диске) представляют типичный пример. Часто эти действия реализованы как часть поведения, связанного с определенным классом типа Persistence или Storable. Чтобы придать такую способность произвольному классу, мы просто добавляем Storable к списку предков класса.
Конечно же, важно различать между наследованием от независимых источников и построением из независимых компонент, что иллюстрируется на примере «автомобиля» и «мотора».

## 13.3. Двусмысленность имен

Часто возникающее затруднение при множественном наследовании состоит в том, что имена могут использоваться для обозначения более чем одной операции. Чтобы проиллюстрировать это, мы рассмотрим еще раз модель карточной игры. Предположим, что уже имеется абстракция карточной колоды CardDeck, которая обеспечивает надлежащую функциональность: тасование колоды (метод shuffle), выбор отдельной карты (метод draw [1] )

---

[1] Здесь используется тот факт, что в английском языке глагол draw кроме значения «рисовать» имеет также и массу других значений (в англо-русском словаре В. К. Мюллера их список занимает почти две колонки текста), среди которых имеются «отбрасывать», «вытаскивать» и «тянуть жребий». — Примеч. перев. и

PDF created with pdfFactory Pro trial version www.pdffactory.com

т. д., но графика при этом не реализована. Предположим далее, что другое множество классов обеспечивает поддержку обобщенных графических объектов. Они содержат поле данных (точку на плоскости) и, кроме того, виртуальный метод с именем draw для графического отображения самих себя.

Программист решает создать класс новой абстракции GraphicalDeck, которая наследует как от класса CardDeck, так и от GraphicalObject. Ясно, что концептуально класс GraphicalDeck является колодой карт CardDeck и тем самым логически выводится из нее. GraphicalDeck является также графическим объектом GraphicalObject. Единственная неприятность — это двойное значение команды draw.

Как отмечает Мейер [Meyer 1988a], проблема однозначно кроется в дочернем классе, а не в родителях. Команда draw имеет недвусмысленное значение для каждого из родительских классов, когда они рассматриваются изолированно. Сложность состоит в их комбинировании. Поскольку загвоздка возникает на уровне дочернего класса, то и решение должно быть найдено здесь же. В данном случае дочерний класс обязан принять решение, как устранить двусмысленность перегруженного имени.

Решение проблемы обычно включает комбинацию переименования и переопределения. Под переопределением мы понимаем изменение в выполняемой операции или команде, как это происходит при модификации в подклассе виртуального метода. Под переименованием мы просто подразумеваем смену имени метода без изменения его

функционирования. В случае колоды карт GraphicalDeck программист может приписать методу draw задачу рисования графического образа, а процесс вытаскивания карты из колоды переименовать в drawCard.

### 13.3.1. Наследование через общих предков

Более сложная проблема возникает, если программист хочет использовать два класса, имеющие общего родителя. Предположим, что программист разрабатывает набор классов для потоков ввода/вывода. Поток данных — это обобщение понятия файла. Элементы первого могут быть более структурированы. Например, часто используются потоки целых чисел или потоки чисел с плавающей точкой. Класс InStream обеспечивает протокол для входных потоков. Пользователь может открыть входной поток путем присоединения его к файлу данных, выбрать очередной элемент из потока и т. д. Класс OutStream обеспечивает похожую функциональность для выходных потоков. Оба класса наследуют от общего родителя с именем Stream. Информация, которая указывает на собственно файл данных, прячущийся под маской потока, содержится в родительском классе.

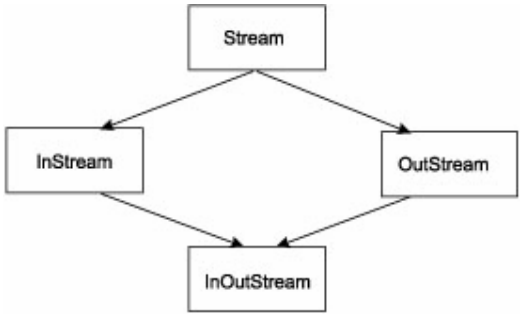



**Рис. 13.6**. Граф множественного наследования

PDF created with pdfFactory Pro trial version www.pdffactory.com

Теперь предположим, что пользователь хочет создать комбинированный поток ввода/вывода InOutStream (рис. 13.6). Имеет смысл объявить его потомком и потока ввода, и потока вывода. Переименование (см. предыдущий раздел) позволяет принять решение по поводу любой функции, определенной одновременно в классах InStream и OutStream. Но что делать со свойствами, наследуемыми от общего прародителя Stream? Трудность состоит в том, что дерево наследования — это направленный граф, а не просто дерево (см. рис. 13.6). Если методы — это единственное, что наследуется от общего родительского класса, то может быть использована описанная выше техника разрешения противоречий. Но если родительский класс определяет также и поля данных (например, указатель на файл), то имеются два варианта. Хотим ли мы иметь две копии полей данных или только одну? Аналогичная проблема возникает, если прародительский класс использует конструкторы или подобные им средства инициализации, которые должны вызываться только однажды. В следующем разделе мы опишем, как с этой проблемой справляется язык C++.

## 13.4. Множественное наследование в C++

Мы проиллюстрируем использование множественного наследования в C++, работая над небольшим примером. Предположим, что для прежнего проекта программист разработал набор классов для манипуляций со связными списками (листинг 13.1). Абстракция списка была разбита на две части: класс Link поддерживает указатели на элементы списка, а класс LinkedList запоминает начало списка. Основной смысл связного списка — добавление новых элементов. Связные списки предоставляют также возможность выполнить некоторую функцию над каждым своим элементом. Функция передается в качестве аргумента. Оба эти действия поддерживаются процедурами в классе Link.

**Листинг 13.1**. Классы реализации связных списков

```
class LinkedList
{
  public:
    Link *elements;
    LinkedList()
    {
      elements = (Link *) 0;
    }
    void add(Link *n)
    {
      if (elements) elements->add(n);
        else elements = n;
    }
    void onEachDo(void f(Link *))
    {
      if (elements) elements->onEachDo(f);
    }
};
class Link
{
  public:
    Link *next;
    Link()
    {
      next = (Link *) 0;
    }
    void setLink(Link *n)
    {
      next = n;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    }
    void add(Link *n)
    {
      if (next) next->add(n);
        else setLink(n);
    }
    void onEachDo(void f(Link *))
    {
      f(this);
      if (next) next->onEachDo(f);
    }
};
```

Мы образуем специализированные списки через определение подклассов Link. Например, класс IntegerLink в листинге 13.2 служит для поддержки списков целых чисел. Листинг 13.2 содержит также короткую программу, которая показывает, как используется эта абстракция данных.

Теперь предположим, что для нового проекта тот же самый программист должен разработать класс Tree (древовидная структура). После некоторого размышления он обнаруживает, что дерево можно представить себе как совокупность связных списков. На каждом уровне дерева поля связи указывают на подветви (деревья, принадлежащие к одному уровню). Однако каждый узел указывает также на связный список, который представляет собой его потомков. Рисунок 13.7 иллюстрирует эту структуру. Здесь наклонные стрелки обозначают указатели на потомков, а горизонтальные — соединения подветвей.

Тем самым узел дерева относится и к классу LinkedList (поскольку он содержит указатель на список своих потомков), и к классу Link (поскольку он содержит указатель на свою подветвь). В языке C++ мы обозначаем множественное наследование, просто перечисляя имена надклассов, разделяя их запятыми (перечисление следует после двоеточия сразу за именем класса при его описании). Как и в случае одиночного наследования, каждому классу должно предшествовать ключевое слово (public или private), которое определяет правило видимости. В листинге 13.3 показан класс Tree, который наследует от классов

**Листинг 13.2**. Уточнение класса Link

```
class IntegerLink : public Link
{
  int value;
  public:
    IntegerLink(int i) : Link()
    {
      value = i;
    }
    print()
    {
      printf("%d\n", value);
    }
};
void display(IntegerLink *x)
{
  x->print();
}
main()
{
  LinkedList list;
  list.add(new IntegerLink(3));
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  list.add(new IntegerLink(17));
  list.add(new IntegerLink(32));
  list.onEachDo(display);
}
```

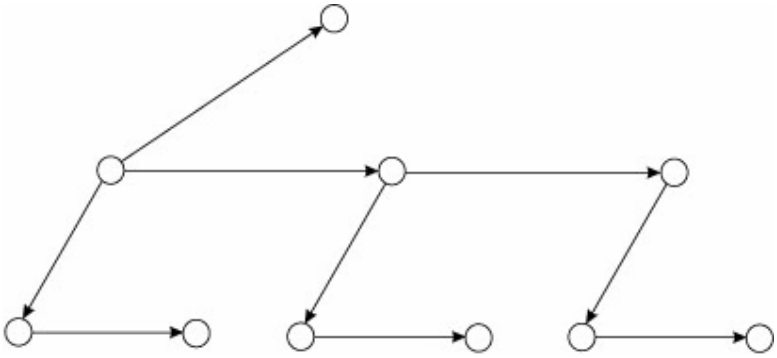


**Рис. 13.7**. Дерево как совокупность связных списков

**Листинг 13.3**. Пример множественного наследования

```
class Tree : public Link; public LinkedList
{
  int value;
  public:
    Tree(int i)
    {
      value = i;
    }
    print()
    {
      printf("%d\n",value);
    }
    void add(Tree *n)
    {
      LinkedList::add(n);
    }
    void addChild(Tree *n)
    {
      Linkedlist::add(n);
    }
    void addSubling(Tree *n)
    {
      Link::add(n);
    }
    void onEachDo(void f(Link *))
    {
      /* сначала обработать потомка */
      if (elements) elements->onEachDo(f);
      /* затем себя */
      f(this);
      /* потом перейти к подветвям */
      if (next) next->onEachDo(f); }
};
main()
{
  Tree *t = new Tree(17);
  t->add(new Tree(12));
  t->addSubling(new Tree(25));
  t->addChild(new Tree(15));
  t->addEachDo(display);
}
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Link и LinkedList с ключевым словом public. Узлы дерева Tree содержат указатели на потомков, а также целочисленные значения.

Теперь необходимо справиться с проблемой неоднозначности. Прежде всего имеется двусмысленность в имени add (добавить), которая отражает двойственность приписываемого ему значения. Для дерева есть два смысла операции «добавить»: присоединить узел-потомок и породить узел-подветвь. Первое обеспечивается функцией add класса LinkedList, второе — функцией add класса Link. После некоторого размышления программист решает оставить функцию add в смысле «добавить узел-потомок», но одновременно вводит две новые функции, имена которых отражают цель операции (добавить подветвь и добавить потомка).

Заметьте, что все три функции по существу — просто переименованные старые. Они не добавляют нового функционирования, а просто передают управление ранее определенным функциям. Некоторые объектно-ориентированные языки (например, Eiffel) позволяют пользователю вводить подобное переименование без создания новой функции.

Двусмысленность в методе onEachDo является более сложной. Здесь правильное действие состоит в выполнении сквозного прохода по всем узлам дерева. Процесс начинается с просмотра узлов-потомков, затем возвращается в исходный узел, а затем переходит к подветвям (которые, естественно, осуществляют рекурсивный проход уже по своим потомкам). То есть действие является комбинацией методов базовых классов Link и LinkedList, как это показано в листинге 13.4.

Переименование время от времени оказывается необходимым из-за пересечения понятий наследования и параметрической перегрузки. Когда в C++ используется перегруженное имя, то сперва вызывается механизм наследования для поиска контекста, в котором определена функция. Затем типы параметров анализируются для снятия двусмысленности в пределах данного контекста. Предположим, что есть два класса A и B, для каждого из которых определен метод display, но у методов разные аргументы (листинг 13.4). Пользователь считает, что так как эти два метода различаются по списку параметров, дочерний класс может наследовать от двух родителей и иметь доступ к обоим методам. К сожалению, здесь наследования недостаточно. Когда пользователь вызывает метод display с целочисленным аргументом, компилятор не может принять решение, использовать ли функцию из класса A (которая соответствует типу аргумента) или же из класса B (которая встречается первой при заложенном в C++ алгоритме поиска; для ее вызова аргумент будет приведен от типа integer к типу double). К счастью, компилятор всегда предупреждает о подобных случаях. Однако предупреждение выдается в точке вызова метода, а не при описании класса.

Выход в том, чтобы переопределить оба метода для дочернего класса C, как это показано в листинге 13.4. Мы избежим конкуренции между наследованием и перегрузкой — в обоих случаях поиск кончается в классе C, где для компилятора уже ясно, что параметрическая перегрузка используется намеренно.

**Листинг 13.4**. Взаимодействие наследования и перегрузки

```
class A
{
  public:
    void virtual display(int i)
    {
      printf("in A %d\n", i);
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    }
};
class B
{
  public:
    void virtual display(double d)
    {
      printf("in B %g\n", d);
    }
};
class C: public B, public A
{
  public:
    void virtual display(int i)
    {
      A::display(i);
    }
    void virtual display(double d)
    {
      B::display(d);
    }
};
main()
{
  C c;
  c.display(13);
  c.display(3.14);
}
```

В предыдущем разделе мы описали трудность, которая возникает, когда класс наследует от двух родителей, имеющих общего предка. Эта проблема была проиллюстрирована на примере классов InStream и OutStream, каждый из которых наследовал от общего класса Stream. Если мы хотим, чтобы порождаемый класс наследовал только одну копию полей данных, определенных в классе Stream, то промежуточные классы InStream и OutStream должны определять, что их наследование от общего родительского класса является виртуальным. Ключевое слово virtual показывает, что надкласс может появляться более одного раза в подклассах, порождаемых из определяемого класса, но при этом нужно оставлять только одну его копию. Листинг 13.5 показывает такой вариант описания для этих четырех классов.

**Листинг 13.5**. Пример виртуального наследования

```
class Stream
{
  File *fid;
  ...
};
class InStream : public virtual Stream
{
  ...
  int open(File *);
};
class OutStream : public virtual Stream
{
  ...
  int open(File *);
};
class InOutStream: public InStream, public OutStream
{
  ...
};
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Такой подход, использованный в языке C++, нельзя признать совершенным (как на это указывает Мейер), поскольку конфликт имен возникает на уровне дочерних классов, а решение (ключевое слово virtual для общего предка) затрагивает родительские классы. То есть «виртуальное предназначение» общей части закладывается в родителях, а не в комбинированном классе.

В редких случаях желательно создавать две копии наследуемых полей данных. Например, и графические объекты, и колоды игральных карт могут быть основаны на связных списках, поэтому оба класса выводятся из класса Link. Поскольку эти два списка являются независимыми, они оба должны содержаться в комбинированном классе GraphicalDeck. В такой ситуации ключевые слова virtual опускаются — желаемый результат будет достигнут. Однако важно гарантировать, что возникший конфликт имен не вызовет ошибочной интерпретации.

Ключевые слова, определяющие видимость, имеют право отличаться в разных родительских классах. Следовательно, виртуальный наследник может быть порожден с различными атрибутами, например public и protected. В таком случае наиболее жесткий уровень защиты (в нашем примере protected) игнорируется и используется менее ограничительная категория.

Когда конструкторы определены в нескольких надклассах, важен порядок выполнения родительских конструкторов и, следовательно, очередность инициализации полей данных. Пользователь может управлять этим, вызывая непосредственно конструкторы родительских классов внутри конструктора потомков. Например, в листинге 13.6 пользователь явно указывает, что при инициализации класса C конструктор класса B вызывается первым, то есть до вызова конструктора класса A. Порядок вызова конструкторов влияет на инициализацию.

Исключение из этого правила составляют виртуальные базовые классы. Они всегда инициализируются лишь один раз вызовом безаргументного конструктора (если обращение к нему не осуществляется пользователем, такой конструктор вызывается системой). Это происходит до какой бы то ни было другой инициализации. В листинге 13.6 инициализация при создании нового элемента класса C будет производиться в таком порядке: инициализируется класс D с помощью безаргументного конструктора, затем — класс B и наконец — класс A. Два кажущихся вызова конструктора класса D внутри конструкторов классов A и B не имеют никакого эффекта, поскольку указано, что родительский класс виртуален.

Если требуется, чтобы для конструктора базового класса задавались аргументы, класс C может законным образом задать нужные значения даже тогда, когда D не

**Листинг 13.6**. Конструкторы при множественном наследовании

```
class D
{
  public:
    D()
    {  ...  }
    D(int i)
    {  ...  }
    D(double d)
    {  ...  }
};
class A : virtual D
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
{
  public:
    A() : D(7)
    {  ...  }
};
class B : virtual D
{
  public:
    B() : D(3.14)
    {  ...  }
};
class C : public A, public B
{
public:
    C() : B(), A()
    {  ...  }
};
```

является непосредственным предком класса C. Это единственная ситуация, когда внутри класса разрешено использовать конструктор другого класса, который не является непосредственно предшествующим родителем. То есть конструктор класса C может быть записан следующим образом:

```
C() : D(12), B(), A() { ... }
```

Конструкторы для виртуальных базовых классов должны вызываться первыми, то есть до конструкторов невиртуальных предков.

Виртуальные методы, определенные в виртуальных базовых классов, также могут быть источником проблем. Предположим, что каждый из четырех классов в листинге 13.5 обладает методом с именем initialize(). Он определен как виртуальный в классе Stream и переопределяется в каждом из последующих трех классов. Методы initialize() в классах InStream и OutStream вызывают Stream::initialize() и, кроме того, выполняют некоторую специфическую для каждого из классов инициализацию.

Теперь рассмотрим метод initialize() для класса InOutStream. Он не может вызвать оба унаследованных метода InStream::initialize() и OutStream::initialize() без того, чтобы не вызвать дважды метод Stream::initialize(). Повторное обращение к методу Stream::initialize(), вероятно, будет иметь побочные эффекты. Способ из-бежать этой проблемы: переписать Stream::initialize() так, чтобы он определял, была ли уже осуществлена инициализация. Другой вариант: переопределить методы классов InStream и OutStream, чтобы они не вызывали метод класса Stream. В последнем случае класс InOutStream должен в явном виде обращаться к процедуре инициализации каждого из трех классов.

## 13.5. Множественное наследование в Java

Язык Java не поддерживает множественное наследование классов, но реализует множественное наследование интерфейсов. Класс может указать, что он поддерживает несколько различных интерфейсов. Например, один интерфейс может требовать запоминания данных на диске, а другой — определять протокол самоотображения объектов. Запоминаемый графический объект будет поддерживать оба эти интерфейса:

```
class graphicalObject implements Storable, Graphical
{
  // ...
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
}
```

В то время как классы не могут наследовать от двух и более классов (расширяя их), интерфейсам это разрешено. Мы имеем право определить интерфейс для запоминаемых графических объектов следующим образом:

```
interface GraphicalObject extends Storable, Graphical
{
  // ...
}
```

### Литература для дальнейшего чтения

Критика множественного наследования встречается у Саккинена [Sakkinen 1988a]. Упомянутая работа является сокращенной адаптированной версией его Ph. D. диссертации [Sakkinen 1992]. Объяснение множественного наследования в языке C++ дается Эллис [Ellis 1990].

### Упражнения

1. Приведите два примера множественного наследования в ситуациях, не связанных с компьютерами.
2. В работе [Wiener 1989] описан «практический пример множественного наследования в C++». Определен класс IntegerArray, который наследует от двух классов Array и Integer. Как вы думаете, является ли это хорошим примером множественного наследования? Обоснуйте свой ответ.
3. Модифицируйте определение класса Tree так, чтобы он мог быть использован как двоичное дерево. Обеспечьте средства для поиска или изменения левого или правого потомка любого узла. Какие предположения вам требуются?
4. Обобщите вашу работу над упражнением 3 так, чтобы создать поисковое двоичное дерево. Оно содержит список целых чисел со следующим свойством: значение в каждом узле больше, чем значения в левой подветви, и меньше или равно значениям в правой подветви.
5. Обсудите виртуальное наследование в языке C++ с точки зрения принципов Парнаса о маскировке информации.

## *Глава 14: Полиморфизм*

Слово полиморфизм греческого происхождения и означает приблизительно «много форм» (poly — много, morphos — форма). Слово morphos имеет отношение к греческому богу сна Морфею (Morphus), который мог являться спящим людям в любой форме, в какой только пожелает, и, следовательно, был воистину полиморфным. В биологии полиморфные виды — это те (наподобие Homo Sapiens), которые характеризуются наличием различных форм или расцветок. В химии полиморфные соединения могут кристаллизоваться по крайней мере в двух различных формах (например, углерод имеет две кристаллические формы — графита и алмаза).

### 14.1. Полиморфизм в языках программирования

В языках программирования полиморфный объект — это сущность (переменная, аргумент функции), хранящая во время выполнения программы значения различных типов. Полиморфные функции — это те функции, которые имеют полиморфные аргументы.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 14.1.1. Полиморфные функции в динамических языках

Полиморфные функции относительно легко составлять в языках программирования с динамическими типами данных (Lisp, Scheme в функциональной парадигме, Smalltalk в объектно-ориентированной парадигме). Следующий пример иллюстрирует метод языка Smalltalk с именем silly, который в зависимости от аргумента x возвращает: (x+1), если x — целое число, обратную величину, если x — дробь, текст в обратном порядке, если x — текстовая строка, и специальное значение nil во всех остальных случаях:

```
silly: x " глупейший полиморфный метод "
    (x isKindOf: Integer) ifTrue: [  x + 1 ].
    (x isKindOf: Fraction) ifTrue: [  x reciprocal ].
    (x isKindOf: String)  ifTrue: [  x reversed ].
     nil
```

Полиморфизм встречается и в языках со строгим контролем типов данных. Его наиболее распространенная форма в стандартных языках программирования — это перегрузка. Так, символ «+» означает сложение и целых и вещественных чисел. Мы будем рассматривать этот вид полиморфизма в следующем подразделе.

Новые функциональные языки программирования (например, ML [Milner 1990]) разрешают использовать разновидность полиморфизма, называемую параметрическим полиморфизмом. При этом подходе параметр может быть описан только частично — например, «список из T», где тип данных T остается неопределенным. Это позволяет определять функции, оперирующие со списками. Такие функции могут вызываться для списков произвольного типа. Аналогичные свойства доступны в некоторых объектно-ориентированных языках через обобщенные функции или шаблоны.

В общем случае полиморфизм в объектно-ориентированных языках отражает принцип подстановки. То есть полиморфной объектно-ориентированной переменной разрешено хранить в себе значение, относящееся к ее объявленному типу данных или же к любому его подтипу.

### 14.1.2. Абстракции низкого и высокого уровней

Один из способов анализа полиморфизма — рассмотреть его с точки зрения абстракций низкого и высокого уровней. Абстракция низкого уровня — это базовая операция (например, над структурой данных), которая построена поверх небольшого количества механизмов. Абстракция высокого уровня отражает планирование более общего характера (например, алгоритм сортировки или изменение размера окна), когда определяется общий подход, без конкретизации деталей.

Алгоритмы обычно описываются абстракциями высокого уровня. Их фактическая реализация также должна быть абстракцией высокого уровня, основанной, однако, на структурах низкого уровня. Приведем пример. На высоком уровне рекурсивный алгоритм, вычисляющий длину списка, записывается следующим образом:

```
function length (list) -> integer
begin
  if list.link is nil
  then
    return 1
  else
    return 1 + length(list.link)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
end
```

Фактическая реализация алгоритма на традиционном языке программирования (таком, как Pascal) потребует конкретной спецификации не только тех данных, которые используются в алгоритме (поле link), но и всех свойств, которые входят в структуру данных, но не задействованы в алгоритме. Тем самым алгоритм должен быть переписан на языке Pascal следующим образом:

```
type
        intlist = record
              value : integer;
              link : ^ intlist;
  end;
function length (x : ^ intlist) : integer;
begin
  if x^.link = nil then
    length := 1
  else
    lehgth := 1 + length(x^.link);
end;
```

Эта функция может быть использована для вычисления длины связного списка, составленного из целых чисел, но она непригодна для определения длины связного списка вещественных чисел. Вычисление длины любого нецелочисленного списка потребует полного переопределения структуры данных.

Большинство программ состоят из абстракций как высокого, так и низкого уровней. Разработчик, знакомый с понятиями структурного программирования и абстракциями данных, сразу же определит абстракции низкого уровня и воплотит их так, чтобы они не зависели от конкретного применения. То есть абстракции низкого уровня — это инструменты, которые могут переноситься из одного проекта в другой. С другой стороны, в традиционных языках программирования абстракция высокого уровня должна базироваться на конкретной структуре данных. Поэтому трудно переносить абстракции высокого уровня из одного проекта в другой. Переносится только общая идея. Как следствие, даже простейшие абстракции высокого уровня (вычисление длины списка или поиск в таблице определенного значения) зачастую переписываются заново для каждого нового приложения.

Сила полиморфизма состоит в том, что он позволяет записывать алгоритмы высокого уровня лишь однажды и затем повторно их использовать с различными абстракциями низкого уровня. Даже относительно сложные алгоритмы могут быть записаны в виде схемы или шаблона и применяться во многих приложениях. Мы будем обсуждать шаблоны в последующих главах, когда рассмотрим механику полиморфизма.

## 14.2. Разновидности полиморфизма

В объектно-ориентированных языках программирования полиморфизм — естественное следствие:

- отношения «быть экземпляром»;
- механизма пересылки сообщений;
- наследования;
- принципа подстановки.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Одно из важнейших достоинств объектно-ориентированного подхода состоит в возможности комбинирования этих средств. В результате получается богатый набор технических приемов совместного и многократного использования кода.
Чистый полиморфизм имеет место, когда одна и та же функция применяется к аргументам различных типов. В случае чистого полиморфизма есть одна функция (тело кода) и несколько ее интерпретаций. Другая крайность наблюдается, когда имеется множество различных функций (то есть тел кода) с одним именем. Такая ситуация называется перегрузкой или полиморфизмом ad hoc. Между этими двумя полюсами лежат переопределяемые и отложенные методы [1].

## 14.3. Полиморфные переменные

За исключением случаев перегрузки полиморфизм в объектно-ориентированных языках программирования возможен только за счет существования полиморфных переменных. Полиморфная переменная многолика: она содержит значения, относящиеся к различным типам данных. Полиморфные переменные реализуют принцип подстановки. Другими словами, хотя для такой переменной имеется ожидаемый тип данных, фактический тип может быть подтипом ожидаемого типа.

В языках с динамическим связыванием (Smalltalk, Objective-C) все переменные являются потенциально полиморфными (любая переменная может содержать значение любого типа). В этих языках от типа требуется только некий набор ожидаемых действий. Например, массив — это то, что по индексу поставляет значение. Иными словами, пользователь может определить свой собственный тип данных (скажем, разреженный массив), и если операции индексирования определены с теми же именами, то новый тип данных может использоваться в уже существующем алгоритме.

В языках со статическими типами данных (таких, как C++, Java, Object Pascal и Objective-C при использовании статических описаний) ситуация немного сложнее. Мы отмечали, что эти языки рассматривают создание подклассов как порождение подтипов данных. Полиморфизм существует в этих языках благодаря различию между декларированным (статическим) классом переменной и фактическим (динамическим) классом значения, которое содержится в переменной. Как мы отметили в главе 10, это достигается через отношение «быть подклассом». Переменная может содержать значение объявленного типа или любого его подтипа.

В языках Object Pascal и Java это справедливо для всех переменных, описанных с типом данных object. В C++ и Objective-C с использованием статических описаний полиморфные переменные существуют только как указатели и ссылки. Опять же, как уже было отмечено в главе 10, когда указатели не используются, динамический класс переменной всегда приводится к ее статическому классу.

---

[1] Опять-таки отметим, что согласие в среде сообщества программистов относительно используемой терминологии весьма невелико. Например, в работах [Horovitz 1984], [Marcotty 1987], [MacLennan 1987] и [Pinson 1988] термин полиморфизм определяется так, что он приблизительно соответствует понятию, называемому в данной книге перегрузкой. В работах [Sethi 1989] и [Meyer 1988a], а также в среде людей, занимающихся функциональным программированием [Wikstrom 1987], [Milner 1990], этот термин резервируется для обозначения того, что здесь называется чистым полиморфизмом. Другие же авторы используют этот термин для обозначения одного-двух или всех механизмов полиморфизма, рассматриваемых в данной главе. Два законченных, но запугивающих избытком технических подробностей анализа могут быть найдены в работах [Cardelli 1985] и [Danforth 1988].

PDF created with pdfFactory Pro trial version www.pdffactory.com

Хорошим примером полиморфной переменной является массив allPiles в карточном пасьянсе из главы 8. Массив был описан как содержащий значения типа CardPile, но на самом деле он хранит значения, принадлежащие подклассам родительского класса. Сообщение (например, показанное ниже сообщение display), передаваемое к элементу этого массива, выполняет метод, связанный с динамическим типом переменной, а не со статическим классом.

```
public class Solitare extends Applet
{
  ...
  static CardPile allPiles[];
  ...
  public void paint(Graphics g)
  {
    for (int i = 0; i < 13; i++)
    {
      allPiles[i].display(g);
    };
  }
  ...
}
```

## 14.4. Перегрузка

Мы говорим, что имя функции перегружено, если имеются два (и более) кода, связанные с этим именем. Заметьте, что перегрузка является обязательной частью переопределения методов, которое рассматривалось в главе 11 (и будет анализироваться снова в следующем разделе), но эти два термина не идентичны, и перегрузка может происходить без переопределения.

При перегрузке полиморфным является имя функции — оно многозначно. Еще один способ представить себе перегрузку и полиморфизм: вообразите, что есть единая абстрактная функция, которая вызывается с аргументами различного типа, а фактический выполняемый код зависит от типа аргументов. Тот факт, что компилятор часто может определить правильную функцию на этапе компиляции (в языках со строгим контролем типов данных) и, следовательно, сгенерировать только нужный код — это просто оптимизация.

### 14.4.1. Перегрузка в реальной жизни

В главе 1 мы встретились с ситуацией, когда перегрузка возникла без переопределения. Помните, я хотел сделать сюрприз моей бабушке и послать ей цветы на день рождения? Одно возможное решение состояло в том, чтобы передать сообщение sendFlowersTo хозяйке цветочного магазина. Согласно другому плану то же самое сообщение следовало послать моей жене. Как хозяйка цветочного магазина, так и моя жена должны были понять сообщение, и обе стали бы как-то действовать, чтобы получить желаемый результат. В определенном смысле я могу думать о сообщении sendFlowersTo как об одной функции, понимаемой как моей женой, так и хозяйкой цветочного магазина. Однако они будут использовать различные алгоритмы в своих действиях.

Заметьте, что в данном примере нет наследования (а следовательно, и переопределения). Первый общий надкласс для хозяйки цветочного магазина и моей жены — это категория Human (человек). Но реагировать на сообщение sendFlowersTo свойственно не всем

PDF created with pdfFactory Pro trial version www.pdffactory.com

людям. К примеру, мой дантист, который несомненно является человеком, вообще не поймет данное сообщение.

### 14.4.2. Перегрузка и приведение типа

В качестве примера более близкого к языкам программирования, рассмотрим разработку библиотеки классов, представляющих структуры данных общего вида. Ряд структур (множества, наборы, словари, очереди с учетом приоритетов) может быть использован для хранения совокупности элементов. Для каждой структуры будет определен метод add, который добавляет к ней новый элемент.

Итак, две совершенно различные функции используются для выполнения семантически аналогичных действий над различными типами данных. Такая ситуация часто встречается в программировании, и не только объектно-ориентированном. Типичный пример — перегрузка оператора сложения «+». Код, генерируемый компилятором при сложении целых чисел, часто радикальным образом отличается от кода с плавающей точкой. Программист, однако, думает об этих операциях как о единой сущности — функции «сложение».

В этом примере важно отметить, что здесь происходит не только перегрузка. Семантически отдельная операция — приведение типа — также обычно ассоциируется с арифметическими действиями. Приведение происходит, когда значение одного типа преобразуется в значение другого. Если разрешены арифметические действия со смешанными операндами, то сложение двух значений может интерпретироваться несколькими способами:

- Имеются четыре различные функции, которые соответствуют операциям «целое + целое», «целое + вещественное», «вещественное + целое», «вещественное + вещественное». В этом случае есть перегрузка, но нет приведения типа.
- Есть две различные функции: «целое + целое» и «вещественное + вещественное». Для операций «целое + вещественное» и «вещественное + целое» целое значение приводится к вещественному. В таком случае наблюдаются комбинация перегрузки и приведения типа.
- Есть только одна функция сложения: «вещественное + вещественное». Все аргументы приводятся к типу данных «вещественное число». В этом случае нет перегрузки, а есть только приведение типов.

### 14.4.3. Перегрузка не подразумевает сходство

В определении перегрузки совершенно не подразумевается, что функции, связанные с перегруженным именем, имеют какое-либо семантическое сходство. Рассмотрим, например, пасьянс из главы 8. Метод draw использовался для того, чтобы нарисовать на экране карту. В другом приложении метод draw будет применяться к стопке карт, только на этот раз он будет разыгрывать верхнюю карту колоды. Этот метод draw даже отдаленно не похож с точки зрения семантики на метод draw, определенный для одной карты, и тем не менее они имеют общее имя.

Заметьте, что данная перегрузка одного и того же имени независимыми и не имеющими отношение друг к другу значениями не обязательно является плохим стилем программирования. Как правило, это не вносит путаницы. На самом деле выбор короткого, ясного и значимого имени (вроде add, draw и т. д.) значительно улучшает и облегчает использование объектно-ориентированных компонент. Проще запомнить, что

PDF created with pdfFactory Pro trial version www.pdffactory.com

вы добавляете элемент через метод add, а не вспоминать что-нибудь вроде addNewElement или вызывать процедуру Set_Module_Addition_Method.

Все объектно-ориентированные языки, которые мы рассматриваем, разрешают использовать методы с одинаковыми именами в не связанных между собою классах. В этом случае привязка перегруженного имени производится за счет информации о классе, к которому относится получатель сообщения. Тем не менее это не означает, что могут быть написаны функции или методы, которые принимают произвольные аргументы. Природа языков C++ и Object Pascal, осуществляющих строгий контроль типов данных, по-прежнему требует описания всех имен.

### 14.4.4. Параметрическая перегрузка

Другой стиль перегрузки, при котором процедурам (функциям, методам) в одном и том же контексте разрешается использовать совместно одно имя, а двусмысленность снимается за счет анализа числа и типов аргументов, называется параметрической перегрузкой. Она присутствует в C++ и Java, а также в некоторых директивных языках (например, Ada) и во многих языках, основанных на функциональной парадигме. Мы уже видели примеры такой перегрузки для функций-конструкторов. C++ позволяет любому методу, функции, процедуре или оператору быть параметрически перегруженными, коль скоро аргументы таковы, что выбор может быть произведен однозначно на этапе компиляции. (При автоматическом приведении типа — например, от символов character к целым числам integer или от целых integer к числам с плавающей точкой float — алгоритм, используемый для разрешения неоднозначности в имени перегруженной функции, становится очень сложным. Более подробная информация может быть найдена в работах [Ellis 1990] и [Stroustrup 1986].)

Перегрузка присутствует во всех других формах полиморфизма, которые мы рассматриваем: переопределение, отложенные методы, чистый полиморфизм. Она также часто полезна при «сужении концептуального пространства», то есть при уменьшении количества информации, которую необходимо помнить программисту. Часто эта забота о памяти программиста не менее важна, чем снижение требований к памяти компьютера, достигаемое при совместно используемом коде.

## 14.5. Переопределение

В главе 11 мы описали механику переопределения методов (уточнение и замещение) в различных рассматриваемых нами объектно-ориентированных языках программирования. Поэтому не будем повторяться. Вспомним, однако, следующие существенные элементы этой техники. В одном из классов (обычно в абстрактном надклассе) имеется определенный для конкретного сообщения общий метод, который наследуется и используется подклассами. Однако по крайней мере в одном подклассе определен метод с тем же именем. Это перекрывает доступ к общему методу для экземпляров данного подкласса (или же в случае уточнения делает доступ к методу многоступенчатым). Мы говорим, что второй метод переопределяет первый.

Переопределение часто происходит прозрачно для пользователя класса, и, как и в случае перегрузки, две функции представляются семантически как одна сущность.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 14.5.1. Переопределение в классе Magnitude

Интересным примером переопределения методов является класс Magnitude в системе Little Smalltalk. Magnitude — это абстрактный надкласс, который имеет дело с величинами, обладающими по крайней мере частичным (если не полным) упорядочиванием. Числа — это наиболее характерный пример объектов, обладающих «величиной», хотя время и дата также могут быть упорядочены, равно как и символы, точки на двухмерной координатной плоскости, слова в словаре и т. д.

В классе Magnitude шесть отношений сравнения определяются следующим образом:

```
<= arg
   self < arg or: [ self = arg ]
>= arg
   arg <= self
< arg
   self <= arg and: [ self ~= arg ]
> arg
   arg < self
= arg
   self == arg
~= arg
   (self = arg) not
```

Заметьте, что определения цикличны: каждое из них зависит от некоторых других. Как можно избежать бесконечного цикла при вызове какого-либо конкретного сравнения? Ответ состоит в том, что подклассы класса Magnitude должны переопределять по крайней мере одно из шести сообщений сравнения. Мы оставляем в качестве упражнения для читателя проверку того, что если переопределены сообщения = и либо &lt:, либо >=, то оставшиеся операторы не приведут к бесконечному циклу.

Переопределение методов вносит свой вклад в совместное использование кода, поскольку экземпляры классов, которые не переопределяют данный метод, могут использовать одну копию оригинального кода. Только в тех случаях, когда метод не подходит, создается альтернативный фрагмент кода. Без переопределения методов было бы необходимо для всех подклассов создавать их собственные методы для реагирования на сообщение, даже если большинство методов идентично.

Пользователи языка C++ должны быть осведомлены о тонком семантическом различии между переопределениями виртуального и невиртуального методов. Мы будем обсуждать это более подробно в разделе 14.9.

## 14.6. Отложенные методы

Отложенный метод (иногда называемый абстрактным методом, а в C++ — чисто виртуальным методом) может рассматриваться как обобщение переопределения. В обоих случаях поведение родительского класса изменяется для потомка. Для отложенного метода, однако, поведение просто не определено. Любая полезная деятельность задается в дочернем классе.

Одно из преимуществ отложенных методов является чисто концептуальным: программист может мысленно наделить нужным действием абстракцию сколь угодно высокого уровня. Например, для геометрических фигур мы можем определить метод draw, который их рисует: треугольник Triangle, окружность Circle и квадрат Square. Мы определим

PDF created with pdfFactory Pro trial version www.pdffactory.com

аналогичный метод и для родительского класса Shape. Однако такой метод на самом деле не может выполнять полезную работу, поскольку в классе Shape просто нет достаточной информации для рисования чего бы то ни было. Тем не менее присутствие метода draw позволяет связать функциональность (рисование) только один раз с классом Shape, а не вводить три независимые концепции для подклассов Square, Triangle и Circle.

Имеется и вторая, более актуальная причина использования отложенных методов. В объектно-ориентированных языках программирования со статическими типами данных (C++, Object Pascal) программист имеет право послать сообщение объекту, только если компилятор может определить, что действительно имеется метод, который соответствует селектору сообщения. Предположим, что программист хочет определить полиморфную переменную класса Shape, которая будет в различные моменты времени содержать фигуры различного типа. Это допустимо в соответствии с принципом подстановки. Тем не менее компилятор разрешит использовать метод draw для переменной, только если он сможет гарантировать, что сообщение будет распознаваться в классе переменной. Присоединение метода draw к классу Shape эффективно обеспечивает такую гарантию, даже если метод draw для класса Shape на самом деле никогда не выполняется.

## 14.7. Чистый полиморфизм

Многие авторы резервируют понятие полиморфизм (или чистый полиморфизм) для ситуаций, когда одна функция используется с разными наборами аргументов, и термин перегрузка — для случая, когда определено несколько функций с одним именем1. Эти термины не ограничены исключительно объектно-ориентированным подходом. Например, в языках Lisp и ML легко написать функции, которые обрабатывают списки с различными элементами. Такие функции являются полиморфными, поскольку тип аргумента неизвестен при определении функции. Полиморфные функции — это одна из наиболее мощных объектно-ориентированных техник программирования. Они позволяют единожды писать код на высоком уровне абстрагирования и затем применять его в конкретной ситуации. Обычно программист выполняет подгонку кода с помощью посылки дополнительных сообщений получателю, использующему метод. Эти дополнительные сообщения часто не связаны с классом на уровне абстракции полиморфного метода. Они являются виртуальными методами, которые определяются для классов более низкого уровня.

Следующий пример поможет проиллюстрировать эту концепцию. Как мы отметили в разделе 14.5, посвященному переопределению методов, класс Magnitude в языке Smalltalk — это абстрактный надкласс, который имеет дело с упорядоченными величинами. Рассмотрим метод с именем between:and:, приведенный ниже:

```
between: low and: high
        " проверить, находится ли получатель "
        " между двумя крайними точками "
    ( low <= self ) and: ( self <= high )
```

Этот метод определен в классе Magnitude и проверяет (как это сказано в комментарии), находится ли получатель между заданными точками. Проверка осуществляется посылкой сообщения «меньше или равно» нижней границе с получателем в качестве аргумента и передачей того же сообщения получателю с верхней границей в качестве аргумента (в языке Smalltalk все операторы интерпретируются как сообщения). Только если оба эти выражения дают true, считается, что получатель попадает в заданный интервал.

PDF created with pdfFactory Pro trial version www.pdffactory.com

После того как объекту послано сообщение between:and: с двумя аргументами, фактическое выполнение зависит от конкретного смысла сообщения «меньше или равно». Данное сообщение, хотя оно и определено для класса Magnitude, переопределяется в большинстве подклассов. Для целочисленных значений смысл сообщения — сравнить целые числа. Тем самым сообщение between:and: может использоваться для проверки попадания целого числа в заданный интервал. Для значения с плавающей точкой все происходит аналогично:

```
anInteger between: 7 and: 11
aFloat between: 2.7 and: 3.5
```

Для символов соотношение «меньше или равно» сравнивает ASCII коды. Соответственно сообщение between:and: проверяет, лежит ли символ в интервале между двумя другими символами. Например, чтобы узнать, является ли символ aChar строчной буквой, мы можем использовать следующее выражение (лексема $a обозначает в языке Smalltalk символ a):

```
aChar between: $a and: $z
```

Для точек Points сравнение «меньше или равно» возвращает true, если получатель расположен выше и левее аргумента (то есть и первая и вторая координаты получателя удовлетворяют соотношению «меньше или равно» при сравнении с соответствующими координатами точки-аргумента). Points — базовые объекты языка Smalltalk. Они конструируются из целых чисел с помощью оператора @. Число-получатель становится первой координатой, а число-аргумент — второй. Заметьте, что определение соотношения «<» (меньше) для точек дает лишь частичное упорядочивание. Не все точки являются соизмеримыми. Тем не менее выражение

```
aPoint between: 2@4 and: 12@14
```

дает true, если точка aPoint лежит в прямоугольнике с координатами (2,4) для левого верхнего угла и (12,14) для правого нижнего угла.

Важный момент здесь — это то, что во всех случаях используется только один метод between:and:. Он является полиморфным и работает с аргументами многих типов. В каждом случае переопределение сообщений, вызываемых полиморфной подпрограммой (сообщения «меньше или равно»), приспосабливает код к конкретным обстоятельствам.

В главе 18 мы встретим много новых примеров полиморфных подпрограмм, когда будем обсуждать шаблоны.

## 14.8. Обобщенные функции и шаблоны

Совершенно другой тип полиморфизма обеспечивается за счет так называемых обобщенных функций, которые в языке C++ называются шаблонами. Аргументом обобщенной функции (класса) является тип, который используется при ее (его) параметризации. Аналогия с обычными функциями очевидна: последние реализуют необходимый алгоритм без задания конкретных числовых значений. Чтобы проиллюстрировать понятие обобщенной функции, вернемся к началу этой главы. Там мы отметили, что проблема с языками со строгим контролем типов данных состоит в том, что они не разрешают создавать тип вроде Linked List of X (связанный список из объектов X), где идентификатор X — это неизвестный тип данных. Обобщенные функции обеспечивают такую возможность.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Для обобщенных функций или классов аргумент — это тип данных. Он может использоваться внутри определения класса, как если бы он уже был определен, хотя никакие свойства этого типа данных не известны компилятору при считывании описания класса. Далее при определении конкретного объекта параметр-тип связывается с реальным типом данных. Например, связный список может быть описан в языке C++ следующим образом:

```
template  class List
  {
    public:
      void add(T);
      T firstElement();
      // поля данных
      T value;
      List * nextElement;
  };
```

В этом примере идентификатор T используется как обозначение типа. Каждый экземпляр класса List содержит значение типа T и указатель на следующий элемент списка. Функция-член add добавляет новый элемент в список. Первый элемент в списке возвращается функцией firstElement.

Чтобы создать экземпляр класса, пользователь должен обеспечить значение типа данных для параметра T. Следующие команды создают список целых чисел и список чисел с плавающей точкой:

```
List aList;
List bList;
```

Функции (включая функции-члены класса) также могут иметь описания-шаблоны. Ниже приводится описание функции, которая находит количество элементов в списке независимо от его типа.

```
template  int length (List & aList)
  {
    if (aList == 0) return 0;
    return 1 + length(aList.nextElement);
  }
```

В C++ функции-шаблоны интенсивно используются в стандартной библиотеке шаблонов, которая описывается в главе 16.

## 14.9. Полиморфизм в различных языках

В этом разделе обсуждаются механизмы реализации полиморфизма в различных языках программирования.

### 14.9.1. Полиморфизм в C++

Полиморфизм часто является источником затруднений для изучающих C++. Поэтому остановимся на этом вопросе подробнее.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## Полиморфные переменные

Как мы отметили в главе 10, в языке C++ истинные полиморфные переменные возникают только при использовании указателей или ссылок. Когда настоящей переменной (то есть не указателю и не ссылке) присваивается значение типа подкласса, то динамический класс значения вынужденно приводится так, чтобы совпадать со статическим типом переменной.

Однако при использовании указателей или ссылок значение сохраняет свой динамический тип. Чтобы понять этот процесс, рассмотрим следующие два класса:

```
class One
{
  public:
    virtual int value()
    {
      return 1;
    }
};
class Two : public One
{
  public:
    virtual int value()
    {
      return 2;
    }
};
```

Класс One описывает виртуальный метод, который возвращает значение 1. Этот метод переопределяется в классе Two на метод, возвращающий значение 2.

Определяются следующие функции:

```
void directAssign (One x)
{
  printf("by assignment value is %d\n",
      x.value());
}
void byPointer (One * x)
{
  printf("by pointer value is %d\n",
      x->value());
}
void byReference (One & x)
{
  printf("by reference value is %d\n",
      x.value());
}
```

Эти функции используют в качестве аргумента значение класса One, которое передается соответственно по значению, через указатель и через ссылку. При выполнении этих функций с аргументом класса Two для первой функции параметр преобразуется к классу One, и в результате будет напечатано значение 1. Две другие функции допускают полиморфный аргумент. В обоих случаях переданное значение сохранит свой динамический тип данных, и напечатано будет значение 2.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## Виртуальное и невиртуальное переопределение

Приводящий в замешательство аспект переопределения методов в языке C++ — это разница между переопределением виртуального и невиртуального методов. Как мы отмечали в главе 11, ключевое слово virtual не является необходимым для того, чтобы происходило переопределение. Однако семантический смысл сильно меняется в зависимости от того, используется это слово или нет. Если удалить ключевое слово virtual из описания метода в классе One в предыдущем примере (даже если его сохранить в классе Two), то результат «1» будет напечатан для всех трех функций.

Без ключевого слова virtual динамический тип переменной (даже для указателей и ссылок) игнорируется, когда переменная используется как получатель соответствующего сообщения.

Еще большее замешательство возникает, если программист пытается переопределить виртуальную функцию в подклассе, но при этом указывает (возможно, по ошибке) другой тип аргументов. Например, родительский класс содержит описание

```
virtual void display (char *, int);
```

Подкласс пытается переопределить метод:

```
virtual void display (char *, short);
```

Поскольку списки аргументов различаются, то второе определение не распознается как переопределение. Это приводит к тому, что виртуальное переопределение рассматривается как обычное (невиртуальное) определение метода. Поэтому, например, при вызове в форме родительского типа будет выбираться первый метод, а не второй. Такие ошибки чрезвычайно трудноуловимы, поскольку обе формы записи допустимы и надеяться на диагностику компилятора не приходится.

Возникает естественный вопрос: почему разрешены обе формы записи и почему не всякое переопределение виртуально? Имеются по крайней мере два правдоподобных объяснения. Первое состоит в том, что за редкими исключениями невиртуальная форма переопределения — это именно то, что хочет программист, и обязательная виртуальность была бы только помехой. Второе и более убедительное соображение относится к эффективности. Виртуальное переопределение всегда более затратно с точки зрения выполнения программы, чем невиртуальное. Принцип, проходящий красной нитью через весь C++, состоит в том, что если какое-то свойство языка не востребовано или же используется только для конкретной проблемы, то программист не должен расплачиваться (на этапе выполнения) за его существование. Соответственно, если виртуальное наследование не требуется или же используется лишь в специальном случае, никаких дополнительных накладных расходов быть не должно. Только в тех случаях, когда программист в явном виде утверждает, что ему требуется виртуальная функция, подключается дополнительная надстройка.

## Параметрическая перегрузка

Язык C++ позволяет нескольким функциям иметь одно имя внутри любого контекста до тех пор, пока списки аргументов функций различаются в достаточной степени, чтобы компилятор недвусмысленно определял, какую именно функцию намереваются вызвать. Такая ситуация, как правило, возникает при использовании нескольких конструкторов для

PDF created with pdfFactory Pro trial version www.pdffactory.com

одного и того же класса, каждый из которых имеет свой набор аргументов. Однако таким образом могут описываться любые функции, методы или операции.

Правила для снятия двусмысленности с перегруженных функций являются довольно тонкими, в особенности если разрешено автоматическое приведение типов данных. Один из наиболее важных принципов, которые необходимо помнить: при поиске функции компилятор просматривает наиболее узкую область видимости, в которой определено имя функции, и затем в пределах этой области ищет подходящую функцию, основываясь на типе аргументов. То есть одна функция может скрыть другую с тем же именем, определенную в более широкой области видимости.

### Отложенные методы в C++

В языке C++ отложенный метод (который здесь называется чисто виртуальным методом) должен быть описан в явном виде с ключевым словом virtual. Тело отложенного метода не определяется, вместо этого функции «присваивается» значение 0:

```
class Shape
{
  public:
    ...
    virtual void draw() = 0;
    ...
};
```

Компилятор не разрешает пользователю создавать экземпляр класса, который содержит чисто виртуальные методы. Подклассы должны эти методы переопределять. Переопределение чисто виртуального метода должно произойти при описании его потомков, для которых создаются реальные объекты.

### Обобщенные функции и шаблоны

Обобщенные функции и классы в языке C++ реализуются с помощью ключевого слова template (шаблон). Пример класса-шаблона был приведен выше. Классы-шаблоны и функции-шаблоны интенсивно используются стандартной библиотекой шаблонов языка C++, которую мы обсуждаем в главе 16.

## 14.9.2. Полиморфизм в Java

Язык Java поддерживает как иерархию подклассов (с ключевым словом extends), так и иерархию подтипов (с ключевым словом interfaces). Переменные могут быть объявлены или через класс, или через интерфейс. Все переменные являются полиморфными. Переменная, описанная как класс, может содержать значения, относящиеся к любому подклассу. Переменная, объявленная как интерфейс, может хранить значения любого класса, который реализует этот интерфейс.

Отложенные методы реализуются в языке Java через ключевое слово abstract. Методы, описанные как abstract, не имеют тела; они оканчиваются точкой с запятой. Абстрактные методы должны переопределяться в подклассах. Класс, который включает в себя абстрактный метод, должен в свою очередь быть описан как абстрактный. Не разрешается создавать экземпляры абстрактных классов.

```
abstract class shape
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
{
        // ниже должно переопределяться
        public abstract draw();
        // ...
}
class triangle extends shape
{
        public draw()
        {
        // нарисовать треугольник
        }
        // ...
}
```

Интересным свойством языка Java является модификатор final, который в некотором смысле противоположен ключевому слову abstract. Класс или метод, описанный как final, не может порождать подклассы или переопределяться.

### 14.9.3. Полиморфизм в Object Pascal

### Полиморфные переменные

В языке Object Pascal все переменные потенциально полиморфны при неявном предположении, что подклассы представляют собой подтипы. Все переменные хранят значение или объявленного класса, или его подкласса.

### Отложенные методы в Object Pascal

Как мы отметили в главе 7, версии Apple и Borland языка Object Pascal отличаются тем, как они указывают на переопределение метода. В Object Pascal версии Apple ключевое слово override помещается в описание метода дочернего класса. Версия Delphi требует ключевых слов override и virtual в описании метода в родительском классе.

Язык Object Pascal версии Apple не поддерживает отложенные методы. Они реализуются в виде процедуры, генерирующей сообщение об ошибке:

```
type
  Shape = object
    corner : Point;
    procedure draw();
    ...
  end;
  Circle = object (Shape)
    radius : integer;
    procedure draw(); override;
    ...
  end;
procedure Shape.draw();
  begin
     writeln('descendant should define draw');
     halt();
  end;
```

В языке Delphi Pascal метод может быть объявлен как отложенный с ключевым словом abstract, следующим за ключевым словом virtual (или dynamic) при описании в родительском классе. Для абстрактного метода не задается тело. В отличие от C++ можно

PDF created with pdfFactory Pro trial version www.pdffactory.com

создать объект, класс которого имеет все еще не переопределенные абстрактные методы. То есть язык Delphi поддерживает абстрактные методы, но не абстрактные классы.

```
type
  class TShape
    procedure draw; virtual; abstract;
    ...
  end;
  class TTriangle (TShape)
    procedure draw; override;
    ...
  end;
```

### 14.9.4. Полиморфизм в Objective-C

#### Полиморфные переменные

При описании с ключевым словом id все переменные в Objective-C полиморфны и поэтому могут содержать любое значение. При описании с конкретным классом переменные имеют все свойства (хорошие и не очень) переменных языка C++.

#### Отложенные методы в Objective-C

Не требуется специального указания для описания отложенного метода в языке Objective-C. Чтобы помочь в создании таких методов, в классе Object определяется сообщение subclassResponsibility (которое тем самым доступно для всех объектов). Оно просто печатает строку, показывающую, что выполняемое действие должно быть переопределено в подклассе.

Отложенный метод draw для класса Shape может быть записан, к примеру, следующим образом:

```
@implementation Shape : Object
...
- draw { return [ self subclassResponsibility ]; }
...
@end
```

### 14.9.5. Полиморфизм в Smalltalk

#### Полиморфные переменные

Поскольку Smalltalk — это язык с динамическими типами данных, все переменные являются полиморфными. Они могут хранить любое значение.

#### Отложенные методы в Smalltalk

В языке Smalltalk не требуется специального указания, чтобы описать отложенный метод. В классе Object определено сообщение subclassResponsibility (тем самым доступное любым другим объектам). Оно печатает строку, показывающую, что для некоторого подкласса вызывается действие, которое должно быть переопределено.

Отложенный метод draw для класса Shape может быть записан следующим образом:

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
draw
  " дочерние классы должны переопределять этот метод "
   self subclassResponsibility
```

### 14.10. Эффективность и полиморфизм

Программирование всегда сводится к компромиссу. В частности, полиморфизм подразумевает компромисс между простотой разработки и использования, читаемостью кода и эффективностью. В значительной степени эффективность уже была нами рассмотрена: выяснилось, что потери в эффективности не столь велики. Однако было бы непростительным не принимать полностью во внимание этот момент.

Функция (подобная методу between:and:, описанному в предыдущем разделе), которая не знает тип своего аргумента, вряд ли будет столь же эффективна, как функция, владеющая полной информацией. Сравнительный тест может обнаружить лишь несколько дополнительных команд на языке ассемблера в случае, если аргументом является целое число. Если же аргумент — это объект-точка, то требуются гораздо более продолжительные действия. Тем не менее преимущества быстрой разработки, самосогласованного поведения приложения и возможность многократно использовать программный код обычно значат больше, чем небольшие потери в эффективности.

### Упражнения

1. Как вы думаете, следует ли рассматривать значение nil в языке Pascal, или аналогичную величину NULL в C как полиморфный объект? Обоснуйте свой ответ.
2. Какие еще операции (за исключением арифметических) обычно являются перегруженными в традиционных языках программирования (Pascal и C)?
3. Проведите трассировку методов и классов при вычислении выражения:

   ```
   anInteger between: 7 and: 11
   ```

4. Предположим, что в языке Smalltalk имеются два класса: яблоки Apple и апельсины Orange, которые являются подклассами фруктов Fruit. Какой минимальный объем кода потребуется для сравнения яблок и апельсинов?

## *Глава 15: Учебный пример: контейнерные классы*

Почти все нетривиальные компьютерные программы базируются на простых структурах данных, таких как связные списки, стеки, очереди, деревья, множества, словари. Поскольку эти структуры являются типичными, хорошо было бы иметь их в качестве многократно используемых компонентов. На самом деле можно создать такие компоненты, но возникающие при этом сложности часто очень значительны. Поэтому разработка многократно используемых контейнерных классов является хорошим учебным примером. Он иллюстрирует, как свойства языка программирования влияют на стиль разработки, а также демонстрирует некоторые достоинства и ограничения объектно-ориентированных методов.

Далее мы рассмотрим три тесно связанных вопроса:

- Можно ли сконструировать многократно используемую абстракцию контейнера данных общего назначения, которая не зависит от типа хранимых элементов и тем самым переносима из одного проекта в другой?

PDF created with pdfFactory Pro trial version www.pdffactory.com

- Должен ли такой контейнер содержать данные только одного типа (так называемые однородные контейнеры) или удастся построить контейнер, содержащий значения различного типа (разнородные контейнеры)?
- Можно ли разрешить пользователю контейнера данных доступ к хранимым в нем элементам, гарантируя при этом защиту от удаления и маскировку внутренней реализации контейнера?

## 15.1. Использование традиционных подходов

Чтобы увидеть проблему в перспективе, мы должны сперва рассмотреть, как структуры данных обычно реализуются в традиционных языках программирования (скажем, C и Pascal). Используем связный список целых чисел как пример моделируемой абстракции. В языке Pascal связный список образуется из записей двух типов. Первый тип — это начало списка, который содержит указатель на первый элемент:

```
type
  List = record
    firstLink :  Link;
  end;
```

Начало (голова) списка может быть размещено статически, поскольку размер требуемой памяти (а именно единственный указатель) остается постоянным во время выполнения. Второй тип записей используется, чтобы хранить сами значения. Каждый узел Link содержит одно целое число и указатель на следующий элемент списка:

```
type
  Link = record
    value : integer;
    nextElement :  Link;
  end;
```

Элементы должны размещаться и удаляться динамически, хотя такие подробности следует спрятать от пользователя. Это достигается с помощью разработки функций, которые добавляют значение в начало списка, возвращают первый элемент списка, удаляют его и т. д.

```
procedure addToList(var aList : List,
          newVal : integer);
(* добавляет новый элемент в список *)
var
  newLink :  Link;
begin
  (* создать и проинициализировать новый элемент *)
  new(newLink);
  newLink.value := newVal;
  (* поместить его в начало списка *)
  newLink.nextElement := aList.firstLink;
  aList.firstLink := newLink;
end;
function firstElement (var aList : List) : integer;
(* удаляет из списка и возвращает первый элемент *)
var
  firstNode :  Link;
begin
  firstNode := aList.firstLink;
  firstElement := firstNode^.value;
  aList.firstLink := firstNode^.nextElement;
  dispose(firstNode);
end;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Главное здесь не детали реализации связного списка (их можно найти в любом учебнике по структурам данных), но возможности многократного использования. Предположим, что программист реализовал абстракцию связного списка, приведенную выше. Теперь он хочет использовать наряду со связным списком целых чисел связный список вещественных чисел.

Проблема состоит в том, что язык программирования слишком строго проверяет типы данных. Тип integer, используемый для значения хранимого в списке, является неотъемлемой составной частью описания. Единственный способ ввести новый тип данных — это создать совершенно новую структуру RealLink, новую структуру начала списка RealList и подпрограммы для доступа к этим структурам данных.

Нечто вроде записей с вариантными полями (тип данных union в языке C) может помочь использовать одну и ту же абстракцию списка для хранения как целых, так и вещественных чисел. В действительности вариантные записи позволяют определить разнородный список, который содержит и целые числа, и числа с плавающей точкой. Но вариантные записи — это только часть решения проблемы. Нельзя определить функцию, которая возвращает вариантную запись, так что по-прежнему требуется создавать отдельные функции для получения первого элемента списка. Более того, вариантная запись имеет только конечный набор допустимых альтернативных вариантов. Что если для следующего проекта потребуется совершенно новый тип списка (например, с текстовыми строками)?

Теперь рассмотрим проблему доступа к произвольному элементу без удаления предшествующих ему элементов из контейнера. Типичный цикл, который печатает значения из списка, выглядит примерно так:

```
var
  aList : List;        (* обрабатываемый список *)
  p   : Link;  (* указатель, используемый в цикле *)
begin
  ...
  p := aList.firstLink;
  while (p <> nil) do
  begin
    writeln(p.value);
    p := p.nextElement;
  end;
  ...
```

Заметьте, что для прохождения цикла необходимо было ввести дополнительную переменную, названную здесь p. Она должна принадлежать типу данных Link, который мы намеревались замаскировать. Точно так же сам цикл требует доступа к полям переменной Link, которые мы также не хотели бы открывать.

Итак, как мы видим, в традиционных языках программирования с контролем типов нет средств, необходимых для создания и обработки контейнеров, которые были бы истинно многократно используемыми.

## 15.2. Контейнеры в динамических языках

Создание многократно используемых абстракций контейнеров происходит намного проще в языках программирования с динамическими типами данных (Smalltalk, Objective-C). На самом деле такие языки обычно уже содержат большой набор готовых абстракций

PDF created with pdfFactory Pro trial version www.pdffactory.com

данных, тем самым освобождая программиста от необходимости создавать контейнеры. Как мы видели выше при обсуждении вопроса о времени связывания, в языках программирования с динамическими типами данных знание о типе хранит в себе значение, а не переменная, с помощью которой осуществляется доступ к значению. Например, наша абстракция связного списка может быть определена через следующие структуры языка Objective-C:

```
@ interface List : Object
{
  id firstLink;
}
- (void) addToList : value
- id   firstElement
@ end
@ interface Link : Object
{
  id Value
  id NextElement
}
+
- id value
- id nextElement
@ end
```

О значении, помещаемом в такую структуру, известно, что оно типа id (то есть типа данных «объект»). Аналогично значение, извлекаемое из списка, тоже типа id, но оно может быть присвоено любой объектной переменной, поскольку таким переменным разрешается хранить объект произвольного типа.

Чтобы создать новый список, программист посылает сообщение new фабрике объектов в классе List:

```
id aList
...
aList = [ List new ];
```

Чтобы поместить значение в список, программист использует соответствующую функцию-член:

```
[ aList addToList: aValue ];
```

Операции со списком не требуют никаких дополнительных знаний о типе значений, которые содержатся в списке, за исключением того, что это объекты:

```
@ implementation List
- (void) addList: newElement
 /* добавить в список новый элемент */
 {
   id newLink;
   newLink = [ Link new ];
   [ newLink setValue: newElement link: firstLink ];
   firstLink = newLink;
 }
- id firstElement
 /* удалить и вернуть первый элемент списка */
 {
   id result;
   result = [ firstLink value ];
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
   firstLink = [firstLink nextElement ];
   return result;
 }
@ end
```

Аналогичным образом в языках программирования с динамическими типами данных можно обрабатывать итерации без того, чтобы выставлять на обозрение внутреннюю структуру контейнеров. Мы опишем две такие техники: одна используется в Smalltalk, а другая повсеместно применяется в разнообразных языках программирования.

Мы отметили ранее, что в языке Smalltalk операторы могут быть сгруппированы в конструкцию, называемую block. Она во многом аналогична функции. Подобно функции, блок может иметь список аргументов. Стандартный способ выполнения итерации в языке Smalltalk — это передать блок как аргумент вместе с сообщением структуре, к которой осуществляется доступ. Цикл, который печатает значения списка, может быть записан следующим образом:

```
aList do: [ :ele s ele print ]
```

Класс списка просто пересылает блок классу элемента. Каждый элемент вызывает блок с использованием своего текущего значения, и затем пересылает блок следующему элементу.

```
linkDo: aBlock
 " выполнить блок, передать его следующему элементу списка "
   aBlock value: value.
   nextLink notNil
   ifTrue: [ nextLink linkDo: aBlock ]
```

При таком подходе удается производить разнообразные итерации, причем все — без показа структуры списка.

В языке Objective-C и других объектно-ориентированных языках решение задачи об итерациях будет немного более сложным из-за отсутствия блоков.

**Листинг 15.1**. Итератор в языке Objective-C

```
@ implementation ListIterator
{
  currentLink : id;
}
+ newIterator : aList
{
  self = [ ListIterator new ];
  currentLink = [ aList firstLink ];
  return self;
}
- id value
{
  return [ currentLink value ]
}
- int atEnd
{
  return currentLink == nil;
}
- void advance
{
  if (! [ self atEnd ] )
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    currentLink = [ currentLink nextElement ];
}
@end
```

Стандартной альтернативой является создание вспомогательного объекта, называемого итератором (iterator). Этот объект поставляется разработчиком контейнерного класса (например, связного списка). Единственное назначение такого объекта — обеспечить доступ к элементам контейнера (один элемент за одно обращение) без показа внутренней структуры списка. Обычно итератор содержит указатель, с которым производятся всевозможные манипуляции. Листинг 15.1 иллюстрирует, как можно определить итератор для абстракции связного списка.

Итератор обычно определяется самим списком как реакция на сообщение. Поэлементный цикл производится следующим образом:

```
id aList;  /* объявление списка  */
id itr;   /* объявление итератора */
for (itr = [ aList iterator ]; ! [ itr atEnd ];
  [itr advance ])
  print( [ itr value ] );
```

Заметьте, что хотя цикл потребовал объявления дополнительной переменной-итератора, ее использование не подразумевает знание внутренней структуры связного списка.

Легкость, с которой конструируются и обрабатываются абстракции данных, — это один из основных рекламных лозунгов языков с динамическими типами. Контейнеры имеют совершенно общий вид и могут даже содержать разнородные наборы данных различных типов. К сожалению, такой выигрыш дается недаром. Как мы отметили ранее, существует дилемма между легкостью в использовании и эффективностью выполнения. Программы на динамических языках редко выполняются столь же эффективно, как в языках с более строгим контролем типов.

## 15.3. Контейнеры в языках со строгим контролем типа данных

Мы переходим теперь к рассмотрению того, как контейнерные классы конструируются в языках со строгим контролем типа данных (Object Pascal, C++). Есть мнение, что принцип подстановки сам по себе обеспечивает решение проблемы контейнерных классов в таких языках. Согласно главе 6 принцип подстановки утверждает, что переменной, которая объявлена с определенным типом, можно на самом деле присвоить значение подтипа. Принцип подстановки на самом деле до некоторой степени облегчает решение наших проблем, но далеко не всех.

Чтобы использовать подстановку, мы прежде всего должны создать класс, который бы был родителем всего, что мы захотим хранить в нашей структуре данных. Назовем этот гипотетический класс ListElement. Затем создадим абстракцию списков с элементами. Листинг 15.2 показывает, как это можно сделать в Object Pascal.

Объекты, хранящиеся в списке, должны быть описаны как подклассы класса ListElement. Соответственно мы не можем построить список с целыми или вещественными числами, пока не породим эти типы данных из класса ListElement. Обычно это не очень серьезная проблема. На самом деле мы получили разнородный список со значениями различного типа, если только все они являются подклассами базового класса ListElement.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Настоящая проблема возникает, когда мы хотим сделать что-либо с элементом, извлеченным из списка. Контроль типов данных, который определяет результат как принадлежащий классу ListElement, встает на нашем пути. Мы должны «отменить» подстановку дочернего типа данных вместо родительского типа. Предположим, к примеру, что мы создали два подкласса для класса ListElement. Подкласс WhiteBall представляет белые шары, а подкласс BlackBall — черные. Мы имеем список шаров и хотим извлечь первый элемент списка, присвоив его значение переменной типа WhiteBall.

**Листинг 15.2**. Описание контейнерного класса на языке Object Pascal

```
type
  List = object
    firstLink :  ListElement;
    procedure addToList(var newValue : ListElement);
    function firstValue : ListElement;
  end;
  ListElement = object
    next :  ListElement;
  end;
procedure List.addToList(var newValue : ListElement);
(* добавляет значение к началу списка *)
begin
  (* поле связи должно указывать на текущий элемент *)
  newValue.next := firstLink;
  (* изменить ссылку в первом элементе списка *)
  firstLink := newValue;
end;
function firstValue : ListElement;
(* исключить из списка первый элемент и вернуть его *)
var
  first : ListElement;
begin
  first := firstLink;
  firstValue := firstLink;
  firstLink := first.next;
end;
```

Мы говорили при обсуждении связывания, что здесь на самом деле имеются два момента, которые, однако, в некоторых объектно-ориентированных языках соединены:

- Можем ли мы определить тип значения, полученного из списка?
- Если да, то позволит ли компилятор выполнить присваивание безопасно с точки зрения сохранения типа?

Вспомним, что в Object Pascal первая из двух проблем решается через использование логической функции Member, которая говорит нам, содержит ли переменная значение, относящееся к заданному классу. Если функция Member показывает, что преобразование является законным, то может быть задействовано приведение типа для преобразования значения к соответствующему типу данных:

```
var
  aBall : WhiteBall;
  aList : List;
  aValue : ListElement;
...
  (* извлечь элемент из списка *)
  aValue := aList.firstElement;
  (* тип данных соответствует? *)
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  if Member(aValue, WhiteBall) then
    (* присваивание законно *)
    aBall := WhiteBall(aValue);
```

То есть извлечение элемента из структуры данных может выполняться в несколько этапов, но это именно та техника, которая используется во многих коммерчески доступных структурных классах. Сложность в использовании этих структур привела программистов к необходимости рассмотреть альтернативные варианты.

Циклы часто создаются через структуры, подобные итераторам (см. приведенное выше решение этой проблемы в языке Objective-C). Однако, как и в случае функции firstElement, результатом работы итератора может быть только значение типа ListElement. Обязанностью программиста является привести его к другому типу данных:

```
var
  aList : List;
  aValue : ListElement;
  itr  : ListIterator;
...
  itr := aList.iterator;
  while (not itr.atEnd) do
  begin
               aValue := itr.current;
               if Member(aValue, WhiteBall) then
               ...
               itr.advance; (* следующее значение итератора *)
  end;
```

До того как была введена система RTTI (Run-Time Typing Information — идентификация типа во время выполнения), в языке C++ значения обычно не знали своего собственного динамического типа. Это усложняло построение контейнеров, поскольку программист должен был не только приводить тип, но и обеспечивать собственный механизм определения динамического типа значений. Недавнее появление функции dynamic_cast призвано решить именно эту проблему.

## 15.4. Скрытое приведение типа данных при наследовании

Принципиальные сложности в использовании техники, описанной в предыдущем разделе, состоят в следующем:

- Структура данных может хранить только те значения, которые относятся к подклассу класса ListElement.
- Язык программирования должен поддерживать принцип подстановки.
- Объекты обязаны знать свой собственный динамический тип.
- При извлечении данных требуется как явная проверка типа, так и операция приведения типа.

Приведение типа — это довольно опасная конструкция, которую при программировании следует избегать где только возможно. Вдобавок при создании абстракций данных в C++ существенной становится вторая трудность. Вспомним, что C++ не поддерживает подстановку для объектов, объявленных обычным образом (поддержка осуществляется только для указателей и ссылок). По этой причине многие структуры данных в языке C++ разработаны так, чтобы хранить указатели на значения, а не сами значения.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Стандартно предлагаемая техника программирования использует дочерние классы и наследование, чтобы замаскировать необходимость приведения типа в такой ситуации. Предположим, например, что мы уже имеем абстракцию данных со следующим интерфейсом:

```
class GenericList // список указателей общего вида void *
{
  public:
    void addToList (void * newElement);
    void * firstElement();
  private:
    GenericLink * firstLink;
};
class GenericLink
{
  public:
    void    * value;
    GenericLink * nextLink;
};
```

Соответственно наш список общего вида содержит указатели void. Они могут указывать на что угодно. В теории такая совокупность может быть сколь угодно разнородной при использовании указателей на объекты различного типа. Теперь предположим, что мы хотим создать список указателей на окна (структура Window). Единственное, что надо сделать, — это определить подкласс класса общего вида и изменить типы аргументов и результата в методах, возвращающих элемент списка. В любом случае фактическую работу выполняет родительский класс.

```
class WindowList : public GenericList
{
  public:
    void addToList (Window * newElement)
    {
      GenericList::addToList (newElement);
    }
    Window * firstElement ()
    {
      return (Window *) GenericList::firstElement;
    }
};
```

Таким способом удается создать структуры данных, которые хранят значения, отличные от указателей (целые и вещественные числа). Но реализация требует определения подклассов как для класса List, так и для класса Link, а также, вероятно, создания новых классов-итераторов.

Мы достигли до некоторой степени многократного использования кода, но только за счет того, что заставили программиста вводить новые подклассы всякий раз, когда он хочет применить абстракцию данных. Многие программисты отвергают такое решение просто потому, что оно доставляет не меньше хлопот, чем написание структур данных «с нуля».

## 15.5. Параметризованные классы

Последний наш тезис состоял в том, что (по крайней мере для языков со строгим контролем типа данных) само по себе наследование недостаточно для создания простых в использовании контейнерных классов. Должен применяться другой механизм. Такой механизм существует. Он заключается в определении класса с типом в качестве

PDF created with pdfFactory Pro trial version www.pdffactory.com

параметра. Такие классы называются шаблонами в C++ и обобщенными классами в некоторых других языках.

Шаблоны дают программисту возможность определять типы данных, в которых информация о типе преднамеренно остается незаданной. Этот пробел заполняется позднее. Чтобы лучше понять параметризацию, представьте себе, что описанию класса тип поставляется как аргумент процедуры или функции. Точно так же, как при вызове функции ей могут передаваться различные значения через список аргументов, так и разные «воплощения» параметризованного класса получают информацию о типе-параметре.

Параметризованное описание класса для абстракции связного списка записывается в C++ следующим образом:

```
template
class List
{ public:
    void addElement (T newValue);
    T  firstElement ();
    ListIterator iterator();
  private:
    Link * firstLink;
};
template
class Link
{
  public:
    T value;
    Link * nextLink;
    Link(T, Link *);
};
```

Внутри шаблона класса аргумент шаблона (в данном случае идентификатор T) может использоваться как имя типа данных. Соответственно, разрешается определять переменные типа T, вводить функции, возвращающие результат типа T, и т. д.

Функции-члены, которые определяют методы в шаблоне класса, должны также описываться как шаблоны:

```
template
void List::addElement(T newValue)
{
  firstLink = Link new (newValue, firstLink);
}
template
T List::firstElement()
{
  Link first = firstLink;
  T result = first.value;
  firstLink = first->nextLink;
  delete first;
  return result;
};
template
Link::Link(T v, Link * n) : value(v); nextLink(n)
{ }
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Пользователь создает различные виды списков, указывая конкретные типы. Например, следующие операторы определяют списки целых и вещественных чисел:

```
List  listOne;
List listTwo;
```

Таким способом могут быть созданы только однородные списки.

Шаблоны — элегантное решение проблемы контейнерных классов. Они позволяют достичь истинного многократного использования, создавать и обрабатывать компоненты общего назначения с минимумом сложностей, а также гарантировать безопасность при обращении с типами, что является целью языков программирования со строгим контролем типов данных.

Недостатки имеются и у шаблонов. Они не позволяют определять списки разнородных данных, поскольку все элементы должны соответствовать объявленному типу. (Эту проблему можно обойти за счет хранения указателей на значения вместо самих значений.) Более важный недостаток: реализация шаблонов сильно варьируется в отношении как легкости использования, так и качества получаемого кода в различных компиляторах. Большинство из них не делают ничего, кроме интерпретации шаблонов в виде сложных макросов, так что для каждого нового параметра-типа создается совершенно новое определение класса и полностью независимые тела методов. Не нужно говорить, что это приводит к значительному увеличению размера кода.

Тем не менее, поскольку шаблоны освобождают программиста от большого количества нудной работы (а именно от переписывания классов для новых структур данных в очередном приложении), они пользуются большой популярностью. В следующей главе мы рассмотрим библиотеку шаблонов.

### 15.5.1. Циклы и итерации в C++

Существование механизма шаблонов как для классов, так и для индивидуальных функций позволяет определить не один, а два различных способа итераций в C++. Оба они включены в недавно разработанную стандартную библиотеку шаблонов для C++, которая рассматривается в главе 16.

Первый способ использует итератор. Контейнерный класс определяет тип данных для итератора, а также функции, которые возвращают значение итератора. Например, итератор для нашего класса связных списков записывается следующим образом:

```
template
class List
{
  public:
    typedef ListIterator iterator;
    ...
    iterator begin()
    {
      // верни мне итератор в исходном состоянии
      return ListIterator (firstLink);
    }
    iterator end()
    {
      // конец
      return ListIterator (0);
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    }
}
template
class ListIterator
{
  public:
    ListIterator (Link * sl) : currentLink(sl)
    { }
    void operator ++ ()
    {
      // переход к следующему элементу
      currentLink = currentLink->nextLink;
    }
    T operator * ()
    {
      // вернуть текущий элемент
      return currentLink->value;
    }
    bool operator == (ListIterator & right)
    {
      return currentLink == right.currentLink;
    }
  private:
    Link * currentLink;
};
```

Теперь итератор может быть объявлен и инициализирован заданным списком. Поэлементный цикл выполняется без знания внутренней структуры списка:

```
List::iterator start = aList.begin();
List::iterator end  = aList.end();
for (; itr != end; itr++)
{
  cout << (*itr) << endl;
};
```

Второй вид итераций, называемый иногда итерациями с применением, в некотором отношении похож на циклы в языке Smalltalk. При такой итерации контейнер получает функцию в качестве аргумента, и он сам применяет ее к каждому элементу совокупности. Эти два действия объединяются в функции for_each, которая применяет передаваемую как аргумент функцию к каждому элементу списка:

```
void printOut(int n)
{
  cout << "the collection contains a " << n << "\n";
}
...
for_each (aList.begin(), aList.end(), printOut);
```

Проблема с итерациями такого типа состоит в том, что они требуют создания или использования функции, передаваемой в качестве аргумента. Если контейнер является чисто локальным объектом, такую функцию зачастую довольно сложно определить. В таких ситуациях цикл с использованием итератора может быть проще.

## Упражнения

1. Контейнерные классы в объектно-ориентированном программировании: успех или неудача?

PDF created with pdfFactory Pro trial version www.pdffactory.com

2. Структуры данных разделяются на те, которые характеризуются своей реализацией (связные списки, деревья), и на те, которые определяются их предназначением (стеки, множества). Опишите, как можно использовать ООП для упрощения второго типа структур и маскировки деталей их реализации. Приведите в качестве иллюстрации структуру данных с единым интерфейсом и двумя радикально отличающимися реализациями.
3. Дайте пример разнородного контейнера, то есть контейнера значений различного типа.
4. Подход Smalltalk к построению итераций состоит в том, чтобы сгруппировать операции, которые надо выполнить, в блок и передать его структуре данных. Напротив, итератор — это структура, которая передает данные одно за другим внешней процедуре, где с ними выполняются нужные действия. Можно ли реализовать подход Smalltalk в других языках программирования (Object Pascal, C++)? Как при этом сказывается строгий контроль типов?
5. Приведите пример шаблонов, не связанных с контейнерными классами.

## *Глава 16: Пример: STL*

Богатая коллекция структур данных, основанная на использовании шаблонов, недавно была добавлена к стандартной библиотеке языка C++. Эти структуры содержат классы для векторов, списков, множеств, карт (словарей), стеков, очередей, очередей с приоритетами. Когда этот стандарт станет распространен повсеместно, программисты на языке C++ будут освобождены от необходимости постоянно переопределять и повторно реализовывать один и тот же набор классов для структур данных. Более подробная информация о библиотеке STL (STL — Standard Template Library, стандартная библиотека шаблонов) может быть найдена в работах [Musser 1996, Glass 1996].

Создание STL является результатом многолетних исследований под руководством Александра Степанова и Менга Ли из компании Hewlett-Packard и Дэвида Мюссера из Rensselaer Polytechnic Institute. Создание STL вдохновлялось не только предшествующими объектно-ориентированными библиотеками, но и многолетним опытом работы ее создателей в области функциональных и директивных языков программирования (Scheme и Ada).

Одна из наиболее необычных идей в STL — это обобщенные алгоритмы. Она заслуживает отдельного рассмотрения, поскольку выглядит вызовом объектно-ориентированным принципам, которые мы обсуждали до сих пор, и в то же время является источником огромной мощи библиотеки STL. Обобщенные алгоритмы в STL напоминают классы-шаблоны. Идея шаблонов применяется к отдельным функциям. Чтобы понять концепцию обобщенных алгоритмов, мы сперва должны описать использование инкапсуляции в большинстве объектных библиотек.

Объектно-ориентированное программирование рассматривает инкапсуляцию как главную цель. Хорошо разработанный объект старается инкапсулировать все состояние и поведение, необходимые для выполнения задачи, и в то же время скрывает как можно больше деталей внутреннего устройства. Во многих предшествующих объектно-ориентированных библиотеках этот философский подход воплощался в контейнерных классах, обладающих широкой функциональностью и богатым интерфейсом.

Разработчики STL пошли в совершенно другом направлении. Поведение, обеспечиваемое их стандартными компонентами, является минимальным, почти что спартанским. Вместо этого каждый компонент предназначен для функционирования совместно с большим

PDF created with pdfFactory Pro trial version www.pdffactory.com

набором обобщенных алгоритмов, имеющихся в библиотеке. Эти алгоритмы не зависят от контейнеров и поэтому могут работать со многими различными типами.

Отделяя функционирование алгоритмов от контейнерных классов, библиотека STL много выигрывает в размере — как в объеме самой библиотеки, так и в генерируемом коде. Вместо того чтобы дублировать алгоритмы для дюжины или около того контейнерных классов, одно-единственное описание библиотечной функции может использоваться с любым контейнером. Более того, описание этих функций является настолько общим, что они могут применяться с обычными массивами и указателями (в смысле языка C), и с другими типами данных.

Наш пример проиллюстрирует некоторые основные свойства стандартной библиотеки шаблонов. Обобщенный алгоритм find находит первое вхождение заданного значения в коллекцию. Итераторы стандартной библиотеки состоят из пар значений, отмечающих начало и конец структуры. Алгоритм find использует пару итераторов и ищет первое вхождение заданного значения. Он определен следующим образом:

```
template
InputIterator find (InputIterator first,
          InputIterator last,
          const T& value)
{
  while (first != last && *first != value)
  {
    ++first;
  };
  return first;
}
```

Алгоритм будет работать со структурой любого типа, в том числе и с обычными массивами языка C. Чтобы найти первое вхождение числа 7 в массив целых, пользователь выполняет следующий код:

```
int data[100];
...
int * where;
where = find(data, data+100, 7);
```

Поиск первого значения в целочисленном списке ничуть не сложнее:

```
list aList;
...
list::iterator where;
where = find(aList.begin(), aList.end(), 7);
```

В этой главе мы можем описать только основные свойства STL. Следующие два раздела посвящены базовым концепциям, используемым в библиотеке, а именно итераторам и объектам-функциям. Затем три примера проиллюстрируют в действии контейнеры и обобщенные алгоритмы библиотеки STL.

## 16.1. Итераторы

Итераторы — это фундамент, на котором основано использование контейнерных классов и соответствующих алгоритмов из стандартной библиотеки. Абстрактно итератор — это

PDF created with pdfFactory Pro trial version www.pdffactory.com

просто объект типа указателя, который применяется для прохода по всем элементам в контейнере.

Подобно тому как указатели по-разному используются в традиционном программировании, итераторы применяются для различных целей. Итератор может обозначать конкретное значение (как указатель указывает на конкретную область памяти). С другой стороны, пара итераторов может задавать диапазон значений (два указателя отмечают границы непрерывного участка памяти). Однако в случае итераторов описываемые значения расположены друг за другом не физически, а логически. Это происходит, поскольку они взяты из одного контейнера и, следовательно, следуют друг за другом в том порядке, как они хранились в контейнере.

Традиционные указатели иногда принимают значение null — то есть не указывают ни на что. Итераторы аналогичным образом могут не определять какое-либо конкретное значение. Подобно тому, как логической ошибкой является разыменование и использование указателя со значением null, нельзя разыменовывать и применять итератор, который не определяет никакого значения.

Когда в языке C++ используются два указателя, которые ограничивают область памяти, по соглашению второй указатель не рассматривается как часть области. Например, массив с именем x и длиной 10 иногда описывается как занимающий область от x до x+10, хотя элемент x+10 не является частью массива. На самом деле указатель x+10 ссылается на запредельный элемент, то есть элемент, следующий за последним элементом описываемого диапазона. Итераторы определяют диапазон аналогичным образом. Второе значение рассматривается не как часть определяемого диапазона, но как запредельный элемент, описывающий значение, следующее в последовательности за последним значением из заданного диапазона.

Подобно традиционным указателям, основное действие, которое модифицирует итератор, это инкрементация (оператор ++). Когда инкрементация применяется к итератору, который указывает на последнее значение в последовательности, итератор принимает описанное выше запредельное значение. Оператор разыменования (*) осуществляет доступ к данным, определяемым итератором.

Диапазон может описывать весь контейнер при задании итератора, указывающего на первый элемент, и итератора, имеющего специальное «последнее» значение. Диапазоны могут описывать подпоследовательности, входящие в контейнер (двум итераторам присваиваются конкретные значения). В стандартных контейнерах итератор начала контейнера возвращается функцией begin(), а итератор, обозначающий конец контейнера, — функцией end().

## 16.2. Объекты-функции

Ряд обобщенных алгоритмов из библиотеки STL требует функций в качестве аргументов. Простым примером служит обобщенный алгоритм for_each(), который вызывает функцию, переданную в качестве аргумента, для каждого значения в контейнере. Следующий код используется для того, чтобы вывести полный набор значений в целочисленном списке:

```
void printElement(int value)
{
  cout << "The list contains " << value << endl;
}
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
main ()
{
  list aList;
  ...
  for_each(aList.begin(), aList.end(), printElement);
}
```

Понятие функции было обобщено до понятия объекта-функции. Объект-функция — это экземпляр класса, в котором определен метод «круглые скобки» (). В ряде случаев удобно заменить функции на объекты-функции. Когда объект-функция используется в качестве функции, то при ее вызове используется оператор «круглые скобки».

Чтобы проиллюстрировать это, рассмотрим следующее определение класса:

```
class biggerThanThree
{
  public:
    bool operator () (int v)
    {
      return V > 3 ;
    }
};
```

Если мы создадим экземпляр класса biggerThanThree, то каждый раз, когда мы будем ссылаться на него с использованием синтаксиса вызова функции, будет вызываться метод, соответствующий оператору «круглые скобки». Следующий шаг — обобщить этот класс, добавив к нему конструктор и неизменяемое поле данных, которое устанавливается конструктором:

```
class biggerThan
{  public:
     biggerThan (int x) : testValue(x) { }
     const int testValue;
     bool operator () (int val)
     {  return val > testValue;  }
};
```

Результатом является функция общего вида, выполняющая целочисленное сравнение «больше чем X», где значение X определяется при создании экземпляра класса. Это может быть сделано, например, при передачи одной из обобщенных функций аргумента — логической функции. Мы ищем в списке первое значение, большее 12, с помощью кода

```
list::iterator firstBig = find_if(aList.begin(),
                    aList.end(),
                    biggerThan(12));
```

## 16.3. Пример программы: инвентаризация

Наш первый пример иллюстрирует создание и обработку объектов в библиотеке STL. Допустим, фирме WorldWideWidgetWorks потребовалась программа учета виджетов. Виджеты — это простые устройства, различаемые по идентификационным номерам:

```
class Widget
{
  public:
    Widget(int a) : id(a) { }
    Widget() : id(0) { }
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    int id;
};
ostream & operator << (osteam & out, Widget & w)
{
  return out << "Widget " << w.id;
}
bool operator == (const Widget & lhs, const Widget & rhs)
{
  return lhs.id == rhs.id;
}
bool operator < (const Widget & lhs, const Widget & rhs)
{
  return lhs.id < rhs.id;
}
```

Инвентарь описывается двумя списками. Один представляет собой виджеты, имеющиеся в данный момент на складе. Второй список содержит типы виджетов, которые были заказаны покупателями. Первый список содержит собственно виджеты, а второй — идентификационные типы виджетов. Чтобы управлять инвентарем, требуются две команды:

- order() обслуживает заказы;
- receive() следит за поставками новых виджетов.

```
class inventory
{
  public:
    void order (int wid);
      // обработка заказа виджета типа wid
    void receive (int wid);
      // получение виджета типа wid
  private:
    list on_hand;
    list  on_order;
};
```

Когда поступает новый виджет, мы сравниваем его идентификационный номер со списком заказанных виджетов. Мы используем функцию find() для поиска в списке заказов и немедленно пересылаем виджет покупателю, если он (виджет) был заказан. В противном случае он добавляется к списку виджетов на складе.

```
void inventory::receive(int wid)
{
  cout << "Пришла партия виджетов типа" << wid << endl;
  list::iterator weNeed = find(on_order.begin(),
                   on_order_end(),
                   wid);
  if ( weNeed != on_order.end() )
  {
    cout << "Поставить " << Widget(wid)
       << " чтобы пополнить запас" << endl;
    on_order.erase(weNeed);
  }
  else
  {
    on_hand.push_front(Widget(wid));
  };
}
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Когда покупатель заказывает новый виджет, мы просматриваем с помощью функции find_if() список имеющихся на складе виджетов, чтобы определить, нельзя ли обслужить заказ немедленно. Для этого определена унарная функция, которая берет в качестве аргумента виджет и определяет, соответствует ли он требуемому типу. Эта функция записывается следующим образом:

```
class WidgetTester
{ public:
    WidgetTester(int t) : testid(t) { }
    const int testid;
    bool operator () (const Widget & wid)
    {  return wid.id == testid;
    }
};
```

Функция, обслуживающая заказы виджетов, выглядит так:

```
void inventory::order(int wid)
{
  cout << "Получен заказ на виджеты типа " << wid << endl;
list::iterator weHave = find_if(on_hand.begin(),
                      on_hand.end(),
                      WidgetTester(wid));
  if ( weHave != on_hand.end() )
  {
    cout << "Поставить " << *weHave << endl;
    on_hand.erase(weHave);
  }
  else
  {
    cout << "Заказать виджет типа " << wid << endl;
    on_order.push_front(wid);
  };
}
```

## 16.4. Пример программы: графы

Второй и третий примеры используют тип данных «карта». Карта map — это индексированный словарь, то есть набор пар ключ-значение.

Карта, в которой элементами тоже являются карты, служит естественным представлением направленного графа. Предположим, к примеру, что мы используем текстовые строки для кодирования названий городов и хотим сконструировать карту, где значения ребер графа — это расстояние между двумя соответствующими городами. Такой граф показан на рис. 16.1.

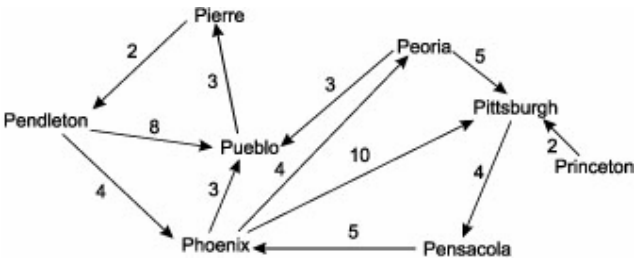



**Рис. 16.1**. Взвешенный граф

PDF created with pdfFactory Pro trial version www.pdffactory.com

**Листинг 16.1**. Операторы инициализации графа

```
typedef map stringVector;
typedef map graph;
string pendleton("Pendleton");
string pensacola("Pensacola");
string peoria("Peoria");
string phoenix("Phoenix");
string pierre("Pierre");
string pittsburg("Pittsburg");
string princeton("Princeton");
string pueblo("Pueblo");
graph cityMap;
cityMap[pendleton][phoenix] = 4;
cityMap[pendleton][pueblo] = 8;
cityMap[pensacola][phoenix] = 5;
cityMap[peoria][pittsburgh] = 5;
cityMap[peoria][pueblo] = 3;
cityMap[phoenix][peoria] = 4;
cityMap[phoenix][pittsburg] = 10;
cityMap[phoenix][pueblo] = 3;
cityMap[pierre][pendleton] = 2;
cityMap[pittsburg][pensacola] = 4;
cityMap[princeton][pittsburg] = 2;
cityMap[pueblo][pierre] = 3;
```

Тип данных stringVector — это вектор целых чисел, индексированный строками. Тип данных graph — это двумерный разреженный массив, индексированный строками и содержащий целые числа. Последовательность операторов присваивания инициализирует граф. Граф, показанный на рис. 16.1, реализован в листинге 16.1.

Можно использовать набор классических алгоритмов для обработки таких графов. Одним из примеров является алгоритм Дейкстры поиска кратчайшего пути. Требуется указать город, с которого начинается путь. При этом создается приоритетная очередь для пар расстояние/город. Она инициализируется нулем для начального города. Приоритетная очередь содержит города в порядке от ближайшего до самого дальнего. Определение типа данных DistancePair выглядит следующим образом:

```
struct DistancePair
{ unsigned int first;
  string  second;
  DistancePair() : first(0) { }
  DistancePair(unsigned int f, string & s)
    : first(f), second(s) { }
};
bool operator < (DistancePair & lhs, DistancePair & rhs)
{ return lhs.first < rhs.first; }
```

При каждой итерации цикла мы извлекаем из очереди город. Если для него еще не найдено кратчайшего пути, записывается текущее расстояние и путем проверки графа вычисляется расстояние до соседних городов. Этот процесс продолжается до тех пор, пока не будет исчерпана приоритетная очередь.

```
void shortestDistance(const graph & cityMap,
           const string & start,
           stringVector & distances)
{
  // обработать приоритетную очередь расстояний до городов
  priority_queue,
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
          greater > que;
  que.push(DistancePair(0, start));
  while (! que.empty() )  {
    // выбрать ближайший город из очереди
    int distance = que.top().first;
    string city = que.top().second;
    que.pop();
    // если мы еще не встречали такой город, обработать его
    if (0 == distances.count(city))
    {
      // затем добавить его к карте
      // кратчайших расстояний
      distances[city] = distance;
      // и занести в очередь новые значения
      stringVector::iterator start, stop;
      start = cityMap[city].begin();
      stop = cityMap[city].end();
      for (; start != stop; ++start)
      que.push(DistancePair(distance + (*start).second,
                  (*start).first));
    }
  }
}
```

## 16.5. Пример программы: алфавитный указатель

Последним примером является алфавитный указатель. Программа использует связные списки и тип данных multimap. multimap — это разновидность карт, которая позволяет индексировать одним ключом несколько данных.

Алфавитный указатель представляет собой список упорядоченных по алфавиту слов, который показывает номера строк, где встречается то или иное слово. Эти значения хранятся в алфавитном указателе в виде мультикарты multimap, индексированной словами (тип данных string). Ее содержимое — целые числа (номера строк в тексте). Мультикарта используется потому, что одно и то же слово встречается в нескольких местах. На самом деле обнаружение таких повторений — главное предназначение алфавитного указателя.

Тип данных для алфавитного указателя задается следующим образом:

```
class concordance // алфавитный указатель
{
  typedef multimap wordDictType;
  public:
    concordance() : wordMap() { }
    void addWord (string, int);
    void readText(istream &);
    void printConcordance(ostream &);
  private:
    wordDictType wordmap;
};
```

Создание алфавитного указателя разбивается на два шага. Сперва он генерируется программой (строки текста считываются из входного потока), а затем результат печатается в выходной файл. Эти два шага реализуются в функциях-членах readText() и printConcordance(). Первая из них записывается следующим образом:

```
void concordance::readText(istream & in)
// считать текст из входного потока,
// занося слова в алфавитный указатель
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
{
  string line;
  // читать строки входного потока
  for (int i=1; getline(in, line); i++)
  {
 allLower(line); // преобразовать в нижний регистр
 list words;
 split(line, " ,.;:", words); // разбить строку на слова
 list::iterator wptr; // внести слова в коллекцию
 for (wptr = words.begin(); wptr != words.end(); ++wptr)
 addWord(*wprt, i);
  }
}
```

Строки текста считываются из входного потока одна за другой. Их текст прежде всего преобразуется к нижнему регистру. Затем строка разбивается на слова с помощью функции split(). Набор литер-разделителей и исходный текст передаются как аргументы. Эта функция иллюстрирует некоторые из возможностей обработки текстовых строк, заложенных в библиотеку STL:

```
void split(string & text, string & separators,
      list & words)
{
  int n = text.length();
  int start, stop;
  // найти первую букву, не разделитель
  start = text.find_first_not_of(separator);
  while ( (start >= 0) && (start < n) )
  {
    // найти конец текущего слова
    stop = text_find_first_of(separators, start);
    if ( (stop < 0) || (stop > n) ) stop = n;
    // добавить слово к списку слов
    words.push_back(text.substr(start, stop-start));
    // найти начало следующего слова
    start = text.find_first_not_of(separators,stop+1);
  }
}
```

Вслед за вызовом функции split() каждое найденное слово добавляется в алфавитный указатель с помощью следующего метода:

```
void concordance::addWord(string word, int line)
{
  // проверить, не встречается ли слово в списке
  // сначала найти диапазон вхождений с нужным ключом
  wordDictType::iterator low =
    wordMap.lower_bound(word);
  wordDictType::iterator high =
    wordMap.upper_bound(word);
  // цикл по вхождениям, соответствуют ли они строке
  for (; low != high; ++low)
    if ( (*low).second == line ) return;
  // не встретилась, добавим
  wordMap.insert(make_pair(word, line));
}
```

Метод addWord() гарантирует, что значения в карте слов не дублируются, если какое-либо слово встречается в одной строке дважды. Это достигается благодаря проверке записей, которые соответствуют ключу поиска. Каждая запись проверяется на совпадение номеров

PDF created with pdfFactory Pro trial version www.pdffactory.com

строк, и в случае совпадения запись в указатель не добавляется. Только если цикл проверки завершается без обнаружения записи с тем же номером строки, новая пара слово/номер строки добавляется в карту.

Последний шаг — распечатать алфавитный указатель. Это делается следующим образом:

```
void concordance::printConcordance(ostream & out)
{
  string lastword("");
  wordDictType::iterator pairPtr;
  wordDictType::iterator stop = wordMap.end();
  for(pairPtr = wordMap.begin(); pairPtr != stop;
      ++pairPtr)
    // если слово такое же, что и предыдущее,
     // просто напечатать номер строки
    if ( lastword == (*pairPtr).first)
      out << " " << (*pairPtr).second;
    else
    // первое вхождение слова
    {
      lastword = (*pairPtr).first;
      out << endl << lastword << ": " <<
        (*pairPtr).second;
    }
  cout << endl; // завершить последнюю строку вывода
}
```

Итератор циклически проходит по всем элементам, содержащимся в списке слов. Каждое новое слово порождает новую строку вывода; следующие за словом номера строк разделяются пробелами. Если, к примеру, на входе задан текст

```
It was the best of times,
it was the worst of times.
```

то на выходе будут напечатаны слова от «best» до «worst»:

```
best: 1
it: 1 2
of: 1 2
the: 1 2
times: 1 2
was: 1 2
worst: 1
```

## 16.6. Будущее ООП

Мы отмечали, что во многих отношениях библиотека STL не является объектно-ориентированной. Скорее, она написана под воздействием функционального программирования. Означает ли включение STL в список стандартных библиотек C++, что объектно-ориентированное программирование выходит из моды?

Безусловно, нет. Объектно-ориентированные техники проектирования и программирования не имеют себе равных при разработке больших сложных систем. В большинстве задач программирования ООП будет оставаться главенствующим. Однако разработка STL (и не только) показывает желанные признаки того, что сообщество объектно-ориентированных программистов признает, что не все идеи должны иметь

PDF created with pdfFactory Pro trial version www.pdffactory.com

выражение в объектно-ориентированном стиле, равно как и не все проблемы обязаны решаться с помощью объектно-ориентированной техники.

### Упражнения

1. Предположите, что имеется непосредственная реализация класса линейной структуры данных (например, класса связных списков из главы 15). Опишите основные свойства класса-итератора для этой структуры. За какой информацией должен следить ваш итератор?
2. Теперь рассмотрите нелинейную структуру данных (например, двоичное дерево). Какую информацию должен поддерживать итератор для того, чтобы выдавать элементы, содержащиеся в контейнере?

## *Глава 17: Видимость и зависимость*

В главе 1 мы выяснили, что взаимозависимость программных компонент является основным препятствием на пути разработки многократно используемого кода. Этот факт давно признан сообществом разработчиков. Имеется литература, посвященная характеристике взаимозависимости компонент, где приведены правила, как избегать «вредных» связей (см., к примеру, [Gillet 1982] и [Fairley 1885]). В этой главе мы исследуем некоторые из этих соображений в контексте объектно-ориентированного программирования.

Будем описывать взаимосвязи в терминах видимости и зависимости. Видимость описывает некоторую характеристику имен объектов. Объект является видимым в некотором контексте, если его имя является правильным и обозначает объект. Близкий термин, часто используемый для описания видимости, это область видимости идентификатора.

Видимость тесно связана с взаимозависимостью: управляя видимостью имени идентификатора, мы можем более четко охарактеризовать использование идентификатора. В языке Smalltalk, к примеру, видимость переменных экземпляра ограничена методами — к таким переменным невозможен непосредственный доступ вне метода. Это не означает, что они не могут изменяться или считываться извне класса. Все такие действия, однако, должны проводиться при посредничестве метода. С другой стороны, в языке Object Pascal версии Apple переменные экземпляра видны всюду, где известно имя соответствующего класса. То есть этот язык не обеспечивает механизмов, гарантирующих, что переменные экземпляра модифицируются только методами. Вместо этого нам приходится полагаться на надлежащее поведение пользователей.

Понятие зависимости соотносит различные части приложения. Если программная система (класс, модуль и т. д.) не может осмысленно существовать в отрыве от другой системы, говорят, что первая система зависит от второй. Например, дочерний класс почти всегда зависит от своего родителя. Зависимости могут быть и гораздо более тонкими, как мы увидим в следующем разделе.

### 17.1. Зацепление и связность

Понятия зацепления и связности были введены Стивенсом, Константайном и Майерсом [Stevens 1981] для оценки эффективного использования модулей. Мы будем обсуждать их, имея в виду языки, поддерживающие модули, и только затем перейдем к объектно-ориентированным языкам.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Зацепление описывает отношения между модулями, а связность — внутри них. Уменьшение взаимозависимости между модулями (или классами) достигается, следовательно, за счет уменьшения зацепления. С другой стороны, хорошо разработанные модули должны служить некоторой цели, то есть все элементы модуля должны быть связаны общей задачей. Это означает, что хорошо разработанный модуль должен быть внутренне связным.

### 17.1.1. Разновидности зацепления

Зацепление между модулями возникает по многим причинам. Некоторые из них являются более приемлемыми или желательными, чем другие. Упорядоченный список причин выглядит примерно следующим образом:

- зацепление внутренних данных;
- зацепление по глобальным данным;
- зацепление при управлении;
- зацепление из-за параметров;
- зацепление подклассов1.

Зацепление по внутренним данным происходит, когда один из модулей изменяет локальные данные в другом модуле. Эта деятельность затрудняет понимание и осознание смысла программы, и ее следует избегать, где только возможно. В одном из следующих разделов мы рассмотрим эвристический подход, который используется для уменьшения зацепления внутренних данных в объектно-ориентированных системах.

Зацепление по глобальным данным происходит, когда два модуля связаны через общие глобальные структуры данных. Опять-таки, это усложняет понимание модулей при изолированном их рассмотрении, но иногда такое зацепление неизбежно.

На практике важно различать два вида глобальных переменных. Во многих программах некоторые глобальные переменные имеют область видимости, ограниченную текущим файлом (file scope), следовательно, они используются только в пределах одного файла. Другие глобальные переменные видны во всей программе (program scope), и, значит, потенциально они могут модифицироваться где угодно. Понимание смысла глобальных переменных, видимых во всей программе, труднее, чем выяснение предназначения переменных, доступных только в пределах одного файла.

В рамках объектно-ориентированного программирования альтернативой зацеплению по глобальным данным является создание нового класса, чьей обязанностью является администрирование нужных данных. Затем можно заменить все обращения к глобальным данным на вызовы методов класса-администратора. (Этот подход напоминает использование функций доступа для защиты локальных данных внутри объекта.) Такой подход сводит зацепление по глобальным данным к зацеплению по параметрам, что легче для понимания и контроля. В языке Java нет глобальных переменных, и все значения должны управляться некоторым классом.

Зацепление по управлению происходит, когда один модуль должен выполнить некие операции в порядке, который определяется другим модулем. Например, система управления базой данных должна инициализироваться, считать текущие записи, обновить их, удалить часть записей, создать отчет. Однако каждое действие осуществляется отдельной процедурой, и последовательность вызовов может зависеть от кода в другом модуле. Наличие зацепления по управлению показывает, что разработчик модуля

PDF created with pdfFactory Pro trial version www.pdffactory.com

руководствовался более низким уровнем абстрагирования, чем требуется (то есть единственная директива «обработать базу данных» разбивается на шаги). Даже если зацепление по управлению неизбежно, здравый смысл подсказывает, чтобы разделенный на этапы модуль сам гарантировал, что они выполняются в должном порядке (а не полагался бы в этом вопросе на благоразумие со стороны вызывающих блоков).

Зацепление по параметрам происходит, когда один модуль должен пользоваться услугами и процедурами другого модуля, и при этом единственная связь между ними осуществляется через входные и выходные параметры. Такая форма зацепления является типичной, простой для отслеживания и легкой для статической проверки (например, параметры при вызове сравниваются с определением функции). То есть это наиболее желательный вариант зацепления.

Зацепление через подклассы специфично для объектно-ориентированного программирования. Оно описывает отношения класса со своим родителем (или родителями в случае множественного наследования). За счет наследования экземпляр дочернего класса может рассматриваться как принадлежащий родительскому классу. Как мы видели в нескольких учебных примерах в книге, данное свойство позволяет разрабатывать объемные программные компоненты (системы с оконным интерфейсом), которые слабо соотносятся (за счет зацепления через подклассы) с другими разделами программы.

### 17.1.2. Разновидности связности

Внутренняя связность модуля — это мера сцепления друг с другом различных элементов внутри модуля. Как и в случае зацепления, связность может быть ранжирована по шкале от слабой (наименее желательной) до сильной (самой желательной) следующим образом:

- связность по совмещению;
- логическая связность;
- временная связность;
- коммуникационная связность;
- последовательная связность;
- функциональная связность;
- связность данных.

Связность по совмещению означает, что элементы модуля группируются без видимой причины — часто как результат произвольного «разбиения на модули» большой программы. Обычно это признак плохой разработки. В объектно-ориентированном подходе мы говорим, что имеет место связность по совмещению, когда класс состоит из методов, не имеющих между собой ничего общего.

Логическая связность возникает, когда имеется логическая связь между элементами модуля (или методами класса), но нет фактического соединения ни по данным, ни по управлению. Библиотека математических функций (синус, косинус и т. д.) служит примером логической связности, если каждая из функций закодирована отдельно, то есть без связи с другими.

Временная связность возникает, когда элементы объединяются вместе, так как все они должны использоваться примерно одновременно. Модуль инициализации программы является типичным примером. В этом случае более удачная разработка могла бы

PDF created with pdfFactory Pro trial version www.pdffactory.com

распределить различные инициализирующие операции по нескольким модулям, которые в большей степени отвечают за соответствующие действия.

Коммуникационная связность возникает, когда элементы (или методы класса) объединены в модуль, поскольку они имеют доступ к одним и тем же устройствам ввода/вывода. Модуль работает как администратор устройства.

Последовательная связность возникает, если элементы модуля должны активизироваться в определенном порядке. Эта связность часто является следствием попытки избежать зацепления по управлению. Опять-таки, обычно находится лучшая схема, если поднять уровень абстракции. (Конечно же, необходимость выполнять действия в определенном порядке должна быть выражена на некотором уровне абстракции. Важно скрыть эту необходимость от других уровней абстракции.)

Функциональная связность желательна. При ее наличии все элементы модуля связаны выполнением единой задачи.

Наконец, связность на уровне данных возникает в модуле, когда он внутренним образом определяет набор данных и экспортирует подпрограммы (процедуры, функции, методы), которые манипулируют этой структурой данных. Связность по данным возникает, если модуль используется для реализации абстрактного типа данных.

Часто можно оценить степень связности модуля, если кратко сформулировать предложение, описывающее его предназначение (вспомните CRC-карточки из главы 2). Следующий набор тестов был предложен Константайном:

1. Если предложение, которое описывает предназначение модуля, составное, то есть содержит запятую и более одного глагола, то модуль (скорее всего) выполняет более одной функции. Вероятно, он обладает последовательной или коммуникационной связностью.
2. Если предложение содержит слова, имеющие отношение ко времени (такие, как «первый», «следующий», «затем», «после», «когда», «начать»), то модуль, вероятно, обладает последовательной или временной связностью. Например: «Подождать, пока экземпляр не получит сигнал, что пользователь вставил электронную карту, затем запросить индивидуальный идентификационный номер».
3. Если предикат предложения не содержит единого, конкретного объекта, следующего за глаголом, то модуль, вероятно, обладает логической связностью. Например, утверждение «редактировать все данные» обладает логической связностью. Высказывание «редактировать исходные данные» может обладать функциональной связностью.
4. Если предложение содержит слова вроде «инициализировать» или «обновить», то модуль скорее всего обладает временной связностью.

### 17.1.3. Зацепление и связность в ООП

В главе 1 мы отметили несколько аспектов, на основании которых классы могут рассматриваться как логическое продолжение модулей. Тем самым правила разработки для модулей легко переносятся на объекты. Различные классы должны быть зацеплены как можно меньше — не только для большей понятности, но также и для того, чтобы их легче было извлечь из одного приложения и повторно использовать в новых проектах. С другой стороны, каждый объект класса должен иметь конкретную цель, а методы обязаны способствовать этой цели тем или иным способом. То есть объект должен быть связным.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 17.1.4. Закон Деметера

Советы по написанию программ бывают абстрактными («модули должны обладать внутренней связностью и минимизировать зацепление») и конкретными («процедуры не должны содержать более 60 строк кода»). Конкретные идеи легче понимать и применять, но они часто убаюкивают программистов (и администраторов проекта) ложным чувством безопасности и отвлекают внимание от настоящей проблемы. В качестве средства снижения сложности программы правило, ограничивающее процедуры размером 60 строк, является в лучшем случае условным. Короткая процедура со сложной логической структурой бывает гораздо более трудной для понимания и правильного кодирования, чем длинная последовательность прямолинейных команд присваивания.

Аналогично, фанатическая попытка некоторых людей несколько лет назад выбросить оператор goto часто уводила в неправильном направлении. Оператор goto сам по себе просто симптом болезни, а не болезнь. Утверждение состояло не в том, что команда goto является плохой от природы и что программы, которые ее избегают, являются однозначно более хорошими, но в том, что при использовании goto труднее понять смысл программы. Важна понятность программ, а не оператор goto. Тем не менее мы не можем игнорировать тот факт, что это простое правило является полезным, его легко применять и оно эффективно в большинстве случаев в смысле достижения желаемого результата. Спросим себя: могут ли быть созданы такие руководящие правила для объектно-ориентированных программ?

Одно из таких правил было предложено Карлом Либерхером в результате его работы над средством объектно-ориентированного программирования под названием Demeter. Правило получило названия закон Деметера [Lieberherr 1989a, Lieberherr 1989b]. Имеются две формы этого закона: слабая и сильная. Обе стремятся уменьшить зацепление объектов за счет ограничения связей между ними.

Закон Деметера. В методе M класса C должны использоваться исключительно методы:

- класса C;
- классов, к которым принадлежат аргументы метода M (возможны аргументы класса C). Глобальные объекты и объекты, создаваемые внутри метода M, рассматриваются как аргументы этого метода.

Если перефразировать этот закон в терминах объектов, а не методов, то мы приходим к следующему утверждению.

Закон Деметера (слабая форма). Только следующие объекты должны выступать в роли источника данных и приемника сообщений метода:

1. Аргументы выполняемого метода (включая объект self).
2. Экземпляры получателя метода.
3. Глобальные переменные (и доступные во всей программе и те, что видны в одном файле).
4. Временные переменные, создаваемые внутри метода.

Этот закон в своей сильной форме разрешает доступ из метода только к экземплярам класса, в котором определен метод. Доступ к экземплярам суперкласса должен осуществляться исключительно посредством функций доступа.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Закон Деметера (сильная форма). Только следующие объекты должны выступать в роли источника данных и приемника сообщений метода:

1. Аргументы выполняемого метода (включая объект self).
2. Экземпляры класса, содержащего выполняемый метод.
3. Глобальные переменные.
4. Временные переменные, создаваемые внутри метода.

Полезно рассмотреть, какие правила доступа вытекают из закона Деметера, и соотнести его с концепциями зацепления и связности, описанными выше. Основной вид доступа, который запрещается правилами Деметера, — это прямая обработка (манипулирование) полей экземпляра другого класса. В противном случае один объект зависит от внутреннего представления другого объекта (что является формой зацепления внутренних данных). Соблюдение этого правила приводит к тому, что классы могут изучаться и быть понятными независимо друг от друга (поскольку они взаимодействуют между собой простым, четко определенным образом). Вирфс-Брок и Вилкерсон еще более ужесточают закон Деметера, считая, что ссылки из метода даже на переменные экземпляра того же класса должны выполняться через функции доступа [Wirfs-Brock 1989a]. Их аргументация состоит в том, что непосредственный доступ к переменным серьезно ограничивает возможность программиста дорабатывать существующие классы.

### 17.1.5. Видимость: на уровне классов и на уровне объектов

Тот факт, что класс может иметь несколько экземпляров, приводит к ряду новых соображений, касающихся контроля над зацеплением. В объектно-ориентированных языках программирования для описания видимости имен используются две модели. Они могут быть описаны как видимость на уровне класса и видимость на уровне объекта. Различие между ними сводится к ответу на вопрос: разрешено ли объекту заглядывать внутрь родственного объекта?

Языки, которые управляют видимостью на уровне классов (например, C++), рассматривают все экземпляры класса одним способом. Как мы скоро узнаем, C++ допускает контроль видимости идентификаторов, но даже в наиболее ограничительном случае (поля данных описаны с ключевым словом private) экземпляр класса всегда имеет возможность доступа к полям данных других экземпляров того же класса. То есть объектам всегда разрешен доступ ко внутреннему состоянию родственных объектов.

С другой стороны, управление видимостью на уровне объектов рассматривает индивидуальный объект как основную единицу контроля доступа. Языки программирования с видимостью на уровне объектов (например, Smalltalk) запрещают объектам доступ ко внутреннему состоянию другого объекта, даже если они оба являются экземплярами одного и того же класса.

### 17.1.6. Активные значения

Активное значение [Stefik 1986] — это переменная, с которой мы хотим выполнять некоторые действия всякий раз, когда изменяется ее значение. Система с активными значениями иллюстрирует, почему зацепление через параметры предпочтительнее, чем другие виды зацепления, особенно для объектно-ориентированных языков. Предположим, что модель ядерного реактора включает в себя класс Reactor, который содержит различную информацию о состоянии реактора. Среди наблюдаемых параметров значится температура теплоотводящей среды (воды, циркулирующей вокруг блока). Далее

PDF created with pdfFactory Pro trial version www.pdffactory.com

предположим, что эта величина модифицируется с применением классического объектно-ориентированного подхода: значение устанавливается через метод setHeat, а считывается через функцию getHeat. Класс выглядит следующим образом:

```
@interface Reactor : Object
{ ...
  double heat; ...
}
- (void)  setHeat: (double) newValue;
- (double) getHeat;
```

Представим, что программа была разработана и находилась в рабочем состоянии, когда программист решил, что было бы неплохо постоянно визуально отображать текущую температуру воды в процессе моделирования. Желательно сделать это с минимальным вторжением в тело программы. В частности, разработчик не хочет менять класс Reactor. (Например, потому, что этот класс был написан другим программистом, или же класс используется в другом приложении, где указанное свойство не требуется.)

Простое решение состоит в том, чтобы породить подкласс класса Reactor (с именем GraphicalReactor), который переопределяет исключительно метод setHeat. Этот метод теперь обновляет графические изображения перед вызовом соответствующего метода надкласса (см. ниже). Таким образом, программист будет создавать объекты не типа Reactor, а типа GraphicalReactor. Это происходит, вероятно, лишь однажды при инициализации. До тех пор пока все изменения значения температуры для объекта Reactor происходят исключительно через метод setHeat, датчик будет отражать значение корректно.

```
@implementation GraphicalReactor : Reactor
- (void) setHeat: (double) newValue
 {
   /* код, необходимый для обновления датчика */
   [ super setHeat: newValue ];
 }
@end
```

Языки Smalltalk и Objective-C поддерживают более общую концепцию, называемую зависимость. Мы обсуждаем ее в разделе 17.4.

## 17.2. Клиенты-подклассы и клиенты-пользователи

Мы несколько раз отмечали, что объект, подобно модулю, имеет две составляющие: открытую (public) и закрытую (private). Открытая часть охватывает все свойства (методы, переменные), к которым имеется доступ вне модуля. Закрытая часть включает общедоступную, а также методы и переменные, доступ к которым возможен только изнутри объекта. Пользователю сервиса, обеспечиваемого объектом (то есть клиенту), требуется знать подробности только про открытую сторону модуля. Детали реализации и другие внутренние свойства, не являющиеся важными для клиента, должны быть от него спрятаны.

Алан Снайдер [Snyder 1986] и другие исследователи отмечали, что наследование в объектно-ориентированных языках программирования означает, что классы имеют еще и третью составляющую — а именно свойства, доступные для подклассов, но не нужные другим пользователям. Разработчику подкласса данного класса потребуется, вероятно, знать больше о внутреннем устройстве исходного класса, чем пользователю экземпляров

PDF created with pdfFactory Pro trial version www.pdffactory.com

класса. Однако и разработчик подкласса не нуждается во всей информации об исходном классе.

Мы можем думать как о разработчике подкласса, так и о пользователе класса как о клиентах класса, поскольку они используют предоставляемые им средства. Однако так как эти два клиента имеют различные требования, полезно отделить клиентов-подклассов от клиентов-пользователей. Последние создают экземпляры класса и посылают им сообщения, а первые конструируют новые классы, основанные на данном классе.

В наборе классов, созданных нами в главе 8 как часть карточного пасьянса, класс Card описывает переменные r и s (содержащие ранг и масть игральной карты) как закрытые. Только методы класса Card могут иметь доступ или модифицировать эти переменные. С другой стороны, данные класса CardPile разбиты на три категории: закрытые private, защищенные protected и открытые public. Закрытая переменная firstCard доступна только изнутри класса CardPile, в то время как защищенные поля данных x и y доступны либо через класс, либо через его подклассы. Единственный общедоступный универсальный интерфейс — через методы. При устранении открытых переменных экземпляра язык программирования гарантирует, что между классом и другими компонентами программы не возникнет зацепления по данным. (Однако язык лишь предоставляет соответствующий механизм. Правильное его использование остается обязанностью программиста — например, путем описания полей данных как private или protected.)

Можно думать о развитии и модификации программного обеспечения в терминах клиентов-подклассов и клиентов-пользователей. Когда разработчик класса декларирует общедоступные свойства класса, он тем самым определяет некий контракт: класс обязан выполнить заявленные обязанности. Программист свободен во внутренней реализации класса до тех пор, пока внешний интерфейс остается без изменений (или, возможно, наращивается). Аналогично, хотя это менее принято и не столь очевидно, разработчик класса должен обеспечить интерфейс для работы подклассов. Здесь возникает стандартный и трудноуловимый источник ошибок в программном обеспечении: при изменении внутренних деталей класса подклассы перестают работать. Отделяя закрытые внутренние части класса от пользовательского интерфейса различного уровня (хотя бы только в силу соглашения), программист устанавливает границы для допустимых изменений и модификаций. Безопасность изменений, вносимых в существующий код, критична при сопровождении больших программных систем длительного использования.

Понятие клиента-подкласса может вызвать недоумение у некоторых читателей, поскольку когда экземпляр подкласса уже создан, то класс и подкласс сплавляются в единый объект. Тем не менее это понятие полезно, когда мы рассматриваем разработчиков класса. Зачастую разработчик класса и разработчик подкласса — это разные люди. Тем самым хорошая практика программирования требует, чтобы разработчик любого класса учитывал возможность порождения подкласса в будущем и обеспечивал необходимую документацию и программные средства, которые облегчат этот процесс.

## 17.3. Управление доступом и видимостью

В этом разделе мы вкратце очертим различные свойства маскировки информации в рассматриваемых нами объектно-ориентированных языках программирования. Мы будем отмечать также поддержку в различных языках концепций, анализируемых в настоящей главе.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 17.3.1. Видимость в Smalltalk

Система Smalltalk обеспечивает скромный набор средств защиты и маскировки данных и методов. Переменные экземпляра всегда рассматриваются как закрытые и доступны только изнутри методов класса-прототипа экземпляра или его подкласса. Доступ к переменным экземпляра извне объекта должен выполняться косвенным путем через функции доступа.

С другой стороны, методы всегда рассматриваются как общедоступные, и доступ к ним открыт любому объекту. Подобно тому, как нет средств, чтобы сделать поля данных экземпляра общедоступными, так нет и средств для маскировки методов. Однако некоторые методы помечаются private. Это означает, что они должны использоваться только классом и не должны вызываться клиентами-пользователями. Хорошим тоном является уважение этого соглашения и мораторий на использование закрытых методов.

### 17.3.2. Видимость в Object Pascal

Язык Object Pascal версии Apple обеспечивает небогатые средства управления видимостью полей объекта. Все поля — как данные, так и методы — доступны и для клиентов-пользователей, и для клиентов-подклассов. Только в силу традиции или соглашения поля данных считаются открытыми для разработчиков подклассов, а методы — для клиентов-пользователей. Даже если руководящие указания по стилю программирования (подобные законам Деметера) и не могут строго контролироваться системой, они все-таки остаются в силе и должны уважаться программистом. Полезно также, если программист указывает в комментариях на те методы класса, которые следует переопределить в подклассах.

Версия языка фирмы Borland является немного более мощной в этом отношении. Delphi поддерживает ключевые слова public, protected и private в смысле, очень близком к их значению в языке C++. Однако внутри раздела implementation библиотек unit все поля рассматриваются как открытые. Это позволяет экземплярам иметь доступ к закрытым полям данных своих родственников.

### 17.3.3. Видимость в C++

Из всех рассматриваемых нами языков C++ обеспечивает наиболее богатый набор средств контроля доступа к информации. Как мы отмечали в предыдущих главах, это обеспечивается тремя ключевыми словами: public, protected и private.

Когда указанные ключевые слова используются при описании полей данных класса, их эффект описывается почти непосредственно в терминах раздела 17.2. Данные, которые следуют за спецификатором доступа public:, доступны в равной мере и клиентам-подклассам, и клиентам-пользователям. Поля, определенные со спецификатором protected:, доступны только внутри класса и его подклассов и поэтому предназначены для клиентов-подклассов, но не для клиентов-пользователей. Наконец, спецификатор доступа private: предшествует полям данных, которые доступны исключительно экземплярам самого класса: они закрыты и для подклассов, и для пользователей. При отсутствии какого-либо явного спецификатора поля данных рассматриваются как private.

С точки зрения общей философии механизмы контроля языка C++ предназначены для защиты от непреднамеренного доступа, но не от злого умысла. Есть несколько способов

PDF created with pdfFactory Pro trial version www.pdffactory.com

обойти защиту. Простейший из них состоит в использовании функций, возвращающих указатель или ссылку. Рассмотрим следующий класс:

```
class Sneaky
{
  private:
    int safe;
  public:
    // инициализировать поле safe значением 10
    Sneaky() { safe = 10; }
    int &sorry() { return safe; }
}
```

Хотя поле данных safe и описано как private, ссылка на него возвращается методом sorry. Следовательно, выражение вида

```
Sneaky x;
x.sorry() = 17;
```

изменит значение поля safe с 10 на 17 даже в том случае, когда вызов метода sorry осуществляется пользовательским кодом.

Более тонким моментом является то, что спецификаторы доступа в языке C++ управляют не видимостью, а доступом к элементам данных. Классы, показанные ниже, иллюстрируют это:

```
int i;  // глобальная переменная
class A
{
  private:
    int i;
};
class B : public A
{
  void f();
};
B::f()
{
  i++; // ошибка: A::i описано как private
}
```

Ошибка возникает, поскольку функция f пытается модифицировать переменную i, наследуемую из класса A, но недоступную (так как она описана как private:). Если бы спецификаторы доступа управляли видимостью, а не доступом, то переменная i класса A была бы не видна и обновлению подверглась бы глобальная переменная i.

## Родственные экземпляры

Модификаторы доступа относятся к классу, а не к его экземплярам. То есть поля данных, описанные как private, в языке C++ не соответствуют в точности концепции, разработанной нами ранее при общем обсуждении понятия видимости. Согласно этой концепции закрытые данные доступны только самому объекту, в то врем как в C++ они открыты любому объекту того же класса. Тем самым в языке C++ объекту разрешается манипулировать закрытыми полями другого экземпляра того же класса.

PDF created with pdfFactory Pro trial version www.pdffactory.com

В качестве примера рассмотрим описание класса, приведенное ниже. Поля данных rp и ip, которые означают вещественную и мнимую части комплексного числа, помечены как private:

```
class Complex
{
  private:
    double rp;
    double ip;
  public:
    Complex (double a, double b)
    { rp = a; ip = b;
    }
    Complex operator + (Complex & x)
    {
      return Complex(rp+x.rp, ip+x.ip);
    }
};
```

Бинарная операция + перегружается с целью правильного сложения двух комплексных чисел. Несмотря на закрытую природу полей rp и ip, оператору-функции разрешен доступ к ним в аргументе x, поскольку аргумент и получатель относятся к одному классу.

Конструкторы и деструкторы, подобные функции Complex в приведенном примере, обычно описываются как public. Объявление конструктора как protected подразумевает, что только подклассы или дружественные классы (см. далее) могут создавать экземпляры этого класса, в то время как описание конструктора с ключевым словом private ограничивает создание новых экземпляров только «друзьями» и экземплярами самого класса.

Слабая форма законов Деметера частично выполняется при описании всех полей как защищенных (protected). Сильная форма реализуется при объявлении закрытых полей (private). Более подробный анализ приложения законов Деметера к языку C++ можно найти в работе [Sakkinen 1988b].

Хотя модификаторы доступа в C++ намного сильнее и гибче, чем в других рассматриваемых нами языках, эффективное использование этих свойств требует предусмотрительности и опыта. Как и в случае выбора между виртуальным и невиртуальным методами, уровень контроля, обеспечиваемый языками C++ или Delphi Pascal, приводит к тому, что легкость порождения подкласса зависит от того, что записал разработчик в исходном классе. Если класс является чрезмерно закрытым (защищенные поля данных объявлены закрытыми), то создание подкласса затруднено. Возникают серьезные проблемы, когда разработчик подкласса не может модифицировать исходную форму класса — например, если исходный класс распространяется как часть библиотеки.

Закрытое наследование

Ключевые слова public и private также предшествуют именам надклассов при описании класса. В данном случае указанные ключевые слова определяют видимость информации, наследуемой из надкласса. Подкласс, который наследует от другого класса открыто (public), соответствует понятию наследования, которым мы руководствовались до сих пор: подкласс является подтипом. Если подкласс порождается закрыто (private), то общедоступные свойства надкласса урезаются до уровня модификатора. В результате данный модификатор указывает, что надкласс используется только для конструирования,

PDF created with pdfFactory Pro trial version www.pdffactory.com

и результирующий класс не должен и не может рассматриваться как подтип исходного класса.

Когда класс порождается закрытым образом, экземпляры подкласса не должны присваиваться идентификаторам надкласса (такое возможно при открытом наследовании). Простой способ запомнить указанное ограничение — воспользоваться условием «быть экземпляром». Наследование через модификатор public означает, что выполнено условие «быть экземпляром», и тем самым экземпляры подкласса могут использоваться везде, где встречаются экземпляры надкласса. Собака Dog «является экземпляром» класса млекопитающих Mammal, и, следовательно, Dog может использоваться во всех ситуациях, где встречается Mammal. Закрытое наследование не подразумевает выполнения условия «быть экземпляром», поскольку экземпляры порожденного класса не могут во всех случаях использоваться вместо экземпляров родителя. Например, бессмысленно применять класс таблиц символов SymbolTable (наследующий от более общего класса словарей Dictionary через модификатор private) там, где требуется использование класса Dictionary. Если переменная описана с типом данных Dictionary, ей нельзя присваивать значение типа SymbolTable (это было бы разрешено, если бы наследование было открытым).

## Дружественные функции

Другой аспект видимости в языке C++ — это дружественные функции. Они представляют собой обычные функции (не методы), которые описаны с модификатором friend в определении класса. Дружественным функциям разрешается читать и записывать в поля данных объекта, описанные и как private, и как protected.

Рассмотрим описание класса, расширяющее приведенное выше определение комплексных чисел:

```
class Complex
{
  private:
    double rp;
    double ip;
  public:
    Complex(double, double);
    friend double abs(Complex&);
};
Complex::Complex(double a, double b)
{
  rp = a; ip = b;
}
double abs(Complex& x)
{
  return sqrt(x.rp*x.rp + x.ip*x.ip);
}
```

Поля данных rp и ip в структуре данных, представляющей комплексные числа, описаны с модификатором private, и тем самым недоступны вне методов класса. Функция abs, которая перегружает функцию с тем же именем, определенную для вещественных значений с двойной точностью, не является методом (это — обычная функция). Однако поскольку она описана как дружественная с модификатором friend в классе комплексных чисел, ей разрешен доступ ко всем полям данных класса, в том числе и к закрытым.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Также разрешается описывать классы и даже отдельные методы классов как дружественные. Наиболее типичная причина использования дружественных функций состоит в том, что

- функции требуется доступ ко внутренней структуре двух классов;
- необходимо вызвать дружественную функцию именно как функцию, а не как сообщение, передаваемое объекту, то есть как abs(x), а не как x.abs().

Дружественные функции являются мощным средством, но они также легко могут стать источником проблем. В частности, они вводят в точности ту разновидность зацепления данных, которая идентифицировалась в начале этой главы как вредная для разработки многократно используемого программного обеспечения. Везде, где только возможно, более объектно-ориентированные методы инкапсуляции (например, методы) должны иметь предпочтение перед дружественными функциями. Тем не менее есть случаи, когда нет других средств — например, функции требуется доступ ко внутренней структуре двух (или более) классов. В таких случаях дружественные функции являются полезной абстракцией [Koenig 1989c].

## Пространства имен

Другое недавнее изменение в языке C++ — введение пространства имен (namespace). namespace помогает предотвратить размножение глобальных имен. Ключевое слово static ограничивает область видимости одним файлом. Поэтому когда прежде требовалось сделать некоторое имя совместно используемым в двух файлах, то единственный выход состоял в том, чтобы сделать его глобальным. Подобные имена могут теперь вкладываться внутрь описаний namespace:

```
namespace myLibrary
{
  int x;
  class A
  {
    ...
  };
  class B : public A
  {
    ...
  };
  ...
}
```

Переменные, описанные внутри namespace, не являются глобальными. Если программист хочет подключить какое-нибудь конкретное пространство имен, он его явно указывает. В результате, все имена верхнего уровня, определенные внутри указанного пространства имен, становятся видимыми:

```
using namespace myLibrary;
```

Индивидуальные элементы также могут экспортироваться из конкретного пространства имен с помощью явного указания либо пространства целиком, либо отдельного имени:

```
myLibrary::A anA; // явно подключаем все пространство имен
using myLibrary::B; // импортируем только класс B
B aNewB;      // теперь B — имя типа данных
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

## Постоянные члены

В языке C++ ключевое слово const используется для указания на значение, которое остается неизменным во время существования объекта. Глобальные переменные, которые описаны таким образом, являются глобальными константами. Переменные, объявленные как const локально в процедуре, доступны только внутри нее и не могут модифицироваться после их инициализации.

Поля данных экземпляра часто ведут себя как константы, но их начальное значение не может быть определено до того, как создан соответствующий объект. Например, поля данных, представляющие собой вещественную и мнимую части комплексного числа в классе Complex (см. выше), никогда не должны изменяться раз, уж комплексное число создано. В главе 3 мы называли такие поля данных неизменяемыми. Они создаются с помощью ключевого слова const.

Так как присваивание постоянным полям данных экземпляра не разрешается, в C++ они инициализируются с помощью той же синтаксической конструкции, которая используется для вызова конструктора родителя (см. главу 7, где обсуждается вызов конструкторов родительских классов). Рассмотрим следующее описание класса:

```
class Complex
{
  public:
    const double rp;
    const double ip;
    Complex(double, double);
};
Complex::Complex(double a, double b) : rp(a), ip(b)
{
  /* пустая команда */
}
```

В этом случае поля данных rp и ip описаны через модификатор const, так что не представляет опасности сделать их полями public, поскольку они все равно не могут быть модифицированы. Чтобы присвоить им начальное значение, конструктор, по-видимому, вызывает rp и ip, как если бы они были надклассами. Это — единственный способ присваивания значений постоянным полям. Когда начинает выполняться тело конструктора, значение постоянных полей данных уже не может быть изменено.

Поля данных, описанные как ссылки, инициализируются аналогичным образом. Ключевое слово const может присоединяться также к описанию функции-члена. Однако обсуждение этой темы выходит за рамки данной книги.

Взаимодействие между перегрузкой и переопределением

Другой приводящий в смущение аспект правил видимости в C++ — это связь понятий перегрузки и переопределения. Имена функций, включая методы классов, могут перегружаться двумя или более определениями до тех пор, пока списки их аргументов достаточно различны с точки зрения компилятора. Это показывает следующее описание класса, которое перегружает функцию test за счет использования целочисленного аргумента в одном случае и вещественного — в другом:

```
class A
{
  public:
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    void test(int a)
    {
      cout << "This is the integer version\n";
    }
    void test(double b)
    {
      cout << "This is the floating point version\n";
    }
};
```

Стараясь подогнать сообщение под подходящий метод, C++ сперва просматривает область имен, в которой определен селектор сообщения, а затем ищет наиболее подходящую функцию, определенную в пределах этой области имен. Даже если есть более подходящая функция, наследуемая из другого пространства имен, она не будет рассматриваться. Это иллюстрируется следующим описанием классов:

```
class A
{
  public:
    void test(double b)
    {
      cout << "This is the floating point version\n";
    }
};
class B : public A
{
  public:
    void test(int a)
    {
      cout << "This is the integer version\n";
    }
};
```

Попытка послать сообщение test с вещественным аргументом экземпляру класса B приведет к предупреждению компилятора, поскольку в области имен, в которой компилятор нашел метод с именем test (а именно, внутри класса B), нет определения функции с вещественным аргументом. Это происходит, несмотря на то, что нужная функция может быть унаследована от класса A. Результат будет тем же самым независимо от того, описана функция test как virtual или нет. Чтобы справиться со всем этим, программисту нужно определить обе версии в классе B, одна из которых будет просто вызывать соответствующую функцию родителя:

```
class B : public A
{
  public:
    void test(double b)
    {
      A::test(b);
    }
    void test(int a)
    {
      cout << "This is the integer version\n";
    }
};
```

### 17.3.4. Видимость в Java

Как мы видели на примерах программ в языке Java, модификаторы private и public размещаются отдельно для каждого поля данных и функции-члена.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Java вводит новый интересный модификатор с именем final. «Финальный» класс не может порождать подклассы. «Окончательный» метод не может переопределяться другим методом. Переменной экземпляра, описанной как final, нельзя присваивать значения. Использование ключевого слова final позволяет компилятору оптимизировать код.

Как и в C++, модификатор private в языке Java относится к классам, а не к экземплярам. Для экземпляров одного класса разрешен доступ к закрытым полям данных друг друга.

Другое средство управления областью видимости, предоставляемое языком Java, — это пакеты. Пакет содержит классы и интерфейсы. Пакеты служат для безконфликтного управления большими областями имен. Пакет описывается с помощью ключевого слова package, которое должно быть первым оператором в файле:

```
package packageName;
```

Код одного пакета может ссылаться на классы и интерфейсы из другого пакета, либо явно указывая пакет, в котором находится объект, либо импортируя весь пакет. Следующие команды иллюстрируют первый механизм:

```
// получить тип данных foo из пакета bar
bar.foo newObj = new bar.foo();
```

Импортирование пакета делает имена всех его открытых классов и интерфейсов доступными так же, как если бы они были определены в текущем файле:

```
// импортировать все объекты и интерфейсы
// из пакета с именем bar
import bar.*;
```

При желании можно задать имена отдельных объектов и интерфейсов вместо универсального символа *.

```
// импортировать идентификатор foo
// из пакета bar
import bar.foo;
```

### 17.3.5. Видимость в Objective-C

В языке Objective-C объявление экземплярных переменных должно помещаться в интерфейсное описание класса. Нельзя объявлять новые поля в разделе реализации (даже несмотря на то, что эти поля не станут частью интерфейса), поскольку они доступны только изнутри методов (в терминах языка C++ они являются защищенными). Видимость экземплярных переменных модифицируется с помощью ключевого слова @public, которое делает все поля, следующие за ключевым словом, доступными для пользователя. Например, следующий пример показывает описание интерфейса класса Ball, который представляет собой графический объект-шар. Положение шара задается координатами, хранящимися в полях данных x и y. Оно общедоступно, в то время как направление движения и энергия шара являются защищенными.

```
@interface Ball : Object
{
  double direction;
  double energy;
@public
  double x;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
  double y;
}
```

В отличие от экземплярных переменных в разделе implementation разрешается описывать методы, не упомянутые в интерфейсной части. Такие методы оказываются видимыми и могут вызываться только в той части программного кода, которая следует за определением нового метода.

Нельзя создать метод, который бы вызывался клиентами-подклассами, но при этом был бы не доступен клиентам-пользователям. Также невозможно ввести истинно закрытые значения экземплярных переменных.

## 17.4. Преднамеренное зацепление

Хотя, как правило, программисты стараются избегать зацепления между фрагментами кода, иногда оно полезно. Представьте себе, например, что вы моделируете некий динамический объект. Макет графически отображается в графическом окне (или окнах), которое должно непрерывно обновляться.

Следуя изложенным выше соображениям, мы должны были бы избегать зацепления в программе, то есть слишком тесного соединения модели и ее графического образа. В частности, модели не следует знать, как она отображается на экране (например, как представляются значения: численно или графически). Как может модель без тесной связи с методами отображения потребовать от них обновления экрана?

Один из способов избежать слишком сильной взаимозависимости между связанными компонентами — использовать администратор зависимостей. Он является стандартной частью run-time библиотек в Smalltalk и Objective-C, но может быть легко сконструирован и для других языков (например, C++). Основная идея состоит в том, что администратор зависимостей действует как посредник, обслуживая список объектов и других компонент, от них зависящих. Модели требуется знать только об администраторе зависимостей. Объекты-отображения «регистрируют» себя с помощью администратора зависимостей, указывая при этом, что они зависят от объекта-модели. Впоследствии объект-модель при своем изменении посылает единственное сообщение администратору зависимостей. Тем самым последний узнает, что модель изменилась и все зависимые от нее компоненты должны быть оповещены об этом. Зависимые компоненты получают сообщения от администратора зависимостей, которые извещают, что модель изменилась и должны быть предприняты надлежащие действия.

Такая система обслуживания зависимостей помогает лучше изолировать компоненты друг от друга, что уменьшает количество связей внутри программы. Однако в отличие от схемы, описанной в подразделе 17.1.6, она работает, только если зависимые компоненты знают, что кто-то ожидает их изменения. Она не будет работать, если, как в случае примера объекта Reactor, желательно минимизировать вмешательство в исходный код модели.

## Упражнения

1. Разработайте средство, обнаруживающее нарушения закона Деметера, в программах, написанных на вашем любимом объектно-ориентированном языке.

PDF created with pdfFactory Pro trial version www.pdffactory.com

2. Сильная форма закона Деметера запрещает доступ к экземплярам класса потомка. Опишите преимущества и недостатки этого ограничения. Рассмотрите такие моменты, как зацепление между классами и понятность кода программы.
3. Не кажется ли вам, что сильная форма закона Деметера должна ограничивать доступ также и к глобальным переменным? Обоснуйте свое мнение. Обдумывая ответ, вы можете обратиться к статье Вульфа и Шоу [Wulf 1973].
4. Какие еще можно предложить конкретные правила, аналогичные закону Деметера, такие, что:
    - их выполнение обычно приводит к системам с меньшей взаимозависимостью;
    - ситуации, когда правила должны нарушаться, встречаются редко.

    Закон Деметера посвящен зацеплению между различными объектами. Можно ли предложить правила, которые стимулировали бы большую связность внутри объекта?

5. Защищенные данные (для клиентов-подклассов) занимают промежуточное положение по степени закрытости между открытыми и закрытыми полями. Представьте себе аналогичную промежуточную ступень между следующими положениями: «доступ к объекту разрешен кому угодно» и «доступ к объекту осуществляет только сам объект». Например: «экземпляры некоторого класса (и только они) могут иметь доступ ко внутреннему состоянию объекта». В языке C++, например, экземпляр класса имеет доступ к любому полю данных другого объекта того же класса, даже если эти поля описаны как private или protected. В других языках (например, в Smalltalk) такое не разрешается. Обсудите преимущества и недостатки каждого из подходов.
6. Другая возможная вариация на тему правил видимости для клиентов-подклассов состоит в том, чтобы разрешить доступ к непосредственному родителю класса, но не к более удаленным предкам. Обсудите достоинства и недостатки этой идеи (см. [Snyder 1986]).
7. Из рассмотренных нами языков только C++ и Delphi Pascal имеют явные средства для отделения свойств, доступных клиентам-подклассам, от свойств, доступных клиентам-пользователям. Тем не менее все языки имеют свой механизм описания намерений программиста. Зачастую используются структурированные комментарии (директивы компилятора) для передачи дополнительной информации системе, обслуживающей язык программирования. Опишите соглашения по комментированию, которые бы помогли определить уровень видимости. Затем охарактеризуйте алгоритм, который должен использоваться программным средством для обеспечения выполнения правил видимости.

## *Глава 18: Среды и схемы разработки*

Класс — это механизм инкапсулирования решения конкретной задачи. Он предоставляет определенный сервис, поставляет данные и обеспечивает необходимое поведение. Классы совместными усилиями выполняют задачу, возложенную на прикладную программу. Механизм классов полезен тем, что он позволяет разбить программу на компоненты, которые анализируются независимо друг от друга. Однако отдельные классы сами по себе редко предоставляют решение всей проблемы.

Учитывая этот факт, за последние несколько лет возрос интерес к способам, с помощью которых классы могут работать совместно над решением проблемы. Наиболее

PDF created with pdfFactory Pro trial version www.pdffactory.com

популярными стали две идеи: среда разработки и схемы разработки. В последующих разделах мы опишем каждую из них более подробно.

## 18.1. Среда разработки

Среда разработки — это набор классов, тесно сотрудничающих между собой. Вместе они представляют разработку, предназначенную для многократного использования и пригодную для решения общих проблем. Самое распространенное применение сред разработки заключается в создании графических интерфейсов пользователя или GUI-приложений. Мы подробно рассмотрим одно из таких приложений в главе 19. Однако данная концепция применяется не только при разработке пользовательского интерфейса. Например, существуют среды, которые рассчитаны на построение разнообразных редакторов, компиляторов, финансовых моделей [Gamma 1995, Deutsch 1989, Weinand 1988].

Мы всегда можем абстрагироваться и обсуждать элементы разработки, на которых основывается среда разработки, по отдельности. Однако, как правило, такая среда остается набором специфических классов, которые обычно реализованы только для одной конкретной платформы. В качестве примера можно привести среду Model-View-Controller в языке Smalltalk или библиотеку OWL на PC.

Среда диктует общую структуру приложения. Она описывает, как распределены обязанности между различными компонентами и как последние должны взаимодействовать между собой. Преимущество среды состоит в том, что разработчику нового приложения нужно всего лишь сконцентрироваться на специфике решаемой в данный момент проблемы. Более ранние разработки в той же среде не требуют длительной подготовки к повторному использованию, а их код не нуждается в переписывании.

Например, Малая Среда Разработки LAF, о которой мы рассказываем в главе 19, содержит примерно дюжину классов. Центральный класс application реализует поведение, ожидаемое от окон: способность к движению и изменению размера. Однако он не реализует функциональность, типичную для конкретного приложения. Ожидается, что пользователь среды создаст новый класс, потомка application. В нем будут переопределены некоторые методы (например, реакция на перемещение мыши или способ перерисовывания окна). Ниже приводится описание класса для карточного пасьянса, подобного тому, что был написан нами в главе 8. Здесь класс разрабатывается на основе LAF.

```
class cardApp : public application
{  public:
               // конструктор
               cardApp();
               // поведение, специфичное для приложения
               virtual void mouseButtonDown(int x, int y);
               virtual void paint();
  private:
    CardPile * piles[13];
};
```

Создавая экземпляр этого класса, пользователь сразу же получает графическое окно для своего приложения. Разработчику нет нужды беспокоиться о том, как написать весь код окна, а значит, можно сосредоточиться на поведении, специфичном для приложения.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Недостаток единой общей среды разработки в том, что она жестко ограничивает форму приложения. Например, среда LAF упрощает создание однооконных приложений, использующих кнопки, меню и текстовые области, но она вряд ли поможет вам при построении многооконного приложения. Снятие этого ограничения требует глобального пересмотра всей среды, тем самым утрачивается первопричина ее использования.

В приложениях традиционного типа код, специфичный для конкретной задачи, определяет общий поток вычислений, время от времени вызывая библиотечные процедуры (такие, как математические подпрограммы или операции ввода/вывода) для выполнения некоторых специфических функций.


При использовании среды разработки поток управления определяется средой и не меняется от приложения к приложению. Переопределяются подпрограммы, вызываемые средой, но общая структура остается универсальной. Таким образом, среда разработки доминирует над специфическим кодом.


Таким образом, мы наблюдаем смену ролей общего и специфического кодов. Поэтому иногда среда разработки называется «библиотекой наизнанку» [Wilson 1990].

### 18.1.1. Java API

Другим примером среды разработки является Java API (Application Programming Interface, программный интерфейс приложений). Он состоит из классов, используемых программистом для конструирования новых приложений Java, называемых апплетами. Как и в случае со средой LAF, создание нового класса, наследника базового класса API, — это основной механизм включения нового поведения в приложение.

Фундаментальный класс для всех Java-приложений называется Applet. Он определяет общую структуру приложения через метод main, который обычно не переопределяется программистами. Этот метод вызывает ряд других методов, которые переопределяются, чтобы обеспечить поведение, специфичное для приложения. Некоторые из вызываемых методов вкратце перечислены ниже:

init () вызывается при инициализации апплета
start () вызывается в начале работы приложения
paint (Graphics) вызывается при перерисовке окна
mouseDown (Event, int, int) вызывается при нажатии кнопки мыши
keyDown (Event, int, int) вызывается при нажатии клавиши
stop () вызывается при удалении окна

PDF created with pdfFactory Pro trial version www.pdffactory.com

Кроме того, среда API предоставляет богатую коллекцию классов для конструирования кнопок и меню, вывода текста с использованием шрифтов, работы с цветом, выполнения математических действий и многого другого.

### 18.1.2. Среда моделирования

Для того чтобы проиллюстрировать, что не все среды должны быть связаны с интерфейсами пользователя, мы вкратце рассмотрим среду моделирования, которую можно использовать, например, при моделировании бильярдного шара (см. главу 6). Как описано в последней части главы, мы могли бы начать с определения всех объектов в модели как подклассов общего класса графических объектов:

```
GraphicalObject = object
  (* поля данных *)
  link : GraphicalObject;
  region : Rect;
  (* инициализирующие функции *)
  procedure setRegion(left, top, right, bottom :
    integer);
  (* операции, выполняемые графическими объектами *)
  procedure draw;
  procedure erase;
  procedure update;
  function intersect(anObj : GraphicalObject) :
    boolean;
  procedure hitBy(anObj : GraphicalObject);
end;
```

Графические объекты занимают область на экране, знают, как нарисовать самих себя, и сообщают о пересечении с другими объектами. Таким образом, среда моделирования создана, коль скоро определен класс общего назначения для работы с графическими объектами:

```
GraphicalUniverse = object
  (* поля данных *)
  moveableObjects : GraphicalObject;
  fixedObjects  : GraphicalObject;
  continueUpdate : boolean;
  (* методы *)
  procedure initialize;
  procedure installFixedObject (newObj :
    GraphicalObject);
  procedure installMovableObject (newObj :
    GraphicalObject);
  procedure drawObjects;
  procedure updateMovableObjects;
  procedure continueSimulation;
end;
```

Основа среды моделирования — это подпрограмма для обновления всех перемещаемых объектов. Она просто проходит по списку перемещаемых объектов, сообщая каждому из них о необходимости обновить себя. Если какой-нибудь из объектов просит продолжить цикл обновления (путем вызова процедуры continueSimulation), его просьба удовлетворяется. В противном случае моделирование прерывается:

```
procedure GraphicalUniverse.updateMovableObjects;
var
  currentObject : GraphicalObject;
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
begin
  repeat
    continueUpdate := false;
    currentObject := movableObjects;
    while currentObject <> nil do
    begin
      currentObject.update;
      currentObject := currentObject.link;
    end
  until not continueUpdate
end;
```

В результате среда моделирования ничего не знает о конкретном приложении, для которого она будет использоваться, и, таким образом, может применяться для моделирования бильярдных шаров, рыбок в аквариуме, в экологическом исследовании о кроликах и волках, а также во множестве других приложений.

Другой тип моделирования «управляется событиями». События хранятся в приоритетной очереди, упорядоченной по «времени» их возникновения. Соответствующие значения извлекаются из очереди и последовательно выполняются. Каждое событие может вызвать новые события, которые также добавляются в очередь событий. Среда для разработки приложений такого типа описана в [Budd 1994].

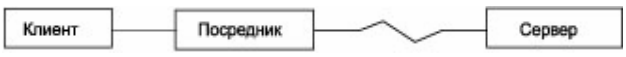
## 18.2. Схемы разработки

Среда разработки — это набор конкретных классов, разработанных для создания приложений определенного рода на базе конкретной платформы. Понятие схемы разработки гораздо более аморфно. Схема разработки воплощает в себе характерные способы решения задач, то есть идеи, которые постоянно используются при реализации приложений в разных предметных областях. Обычно такие схемы определяют взаимодействие между несколькими объектами или классами. Путем систематического изучения схем можно получить представление о широком круге готовых возможных решений любой конкретной проблемы. В будущем это поможет распознать обстоятельства, где применима та или иная схема.

### 18.2.1. Схемы с посредником

Проиллюстрируем данную концепцию на примере. Общая идея схемы может быть выражена одним словом: подстановка. В такой схеме имеется объект-клиент, который думает, что он взаимодействует с другим объектом (посредником). Однако на самом деле посредник не выполняет требуемых действий — их совершает третий объект, называемый исполнителем. Эта простая концепция может иметь множество воплощений. Вот некоторые из них.

**Передача**. Посредник — это агент, который посылает запросы по определенным каналам, например через компьютерную сеть. Фактическое исполнение запроса производит сервер на другом конце сети. Ответ на запрос пересылается обратно. Использование посредника скрывает детали передачи информации, придавая процессу видимость элементарного сообщения.



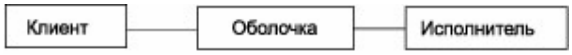

**Фасад**. Фактическая работа по обслуживанию запроса клиента выполняется не одним объектом, а скорее совокупностью взаимодействующих объектов. Посредник действует

PDF created with pdfFactory Pro trial version www.pdffactory.com

как точка фокуса, из которой запросы пересылаются к соответствующему исполнителю. В такой схеме за счет обеспечения единого простого интерфейса клиентам не нужно помнить список всех объектов, выполняющих фактическое обслуживание запросов.


**Транслятор (или адаптер)**. Фактическая работа производится одним объектом, однако протокол, используемый этим объектом, отличается от протокола клиента. Посредник действует как переводчик, переводя сообщения клиента на язык, понятный исполнителю.


**Декоратор (он же упаковщик, он же фильтр)**. В то время как исполнитель выполняет основной объем работ, посредник вносит свою небольшую лепту. При этом интерфейс остается без изменения. К примеру, «декоратор» может обеспечивать обрамление окна (например, полосы прокрутки), в то время как исполнитель обновляет содержимое окна. Клиент взаимодействует с фильтром, как будто это — исполнитель, хотя основная часть работы фактически выполняется другим объектом. При таком подходе фильтр прозрачен для клиента. Часто фильтр — это менее дорогостоящая и более гибкая альтернатива, чем создание подклассов, поскольку его можно добавить или убрать во время исполнения программы.



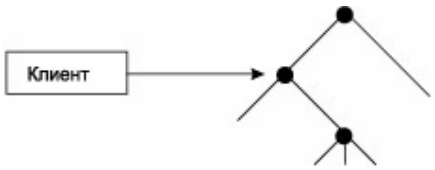

**Мост (или состояние)**. В данном случае посредник опять-таки отвечает за интерфейс, но не за выполнение поставленной задачи, которая решается несколькими возможными способами, причем выбор того или иного способа зависит от времени. Посредник вызывает на исполнение подходящий вариант. Использование посредника обеспечивает единый интерфейс, в то время как фактическое выполнение варьируется в зависимости от времени или условий (состояния).


Схемы с посредником чаще всего возникают в ситуации, когда промежуточный объект относительно «легковесен», то есть он сам не выполняет почти никакой работы, а вместо этого просто передает команды другим агентам. Например, схема интерпретатора, описанная в следующем разделе, могла бы в некотором смысле рассматриваться как разновидность схемы с посредником, поскольку клиент взаимодействует с одним объектом (интерпретатором), который представляет собой фасад гораздо более сложной структуры. Однако интерпретатор по своей природе также довольно «тяжеловесен». В этом смысле он отличается от остальных схем проектирования, описанных в этом разделе.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 18.2.2. Схемы обхода

Еще одна распространенная разновидность схем разработки возникает, когда значения данных организуются в иерархическую структуру (например, двоичное дерево). Типичная задача состоит в том, чтобы пройти по каждому узлу этого дерева. Меняются клиенты, проходящие по узлам, или сами узлы, или же и то и другое.




Посетитель. В первом случае значения узлов в иерархии однотипны, однако имеется много различных посетителей, которые проходят по узлам дерева. Примером служит двоичное дерево, в котором каждый узел относится к следующему классу:

```
template  class TreeNode
{
  public:
    TreeNode (T & initial);
    void visit (Visitor & v);
  private:
    T value;
    TreeNode * left;
    TreeNode * right;
};
```

Здесь каждый узел содержит значение и указатели на правый и левый узлы нижнего уровня; каждый из указателей может быть нулевым.

Все посетители узлов дерева должны происходить из общего класса Visitor. Они переопределяют виртуальный метод с именем action. Метод visit класса TreeNode осуществляет в определенном порядке просмотр узлов, выполняя для каждого узла действие action. Альтернативой может быть кодирование прохода по узлам со стороны клиента, однако обычно проще (к тому же, это более соответствует идеям ООП) поручить проход собственно дереву:

```
TreeNode::visit (Visitor & v)
{
  v.action(value);
  if (left != NULL) left->visit(v);
  if (right != NULL) right->visit(v);
}
```

Изменчивость возникает из-за отложенного метода action. Процедура visit не знает в точности, какое именно действие будет выполняться этим методом. С другой стороны, схема требует, чтобы все посетители были потомками общего класса Visitor.

**Интерпретатор**. Схема, которая внешне напоминает приведенную выше, возникает, когда составная структура все еще является иерархической, но уже не однородной. Примером является абстрактное синтаксическое дерево. Оно представляет структуру регулярного выражения, которое состоит из узлов выбора, повторения и символов. Узлы повторения и переключения содержат в себе поддеревья состояния, которые представляют собой различные альтернативы или повторяющуюся структуру.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Данная схема отличается от первой тем, что действие, которое необходимо выполнить, обычно закодировано в самом дереве, а не определяется внешней структурой данных. Например, нашему абстрактному синтаксическому дереву передается текстовая строка. Задается вопрос: соответствует ли строка регулярному выражению? Чтобы ответить на вопрос, каждый узел в выражении производит различные последовательности действий. Узлы-символы соответствуют только точно заданным величинам. Узел выбора проверит по очереди каждую имеющуюся альтернативу, чтобы выяснить, не подходит ли одна из них, и т. д. Клиент передает строку, которая должна быть проверена на совпадение, корню дерева, а затем получает ответ.
Эта схема также отличается от первой: структура данных не имеет иного взаимодействия с клиентом, кроме как получения входной величины и возвращения вычисленного результата. Как только метод, определенный корнем дерева, вызывается клиентом, дерево производит серию действий, дающую результат.

**Итератор**. Одна из проблем со схемой типа «посетитель» состоит в том, что клиент и структура дерева слишком тесно связаны между собой. Если клиент выполняет обход дерева, то это подразумевает, что он имеет достаточно полное представление о структуре дерева. Если дерево осуществляет обход узлов, то выполняемое действие должно быть ему известно (по крайней мере, в форме отложенного метода). Необходимость согласования между деревом и клиентом может быть уменьшена путем помещения посредника между клиентом и значениями данных.


Такой посредник называется итератором. Он производит просмотр структуры дерева, передавая клиенту значения одно за другим. Тем самым клиент имеет дело с плоской, линейной структурой, а не со сложной иерархией, используемой на самом деле для хранения значений. Интерфейс итератора обычно выглядит следующим образом:

```
initialize     инициализировать итератор корнем дерева
current вернуть текущее значение
atEnd   вернуть true, если значений больше нет
advance верейти к следующему значению
```

Используя итератор, клиент выполняет цикл, аналогичный следующему:

```
itr.initialize();
while not itr.atEnd() do
begin
  ... (* некоторое действие со значением itr.current() *)
  itr.advance();
end;
```

Итератор эффективно разделяет клиента и фактическую структуру данных. Ей нет необходимости знать, как используется ее информация, а клиент не знает о внутреннем представлении данных.

### 18.2.3. Схема двойной диспетчеризации

Схема двойной диспетчеризации применяется, когда существуют два (как минимум) источника изменчивости при обмене данными. Например, допустим, что в случае просмотра дерева, описанного в предыдущей схеме, и клиент, и узлы иерархической структуры варьируют свой тип данных. Можно привести и множество других примеров.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Представим себе, что у нас есть разные многоугольники, представленные в виде базового класса Shape, и соответствующие подклассы (Triangle, Square и т. д.). Предположим, у нас есть два устройства вывода: принтер и терминал. Они представляются подклассами класса Device. Команды, необходимые для выполнения операции вывода на эти устройства, настолько различны, что общий интерфейс невозможен. Вместо этого каждая фигура сама инкапсулирует информацию о том, как печатать ее на принтере и как изображать ее на терминале:

```
class Triangle : public Shape
{
  public:
    Triangle (Point, Point, Point);
    // ...
    virtual void displayOnPrinter (Printer);
    virtual void displayOnTerminal (Terminal);
    // ...
  private:
    Point p1, p2, p3;
};
```

Вопрос: как полиморфную фигуру Shape (или любой ее подкласс) отобразить на полиморфном устройстве типа Device? Ответ состоит в том, что нужно использовать пересылку сообщений для «вылавливания» одного из двух значений. Для определения обеих величин мы просто по очереди посылаем им сообщение. (Обобщение этой идеи на случай трех и более переменных очевидно — перешлите сообщение каждой неизвестной переменной.)

Например, сначала мы передаем команду с аргументом типа Shape на устройство вывода. Эта команда описана как отложенная в классе Device и переопределяется в каждом его подклассе. Поэтому пересылка сообщения выбирает для выполнения правильную функцию:

```
function Printer.display (Shape aShape)
begin
  aShape.displayOnPrinter(self);
end;
function Terminal.display (Shape aShape)
begin
  aShape.displayOnTerminal(self);
end;
```

Обратите внимание: метод display не знает о том, какая фигура рисуется в настоящий момент. Но каждый из методов displayOnPrinter и displayOnTerminal в свою очередь является отложенным (он описан в классе Shape и переопределяется в каждом подклассе). Предположим, что обрабатываемая фигура является треугольником Triangle. К тому времени, когда будет вызван метод класса Triangle, оба источника изменчивости (вид фигуры и тип печатающего устройства) уже привязаны к определенным категориям:

```
procedure Triangle.displayOnPrinter (Printer p)
begin
        // код, специфичный для принтера
        // вывести треугольник
        // ...
end;
procedure Triangle.displayOnTerminal(Terminal t)
{
        // код, специфичный для терминала
        // вывести треугольник
        // ...
}
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Основная сложность, связанная с данной техникой, состоит в большом количестве требуемых методов. Например, каждая фигура должна иметь по методу на каждое печатающее устройство. Несмотря на этот недостаток, данная техника очень эффективна для борьбы с одновременной неопределенностью двух (и более) величин [Ingalls 1986, Budd 1991, LaLonde 1990a, Hebel 1990] [1].

.

### 18.2.4. Классификация схем разработок

Как иллюстрируют приведенные выше примеры, схемы разработки полезны тем, что они предоставляют «словарный запас», необходимый при обсуждении возможных решений проблемы. Единая терминология позволяет группам программистов обмениваться идеями без привязки к конкретному приложению и передавать знания и традиции от одной разработки к другой.

В то время как конкретные схемы описываются довольно легко, более сложной проблемой является их категоризация, которая могла бы использоваться для записи и воспроизведения схем в последующих проектах. Большая часть текущей работы в этой области связана с созданием подобных баз знаний [Gamma 1995, Coplien 1995, Pree 1995].

Обратите внимание на то, что схемы действуют на ином уровне детализации по сравнению со средой разработки. Среда содержит в себе законченную структуру целого приложения. Напротив, схема разработки — это просто контур решения небольших специфических проблем. Реальные приложения часто составлены из кусочков, заимствованных из нескольких разных схем.

И наконец, следует подчеркнуть, что хотя схемы разработок полезны при выборе варианта разработки, определение подходящей схемы не заменяет выработку решения. Схема просто обозначает общие контуры. Проработка деталей может потребовать значительных усилий.

### Упражнения

1. Опишите роль наследования и отложенных методов в среде разработки. Возможно ли создание среды для языка программирования, не являющегося объектно-ориентированным?
2. Опишите другое приложение, которое могло бы быть создано с помощью графической среды моделирования из подраздела 18.1.2.
3. Можете ли вы привести другие примеры схемы типа «посредник»?
4. Объясните, почему инкапсуляция становится более полезной в схеме «посетитель», если проход дерева осуществляется структурой данных (деревом), а не клиентом?
5. Объясните, каким образом в языке программирования без строгого контроля типа данных (например, Smalltalk) для выполнения арифметических действий с операндами смешанного типа может быть использована техника двойной диспетчеризации. Как, в частности, сложить два числа, каждое из которых может быть как целым, так и вещественным?

---

[1] Двойная диспетчеризация впервые была описана Даном Ингалсом [Ingalls 1986], который называл ее множественным полиморфизмом. Альтернативный термин двойная диспетчеризация получил большее распространение, поскольку позволяет избежать путаницы с понятием множественного наследования

PDF created with pdfFactory Pro trial version www.pdffactory.com

# *Глава 19: Учебный пример: среда разработки*

Как мы уже отмечали в главе 18, среда разработки — это совокупность классов, которые определяют структуру, необходимую для решения конкретной проблемы. Возможно, наиболее часто среды разработки используются при создании графических интерфейсов пользователя. Действительно, в большей степени популярность объектно-ориентированных техник можно отнести за счет успеха сред типа GUI (Graphics User Interface — графический интерфейс пользователя).

В этой главе мы рассмотрим очень простую среду разработки — Малую Среду Разработки LAF (LAF — Little Application Framework). Она сознательно сделана гораздо проще, чем большинство коммерческих сред, поскольку рассчитана только на объяснение концепции среды разработки. LAF в целом содержит менее дюжины классов, в то время как в большинстве коммерческих сред их более сотни. Несмотря на это, среда LAF вполне закончена и демонстрирует большинство важных концепций, необходимых для понимания средств разработки графических приложений.

Среда LAF была создана с учетом требований переносимости на различные платформы. В то время как детали реализации могут сильно различаться, интерфейс LAF для различных платформ остается постоянным. Существуют версии среды LAF для Macintosh, PC и UNIX-систем.

## 19.1. Компоненты GUI

Среда LAF поддерживает следующие компоненты GUI: единственное окно, меню, кнопки, текстовые поля, работу с мышью и клавиатурой.

Основной способ, посредством которого программист задает конкретный вид элемента управления (кнопки, меню, окна), — это наследование. Так, существует общий класс кнопок button, из которого порождаются подклассы, то есть кнопки конкретного типа. То же самое справедливо для меню, пунктов меню и окон.

До того как объяснять структуру этих компонентов в среде LAF, нам следует вначале приобрести общее понимание программ, управляемых событиями, к которым относятся и интерфейсы.

## 19.2. Выполнение, управляемое событиями

Первое концептуальное препятствие, которое должен преодолеть программист в попытке понять технику разработки графического интерфейса, — это принцип построения программы на основе событий.

Традиционная программа следует своему собственному сценарию. Мы можем представить себе выполнение как указку, двигающуюся по программе от начала до конца. На разных этапах программа взаимодействует с пользователем, однако набор ответов, с которыми она способна справиться, весьма ограничен. Например, программа может задавать пользователю вопросы и реагировать только на простые ответы типа «да/нет», вводимые с клавиатуры.

Программа, управляемая событиями, контролируется пользователем. Она должна реагировать только на действия пользователя, и ее исполнение в основном сводится к единственному циклу. Следующий пример описывает с помощью псевдокода главный

PDF created with pdfFactory Pro trial version www.pdffactory.com

блок типичной программы, управляемой событиями. Инициализация начинает, а чистка памяти заканчивает приложение. В промежутке между ними программа просто повторяет цикл, ожидая, пока пользователь сделает следующий шаг — например, передвинет мышь, нажмет на клавишу или вставит дискету в дисковод. В ответ на активность пользователя следует реакция программы, а затем она застывает в ожидании следующего действия.

```
program main;
begin
  initialize application;
  while not finished do
  begin
    if user has done something interesting
      then respond to it
  end;
  clean up application;
end
```

«Интересные вещи», которые делает пользователь, называются событиями. Поскольку поток управления диктуется последовательностью событий, это называется выполнением, управляемым событиями. Не все события должны соответствовать действиям пользователя. Например, такие действия, как появление дискеты в дисководе, могут вызвать последовательность событий, генерируемых изнутри программы. Программа отреагирует на них таким же образом, как и на события, вызванные пользователем.

Объектно-ориентированное программирование естественным образом соответствует работе, управляемой событиями, потому что поток управления хорошо структурирован. Относительно просто создать библиотеку классов, которые имитируют реакцию на типичные события, но не выполняют никаких реальных действий. Путем порождения подклассов из классов такой библиотеки и переопределения некоторых методов мы добавляем «рабочую» часть приложения без необходимости переписывать «управляющие» разделы. Это в точности то, что делается средой разработки GUI-приложений.

Стив Барбек из фирмы Apple описал типичную библиотеку, используемую для построения управляемого событиями приложения, как «библиотеку наизнанку» [Wilson, 1990]. В традиционной программе библиотека поставляет части кода, а задачей программиста является «склеить» их в единое целое. В системе GUI приложение уже «склеено», а программист просто наделяет конкретными свойствами отдельные части библиотеки, используя порождение подклассов и переопределение методов, чтобы получить поведение, характерное для данного приложения.

## 19.3. Настройка через наследование

Основное средство, посредством которого среда разработки настраивается на создание новой программы, — это наследование. Среда предоставляет один или несколько абстрактных надклассов, которые будут переопределены программистом. Чтобы объяснить механизм настройки, мы разделим методы, определенные в этих классах, на три категории:

1. Базовые методы. Они обеспечивают выполнение реальных действий, которые полезны для адаптации программы под задачу, но которые с большой вероятностью не будут переопределяться программистом. Примером служат методы, рисующие окружности, прямые линии и другие графические объекты в окне.

PDF created with pdfFactory Pro trial version www.pdffactory.com

2. Алгоритмические методы. Они описывают абстрактный алгоритм, а конкретизация оставляется на долю других методов. В качестве примера можно привести метод, который реализует описанный выше цикл ожидания события. Такие методы обеспечивают структуру типичного приложения без выполнения какой-либо фактической работы. Как и в случае базовых методов, алгоритмические методы обычно не переопределяются новым приложением.
3. Абстрактные методы. Они выполняют фактическую работу приложения. В библиотеке среды разработки абстрактный метод, как правило, является просто пустой процедурой, которая ничего не делает. Путем переопределения абстрактных методов обеспечиваются реальные действия конкретного приложения.

При построении новой программы в среде разработки программист просто порождает несколько подклассов тех классов, которые поставляются средой приложения, переопределяя при этом абстрактные методы с целью задания необходимого поведения, специфичного для данной задачи. Среда разработки предоставляет заготовку основной программы («верхняя» часть приложения) и зачастую обеспечивает определенную базовую функциональность («нижняя» часть), оставляя программисту заполнение деталей посередине.

## 19.4. Классы в среде LAF

В среде LAF имеется шесть основных классов: application, button, menu, menuItem, staticText и editText. Три класса, на основе которых обычно порождаются подклассы, — это application, button и menuItem. Они подробно описаны ниже.

Мы создаем приложение путем порождения подклассов класса application, переопределяя в них некоторые методы. Основной цикл ожидания событий начинается с вызова метода run, унаследованного от класса application. Этот метод реализует цикл ожидания, описанный выше. Выполнение заканчивается, когда пользователь выбирает элемент меню quit, который закрывает окно приложения, то есть вызывает метод quit(), наследуемый из класса application.

Простейшее возможное приложение показано ниже. Обратите внимание, что его код состоит из:

- определения класса приложения;
- создания экземпляра данного класса;
- вызова метода run для экземпляра класса приложения:

```
class simpleApp : public application
{
  public:
    simpleApp() : application("simple application")
    { }
};
void main()
{
  simpleApp theApp;
  theApp.run();
};
```

В данном случае приложение просто отображает окно на экране и ждет, пока пользователь его закроет.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## 19.5. Класс application

Функции, предоставляемые классом application, показаны в табл. 19.1. Функция run() обычно является последней функцией, вызываемой в процедуре main. Функция run() не имеет возвращаемого результата. Метод quit() служит для прекращения работы приложения.

Методы update() и clearAndUpdate() вызываются каждый раз, когда действия пользователя могут привести к обновлению экрана. Чаще всего это необходимо вследствие какого-либо события, например перемещения мыши или нажатия

**Таблица 19.1**. Методы класса application

| Функция | Назначение |
|---|---|
| run | Начать выполнение приложения |
| quit | Выйти из приложения |
| update | Указать на необходимость обновления окна |
| clearAndUpdate | Очистить окно, потребовать обновления |
| paint | Перерисовать изображение на экране |
| mouseButtonDown | Отреагировать на нажатие кнопки мыши |
| keyPressed | Отреагировать на нажатие клавиши |
| top | Возвратить координату верхней границы окна |
| bottom | Возвратить координату нижней границы окна |
| left | Возвратить координату левой границы окна |
| right | Возвратить координату правой границы окна |
| height | Возвратить высоту окна |
| width | Возвратить ширину окна |
| circle | Нарисовать круг с центром в (x, y) и с радиусом r |
| point | Нарисовать точку (маленький круг) |
| line | Начертить линию от точки до точки |
| rectangle | Нарисовать прямоугольник |
| print | Напечатать текст начиная с заданной позиции |
| setPen | Установить характеристики пера |

клавиши на клавиатуре. Метод clearAndUpdate() сначала очищает экран перед обновлением. Чтобы выполнить обновление, вызывается метод paint(). Обычно пользователь не обращается к методу paint() непосредственно.

Метод paint() перерисовывает изображение на экране дисплея. Обычно в классах приложения он переопределяется, чтобы обеспечить требуемое поведение. Он не должен непосредственно вызываться пользователем — данный метод запускается как реакция на вызов методов update() или clearAndUpdate().

PDF created with pdfFactory Pro trial version www.pdffactory.com

Если нажата кнопка мыши, вызывается метод mouseButtonDown(). Два его целочисленных аргумента представляют собой координаты курсора относительно окна приложения на момент нажатия кнопки. Поведение по умолчанию, связанное с этим методом, — не делать ничего. Чтобы выполнить какое-либо действие, данный метод должен быть переопределен в подклассе класса application.

Событие, связанное с нажатием мыши, обычно не вызывает каких-либо действий, непосредственно влияющих на изображение на экране. Соответствующий метод просто записывает всю необходимую информацию и затем вызывает либо update(), либо clearAndUpdate() для перерисовки экрана.

Когда пользователь нажимает клавишу на клавиатуре, вызывается метод keyPressed(). В качестве аргумента он получает символ, эквивалентный нажатой клавише.

Методы top(), bottom(), left() и right() возвращают соответствующие координаты окна приложения относительно экрана, выраженные в пикселях. Обратите внимание на то, что эти величины даются в глобальной системе координат, где точка (0,0) — это верхний левый угол экрана. Почти все прочие функции, которые оперируют с координатами, используют значения в системе координат окна приложения, где точка (0,0) — это верхний левый угол окна.

В классе application доступны несколько элементарных процедур рисования и вывода на печать. Обычно вызовы графических функций выполняются во время обновления экрана. Если используется метод clearAndUpdate(), то любые действия, связанные с отображением графики, выполненные до вызова этой процедуры, будут потеряны. Таким образом, вызовы графических подпрограмм обычно производятся (прямо или косвенно) из метода paint(). Координаты для рисования даются в той же системе отсчета, что и для нажатия мыши. Иначе говоря, величина (0,0) представляет собой верхний левый угол окна, значения по оси OX увеличиваются вправо, а значения по оси OY — вниз. Последнее часто противоречит ожиданиям начинающих программистов.

### 19.5.1. Класс button

Разновидности кнопок создаются путем порождения подклассов из класса button и переопределения виртуального метода pressed. Как только кнопка присоединена к окну, этот метод вызывается автоматически при нажатии кнопки:

```
class quitButtonClass : public button
{
  public:
    quitButtonClass (window * win)
      : button (win, "Quit", 5, 5, 20, 50)
    { }
  protected:
    pressed (window * win)
    {
       ...
     }
};
```

Окно приложения передается в качестве аргумента конструктору кнопки. В процессе инициализации кнопке приписываются некие координаты относительно окна, дабы гарантировать, что кнопка будет изображаться правильно. Чаще всего кнопка описывается

PDF created with pdfFactory Pro trial version www.pdffactory.com

как часть состояния нового класса приложения и инициализируется в конструкторе класса приложения, как это показано ниже:

```
class newApplication : public application
{
  public :
    newApplication : application ("new program"),
             quitButton(this)
    { }
    ...
  private:
    quitButtonClass quitButton;
};
```

Обычно при нажатии на кнопку реакция заключается в вызове функции-члена класса приложения. Мы можем определить единственный класс общего назначения, который поддерживает это действие, и таким образом избежим необходимости определять новые классы для каждого типа кнопки. Поскольку C++ является языком со строгим контролем типов данных, новый класс должен использовать шаблон и параметризироваться с помощью нового класса приложения1. Шаблон выглядит следующим образом:

```
template
class tbutton : public button
{
  public:
    tbutton (window * win, char * t,
         int x, int y, int h, int w,
         void (T::*f)() )
               : button(win, t, x, y, h, w), fun(f)
    { }
  protected:
    void (T::* fun) ();
    virtual void pressed (window * win)
    {
      (((T *) win)->*fun) ();
    }
};
```

Последний из аргументов, f, служит указателем на функцию-член и должен соответствовать какому-нибудь методу класса T. Мы должны признать, что синтаксис, используемый для описания указателя на функцию-член, является довольно бестолковым. Эта функция запоминается в переменной fun и вызывается при нажатии кнопки. С применением этого класса приложение с двумя кнопками может быть создано следующим образом:

```
class helloApp : public application
{
  public:
    helloApp()  :
      application("hello world"),
      quitButton (this, "quit", 5, 5, 50, 20, quit),
      clearButton (this, "clear", 5, 30, 50, 20,
        clearAndUpdate)
      { };
    virtual void mousButtonDown(int, int);
  private:
    tbutton  quitButton;
    tbutton  clearButton;
};
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 19.5.2. Классы menu и menuItem

Меню во многих отношениях похожи на кнопки. В среде LAF существуют два класса для обслуживания меню: menu и menuItem. Первый класс представляет собой категорию линеек меню, а второй — конкретный элемент этой категории. Когда пользователь указывает на меню, на экране высвечиваются соответствующие элементы меню. Когда мышью выбирается элемент меню, выполняется метод selected для элемента меню. Нужное действие элемента меню обеспечивается за счет создания подкласса класса menuItem и переопределения метода selected. Это происходит так же, как и в случае с кнопками.

Меню обычно объявляются как поля данных в классе приложения и инициализируются конструктором, что иллюстрируется следующим примером:

```
class clearMenuItem : public menuItem
{
  public:
    clearMenuItem (menu & m, char * t) : menuItem(m, t)
    { }
  protected:
    virtual void selected (window *);
};
class helloApp : public application
{
  public:
    helloApp ()
     : application("hello world")
      clearMenu (this, "Clear"),
      clearItem (clearItem, "clear/C")
      { };
    virtual void mouseButtonDown(int, int);
  private:
    menu clearMenu;
    clearMenuItem clearItem;
};
void clearMenuItem::selected (window * w)
{
  w->clear () ;
}
```

Класс tmenuItem аналогичен классу tbutton. Если вы помните, класс tbutton устраняет необходимость создания нового класса кнопок, так как все, что делает кнопка — это вызывает функцию-член. Аналогичным образом класс tmenuItem наследует от класса menuItem и добавляет новый аргумент, который должен быть функцией-членом. Функция, передаваемая в качестве добавочного аргумента, должна вызываться при выборе соответствующего элемента меню.

```
template
class tmenuItem : public menuItem
{
  public:
    tmenuItem (menu & m, char * t, void (T::* f) () )
    : menuitem(m, t), fun(f)
    { }
  protected:
    void (T::* fun) ();
    virtual void selected (window * win)
    {
      (((T*) win)->*fun)();
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    }
};
```

Прочие классы среды LAF позволяют программисту размещать в окне текстовые поля (в том числе редактируемые). Эти свойства среды LAF здесь не описаны: подробную документацию по среде LAF, а также исходные тексты можно найти по адресу ftp://ftp.cs.orst.edu/pub/budd/laf.

### 19.6. Резюме

Важность среды LAF заключается не столько в функциональности, которую она обеспечивает (хотя она является простой в использовании средой для создания программ с простым графическим интерфейсом), сколько в ее типичности в качестве среды приложения. Итак, суммируем:

- Среды разработки предназначены для решения узкого круга проблем. В случае среды LAF — это создание простых однооконных приложений с кнопками и меню.
- Среда обеспечивает общую структуру программы, однако опускает все конкретные детали. В случае среды LAF определяется цикл, управляемый событиями, однако программа не знает, как конкретно она будет реагировать на события (например, на щелчок мышью или нажатие клавиши на клавиатуре).
- Среда разработки также обеспечивает средства низкого уровня. Например, среда LAF предоставляет графические примитивы общего назначения, возможность редактирования текста и т. д.
- Наследование — это основная техника, используемая программистом для специализации типового приложения. В среде LAF программист прибегает к наследованию для переопределения методов, описывающих реакцию на щелчок мышью, нажатие клавиши, выбор элемента меню и т. д.

### Упражнения

1. Постройте в среде LAF простое приложение с тремя типами кнопок.
2. Какие три группы методов можно выделить в типичном приложении среды? Приведите примеры методов каждой категории для класса application в среде LAF.
3. Проследите передачу управления различным классам, начиная с нажатия пользователем командной кнопки и заканчивая моментом, когда программа возвращается к ожиданию следующего события. Предположите, что программист выдает команду update в конце подпрограммы, связанной с щелчком мышью, чтобы вызвать обновление экрана.

## *Глава 20: Новый взгляд на классы*

Выше мы полагались в основном на интуитивное понимание того, что значит слово класс. К сожалению, интуиция не у всех одинакова. Некоторые люди думают о классах как о шаблонах, по которым штампуются готовые изделия, или как о сборочном конвейере, выпускающем какие-то предметы. Другие полагают, что класс — это обобщение записи, то есть структура с полями данных и полями функций. Существуют и другие мнения. В этой главе мы поговорим о классах в языках программирования.

Что же в точности представляет собой класс? Ответ на этот вопрос зависит, в частности, от того, какой язык программирования вы рассматриваете. В широком смысле есть две основные школы. Одни языки, такие как C++ или Object Pascal, рассматривают класс как

PDF created with pdfFactory Pro trial version www.pdffactory.com

тип данных, подобный целым числам или записям. Другие языки, такие как Smalltalk или Objective-C, считают, что класс — это объект. В следующих двух разделах мы рассмотрим некоторые вариации этих двух основных точек зрения.

## 20.1. Классы как типы

Чтобы понять смысл высказывания «классы суть типы», мы прежде всего должны попытаться уяснить значение термина тип в языках программирования. К сожалению, понятие типа данных используется для очень многих целей. Следующий список, взятый из [Wegner 1986], иллюстрирует некоторые ответы на вопрос «Что такое тип данных?»:

- Точка зрения прикладного программиста. Типы данных разбивают значения на классы эквивалентности, обладающие общими атрибутами и операциями.
- Эволюционистский подход (объектно-ориентированная точка зрения). Типы данных являются спецификациями поведения, которые могут объединяться и модифицироваться для получения нового поведения. Наследование есть механизм модификации поведения, приводящий к эволюции системы.
- Точка зрения синтаксического анализатора. Тип данных подразумевает синтаксис, который может быть использован в конкретном выражении. Например, индексы применяются только в массивах, точка — в записях или объектах, круглые скобки — в функциях.
- Точка зрения контроля типов. Типы данных налагают семантические ограничения на выражения: операторы и операнды составных выражений должны быть совместимы. Система типов есть набор правил, связывающих тип данных и любые имеющие смысл подвыражения в языке программирования.
- Верификационная точка зрения. Типы данных определяют инварианты поведения, которым удовлетворяют экземпляры.
- Системное программирование и обеспечение безопасности. Типы данных есть комплект доспехов, который предохраняет сырую информацию (битовые строки) от неверной интерпретации.
- Точка зрения реализации. Типы определяют способы распределения памяти для значений.

Приведенный выше список не претендует на полноту. Однако с него можно начать обсуждение классов как типов данных.

Такие языки программирования, как C++ и Object Pascal, рассматривают классы как обобщение структуры или записи. Класс также определяет поля, и каждый экземпляр класса содержит свои собственные значения полей. В отличие от записи класс имеет поля нового типа, которые представляют собой функции или процедуры (в отличие от полей данных, имеется только одна копия такого поля, совместно используемая всеми экземплярами класса).

Такая интерпретация классов хорошо согласуется с большинством перечисленных выше подходов. С точки зрения преимуществ для прикладного программирования и эволюции системы все экземпляры определенного типа имеют единообразные поля, и они по крайней мере реагируют на один и тот же набор команд. Также несомненно, что объектно-ориентированный подход лучше, чем стандартные методы, подходит для защиты неструктурированных битов в памяти компьютера от прямых манипуляций со стороны программиста.

Однако как только мы добавляем наследование к нашей концепции класса, тип данных превращается в нечто более сложное. В некотором смысле мы можем думать о наследовании просто как о средстве, расширяющем существующий тип записи, но такая интерпретация не является полностью адекватной по ряду причин.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 20.1.1. Как наследование усложняет понятие типа

Переопределенный метод не просто заменяет поле в записи, но, скорее, изменяет поведение объекта, причем произвольным образом. Давайте рассмотрим влияние такой замены на процесс верификации [1].

Программист разрабатывает набор классов, определяет для каждого метода входные и выходные условия и, проверяя их выполнение, доказывает правильность программы. Второй программист затем создает подклассы исходных классов, переопределяя некоторые методы. Если наша точка зрения на подклассы соответствует условиям верификации, то для экземпляра подкласса условия также будут выполнены. Тогда мы заменим в программе экземпляры класса экземплярами подкласса и можем надеяться, что получившаяся программа останется правильной. Мы пришли к принципу подстановки, обсуждавшемуся в главе 10.

В то время как обычная программистская практика и здравый смысл диктуют, что переопределенный метод не должен слишком радикально отклоняться от родительского, в общем случае нет гарантии, что поведение переопределенного метода будет как-либо связано с поведением исходного метода в надклассе. (Как правило, дочерний класс и не будет вести себя точно так же, как родительский — иначе не было бы нужды в переопределении.) Таким образом, у нас нет, вообще говоря, уверенности, что некоторые единые входные и выходные условия будут справедливы для обоих методов. Если одно из этих условий нарушено, основа нашей веры в правильность программы оказывается подорванной, и программа, вероятно, даст сбой. Типичным источником пробле

мы неоднозначности (см. главу 7) является замена программистом (возможно, непреднамеренно) одного метода другим, для которого не сохранены некоторые важные аспекты поведения.

### Пример: пасьянс

Если мы хотим доказать правильность программы карточного пасьянса, представленной в главе 8, то мы должны объяснить работу подпрограммы draw в классе TablePile, которая отображает стопку карт на карточном столе. Чтобы создать изображение, подпрограмма просто проходит в цикле по стопке карт снизу вверх, очищая поверхность под каждой картой и перерисовывая изображение. Таким образом, нижележащие карты прорисовываются, а затем частично стираются по мере того, как рисуется стопка карт.

Предположим, что данное приложение создано и мы обосновали — формально или неформально — его корректность. Теперь новый программист решает, что следует улучшить эффективность подпрограмм рисования, используя параллелизм. Поскольку графические операции являются относительно длительными, то, когда объекту-карте требуется отобразить себя, он попросту запускает фоновый процесс, осуществляющий операцию рисования, и продолжает работу в параллельном режиме. Методики, подобные

---

[1] Верифицирование — специальная техника разработки программ, подробно описанная в [Gries 1981]. При таком подходе в программу добавляются специальные условия, оформленные в виде псевдокомментариев. Выполнение условий считается необходимым для правильной работы программы. Они используются (явно или неявно) программистом при разработке и отладке программы и применяются для математически строгого доказательства правильности программы. Верификации (и в особенности их частный случай — инвариант цикла) вместе с техникой формального доказательства правильности программ предложены Т. Хоаром. Имеются специальные компиляторы, которые в режиме отладки превращают псевдокомментарии во фрагменты кода, проверяющие во время выполнения правильность заданных условий. — Примеч. перев.

PDF created with pdfFactory Pro trial version www.pdffactory.com

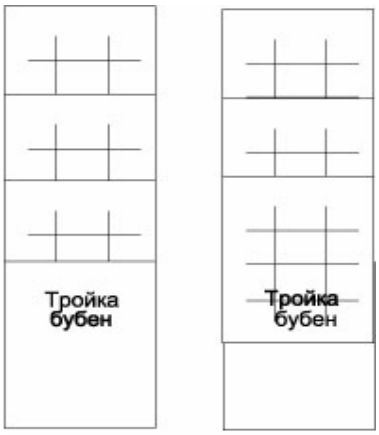



**Рис. 20.1**. Правильное и искаженное отображение стопки карт

описанным нами в главе 7, делают такую замену легкой для программирования. Все, что требуется — это создать новый класс (например, ParallelCard), который наследует все из класса Card и переопределяет только метод draw. Описание такого класса показано ниже. Затем программист может сменить ссылки на класс Card в инициализирующей части программы, чтобы использовать вместо него класс ParallelCard.

```
class ParallelCard : public Card
{
  // наследует все полностью, но изменяет рисование
  public:
    void draw()
    { if (!fork())  // порождаем процесс
       { Card::draw(); exit(0); }
    }
};
```

К сожалению, в результате такого изменения пропадает уверенность в том, что к тому времени, когда рисуется карта, все карты в стопке под ней уже прорисованы. Таким образом, не исключено (а на самом деле вполне вероятно), что будет наблюдаться случайный порядок рисования, который приведет к искажению изображения, показанному на рис. 20.1.

Существенным моментом здесь является то, что интерфейс метода и результат его работы не изменились (как исходный, так и переопределенный методы по-прежнему рисуют карту). Однако было изменено поведение метода draw. Наше доказательство правильности программы, основывавшееся на предположении, что изменений не будет, больше не проходит.

Данный пример иллюстрирует проблемы, возникающие с появлением наследования. Исходная программа была разработана и тщательно протестирована, и, возможно, некий программист поддастся соблазну думать, что до тех пор, пока выдерживается критерий «быть экземпляром», ранее проведенное тестирование будет оставаться справедливым. К сожалению, данное предположение в общем случае несправедливо, и любое изменение должно подвергаться регрессионному тестированию [Perry 1990].

## Использование контролируемых условий

Объектно-ориентированный язык программирования Eiffel [Meyer 1988a, Rist 1995] по крайней мере частично решает эту проблему. К методам присоединяются так называемые условия. Они наследуются и не могут переопределяться в подклассах (хотя могут

PDF created with pdfFactory Pro trial version www.pdffactory.com

дополняться). Компилятор генерирует код, проверяющий выполнение этих условий во время выполнения программы. Тем самым минимальный уровень функциональности метода гарантируется независимо от любых возможных переопределений. Конечно, иногда затруднительно сформулировать некоторые условия — такие, как утверждение о том, что игральная карта полностью нарисована перед выходом из метода.
Заметим, что Java предлагает интересный вариант решения этой проблемы: использование модификатора final. Класс, который объявлен как final, не может порождать подклассы, и тем самым гарантируется сохранение его функциональности.

### 20.1.2. Наследование и память

Наконец рассмотрим связь между типами данных и управлением памятью. Здесь тоже наследование приводит к тонким нюансам, которые отсутствуют для традиционных типов данных (таких, как записи и массивы). Поскольку с точки зрения «классы суть записи» экземпляр подкласса некоторого класса является расширением экземпляра исходного класса, то он может, конечно же, занимать больше места в памяти. Однако, как мы видели в главе 12, это изменение в размере усложняет кажущийся тривиальным предмет обсуждения — присваивание. Вспомним, что при присваивании переменной родительского класса значения типа подкласса требуется, чтобы:

- или при присваивании обрезались поля подкласса, которые не помещаются в области памяти адресата;
- или память для переменной должна выделяться из «кучи», но не из стека.

Суммируя, заключаем, что понятие класса почти точно соответствует большинству наших интуитивных представлений о типах, но совпадение не является полным. Второй взгляд на вещи — «классы суть объекты» — устраняет некоторые проблемы, вообще обходя стороной дискуссию о типах данных.

## 20.2. Классы как объекты

Мы подчеркивали выше, что основная философия объектно-ориентированного программирования — делегирование полномочий индивидуальному объекту. Он ответственен за свое внутреннее состояние и изменяет его в соответствии с несколькими фиксированными правилами поведения. С другой стороны, каждое действие должно быть обязанностью некоторого объекта, или же

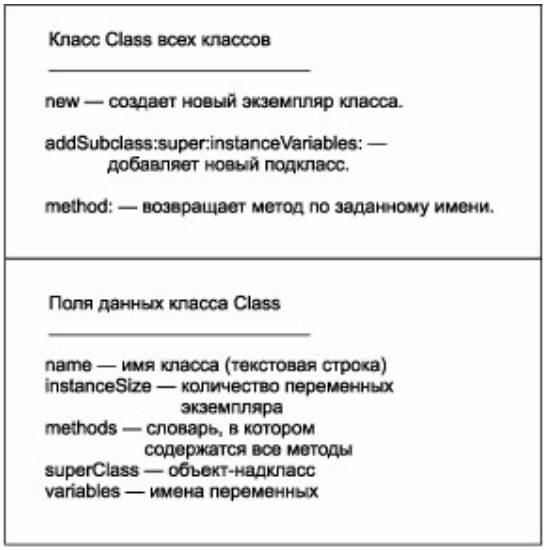



**Рис. 20.2**. CRC-карточка для класса Class

PDF created with pdfFactory Pro trial version www.pdffactory.com

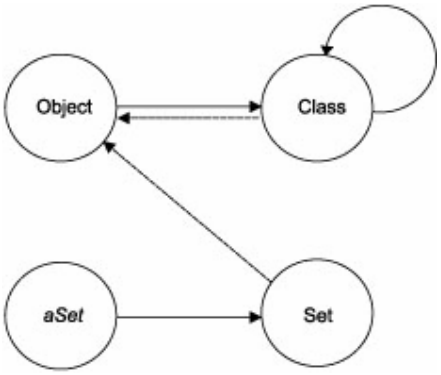



**Рис. 20.3**. Связь экземпляров и связь подклассов

оно не будет выполнено. Несомненно, создание новых экземпляров класса есть некое действие. Вопрос в том, кто (то есть какой объект) должен за это отвечать?

### 20.2.1. Фабрики по созданию объектов

Допустим, у нас есть централизованный объект «для создания объектов». Запрос на динамическое создание нового объекта преобразовывается в обращение к этому объекту. В качестве аргументов передается размер создаваемого объекта и список его методов. Хотя такой механизм работоспособен, он несколько неудачен: внутреннюю информацию о классе (его размер и методы) должен помнить некий внешний объект.

Более удачное решение инкапсулирует эту информацию в самом классе, помещая между пользователем, который хочет создать новый объект, и кодом, который осуществляет распределение памяти, управляющую прослойку. Она знает размеры объектов и их методы, тем самым это не требуется помнить ни пользователю, ни коду распределения памяти.

Действуя таким образом, приходим к схеме, в которой мы имеем один новый объект для каждого класса системы. Основная функция данного объекта состоит в создании новых экземпляров класса. Чтобы сделать это, он должен поддерживать информацию о размере класса и его методах. В практическом отношении этот объект и есть класс.

### 20.2.2. Класс Class

Каждый объект должен быть экземпляром некоторого класса, и описанный выше объект не является исключением. Экземпляром какого же класса он является? Ответ для языков Smalltalk, Objective-C, Java и аналогичных им состоит в том, что он является экземпляром класса, называемого Class. На рис. 20.2 показана CRC-карточка (обе стороны) класса Class в системе Little SmallTalk. По соглашению, значение этого объекта содержится в переменной, которая имеет то же самое имя, что и сам класс. То есть переменная Set содержит объект со структурой, аналогичной изображенной на рис. 20.3. Он отвечает за создание новых экземпляров класса.

Создание объекта инициируется сообщением new, которое определено как метод класса Class. Каждый создаваемый объект является экземпляром некоторого класса и содержит указатель на объект, представляющий этот класс. Во время пересылки сообщения упомянутый указатель используется для того, чтобы найти метод, соответствующий селектору обрабатываемого сообщения.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Чтобы понять все это, вы должны различать взаимосвязь подклассов и взаимосвязь экземпляров. Класс Class является подклассом класса Object — тем самым объект Class указывает на объект Object как на свой надкласс. С другой стороны, объект Object является экземпляром класса Class — тем самым Object указывает обратно на Class. Класс Class является классом сам по себе и тем самым является экземпляром самого себя. Если мы исследуем типичный класс, скажем Set, то объект Set будет являться экземпляром класса Class, но, кроме того, он будет (неявно) и подклассом класса Object. Конкретное множество set является экземпляром класса Set. Эти взаимоотношения проиллюстрированы на рис. 20.3, на котором сплошные линии представляют взаимосвязь экземпляров, а пунктирные линии обозначают связь подклассов.

### 20.2.3. Метаклассы и класс-методы

Выше мы отмечали, что инициализация является важной частью процесса создания объекта. Поскольку объекты отвечают за поддержание собственного состояния, было бы полезно, если бы объект, заведующий созданием новых экземпляров класса, мог бы обеспечить также надлежащую инициализацию объектов. Значение глагола «инициализировать», однако, различается от класса к классу. Если все объекты класса являются экземплярами одного и того же класса, мы не должны ожидать особенностей в их поведении: все они обязаны выполнять один и тот же набор методов и тем самым вести себя одинаково.

Связанная с этим проблема состоит в том, что создание и инициализация часто требуют больше информации, чем показано на рис. 20.2. Например, чтобы создать массив, нам нужно знать число позиций, которые надо зарезервировать для него. Другие классы могут требовать еще более подробной информации.

Обе эти проблемы требуют, чтобы мы сделали специфичным поведение объекта класса, поэтому не удивительно, что один и тот же механизм решает обе проблемы. Мы всегда настаивали на том, чтобы поведение было связано с классом, а не с индивидуальными объектами (все экземпляры класса будут совместно использовать одни и те же функции). Если мы хотим, чтобы экземпляры класса имели свое собственное поведение, единственное решение — сделать их экземплярами своих собственных классов.

Метакласс является классом классов. В Smalltalk-80 метакласс неявно и автоматически конструируется для любого класса, определенного пользователем. Каждый метакласс имеет только один экземпляр, который является собственно классом. Метаклассы организованы в иерархию «класс–подкласс», которая отражает аналогичную иерархию для исходного класса. Эта иерархия содержит метаклассы и имеет корень в классе Class, а не в классе Object.

Ниже показана часть иерархии классов в Smalltalk-80:

```
Object — надкласс всех объектов
  Collection — абстрактный надкласс всех совокупностей
    Bag — класс мультимножеств
```

Соответствующая иерархия метаклассов выглядит так:

```
Object — надкласс всех объектов
 Class — поведение, общее для всех классов
  Metaclass-Object — инициализация всех объектов
   Metaclass-Collection — инициализация коллекций collections
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
    Metaclass-Bag – инициализация мультимножеств bags
```

Код, специфический для отдельного класса, связывается с метаклассом этого класса. Например, экземпляры класса Bag содержат словарь, который используется для хранения фактических значений. Метакласс для класса Bag переопределяет

**Листинг 20.1**. Метод newDay:year: класса Date

```
newDay: dayCount year: referenceYear
  " Возвращает дату, отстоящую на dayCount дней "
  " от начала года referenceYear "
S day year daysInYear S
day <- dayCount.
year <- referenceYear.
[ day > (daysInYear <- self daysInYear: year) ]
  whileTrue:
    [ year <- year + 1
     day <- day – daysYear ].
[day <= 0]
  whileTrue:
    [ year <- year – 1
     day <- day + (self daysInYear: year) ]
 self new day: day year: year
```

метод new, чтобы производить инициализацию словаря при создании новых экземпляров. Это действие осуществляется методом new, определенном в классе Metaclass-Bag и переопределяющем метод класса Class. Сам метод приведен ниже:

```
new
  " создать и инициализировать новый экземпляр "
   super new setdictionary
```

Поскольку класс Bag является экземпляром класса Metaclass-Bag,

**Листинг 20.2**. Метод daysInYear: класса Date

```
daysInYear: yearInteger
  " возвращает число дней в году, yearInteger "
 365 + (self leapYear: yearInteger)
leapYear: yearInteger
  " 1, если год yearInteger – високосный, "
  " иначе 0 "
( yearInteger \\ 4 ~= 0 or:
  [ yearInteger \\ 100 = 0 and:
  [ yearInteger \\ 400 ~= 0 ]])
      ifTrue: [  0 ]
      ifFalse: [  1 ]
```

то указанный метод будет вызываться в ответ на сообщение new. Прежде всего, метод пересылает сообщение надклассу, который осуществляет действия, аналогичные обязанностям, показанным на CRC-карте, приведенной ранее. Таким способом надкласс создает новый объект. Как только новый объект возвращен, ему посылается сообщение setDictionary. Соответствующий метод, показанный ниже, устанавливает значения полей экземпляра для вновь созданного словаря:

setDictionary

```
  " установить новый словарь "
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
contents <- Dictionary new
```

Не вы одни считаете эти рассуждения запутанными. Концепция метаклассов имеет репутацию одного из наиболее труднодоступных аспектов в Smalltalk-80. Несмотря на это, она полезна тем, что позволяет нам придать конкретные свойства функции инициализации для индивидуальных классов, не выходя за рамки чистого объектно-ориентированного подхода. Однако, учитывая запутанную природу метаклассов, большинство программистов очень благодарны системе, которая позволяет их не замечать. Например, в Smalltalk-80 класс-методы определяются при просмотре броузером базового класса, а не метакласса. Щелчок «класса» или «экземпляра» во втором окошке показывает, описывается ли метакласс или собственно класс. Аналогично в языке Objective-C методы, перед которыми в первой колонке стоит знак «плюс» (так называемые методы-«фабрики»), связаны с метаклассами, в то время как методы, которые начинаются с «минуса», являются класс-методами.

Дальнейшая информация о метаклассах и метапрограммировании может быть найдена в работе [Kiczales 1991].

### 20.2.4. Инициализация объектов

Мы продемонстрировали, как методы классов, определяемые посредством метаклассов, используются для решения одной из проблем, очерченных в начале этого раздела — проблемы специализированной инициализации. Вернемся теперь ко второй проблеме, — инициализации объектов, для которых требуется больше информации, чем просто их размер. Мы покажем (наряду с другими вопросами), как методы класса могут быть использованы для решения этой проблемы.

В качестве примера используем класс Date для языка Smalltalk-80. Экземпляры класса представляют собой дату определенного года. Каждый экземпляр класса Date хранит два значения: номер года и число в диапазоне от 1 до 366, то есть день. Новый экземпляр класса Date может быть создан различными способами. Например, сообщение Date today порождает экземпляр класса Date, представляющий текущую дату. Даты могут быть также в явном виде определены пользователем путем задания года и дня. В этом случае вызывается код, показанный в листинге 20.1. Он осуществляет небольшую проверку, гарантируя, что номер дня положителен и не превосходит числа дней в году. Когда есть уверенность, что значения допустимы, код использует метод new для создания нового объекта и инициализации его заданными значениями.

Сообщение daysInYear:, вызываемое в методе, показанном в листинге 20.1, иллюстрирует другое использование методов класса: здесь обеспечивается поведение, связанное с классом в целом, а не с каким-либо конкретным экземпляром. Объект Date на запрос о количестве дней в определенном году возвращает требуемое целое число без фактического построения нового экземпляра класса. Он делает это, используя тот же самый механизм определения метода класса за исключением того, что в данном случае метод класса возвращает целое число, а не новое значение. Методы daysInYear и leapYear (високосный-год), которые вызываются при этом, показаны в листинге 20.2.

### 20.2.5. Подстановки в Objective-C

Интересной иллюстрацией точки зрения «классы суть объекты» является понятие подстановки класса в языке Objective-C. Мы часто видели ситуации, когда программист желает подставить вместо одного класса другой в уже существующем приложении.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Обычно новый класс наследует все из исходного класса и модифицирует только небольшую часть поведения. Примерами являются классы GraphicalReactor из главы 12 и ParallelCard, описанный выше в этой главе. В обоих случаях было бы желательно изменить все сообщения о создании объектов, чтобы использовать новый класс вместо исходного.

Язык Objective-C обеспечивает уникальную альтернативу для этих действий, которая значительно меньше затрагивает исходный код. Любому классу можно дать инструкцию подставить себя вместо другого класса. В результате подставляемый объект занимает место объекта исходного класса. Например, пользователь может написать следующий оператор:

```
[ GraphicalReactor poseAs: [ Reactor class ]];
```

Все последующие ссылки на класс Reactor, включая сообщения о создании экземпляров класса, будут отсылаться к классу GraphicalReactor. Наиболее часто объект, подставляемый вместо другого, — это подкласс замещаемого класса. Тем самым большинство сообщений будет переадресовано обратно исходному

**Листинг 20.3**. Описание класса со статическими полями

```
//
// --------- класс Card
//
class Card
{
  public:
    // конструктор
    Card(int s, int c);
    // константы
    static const int CardWidth;
    static const int CardHeight;
...
};
const int Card::CardWidth = 68;
const int Card::CardHeight = 75;
```

классу (являющемуся теперь надклассом).

## 20.3. Данные класса

Независимо от того, какого взгляда на классы мы придерживаемся, часто желательно иметь область данных, которая является общей для всех экземпляров класса. Например, все окна Windows можно разместить в одном связном списке или все карты Cards — в единой колоде.

Стандартное решение состоит в использовании глобальной переменной. Она несомненно доступна и является общей для всех экземпляров класса, поскольку она является таковой для всех объектов. Однако такая широкая доступность оскорбляет объектную философию, которая состоит в ограничении доступа и передачи ответственности за поведение отдельным индивидуумам. Таким образом, объектно-ориентированная точка зрения требует найти другую альтернативу. Она состоит в том, чтобы создать значения, доступные для экземпляров класса, но недоступные для объектов других типов. Такие значения называются переменными класса.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Тонкость состоит в инициализации переменных класса. В определенном смысле все экземпляры класса равны: они одинаково функционируют, и имеется только одна копия любой переменной класса. Однако необходимо:

- инициализировать данные класса;
- сделать это не более одного раза.

Как в языке Smalltalk, так и в Objective-C система гарантирует, что сообщение initialize пересылается прежде любого другого. Отклик на метод initialize может быть использован затем для установления значения любой переменной класса. Забота о сообщении initialize обеспечивается неявным образом, и здесь нет необходимости вообще в явном виде вызывать метод initialize. Язык Java имеет аналогичное средство, только в этом случае блок инициализации даже не имеет имени.

Реализация переменных класса поддерживается не всеми языками программирования, которые мы рассматриваем, а там, где это возможно, механизмы являются различными. В последующих разделах мы опишем, как создаются переменные класса в различных языках программирования.

### 20.3.1. Переменные класса в Smalltalk

Мы определяем переменные класса, просто перечисляя их по именам при создании нового класса. Например, ниже показано описание класса Date. Как сказано в подразделе 20.2.4, экземпляры класса Date используются для представления даты. Переменные класса содержат несколько массивов информации, которая полезна при манипулировании с датами, включая число дней в месяцах невисокосных лет, названия месяцев и дней недели и т. д. Внутренним образом такие переменные обрабатываются как переменные экземпляра метакласса, связанного с классом. Инициализация переменных класса выполняется как часть метода класса initialize.

```
Magnitude subclass: #Date
  instanceVariableNames: 'day year'
  classVariableNames:
'DaysInMonth FirstDayOfMonth MonthNames SecondsInDay
    WeekDayNames'
  poolDictionaries: ''
  category: 'Numeric-Magnitudes'
```

### 20.3.2. Переменные класса в C++

Язык C++ особым образом воспринимает ключевое слово static, когда оно используется в описании класса. Здесь это слово подразумевает, что создается одна копия значения, которая используется совместно всеми экземплярами класса. Такие значения (переменные класса в нашей терминологии) называются статическими элементами в C++. Обычные правила видимости (определяемые ключевыми словами private, protected или public) применяются к статическим элементам для ограничения доступа к ним извне методов, связанных с классами.

Статический элемент инициализируется вне определения класса способом, похожим на инициализацию глобальных переменных, причем однозначность имени класса помогает связать определение с нужным классом (листинг 20.3). Как и в случае глобальных переменных, инициализация статического элемента производится в программе только один раз.

PDF created with pdfFactory Pro trial version www.pdffactory.com

Поскольку в классе используется только одна копия статического элемента, доступ к открытому (public) статическому элементу может осуществляться напрямую. Например, класс CardPile отображает стопку карт следующим образом:

```
void CardPile::display()
{
  if (top == nilLink)
    game->clearArea(x, y, x+Card::CardWidth,
                y+Card::CardHeight);
  else
    top->draw();
}
```

Определив поля CardWidth и CardHeight класса Card таким образом, мы избегаем создания отдельных констант в каждом экземпляре класса.

Заметьте, что статические элементы не обязаны быть объявлены открытыми — если они не открыты, то их доступность подчиняется обычным правилам видимости. Язык C++ также допускает, чтобы методы были объявлены как static. Статические методы могут обращаться только к статическим данным и во многих отношениях похожи на класс-методы в языках Smalltalk и Objective-C.

### 20.3.3. Переменные класса в Java

Язык Java, следуя C++, использует ключевое слово static для указания переменной класса. Поля данных и методы, описанные в стеке, могут быть помечены как static, тогда они будут применимы к самому классу, а не к его экземплярам.

Статические переменные могут иметь инициализаторы, которые выполняются при загрузке определения класса. Кроме того, блок кода разрешается помечать как static, тогда он также будет выполняться во время загрузки. Следующий пример иллюстрирует это. Здесь описывается статический массив значений, который инициализируется числами от 1 до 12:

```
class statTest
{
        static final size = 12;
        static int arr[] = new int[size]; // объявить массив
        static { // эти команды выполняются при загрузке класса
       for (int i = 0; i < arr.length; i++)
         {
           arr[i] = i + 1;
         }
       }
}
```

Комбинация ключевых слов static и final создает инициализированное поле, которому не может быть присвоено значение.

Как статические переменные, так и статические методы обычно доступны через имя класса. Однако для удобства к ним можно обратиться и через экземпляр класса.

PDF created with pdfFactory Pro trial version www.pdffactory.com

### 20.3.4. Переменные класса в Objective-C

В языке Objective-C нет явной поддержки переменных класса. Мы можем получить нечто похожее, объявляя простые статические (static) переменные языка C в части implementation описания класса. Такие переменные доступны только внутри файла, содержащего реализацию, и к ним нет доступа у клиентов-подклассов и у клиентов-пользователей. Например, методы в нижеследующем классе Date могут ссылаться на статический массив dayNames. (Этот трюк работает также в C++ до тех пор, пока все функции,

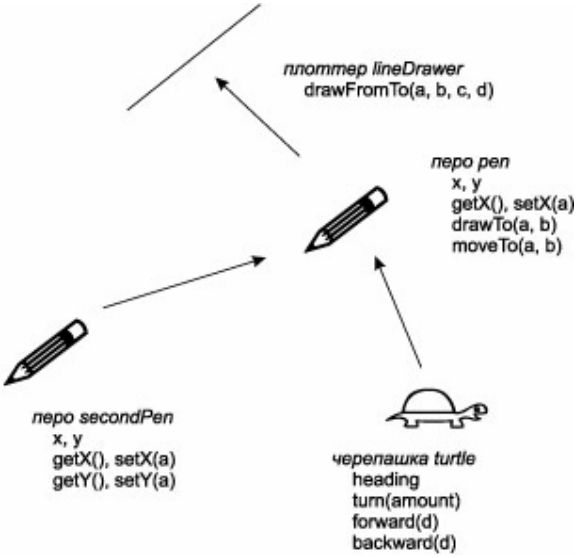



**Рис. 20.4**. Схемы делегирования

обращающиеся к данным, находятся в одном файле.)

```
# import "Date.h"
static char *dayNames[ ] = {"Воскресенье",
        "Понедельник", "Вторник", "Среда", "Четверг",
        "Пятница", "Суббота"};
@implementation Date
...
@end
```

## 20.4. Нужны ли классы?

Учитывая все нюансы объектов, экземпляров, классов, метаклассов и тому подобного, невольно задумаешься, нельзя ли создать объектно-ориентированный язык без привлечения классов? Оказывается, можно, хотя не вполне ясно, чем программирование в таких «деклассированных» объектно-ориентированных языках оказывается проще или быстрее, чем в языках с классами. Также не ясно, являются ли получающиеся программы в чем-то более эффективными.

PDF created with pdfFactory Pro trial version www.pdffactory.com

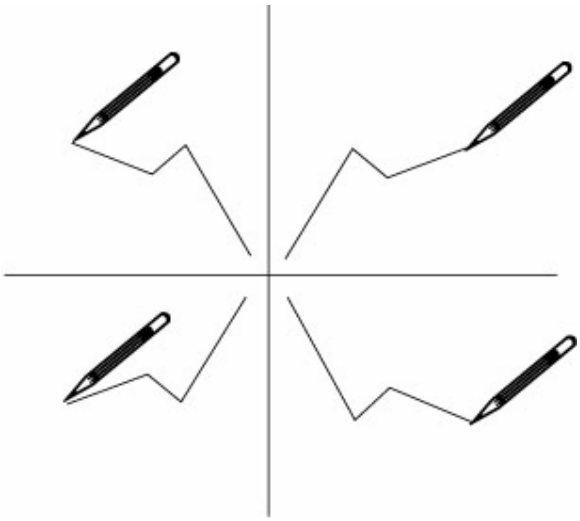


**Рис. 20.5**. Результат работы калейдоскопического пера

### 20.4.1. Что такое знание?

Объектно-ориентированный подход связан с тем представлением, что знание есть отнесение конкретной вещи к абстрактному типу. Платон, например, утверждал, что наше понятие лошади не основывается на какой-либо конкретной лошади, но выводится из идеи «лошадности». Все физические объекты — только приближения к более совершенной абстракции; они — всего лишь бледная тень идеи.

Альтернативное представление утверждает, что люди получают информацию путем изучения конкретных объектов, а затем отбрасывают несущественные частности, чтобы построить общую абстракцию. Например, концепция слона основывается на конкретном слоне, которого данный человек в молодости видел в зоопарке. Затем человек создает образ второго слона, соотнося его с более ранним образцом. У этого нового слона уши могут быть меньше, и, таким образом, познающий заключает, что размер уха не является существенной характеристикой «слоновости». Путем многократного повторения такого процесса сравнения человек вырабатывает общее представление. Каждая абстракция приводит обратно к конкретным экземплярам, на которых она и основывается.

### 20.4.2. Делегирование полномочий

В языках программирования идея общности между конкретными экземплярами известна как делегирование [Lieberman 1986]. При делегировании нет классов — вместо этого программист создает конкретные экземпляры объектов, и вся функциональность связывается с конкретными объектами. Всякий раз, когда объект оказывается похожим на уже существующий объект, программа может делегировать часть поведения нового объекта объекту-оригиналу. Любое сообщение, которое не распознается новым объектом, будет переадресовано объекту, которому осуществлено делегирование. Он в свою очередь может делегировать свое поведение другому объекту и т. д.

Таким образом, мы можем построить язык программирования из объектов (которые содержат переменные и методы) и делегирования (посредством которого объект переадресовывает выполнение любых нераспознаваемых методов другому объекту). Совместное использование кода происходит в таких языках через применение общих делегатов.

PDF created with pdfFactory Pro trial version www.pdffactory.com

В качестве иллюстрации построим простую графическую систему. Предположим, что у нас есть объект, который рисует линии и реагирует на единственное сообщение drawFromTo(a, b, c, d). В качестве реакции на это сообщение рисуется отрезок сплошной линии из точки с координатами (a,b) к точке с координатами (c,d).

Прежде всего мы построим перо, являющееся инструментом рисования, которое помнит свои координаты. Объект-перо инкапсулирует две переменные, x и y, а также определяет методы установки и сообщения этих переменных: getX(), getY(), setX(a), setY(b). Затем объект-перо определяет два метода рисования — а именно, moveTo(a,b), который только перемещает перо без рисования, и drawTo(a,b), который рисует линию. Эти методы могут быть определены с помощью следующего псевдокода:

```
method moveTo(a, b)
    self setX(a)
    self setY(b)
end
method drawTo(a, b)
    self drawFromTo(self getX(), self getY(), a, b)
    self moveTo(a,b)
end
```

Объект-перо делегирует ответственность за метод drawFromTo объекту-линии (рис. 20.4).

Предположим, что программист хочет создать второе перо. Используя технику делегирования, он прежде всего обеспечивает описание объекта, соотнося его (по возможности) с уже существующими объектами. Одна из форм описания может выглядеть так: «второе перо должно вести себя в точности как первое, но поддерживать свои собственные координаты». Из этого описания ясно, что второе перо обязано содержать свои собственные переменные и определять методы для setX и т. д. Однако поскольку оно делегирует себя первому перу, это будут единственные методы, которые надо определить; оставшиеся детали поведения будут унаследованы от первого пера. Когда второму перу посылается сообщение, то получатель (второе перо) пересылается как составная часть сообщения дальше по пути делегирования. Когда следующие сообщения посылаются объекту self (клиенту в терминологии Либермана), поиск начинается снова с исходного получателя. Тем самым сообщения setX и getX, — используемые, например, в методе drawTo, будут сопоставляться с методами второго пера, а не первого. Этот процесс сопоставления аналогичен способу, при котором связывание метода всегда начинается с базового класса получателя. Проблемы, возникающие при этом способе, проявляются и в своем «делегированном» эквиваленте.

Делегирующие объекты не всегда должны переопределять переменные. Предположим, мы хотим создать калейдоскопическое перо, которое производит отражение относительно осей x и y, рисуя четыре линии вместо одной линии для исходного пера (рис. 20.5). Мы можем ввести объект, который переопределяет только метод drawTo; все остальное поведение делегируется первоначальному перу. Поскольку координаты x и y являются координатами исходного пера, изменения в пере типа «калейдоскоп» приводят к изменениям в первоначальном пере. Новый метод drawTo выглядит следующим образом:

```
method drawTo(a, b)
  self drawFromTo(self getX(), self getY(), a, b)
  self drawFromTo(- self getX(), self getY(), – a, b)
  self drawFromTo(self getX(), – self getY(), a, – b)
  self drawFromTo(- self getX(), – self getY(), – a, – b)
  self moveTo(a,b)
end
```

PDF created with pdfFactory Pro trial version www.pdffactory.com

Теперь предположим, что программист хочет определить перо типа «черепашка», которое сохраняет не только координаты, но и ориентацию [Abelson 1981]. Кроме собственно рисования, «черепашку» можно научить поворачиваться и двигаться вперед или назад относительно текущей ориентации. Если мы используем существующее перо для сохранения координат «черепашки», ей потребуется определить единственную переменную, а также методы turn(amount), forward(amount) и backward(amount).

Интересным свойством систем, основанных на принципе делегирования, является их способность динамически изменять делегатов. После того как было сконструировано устройство «черепашка», а затем пользователь перенес делегирование с обычного пера на перо dasedPen, «черепашка» внезапно развивает в себе способность рисовать не только сплошные, но и пунктирные линии. (Естественно, делегирование к объекту, который не понимает всех требуемых сообщений, может привести к краху программы.)

В определенном смысле отношения объект/делегат подобны отношениям экземпляр/класс, за исключением того, что в первом случае нет двух сущностей — делегат просто является другим объектом. Тем не менее существует распространенная практика создания классо-подобных объектов-фабрик, которые не делают ничего, кроме дублирования существующего объекта. Например, удается произвести «фабрику черепашек», которая по запросу выпускает новую «черепашку», независимую от всех других «черепашек».

Основной литературой по делегированию является работа [Lieberman 1986]. Его статья показывает, как с помощью делегирования мы можем смоделировать механизм наследования. Обратное утверждение «с помощью наследования удается смоделировать делегирование» также было продемонстрировано [Stein 1987]. Язык программирования Self, основанный на делегировании, был описан Унгаром [Ungar 1987]. Томлинсон [Tomlinson 1990] приводит интересный анализ затрат времени и памяти при делегировании и при наследовании, приходя к заключению, что наследование в общем случае является более быстрым и, как ни удивительно, требует меньше памяти.

### Упражнения

1. Какие аспекты понятия тип данных не описываются категориями Вегнера [Wegner 1986]? Определите новые точки зрения, чтобы отразить эти аспекты.
2. Изучите технические приемы верификации в стандартных языках программирования (хорошее объяснение их приводится в работах [Gries 1981, Dijkstra 1976]).

## *Глава 21: Реализация объектно-ориентированных языков*

Целью данной книги не является снабдить читателя подробными инструкциями по реализации языков программирования. Тем не менее общее понимание проблем, возникающих при воплощении объектно-ориентированных языков, а также различные способы борьбы с ними во многих случаях помогут читателю лучше понять объектно-ориентированные технологии. В частности, станет понятно, в чем именно объектно-ориентированные системы отличаются от более традиционных средств. В этой главе содержится обзор некоторых наиболее важных техник реализации, а также ссылки на необходимую литературу для читателей, которые захотят продвинуться дальше.

PDF created with pdfFactory Pro trial version www.pdffactory.com

## 21.1. Компиляторы и интерпретаторы

В широком смысле имеются два основных подхода к воплощению языков программирования: компиляторы и интерпретаторы. Компилятор переводит программу пользователя в машинный код компьютера, на котором будет выполняться программа. Вызов компилятора осуществляется как процесс, не зависящий от выполнения программы. Интерпретатор же обязательно присутствует во время исполнения программы и является собственно той системой, которая выполняет программу [1].

В общем случае программа, оттранслированная компилятором, будет выполняться быстрее, чем программа, выполняемая под управлением интерпретатора. Но полное время, затраченное на проектирование, ввод текста и запуск на выполнение для компилирующей системы может быть больше, чем для интерпретатора. Более того, когда во время выполнения программы возникает ошибка, компилятор практически всегда может предложить для локализации места вероятной ошибки всего лишь сгенерированный ассемблерный текст. Интерпретатор же обычно покажет ошибку в исходном тексте программы, введенном пользователем. Тем самым имеются достоинства и недостатки у обоих подходов.

Хотя одни языки обычно компилируются, а другие, как правило, интерпретируются, в собственно языке программирования нет внутренних причин, которые вынуждают программиста, реализующего язык, выбирать один путь, а не другой. C++ обычно компилируется, но имеются и интерпретаторы C++. С другой стороны, язык Smalltalk почти всегда интерпретируется, однако созданы и экспериментальные Smalltalk-компиляторы.

## 21.2. Компиляторы

Типичной отличительной чертой компиляторов является то, что некоторая информация теряется при переводе исходного текста программы в машинный код. Это наиболее заметно при преобразовании символических имен в адреса ячеек памяти. Так, к локальным переменным внутри процедуры скомпилированный код адресуется не по их именам, а через фиксированный сдвиг относительно начала блока памяти, создаваемого при входе в процедуру1. Аналогично поля данных в записи описываются путем указания их сдвига относительно начала блока, а не через имена.
Предположим, что процедура содержит переменную x и запись d, которая в свою очередь имеет поле данных y. Допустим, что x запоминается в ячейке локального блока выделенной памяти с индексом 20, а d начинается с ячейки 28. Пусть поле данных y начинается с восьмого байта записи d. При этих предположениях оператор присваивания

```
x:= d.y
```

может быть оттранслирован в одну ассемблерную команду, которая перемещает содержимое слова, расположенного в тридцать шестом байте от начала блока, в двадцатую ячейку от начала блока:

---

[1] Как и в случае большинства классифицирующих разбиений, между чистыми конечными точками посередине имеется большая серая область. Существуют компиляторы, которые компилируют в интерактивном режиме, иногда даже во время выполнения программы (и по крайней мере на стадии пошаговой отладки). Такие компиляторы обладают некоторыми достоинствами интерпретаторов, в то же время обеспечивая во время выполнения программы преимущества, свойственные компилирующим технологиям. Аналогично некоторые интерпретаторы могут переводить программу в промежуточное представление или псевдокод.

PDF created with pdfFactory Pro trial version www.pdffactory.com

```
move AR+36, AR+20
```

Обратите внимание: операторы ассемблера используют не символические имена переменных, а только их смещения относительно начала блока.

Как мы заметили в предыдущих главах, объект напоминает запись или структуру в традиционных языках программирования. Подобно записям, полям данных (переменным экземпляра) приписываются фиксированные смещения относительно начала объекта. Подклассы могут только расширять эту область памяти, но не сокращать ее, так что объем памяти, выделенной для класса-потомка, строго больше, чем у класса-предка. Смещения данных для подклассов должны соответствовать расположению аналогичных полей в надклассах (рис. 21.1).

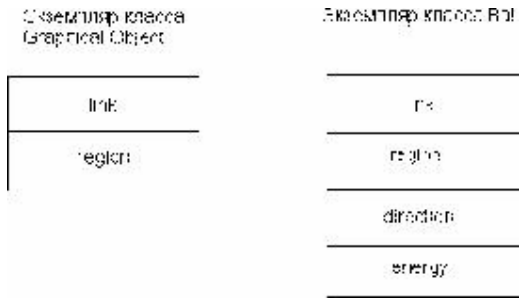



**Рис. 21.1**. Соответствие между полями данных в родительском и дочернем классах

Рассмотрим классы GraphicalObject и Ball для модели бильярда, описанной в главе 6. Класс GraphicalObject содержит поля link и region, а класс Ball добавляет к ним поля direction и energy, одновременно сохраняя для полей класса-предшественника в точности те же смещения. То, что смещения полей для родительского класса сохраняются в дочернем, очень важно: это позволяет методам, определенным для родительского класса, обрабатывать данные экземпляра с использованием постоянных смещений, так что эти функции будут работать правильно независимо от того, к какому классу (родителю или потомку) относится аргумент. Например, метод moveTo класса GraphicalObject будет вести себя как полагается независимо от класса объекта-получателя, поскольку этот метод пользуется только областью region объекта.

### 21.2.1. Проблема «срезки»

То обстоятельство, что дочерний класс может только расширять область данных, определенную родительским классом, решает проблему, описанную в предыдущем разделе. Это позволяет компилятору так генерировать код для процедуры родительского класса, что она будет работать и для объектов, принадлежащих к дочернему классу. Но это же свойство создает другую проблему.

Как мы видели в предыдущих главах, полиморфная переменная — это переменная, которая объявлена как принадлежащая к одному типу данных, тогда как на самом деле она содержит значение, относящееся к другому типу. В объектно-ориентированных языках переменная типа родительского класса, как правило, может содержать значения типа потомков.

Когда компилятор устанавливает размер локального блока выделяемой памяти, он, вообще говоря, знает только декларированный тип переменной, а не тип, который она будет иметь во время исполнения программы. Таким образом, возникает вопрос: сколько

PDF created with pdfFactory Pro trial version www.pdffactory.com

памяти должно быть выделено под локальный блок? Как мы выяснили в главе 12, большинство языков программирования выбирают одно из двух решений этой проблемы:

- блок выделяемой памяти содержит указатели, а не сами значения;
- блок выделяемой памяти содержит только поля данных родительского класса. При операции присваивания те поля данных потомка, которые выходят за пределы родительского класса, отбрасываются (срезаются).

У каждой из двух альтернатив имеются свои преимущества, так что мы не будем комментировать, какая из них лучше. Однако для вас как для программиста важно понимать, какая именно техника используется в системе, в которой вы работаете.

### 21.2.2. Соответствие между методами и сообщениями

Наиболее новаторское свойство объектно-ориентированного программирования с точки зрения реализации состоит в том, что интерпретация сообщения может зависеть от типа (класса) получателя. То есть различные классы объектов могут выполнять различные процедуры в качестве реакции на одно и то же сообщение. Для нашей графической модели класс Wall реагирует на сообщение draw иным образом, чем класс Ball.

По этой причине каждый объект обязан содержать какой-то способ определения, какая процедура должна вызываться для сообщения, воспринимаемого объектом. Более того, мы хотим, чтобы механизм, который используется для связывания метода и процедуры, работал как можно быстрее. Для компилирующих языков программирования техника, наиболее часто используемая с целью обеспечить скорость, называется таблица виртуальных методов.

### 21.2.3. Таблицы виртуальных методов

Одно из возможных решений проблемы соответствия между методами и сообщениями состоит в том, чтобы разместить поля для методов в точности тем же способом, как выделяется память для полей данных. Значениями полей методов являются указатели на соответствующие функции, как это показано на рис. 21.2.

Для того чтобы вызвать нужный метод, достаточно взять значение по правильному смещению внутри объекта, разыменовать его, чтобы получить процедуру, и затем вызвать ее. Однако такой подход, хотя он и приемлем по скорости выполнения, является затратным с точки зрения другого важного ресурса — а именно памяти. Каждый объект должен отводить память (один указатель) для каждого метода. Более того, создание объекта включает в себя инициализацию всех полей методов, что представляет собой ненужные затраты. Возможен компромисс между скоростью и памятью, который основан на том, что все экземпляры одного класса должны совместно использовать одни и те же методы. При таком подходе для каждого класса создается единственная таблица, называемая таблицей виртуальных методов, и все экземпляры класса содержат один указатель на нее (рис. 21.3). Инициализация нового экземпляра подразумевает установку этого указателя на таблицу виртуальных методов.

PDF created with pdfFactory Pro trial version www.pdffactory.com


**Рис. 21.2**. Методы, реализованные как поля

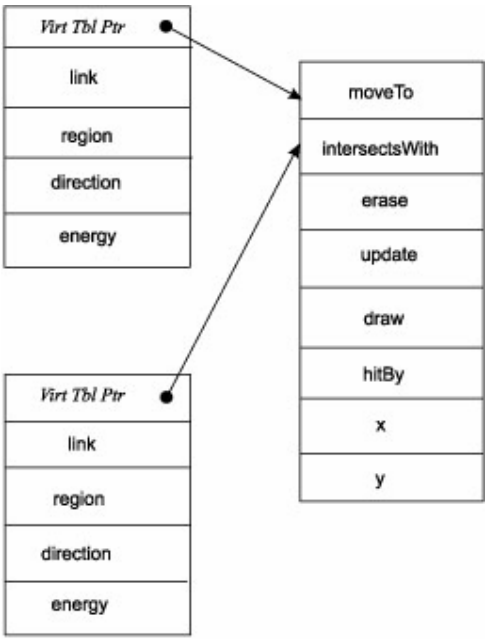



**Рис. 21.3**. Два бильярдных шара с общей таблицей виртуальных методов

Значения полей в таблице виртуальных методов — это указатели на процедуры. Если мы предположим, что эти процедуры известны компилятору и не изменяются во время выполнения программы, то таблица может быть создана статически во время компиляции. Чтобы выполнить метод, нам нужно знать только его смещение в таблице виртуальных методов.

Как и область данных, таблица виртуальных методов класса-предшественника входит во все таблицы классов-потомков, а смещения методов в таблице родителя будут теми же самыми в таблицах потомков. Класс, который наследует методы надкласса, просто копирует общую часть из его таблицы виртуальных методов в свою. Когда в подклассе переопределяется метод, необходимо только изменить запись для этого метода. На рис. 21.4 показана таблица виртуальных методов для классов Wall и Ball, каждый из которых является потомком класса GraphicalObject. Заметьте, что у них общие указатели на методы, унаследованные от родительского класса, и что порядок методов совпадает с тем, который задан в родительском классе.

PDF created with pdfFactory Pro trial version www.pdffactory.com

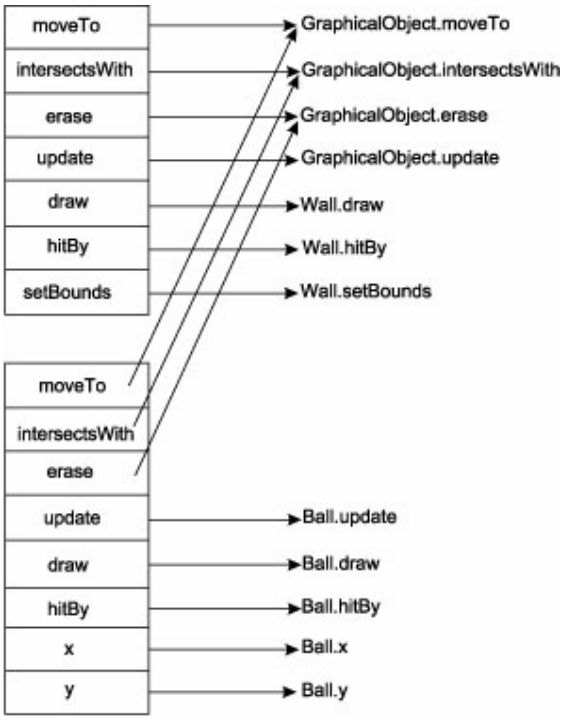



**Рис. 21.4**. Таблицы виртуальных методов для классов Wall и Ball

Раз уж компилятор знает, где найти указатель на метод, то метод может быть вызван как стандартная процедура. Получатель рассматривается как если бы он был первым параметром в списке аргументов, и тем самым он доступен как значение переменной self (this в C++). Предположим, к примеру, что vtab — это внутреннее имя поля, представляющее указатель на таблицу виртуальных методов в объекте x, и что смещение для метода hitBy в таблице равно 12. Тогда вызов метода

```
x.hitBy(y)
```

будет преобразован в следующее внутреннее представление:

```
(*(*(x.vtab))[12]) (x,y)
```

Заметьте, что имя метода не появляется в выходном коде, и надлежащий метод будет выбираться независимо от того, является ли x объектом класса Ball, или класса Wall, или каким-либо другим графическим объектом GraphicalObject. В терминах выполнения программы заголовок посылаемого сообщения требует две операции с указателями и одну операцию нахождения элемента в массиве.

### 21.2.4. Кодирование имен

Поскольку все методы известны во время компиляции и не могут быть изменены во время выполнения, то таблицы виртуальных методов — это просто статические области данных, устанавливаемые компилятором. Они состоят из указателей на соответствующие методы. Поскольку редакторы связей (linkers) и загрузчики (loaders) превращают ссылки в смещения (разрешают ссылки), исходя из символических имен, необходимо обеспечить некоторый механизм, который позволил бы избежать противоречия в ситуации, когда два метода имеют одно и то же имя. Типичная схема комбинирует имя класса и имя метода. Так, метод draw из класса Ball превращается, например, в Ball::draw при внутреннем

PDF created with pdfFactory Pro trial version www.pdffactory.com

представлении. Обычно пользователю никогда не требуется знать это имя, если только нет необходимости анализировать ассемблерный код, созданный компилятором.

В языках программирования, подобных C++, позволяющих еще более перегружать методы за счет снятия неоднозначности путем анализа типов аргументов, требуется более сложное кодирование в стиле Гёделя1 с использованием имени класса, имени метода и типов аргументов. Например, три конструктора класса Complex, описанные в предыдущих главах, получат соответственно внутренние имена Complex::Complex, Complex::Complex_float и Complex::Complex_float_float. Такие внутренние имена называются кусочно-составными (mangled). Они бывают очень длинными. Как мы видели, внутреннее имя не используется для пересылки сообщения; оно применяется только при конструировании таблиц виртуальных методов с целью сделать имена уникальными для редактора связей.

Множественное наследование несколько усложняет использование таблиц виртуальных методов. Детали, однако, выходят за рамки данной книги. Заинтересовавшиеся читатели могут найти более полное описание в [Ellis 1990].

### 21.2.5. Таблицы диспетчеризации

Поскольку языки программирования, подобные C++ и Object Pascal, являются языками со статическими типами данных, они могут определить на этапе компиляции по крайней мере тип родительского класса любого «объектного» выражения. Тем самым таблица виртуальных методов должна быть ровно такой, чтобы вместить методы, реализованные для класса. Для языков программирования с динамическими типами данных (вроде Objective-C) таблица виртуальных методов обязана включать все сообщения, понимаемые всеми классами, и она повторяется для каждого класса. К примеру, если приложение содержит 20 классов и в среднем каждый класс использует 10 методов, то нам требуется 20 таблиц, каждая из которых состоит из 200 записей. Требования к размеру таблиц быстро становятся чрезмерными, и тогда возникает потребность в более удачном подходе.

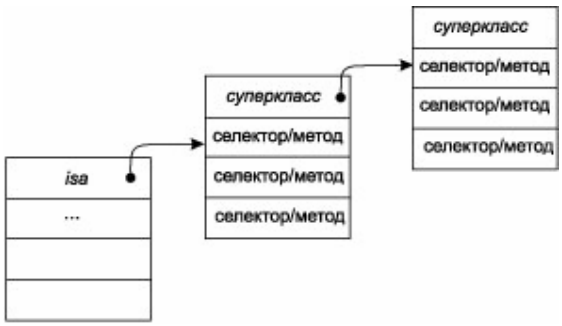



**Рис. 21.5**. Объект и его таблица диспетчеризации

Альтернативный подход состоит в том, чтобы связать с каждым классом таблицу, которая, в отличие от таблицы виртуальных методов, состоит из пар «селектор/метод». Она называется таблицей диспетчеризации. Селекторы соответствуют только тем методам, которые фактически реализованы для данного класса. К наследуемым методам доступ осуществляется через указатель из этой таблицы, который указывает на таблицу диспетчеризации родительского класса (рис. 21.5).

Как и в системе с таблицами виртуальных методов, при использовании таблиц диспетчеризации каждый объект содержит внутри себя неявный (то есть необъявленный)

PDF created with pdfFactory Pro trial version www.pdffactory.com

указатель на таблицу диспетчеризации, связанную с его классом. Этот неявный указатель называется указателем связи isa (isa link) (не смешивать с условием «быть экземпляром» (is-a relation) для классов). Пересылка сообщения в языке Objective-C вроде следующего выражения из задачи о восьми ферзях

```
[neighbor checkrow: row column: column]
```

преобразуется компилятором языка Objective-C1 в код

```
objc_msgSend(neighbor, "checkrow:column:", row, column)
```

Функция objc_msgSend, называемая функцией пересылки сообщений, следует по указателю связи isa для первого аргумента с тем, чтобы найти соответствующую таблицу диспетчеризации. Затем функция пересылки сообщений просматривает таблицу диспетчеризации в поисках записи, соответствующей селектору. Если такая запись найдена, вызывается нужный метод. Если подобного метода не обнаружено, поиск продолжается в таблице диспетчеризации надкласса. Если в конце концов достигнут корневой класс Object, а метод так и не найден, выдается сообщение об ошибке этапа выполнения.

## Кэширование методов

Хотя для языков программирования с динамическими типами данных таблицы диспетчеризации являются более экономичными по объему, чем таблицы виртуальных методов, затраты времени на вызов метода значительно больше. Кроме того, эти затраты пропорциональны глубине наследования. Если бы эти накладные расходы были непреодолимы, то разработчики скорее отказались бы вообще от механизма наследования, согласившись на потерю мощи ради выигрыша в эффективности.

К счастью, мы можем в значительной степени снизить эти потери во время выполнения программы за счет следующего простого подхода. Будем хранить единую для всей системы кэш-таблицу методов, к которым недавно осуществлялся доступ. Она индексируется хэш-кодом [2]

определяемым по селектору метода. Каждая запись в кэш-таблице представляет собой тройку, состоящую из указателя на класс (для этой цели служит собственно таблица диспетчеризации), значения селектора и указателя на метод.

Когда функцию пересылки сообщений просят найти метод, который соответствует паре «класс–селектор», она прежде всего осуществляет поиск в кэш-таблице (рис. 21.6). Если запись в кэше по месту расположения хэш-кода соответствует требуемым селектору и классу, то соответствующий метод может быть выполнен немедленно. Если нет — проводится процесс поиска, описанный выше. В результате поиска непосредственно перед выполнением найденного метода происходит обновление кэш-таблицы. При этом запись, содержавшаяся по месту соответствующего хэш-кода (рассчитываемому по селектору

[2] Хэширование (hashing) — метод приближенного индексирования для сокращения поиска нужных данных. При хэшировании каждому данному ставится в соответствие хэш-код (hash-code), который используется для упорядочивания данных в хэш-таблице и служит селектором при их поиске. Характерная черта хэш-кода состоит в том, что он, вообще говоря, не является уникальным (то есть различные данные могут порождать при вычислениях одинаковый хэш-код). Однако он быстро вычисляется и позволяет значительно сократить область поиска данных. Примером хэширования служит словарь с закладками по месту смены первой буквы слов. Здесь первая буква слова выступает в качестве его хэш-кода. — Примеч. перев.

PDF created with pdfFactory Pro trial version www.pdffactory.com

сообщения), перезаписывается новыми данными. Обратите внимание, что значение класса в кэше соответствует классу, с которого был начат поиск метода, а не классу, в котором метод был в конце концов найден.

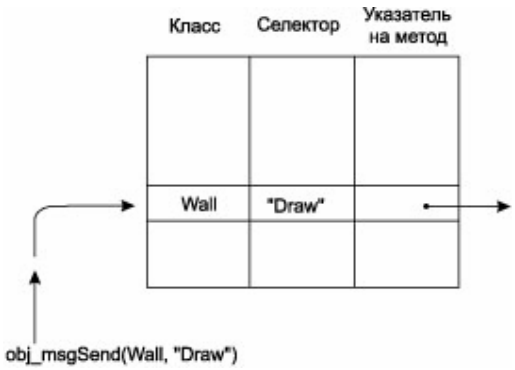



**Рис. 21.6**. Функция пересылки сообщения, проверяющая кэш таблицу

При надлежащем выборе функций вычисления хэш-кода и размера кэша можно достичь частоты попадания в кэш около 90–95 процентов, что уменьшает накладные затраты на передачу сообщения до уровня, лишь примерно в два раза большего, чем расходы на стандартный вызов процедуры [Cox 1986]. Этот показатель вполне благоприятно сравнивается с затратами при использовании таблиц виртуальных методов.

## 21.3. Интерпретаторы

Интерпретаторы обычно предпочтительнее компиляторов, когда изменчивость действий программы превышает некоторый порог, приемлемый для фиксированного ассемблерного кода. Изменчивость действий порождается разнообразными источниками. Например, для языков с динамическими типами данных на этапе компиляции нельзя предсказать, к какому типу данных будет относиться значение переменной (хотя язык Objective-C служит примером языка программирования с динамическими типами данных, который тем не менее, компилируется). Другой источник изменчивости возникает, если пользователю позволено переопределять методы во время выполнения программы.

Подход, обычно используемый для интерпретаторов, — преобразовывать исходную программу в «ассемблерный язык» высокого уровня, часто называемый байт-кодом, поскольку обычно каждая инструкция закодирована в одном байте. На рис. 21.7 показан байт-код системы Little Smalltalk. Старшие четыре бита используются для кодирования операции, а младшие используются для указания номера операнда. Если требуются номера операндов большие шестнадцати, применяется расширенная форма инструкции, и последующий байт целиком содержит значение операнда. Немногие инструкции (типа «послать сообщение» и некоторые специальные) требуют дополнительных байтов.

PDF created with pdfFactory Pro trial version www.pdffactory.com


**Рис. 21.7**. Байт-код в системе Little Smalltalk

Сердцем интерпретатора является цикл, который охватывает огромный оператор переключения switch (case, select и т. д.). Цикл считывает последовательные байт-коды, а оператор выбора передает управление той цепочке кода, которая выполняет требуемое действие. Мы не будем вдаваться в дискуссии о внутреннем представлении программы (заинтересованные читатели могут обратиться к работе [Budd 1987]) и сконцентрируемся исключительно на обработке передачи сообщений.

```
while (timeslice-- > 0)
{
  high = nextByte();   // считать следующий байт-код
  low = high & 0x0F;           // выделить младшие 4 бита
  high >>= 4;          // сдвинуть старшие четыре бита
  if (high == 0)       // это расширенная форма?
  {                    // если так,
    high = low;        // код операции находится в low
        low = nextByte();      // присвоить настоящий операнд
  }
  switch (high)
  {
      case PushInstance: ...
      ...
      case PushArgument: ...
      ...
  }
}
```

Подобно тому как все объекты в рассмотренных ранее компилируемых системах содержали указатель на таблицу виртуальных методов, все объекты в системе Smalltalk поддерживают указатель на свой класс. Разница состоит в том, что, как мы видели в главе 20, класс сам по себе является объектом. Среди полей, содержащихся в объекте-классе, есть набор всех методов, которые соответствуют сообщениям, распознаваемым экземплярами класса (рис. 21.8). Другое поле указывает на надкласс этого класса.

PDF created with pdfFactory Pro trial version www.pdffactory.com

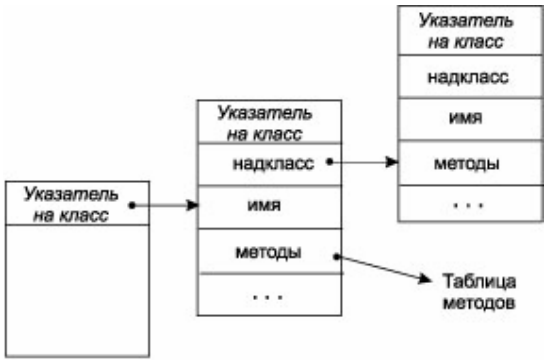



**Рис. 21.8**. Внутренняя структура класса

Когда должно быть послано сообщение, интерпретатор обязан прежде всего определить получателя. Через указатель на класс внутри получателя интерпретатор находит объект, соответствующий классу получателя. Затем интерпретатор производит поиск в наборе методов, стараясь найти тот, который соответствует пересылаемому сообщению. Если такого метода не обнаружено, интерпретатор следует по цепочке наследования, проводя поиск среди методов надклассов, пока либо подходящий метод не будет найден, либо цепочка надклассов не будет исчерпана. В последнем случае интерпретатор сообщает об ошибке. Это в точности та же последовательность действий, которая выполняеся функцией пересылки сообщений при использовании таблиц диспетчеризации. Как и в случае указанной техники, здесь может быть использовано кэширование для ускорения процесса поиска метода.

Язык программирования Java использует подобный интерпретатор байт-кода, хотя реальные коды отличаются от описанных выше.

## Литература для дальнейшего чтения

Для читателя, заинтересованного в получении дополнительной информации по реализации объектно-ориентированных языков, можно порекомендовать работу Кокса [Cox 1986], содержащую детальный анализ затрат времени/памяти для различных схем реализации. Реализация множественного наследования в C++ вкратце описана в работе [Ellis 1990], которая основывается на более раннем алгоритме языка Simula [Krogdahl 1985]. Детальное описание реализаций языка C++ приводится Липманом [Lippman 1996].

Интерпретатор языка Smalltalk-80 описан в работе [Goldberg 1983]. Сборник [Krasner 1983] содержит несколько статей, которые описывают методы улучшения эффективности системы Smalltalk-80. Упрощенный интерпретатор языка Smalltalk детально представлен в работе [Budd 1987]. Камин [Kamin 1990] приводит хороший общий обзор приемов реализации нетрадиционных языков программирования.

## Упражнения

1. Как следует видоизменить метод таблиц диспетчеризации чтобы разрешить множественное наследование?
2. Компилятор языка Objective-C допускает необязательные описания переменных объекта. Объясните, как компилятор может использовать подобные описания для ускорения обработки сообщений, включающих такие значения. Рассмотрите, что происходит при операции присваивания и как пересылка сообщений может быть сделана более эффективной.

PDF created with pdfFactory Pro trial version www.pdffactory.com

3. Объясните, почему методы, которые в языке C++ не описаны как виртуальные, могут вызываться более эффективно, чем виртуальные методы. Как бы вы произвели замеры времени, чтобы определить, является ли разница существенной?
4. Исследуйте технику кэширования, описанную в разделе 21.2. Объясните, почему класс, запоминаемый в кэш-таблице, — это класс, с которого начинается поиск, а не тот класс, внутри которого найден метод. Объясните, как бы изменился алгоритм просмотра кэш-таблицы, если бы использовалось второе значение. Как вы думаете, новый алгоритм будет работать быстрее или медленнее? Обоснуйте свой ответ.
5. Опишите вкратце структуру интерпретатора языка Smalltalk, основанного на байт-кодах, представленных в тексте.

PDF created with pdfFactory Pro trial version www.pdffactory.com