ГЛАВА 1

Принципы программирования и разработки программного обеспечения

В этой главе...

Введение. В этой главе излагаются фундаментальные принципы, лежащие в основе решения больших и сложных задач. В ней излагаются основные принципы программирования, а также показано, что тщательно продуманные и хорошо описанные программы являются экономически эффективными. Глава содержит краткое описание алгоритмов и абстракции данных. Демонстрируется связь этих понятий с главной темой книги, а именно способами решения задач и методами программирования. В последующих главах акцент будет сделан на способах организации и обработки данных. Тем не менее нужно ясно понимать, что при решении любых задач необходимо твердо придерживаться основных принципов, изложенных в данной главе.

Решение задач и разработка программного обеспечения

Кодирование без предварительного проектирования увеличивает время отладки

С чего вы начинали, создавая свою последнюю программу? Многие начинающие программисты, прочитав постановку задачи, сразу же начинают писать код. Очевидно, они стремятся к тому, чтобы их программы работали, причем, по возможности, правильно. С этой целью они запускают свои программы, исследуют сообщения об ошибках, вставляют пропущенные точки с запятыми, изменяют логику, удаляют точки с запятыми, молятся и подвергают свои программы другим издевательствам, пока те не заработают правильно. Большую часть времени такие программисты затрачивают на вылавливание синтаксических ошибок и проверку логики работы программы. Очевидно, сейчас, когда вы уже написали свою первую программу, ваши программистские навыки намного улучшились, однако готовы ли вы создать на самом деле большую программу, используя те способы, которые мы описали только что? Может быть и готовы, однако лучше поступать иначе.

Технологии программирования облегчают разработку программ

Поймите, над разработкой очень больших программных проектов трудятся команды программистов, а не одиночки. Для командной работы нужен подробный план, четкая организация и полное взаимопонимание. Бессистемный подход к программированию здесь совершенно неприемлем и экономически неэффективен. К счастью, применение **технологий программирования** (software engineering) позволяет облегчить разработку компьютерных программ.

В книгах, предназначенных для начинающих программистов, основное внимание обычно уделяется приемам программирования. В нашей книге рассматривается более широкий круг вопросов, связанных с решением задач. Сначала мы рассмотрим весь процесс решения задачи и различные способы достижения результата.

Решение задачи

Термин **решение задачи** (solving problem) охватывает все этапы, начиная с постановки задачи и заканчивая разработкой компьютерной программы для ее решения. Этот процесс состоит из многих этапов — раскрытие смысла задачи, разработка концептуального решения, реализация решения в виде компьютерной программы.

Решение состоит из алгоритмов и способов хранения данных

Что именно называется решением? Обычно **решение** (solution) состоит из двух компонентов: алгоритма и способов хранения данных. **Алгоритм** (algorithm) — это пошаговое описание метода решения задачи за конечный отрезок времени. Алгоритмы часто работают со структурами данных. Например, алгоритм может вносить новые данные в структуру, удалять их оттуда либо просматривать.

Возможно, такое описание решения создает ложное впечатление, что вся сложность заключается в разработке подходящего алгоритма, а способы хранения данных играют вспо-

могательную роль. Это далеко от истины. Для решения задачи нужно не просто хранить данные, но и организовывать их таким образом, чтобы ускорить выполнение алгоритма. Фактически б*о*льшая часть книги посвящена именно способам организации данных в различных структурах.

Для решения задач можно применять методы, описанные в этой главе. Более детально они изложены в последующих главах.

Жизненный цикл программного обеспечения

Разработка хорошего программного обеспечения должна учитывать долгий и продолжительный процесс, называемый **жизненным циклом программного обеспечения** (software's life cycle). Этот процесс начинается с первоначальной идеи, включает в себя написание и отладку программ и продолжается многие годы, в течение которых в исходное программное обеспечение вносятся изменения и улучшения. На рис. 1.1 показаны девять этапов жизненного цикла программного обеспечения в виде сегментов водяного колеса.[1] Это означает, что этапы представляют собой части некоторого умозрительного круга, а не простого линейного списка. Хотя все начинается с постановки задачи, обычно переход от одного этапа к другому не бывает последовательным. Например, тестирование программы может предполагать внесение изменений как в постановку задачи, так и сам проект. Кроме того, обратите внимание, что все девять сегментов расположены вокруг документирования, расположенного в центре круга. Документирование программы не является отдельным этапом ее жизненного цикла, как можно было бы подумать, а сопровождает ее на протяжении всей жизни.

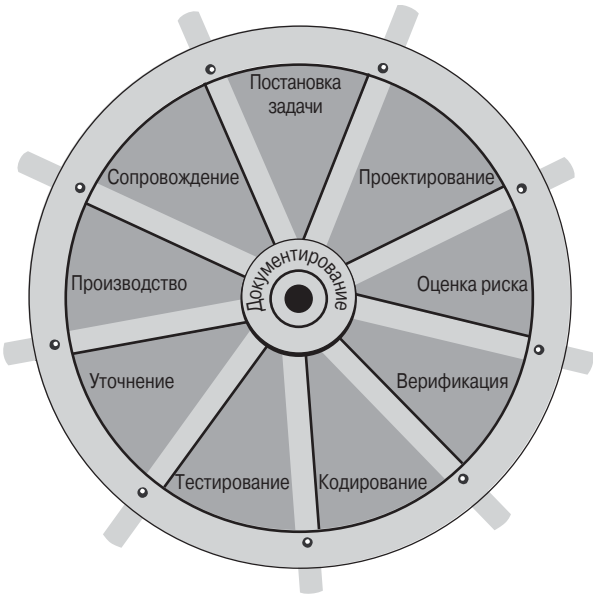

Рис. 1.1. Жизненный цикл программного обеспечения в виде вращающегося водяного колеса

На рисунке изображены этапы жизненного цикла типичного программного обеспечения. Несмотря на то что все они важны, в книге обсуждаются только некоторые из них.

[1] Благодарю Реймонда Падена (Raymond L. Paden) за подсказанную аллегорию.

Этап 1. Постановка задачи. Получив задание, мы должны ясно представлять все его аспекты. Часто люди, формулирующие задачи, не являются программистами, поэтому исходная постановка задачи может быть неточной. Следовательно, на первом этапе в ходе тесного общения программисты и непрограммисты должны совместными усилиями уточнить и детализировать исходную задачу.

Постановка задачи должна быть точной и подробной

Вот вопросы, на которые следует ответить. Каковы входные данные? Какие данные считаются корректными, а какие — нет? Для кого предназначено программное обеспечение? Какой пользовательский интерфейс следует применить? Какие сообщения об ошибках следует предусмотреть? Какие ограничения накладываются на программу? Существуют ли особые ситуации? В каком виде следует представлять выходные данные? Какая документация должна сопровождать программу? Какие усовершенствования программного обеспечения предусмотрены в будущем?

Макетные программы позволяют прояснить постановку задачи

Для полного взаимопонимания между заказчиками и исполнителями можно написать **макетные программы** (prototype programms), имитирующие поведение отдельных частей создаваемого программного обеспечения. Например, простая — пусть даже не эффективная — программа может демонстрировать предполагаемый пользовательский интерфейс. Лучше выявить все подводные камни либо изменить подход к решению задачи на этом этапе, а не в процессе программирования или при эксплуатации программного обеспечения.

Возможно, прежде ваш работодатель сам формулировал спецификации программы за вас. Скорее всего, не все аспекты этого описания были вам понятны, и вы нуждались в разъяснениях, но, вероятнее всего, у вас нет практики создания собственных спецификаций программы.

Этап 2. Разработка. Завершив этап постановки задачи, мы переходим к ее решению. Многие люди, разрабатывающие программы среднего размера и сложности, считают, что с целой программой справиться трудно. Лучше всего упростить процесс решения задачи, разбив большую задачу на несколько маленьких, которыми было бы легче управлять. В результате программа будет состоять из нескольких **модулей** (modules), представляющих собой самостоятельные единицы кода. Модуль может содержать одну или несколько функций, а также другие блоки кода. Следует стремиться к тому, чтобы модули были как можно более независимыми, или **слабо связанными** (loosely coupled) друг с другом. Разумеется, это не относится к их **интерфейсам** (interfaces), представляющим собой механизм их взаимодействия. Умозрительно модули можно считать изолированными друг от друга.

Слабо связанные модули являются независимыми

Узкоспециализированные модули предназначены для решения общей точно определенной задачи

Каждый модуль должен выполнять свою, точно определенную задачу. Следовательно, он должен быть **узкоспециализированным** (highly cohesive). Таким образом, **модульность (modularity)** — это свойство программ, состоящих из слабо связанных и узко специализированных модулей.

Указывайте предназначение каждого модуля, условия его применения, а также входные и выходные данные

На этапе проектирования важно точно указывать не только предназначение каждого модуля, но и **поток данных** (data flow) между модулями. Например, разрабатывая модуль, нужно ответить на следующие вопросы. Какие данные доступны данному модулю во время его выполнения? В каких условиях можно выполнять данный модуль? Какие действия выполняет модуль и как изменяются данные после завершения его работы? Таким образом, нужно детально сформулировать предположения, а также входные и выходные данные для каждого модуля.

Например, если при разработке программы потребовалось упорядочить массив целых чисел, можно написать следующую спецификацию функции сортировки.

- Функция получает на вход *num* целых чисел, где *num* > 0.
- Функция возвращает упорядоченный массив, состоящий из целых чисел.

Спецификации — это контракт

Эту спецификацию можно рассматривать как **контракт** (contract) между вашей функцией и вызывающим ее модулем.

Если вы разрабатываете программу самостоятельно, этот контракт поможет систематически разбить исходную задачу на более мелкие части. Если над проектом работает команда программистов, контракт поможет разделить ответственность между ними. Программист, разрабатывающий функцию сортировки, должен выполнять контракт. Контракт законченной функции сортировки сообщает остальным программистам, как ее вызывать и какие результаты она должна возвращать.

Спецификация модуля не должна описывать метод решения задачи

Однако следует особо подчеркнуть, что контракт модуля не связывает его с конкретным методом решения задачи. Делать в другой части программы какие-либо предположения, касающиеся этого метода, не следует. Тогда, например, если в дальнейшем вы перепишете свою функцию и примените другой алгоритм сортировки, вносить изменения в остальной код не потребуется вообще. Если новая функция выполняет условия старого контракта, о других модулях можно не заботиться.

Спецификации функции состоят из точных пред- и постусловий

Все вышеизложенное не должно быть для вас новостью. Хотя до сих пор вы могли не использовать в своей речи слово “контракт”, его концепция должна быть вам ясна. Формулируя **предусловие** (precondition) и **постусловие** (postcondition) функции, вы пишете ее контракт, состоящий из условий, которые должны выполняться перед ее вызовом и после завершения ее работы, соответственно. Например, псевдокод функции сортировки, придерживающейся приведенного выше контракта, выглядит следующим образом.[2]

Черновой набросок спецификаций

```
sort(anArray, num)
// Сортировка массива.
// Предусловие: переменная anArray является массивом,
// состоящим из num целых чисел; num > 0.
// Постусловие: целые числа в массиве anArray упорядочены.
```

На самом деле в данном случае этих пред- и постусловий недостаточно. Например, в каком порядке упорядочен массив: возрастающем или убывающем? Насколько большим может быть число *num*? Реализуя эту функцию, вы могли предполагать, что массив упорядочивается в возрастающем порядке, а число *num* не должно превышать 100. Представьте себе трудности, с которыми столкнется человек, который попытается применить функцию `sort` для сортировки 500 чисел в убывающем порядке. Этот пользователь ничего не знает о ваших предположениях, пока вы ясно не укажете их в пред- и постусловиях.

Пересмотренная спецификация

```
sort(anArray, num)
// Сортировка массива в возрастающем
// порядке.
// Предусловие: переменная anArray является массивом,
// состоящим из num целых чисел; 1 <= num <= MAX_ARRAY,
// где MAX_ARRAY — это глобальная константа, задающая
// максимальный размер массива anArray.
```

[2] Псевдокоды в книге набраны курсивом.

```
// Постусловие: anArray[0] <= anArray[1] <= ... <=
// anArray[num-1]; число num не изменяется.
```

В предусловии описываются входные аргументы функции, указываются все глобальные именованные константы, использующиеся в ней, и перечисляются все ограничения, которые накладываются функцией. Аналогично, в постусловии описываются результаты работы функции — либо возвращаемое функцией значение — и все последствия ее работы.

Документация должна быть точной

Новички стремятся приуменьшить значение точной документации, особенно когда они одновременно являются и разработчиками, и программистами, и пользователями программы. Если вы разработали функцию `sort`, но не указали условия ее контракта, вспомните ли вы о них при ее реализации? А через неделю? Что лучше освежает память —код на языке C++ или пред- и постусловия, сформулированные простым языком? При увеличении размера программы важность хорошей документации возрастает, независимо от того, в одиночку вы пишете программу или в команде.

Использование компонентов существующего программного обеспечения в собственном проекте

Не следует пренебрегать возможностью применения готовых модулей, решающих вашу задачу. Возможности повторного использования кода, предоставляемые языком C++, обычно реализуются в виде компилируемых библиотек. Это означает, что вы не всегда будете иметь доступ к исходному коду функции. Библиотеки представляют собой яркий пример коллекции готовых компонентов программного обеспечения. Например, вы знаете, как использовать стандартную функцию `sqrt`, содержащуюся в математической библиотеке языка C++ (*math.h*), однако не можете увидеть ее исходный текст. Если же функции `sqrt` передать число с плавающей точкой или соответствующее выражение, она извлечет из него квадратный корень и вернет его в вызывающий модуль. Функцию `sort` можно применять, ничего не зная о деталях ее реализации. Более того, она вообще может быть написана на другом языке! Функцию `sqrt` можно применять вслепую, *поскольку нам известна ее спецификация*.

Итак, если в прошлом вы не задерживались на этапе разработки программы, вам следует немедленно отказаться от этой привычки! Результатом этого этапа должно быть модульное решение, которое легко выразить с помощью конструкций конкретного языка программирования. Уделив должное внимание этому вопросу, вы сэкономите время, необходимое для написания и отладки вашей программы.

Позднее мы еще вернемся к обсуждению модульной структуры программ.

Некоторые, но не все, проблемы можно предсказывать и предотвращать

Этап 3. Оценка риска. Создание программного обеспечения сопряжено с риском. Некоторые проблемы присущи всем проектам, а некоторые характерны лишь для определенных разработок. Кое-какие из них можно предвидеть, в то время как другие остаются в тени. Они могут влиять на график и стоимость выполнения работ, экономические успехи и даже на жизнь и здоровье людей. Некоторые опасности можно предотвратить или смягчить, а некоторые — нет. Для идентификации, оценки и предотвращения опасностей, возникающих при разработке программного обеспечения, существуют специальные методы. Вы познакомитесь с ними при освоении более сложного курса программирования. Результат оценки риска влияет на все этапы жизненного цикла программного обеспечения.

Этап 4. Верификация. Для проверки правильности алгоритмов существуют формальные методы. Хотя полностью эта задача еще не решена, стоит напомнить о некоторых аспектах процесса верификации программ.

Диагностическое утверждение (assertion) — это формальное высказывание, описывающее конкретные условия, которые должны выполняться в определенной точке программы. Пред- и постусловия представляют собой пример простых утверждений об условиях, кото-

рые должны выполняться в начале и в конце функции. **Инвариант** (invariant) — это условие, которое всегда должно быть истинным в конкретной точке алгоритма. **Инвариант цикла** (loop invariant) — это условие, которое должно выполняться до и после каждого выполнения цикла, являющегося частью алгоритма. Как мы убедимся в дальнейшем, инварианты цикла оказываются полезными для создания правильных циклов. Используя инварианты, легче обнаруживать ошибки, следовательно, сокращается время отладки и тестирования программы. Короче говоря, инварианты позволяют сэкономить время.

Правильность некоторых алгоритмов можно доказать

Доказательство правильности алгоритма напоминает доказательство теоремы в геометрии. Например, чтобы доказать, что функция работает правильно, нужно начать с проверки ее предусловия, аналогичного аксиомам и предположениям в геометрии, и продемонстрировать, что шаги алгоритма в итоге приводят к выполнению постусловия. Для этого нужно проверить каждый шаг алгоритма и показать, что из диагностического утверждения, относящегося к моменту времени, предшествующему выполнению конкретного шага, следует диагностическое утверждение, относящееся к моменту времени после выполнения этого шага.

Доказав корректность отдельных операторов, можно доказать правильность последовательности операторов, затем функций, и в итоге — всей программы. Допустим, мы доказали, что если диагностическое утверждение A_1 истинно и выполняется оператор S_1, то утверждение A_2 также истинно. Кроме того, предположим, что утверждение A_2 и оператор S_2 приводят к выполнению утверждения A_3. Отсюда следует, что если утверждение A_1 истинно, то выполнение операторов S_1 и S_2 приводит к истинности утверждения A_3. Продолжая в том же духе, в конце концов можно доказать правильность программы в целом.

Очевидно, что если в процессе верификации программы обнаружилась ошибка, алгоритм можно исправить, а постановку задачи немного изменить. Таким образом, используя инварианты, можно доказать, что ошибка содержалась не в коде, а *в самом алгоритме*. В результате время, затраченное на отладку программы, существенно сократится.

С помощью формальных методов можно доказать правильность разных конструкций, в частности операторов *if*, циклов и операторов присваивания. Для проверки правильности итерационных алгоритмов широко используются инварианты циклов. Например, мы докажем, что приведенный ниже цикл вычисляет сумму первых *n* элементов массива *item*.

```
// Вычисляет сумму элементов item[0], item[1], ...,
// item[n-1] для любого n>=1.
int sum = 0;
int j = 0;
while (j < n)
{
  sum += item[j];
  ++;
} // конец оператора while
```

Перед началом этого цикла значения переменной *sum* и *j* равны 0. После первого выполнения цикла значение переменных *sum* равно *item[0]*, а значение переменной *j* равно 1. Итак, можно сформулировать инвариант данного цикла.

Инвариант цикла

Значение переменной *sum* равно сумме элементов от *item[0]* до *item[j-1]*.

Инвариант правильного цикла должен выполняться в следующих точках.

- После каждого шага инициализации переменных, но до начала выполнения цикла.
- Перед каждым повторением цикла.

- После каждого повторения цикла.
- После завершения цикла.

В предыдущем примере перечисленные точки находятся в следующих местах программы.

```
// Вычисляет сумму элементов item[0], item[1], ...,
// item[n-1] для любого n>=1.
                  <- здесь должен выполняться инвариант
int sum = 0;
int j = 0;
while (j < n)
{                 <- здесь должен выполняться инвариант
sum += item[j];
  ++;
                  <- здесь должен выполняться инвариант
} // конец оператора while
                  <- здесь должен выполняться инвариант
```

Эти рассуждения можно применять при доказательстве правильности итерационного алгоритма. В нашем примере нужно доказать, что инвариант выполняется в каждой из следующих четырех точек.

Шаги, которые следует выполнить для доказательства правильности алгоритма

1. **Инвариант должен быть истинным изначально**, до начала первой итерации. В предыдущем примере инвариант утверждает, что значение переменной *sum* равно сумме элементов от *item[0]* до *item[-1]*. Это утверждение истинно, поскольку в этом диапазоне индексов элементов нет.
2. **Выполнение цикла должно сохранять инвариант**. Это означает, что если перед каждой итерацией цикла инвариант является истинным, нужно показать, что он остается истинным и после ее выполнения. В нашем примере цикл добавляет элемент *item[j]* к переменной *sum*, а затем увеличивает значение переменной *j* на единицу. Таким образом, после выполнения цикла к переменной *sum* добавляется последний элемент, т.е. *item[j-1]*. Таким образом, после выполнения цикла инвариант остается истинным.
3. **Из выполнения инварианта должна следовать правильность алгоритма.** Нужно показать, что если после завершения цикла инвариант остается истинным, то алгоритм является корректным. В предыдущем примере по завершении цикла переменная *j* содержит значение *n*, следовательно, инвариант цикла остается истинным: переменная *sum* содержит сумму элементов от *item[0]* до *item[n-1]*, что и требовалось доказать.
4. **Цикл должен завершиться.** Нужно доказать, что цикл завершится после выполнения конечного количества итераций. В нашем примере переменная *j* сначала равна 0, а затем при каждой итерации увеличивается на 1. Таким образом, в конце концов переменная *j* станет равной числу *n* при любом *n*>=1. Этот факт и оператор *while* гарантирует, что цикл в конце концов завершится.

Инварианты можно применять не только для доказательства правильности цикла, но и для доказательства его неправильности. Например, допустим, что в предыдущем примере в операторе *while* вместо условия *j<=n* поставлено условие *j<n*. Шаги 1 и 2 в доказательстве правильности программы остаются без изменения, а вот шаг 3 изменится: по завершении цикла переменная *j* будет содержать число *n+1*, и, поскольку инвариант цикла должен быть истинным, переменная *sum* станет содержать сумму элементов от *item[0]* до *item[n]*. По-

скольку при этом мы получаем неверное решение задачи, цикл следует признать неправильным.

Обратите внимание на очевидную связь между описанным выше процессом доказательства и **математической индукцией** (mathematical induction).[3] Доказательство истинности инварианта в начальный момент называется **базисом индукции** (base case). Оно аналогично доказательству, что некоторое свойство выполняется для натурального числа 0. Доказательство истинности инварианта на каждой итерации цикла называется **шагом индукции** (induction step). Он аналогичен доказательству утверждения, что если некоторое свойство выполняется для произвольного натурального числа *k*, то оно выполняется и для числа *k+1*. После выполнения четырех шагов, перечисленных выше, мы приходим к выводу, что инвариант является истинным после каждой итерации цикла, точно так же, как, следуя принципу математической индукции, можно доказать, что некоторое свойство выполняется для любого натурального числа.

Идентификация вариантов цикла позволяет конструировать правильные циклы. Инвариант нужно формулировать в виде комментария либо перед циклом, либо в его начале. Например, в предыдущем фрагменте программы следует поместить такой комментарий.

Формулируйте инварианты цикла в своих программах

```
// Инвариант: 0 <= j <= n и
// sum = item[0] + ... + item[j-1]
while (j < n)
...
```

В приведенном ниже примере нужно подтвердить, что инварианты двух не связанных друг с другом циклов являются корректными. Напомним, что каждый инвариант должен быть истинным как до, так и после каждой итерации цикла, включая последнюю итерацию. Кроме того, инвариант цикла *for* легче понять, если этот цикл временно преобразовать в эквивалентный цикл *while*.

Пример инвариантов цикла

```
// Вычисляет n! для целого числа n>=0
int f = 1;
// Инвариант: f == (j-1)!
for (int j = 1; j <= n; ++j)
   f *= j;

// Вычисляет приближенное значение функции e^x
// для действительного числа x
double t = 1.0;
double s = 1.0;
int k = 1;
// Инвариант: t == x^(k-1)/(k-1)! и
// s == 1+x+x^2/2!+...+x^(k-1)/(k-1)!
while (k <= n)
{ t *= x/k;
  s += t;
  ++k;
} // конец цикла while
```

Этап 5. Кодирование. Кодирование заключается в переводе алгоритма на конкретный язык программирования с последующим исправлением синтаксических ошибок. Вполне вероятно, именно кодирование

Кодирование — это относительно небольшая часть жизненного цикла программного обеспечения

[3] Принцип математической индукции изложен в Приложении Г.

многие считают собственно программированием. И все же следует понимать, что кодирование — это не самое главное, это лишь один из этапов жизненного цикла программного обеспечения

Этап 6. Тестирование. На этапе тестирования нужно выявить и исправить как можно больше логических ошибок. Для этого можно прибегнуть к проверке отдельных функций, применяя их к выбранным данным и сравнивая с заранее известным результатом. Если входные данные изменяются в каком-то диапазоне, обязательно проверьте их крайние значения. Например, если входное значение n может изменяться от 1 до 10, обязательно протестируйте программу при значениях 1 и 10. Кроме того, проверьте, как работает программа, если в нее ввести заведомо неверные данные, и может ли она обнаруживать такие ошибки. Попробуйте ввести в программу случайно выбранные данные, а затем примените ее для реального набора данных. Тестирование — это и наука, и искусство одновременно.

> Разработайте набор тестовых данных для проверки вашей программы

Этап 7. Уточнение решения. Результатом выполнения этапов 1–6 является работающая программа, которую интенсивно тестировали и отлаживали. Если программа действительно решает поставленную задачу, возникает вопрос: зачем уточнять решение?

Лучше всего решать задачу при наиболее простых предположениях, постепенно усложняя программу. Например, можно предположить, что входные данные имеют определенный формат и являются правильными. Создав простейший вариант, можно дополнять его более сложными процедурами ввода и вывода данных, оснащать дополнительными возможностями и средствами для обнаружения ошибок.

> Разрабатывайте программу при упрощающих предположениях, постепенно усложняя ее

Таким образом, если вы применяете подход “от простого — к сложному”, этап уточнения решения становится необходимым. Разумеется, окончательное уточнение решения не должно приводить к полному пересмотру программы. Каждое уточнение решения является довольно очевидным, особенно если программа имеет модульную структуру. Фактически постепенное уточнение решения представляет собой основное преимущество модульного подхода к разработке программ! Кроме того, после каждой, даже простейшей, модификации программы, ее нужно снова тщательно протестировать.

> Измененную программу следует протестировать снова

Как видим, этапы жизненного цикла программного обеспечения не изолированы друг от друга и не следуют один за другим. Сделав реалистичные упрощающие предположения в самом начале процесса разработки программы, вы должны точно предвидеть, как учесть их в дальнейшем. Тестирование программы может вынудить внести в программу изменения, однако модифицированную программу придется снова тестировать.

Этап 8. Производство. После завершения разработки программного продукта он распространяется среди пользователей, инсталлируется на их компьютерах и применяется.

Этап 9. Сопровождение. Поддержка программы не имеет ничего общего с обслуживанием автомобиля. Программное обеспечение не износится, если за ним не ухаживать. Однако пользователи ваших программ могут обнаружить ошибки, оставшиеся незамеченными при тестировании. Кроме того, со временем программное обеспечение нужно совершенствовать, добавляя в него новые функциональные возможности или модифицируя его компоненты. Авторы программ занимаются этим довольно редко, тем важнее становится наличие хорошей документации.

> Сопровождение программного обеспечения заключается в исправлении ошибок, обнаруженных пользователем, и его усовершенствовании

Необходимо ли точно следовать описанным выше этапам в реальной работе? Конечно да! Этапы 1–7 — это компоненты процесса решения задачи. Используя эту стратегию, сначала нужно разработать и реализовать решение (этапы 1–6), основываясь на некоторых первоначальных упрощающих предположениях. В результате вы получите хорошо организованную программу, решающую несколько упрощенную задачу. На последнем этапе эта программа усложняется и должна полностью соответствовать исходной постановке задачи.

Хорошее решение задачи

Перед тем как приступить к изучению методов решения задач, следует вначале убедиться, что овладение этими приемами действительно приводит к хорошим результатам. Очевидно, что применение этих методов позволяет получить хорошее решение задачи. Тогда возникает более существенный вопрос: а что считается хорошим решением? Попробуем на него ответить.

Поскольку окончательное решение задачи выражается в виде компьютерной программы, рассмотрим, какими свойствами обладает хорошая компьютерная программа. По-видимому, программа создается для решения конкретной задачи. Решение этой задачи имеет реальную и вполне ощутимую **стоимость** (cost). В нее входят ресурсы компьютера (время вычислений и память), потребленные программой, неудобства, с которыми сталкиваются пользователи программы, и последствия, к которым приводит ее неправильная работа.

Однако это еще не все. Эти факторы относятся лишь к одному из этапов жизненного цикла программы — этапу ее поддержки. Стремясь ответить на вопрос, насколько хорошее решение получено вами, нужно рассмотреть все этапы разработки программы. Каждый из этих этапов также имеет свои затраты. Общая стоимость решения должна учитывать объем рабочего времени, затраченного программистами, которые его разрабатывали, уточняли, кодировали, отлаживали и тестировали. Кроме того, необходимо учесть стоимость поддержки, модификации и усовершенствования программы.

Таким образом, вычисляя общую стоимость решения, нужно принимать во внимание разнообразные факторы. Встав на такую многомерную точку зрения, можно сформулировать следующий критерий.

Многомерная точка зрения на стоимость решения

- Решение считается хорошим, если его общая стоимость минимальна.

Интересно проследить, как изменялась относительная важность разных компонентов в ходе эволюции программирования. Вначале доля стоимости работы компьютера по сравнению со стоимостью работы программистов была чрезвычайно высока. Кроме того, программы разрабатывались для решения очень специфичных, узко поставленных задач. Если постановка задачи изменялась, создавалась новая программа. Поддержка программ во внимание не принималась, их читабельность не имела никакого значения. Программу обычно использовал только один человек, ее автор. Как следствие, программистов не интересовало, удобно ли работать с программой. Интерфейс программы не считался важным фактором.

В такой среде программирования все перевешивала стоимость компьютерных ресурсов. Если две программы решали одну и ту же задачу, лучшей считалась та, которая работала быстрее и занимала меньший объем памяти. Как все изменилось с тех пор! Сейчас стоимость компьютерного времени резко снизилась, и время, затраченное разработчиками и программистами, стало более значительным фактором, влияющим на общую стоимость решения задачи. Другим следствием падения стоимости вычислений стало широкое использование компьютеров в разных сферах деятельности человека, многие из которых не связаны с наукой. Люди, работающие на компьютерах, часто не имеют специального опыта и знаний, не-

обходимых для работы с программами. Следовательно, программы должны быть легкими в эксплуатации.

Программы должны быть хорошо организованными и сопровождаться подробной документацией

В настоящее время программы становятся все более сложными и большими. Часто они настолько велики, что в их разработку и эксплуатацию вовлекается много людей. Хорошая структура и документация в этих условиях приобретают чрезвычайно важное значение. Чем более важную задачу решают программы, тем серьезнее последствия их неправильной работы. Таким образом, людям нужны хорошо организованные программы и способы их формальной верификации. Люди не хотят рисковать, используя программы, с которыми могут работать лишь их авторы.

Как видим, развитие технологии привело к тому, что в настоящее время самое эффективное решение не всегда является наилучшим. Если две программы решают одну и ту же задачу, то лучшей из них не обязательно является та, которая быстрее работает. Программисты, стремящиеся использовать любую возможность, чтобы сэкономить несколько миллисекунд вычислений, отстали от жизни. В настоящее время, создавая программы, нужно ориентироваться не только на компьютеры, но и на людей, которые будут их использовать.

Эффективность — лишь один из многих аспектов, влияющих на стоимость решения

В то же время, не следует считать, что эффективность решения больше не имеет значения. Во многих ситуациях она очень важна. Просто нужно иметь в виду, что эффективность решения — это всего лишь один из многих факторов, которые следует учитывать. Если два решения обладают примерно одинаковой эффективностью, на сцене появляются другие аспекты, влияющие на выбор. Однако, если решения *значительно* отличаются по эффективности, этот факт может перекрыть остальные соображения. Выбирая или разрабатывая методы решения задачи, следует иметь это в виду. Выбор компонентов решения — алгоритмов и способов хранения данных — влияет на эффективность решения больше, чем непосредственное кодирование.

В книге последовательно отстаивается многомерная точка зрения на стоимость решения. В сегодняшних условиях эта точка зрения вполне разумна, и нам кажется, что в ближайшие годы это положение вещей не изменится.

Модульный подход

Мы уже убедились, насколько важно сопровождать каждый модуль точным описанием пред- и постусловий, но как разбить программу на эти модули? Решению именно этой проблемы и посвящена вся книга. В этом разделе мы рассмотрим два важных способа проектирования. Оба эти способа используют абстракцию, поэтому начнем с определения этого понятия.

Абстракция и сокрытие информации

Каждый модуль, из которого состоит решение задачи, начинается строками, в которых указано, для чего он предназначен, но не написано, как именно он работает. Ни один модуль не может “знать”, как работает другой модуль, — в лучшем случае, он может знать, лишь для решения какой задачи предназначены другие модули.

Указывайте, что делает модуль, но не описывайте, как он это делает

Например, если в какой-то части программы данные должны упорядочиваться, то в одном из модулей выполняется алгоритм сортировки (рис. 1.2). Другие модули знают о том, что здесь выполняется сортировка данных, но не знают, как именно она осуществляется. Таким образом, разные части решения изолируются друг от друга.

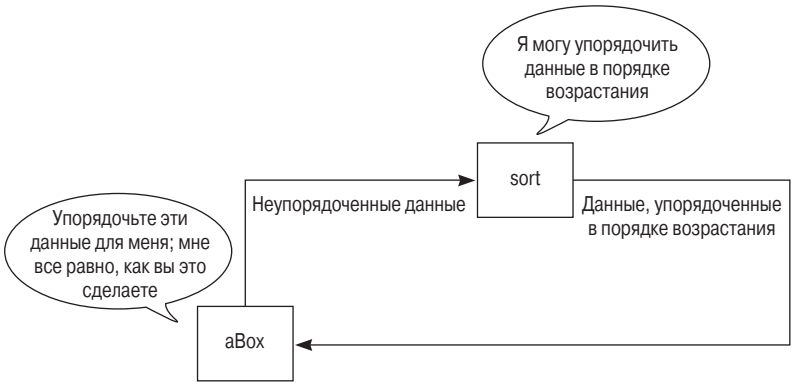

Рис. 1.2. Детали алгоритма сортировки скрыты от других частей программы

Спецификация каждого модуля создается до его реализации

Абстракция (abstraction) отделяет предназначение модуля от его реализации. Модульность и абстракция дополняют друг друга. Модульный подход позволяет разделить решение задачи на блоки; абстракция определяет содержание модуля *до* его реализации на конкретном языке программирования.

В спецификациях не указывается, как именно реализован модуль

Например, в спецификации модуля указывается, какие условия должны выполняться и что именно в нем происходит. Такие спецификации облегчают решение задачи, позволяя сосредоточиться только на функциональных возможностях высокого уровня, не вникая в детали их реализации. Кроме того, эти принципы позволяют модифицировать части решения независимо друг от друга. Например, можно ли изменить алгоритм сортировки, приведенный выше, не затрагивая остальную часть решения?

Указывайте, что делает функция, но не описывайте, как она это делает

В ходе решения задачи содержание каждого модуля постепенно уточняется, воплощаясь в итоге в виде функций на языке C++. Предназначение функции следует отделять от ее реализации. Этот процесс называется **функциональной** (или **процедурной**) **абстракцией** (functional, or procedural abstraction). Готовую функцию можно применять, не вникая в детали реализации алгоритма, поскольку для использования достаточно знать ее предназначение и описание аргументов. Если функция сопровождается соответствующей документацией, ее можно использовать, зная лишь объявление и первичное описание, реализацию можно не изучать.

Функциональная абстракция играет важную роль в командных проектах. В таких ситуациях участники проектов должны применять функции, разработанные другими программистами, не вникая в детали их алгоритмов. Неужели можно применять функцию, не зная ее кода? Но ведь именно так вы и поступаете, вызывая функцию `sqrt` из математической библиотеки языка C++.

Указывайте, что именно вы хотите сделать с данными, но не описывайте, как это нужно сделать

Рассмотрим теперь совокупность данных и набор операций над ними. В этом наборе могут быть операции добавления данных в совокупность, удаления их оттуда или операции поиска. **Абстракция данных** (data abstraction) сосредоточивает внимание на предназначении операций, а не на деталях их выполнения. Другие модули программы будут "знать", *что именно* делает та или иная операция, но не смогут узнать, *как* при этом хранятся данные или *как* именно выполняется данная операция.

В предыдущих примерах мы использовали массив. А что, собственно, он собой представляет? В книге приведено много иллюстраций, посвященных массивам. Они не могут точно соответствовать их машинной реализации, а могут лишь отдаленно напоминать о ней. Дело в том, что нам не важно, что именно представляет собой массив, т.е. как он реализован. Мы и без этого можем его использовать. Несмотря на то что разные операционные системы реализуют массивы по-разному, программисту это безразлично. Например, независимо от реализации массива *years*, число 1492 всегда можно записать в ячейку массива с номером *index*, используя следующий оператор.

```
years[index] = 1492;
```

Позднее, это значение можно вывести на экран, воспользовавшись оператором

```
cout << years[index] << endl;
```

Таким образом, мы вполне способны использовать массив, ничего не зная о способе его реализации, точно так же, как функцию *sqrt* мы можем вызывать, не зная, как она извлекает квадратный корень из своего аргумента.

Бо*л*ьшая часть книги посвящена абстракции данных. Чтобы научить вас думать о данных абстрактно — т.е. фокусировать внимание на операциях с данными, а не на деталях их реализации, — нужно дать определение **абстрактного типа данных**, или **АТД** (abstract data type). АТД — это совокупность данных *и* множество операторов над ними. Операции АТД можно применять, если известны их спецификации, при этом не обязательно знать детали их реализации или способы хранения данных.

АТД — это не синоним структуры данных

Для реализации АТД можно использовать **структуру данных** (data structure), представляющую собой конструкцию, определенную в языке программирования для хранения совокупности данных. Например, данные можно хранить в массивах целых чисел, объектов или массивах массивов.

Разработка алгоритма и АТД должны быть связаны друг с другом

В процессе решения задачи абстрактные типы данных помогают реализовывать алгоритм, а алгоритмы диктуют выбор абстрактного типа данных. Разработка алгоритма и АТД должны быть связаны друг с другом. Глобальный алгоритм, предназначенный для решения задачи, предполагает выполнение последовательности операций над данными, что, в свою очередь, приводит к определению АТД и алгоритмов, выполняющих эти операции. Однако процедуру решения задачи можно выполнять и в обратном порядке. Вид применяемого АТД может диктовать выбор стратегии глобального алгоритма решения задачи. Таким образом, зная, какие операции над данными выполнять легко, а какие — трудно, можно существенно повысить эффективность решения задачи.

Возможно, вы уже догадались, что обычно трудно четко отделить проблемы, связанные с алгоритмами, от проблем, связанных со структурами данных. Часто невозможно понять, благодаря чему достигается эффективность программы: остроумному алгоритму или удачному выбору структуры данных.

Все модули и АТД иногда нужно скрывать

Сокрытие информации. Как видим, абстракция вынуждает создавать функциональные спецификации для каждого модуля, делая его **открытым** (public) для внешнего мира. Однако она позволяет идентифицировать детали, которые должны быть скрыты от публичного обозрения, — т.е. быть **закрытыми** (private). Принцип **сокрытия информации** (information hiding) гарантирует, что такие детали будут не только скрыты внутри модуля, но и ни один другой модуль не будет даже подозревать об их существовании.

Принцип сокрытия информации ограничивает способы работы с функциями и данными. Пользователь модуля не должен интересоваться деталями его реализации. Разработчик модуля не должен заботиться о способах его использования.

Объектно-ориентированное проектирование

Объекты инкапсулируют данные и операции

Один из способов модульного решения задачи — идентификация **объектов** (objects), объединяющих в единое целое данные и операции над ними. В результате такого **объектно-ориентированного подхода** (object-oriented approach) к модульному решению задачи возникает совокупность объектов, обладающих определенным поведением.

Инкапсуляция скрывает внутренние детали

Не зная этого, вы уже встречались с объектами. Будильник, разбудивший вас сегодня утром, **инкапсулирует** (encapsulates) время и операции, например "звонок". Инкапсулировать — значит "упаковывать" или "вкладывать". Таким образом, инкапсуляци — это способ сокрытия внутренних деталей. Функции инкапсулируют действия, объекты инкапсулируют данные вместе с действиями. Когда вы хотите, чтобы будильник зазвенел, вы не знаете, как он это сделает. Вы увидите лишь результат этой операции.

Допустим, мы хотим написать программу, выводящую на экран циферблат часов. Для простоты предположим, что это электронные часы без будильника, как показано на рис. 1.3. Начать решение задачи можно с идентификации объектов.

Для идентификации объектов существуют несколько способов, но все они не идеальны. Один из простых способов[4] основан на распознавании имен существительных и глаголов, входящих в описание задачи. Имена существительные можно считать объектами, действия которых обозначаются глаголами. Тогда поставленную выше задачу можно описать следующим образом.

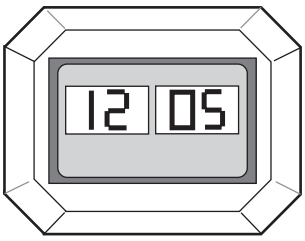

Рис. 1.3. Электронные часы

Спецификации программы для вывода на экран циферблата электронных часов

Программа имитирует работу электронных часов, показывающих часы и минуты. Цифровые индикаторы часов и минут позволяют отображать числа от 1 до 12 и от 0 до 60 соответственно. Время задается с помощью установок индикаторов часов и минут, причем программа должна постоянно обновлять их показания.

Даже не имея детального описания задачи, можно идентифицировать по крайней мере один объект — сами часы. Эти часы выполняют следующие операции.

- *Установка времени*
- *Изменение времени*

[4] Этот метод не слишком надежен. Спефикация задачи может состоять как из имен существительных, так и глаголов. Так, например, слово "звонок" иногда может означать как существительное, так и глагол. В этом случае идентифицировать объекты и операции будет непросто.

- *Вывод показаний на экран*

Индикаторы часов и минут также являются объектами, причем они очень похожи. Каждый из них выполняет следующие операции.

- *Установка значения*
- *Изменение значения*
- *Вывод значения на экран*

Объект — это экземпляр класса

Фактически оба индикатора представляют собой один и тот же тип объекта. Множество объектов, имеющих один и тот же тип, называется **классом** (class). Таким образом, нам нужно указать не конкретный объект, а класс объектов: класс часов и класс индикаторов. Объект, обозначающий часы, представляет собой **экземпляр** (instance) класса часов. Он состоит из двух объектов, представляющих собой экземпляры класса индикаторов.

Классы определяют данные и операции над объектами. Отдельные элементы данных, определенных в классе, называются **данными-членами** (data members), **полями данных** (data fields) или **атрибутами** (attributes). Операции, заданные в классе, называются **методами** (methods) или **функциями-членами** (member functions).

Инкапсуляция будет рассмотрена в главе 3. В частности, там будут определены классы языка C++. В последующих главах мы изучим различные абстрактные типы данных и их реализации в виде классов. Основное внимание будет уделено абстракции данных и инкапсуляции. Такой подход к программированию называется **объектным** (object based).

Объектно-ориентированное программирование (object-oriented programming), или **ООП**, дополняет инкапсуляцию двумя новыми принципами.

ОСНОВНЫЕ ПОНЯТИЯ

Три принципа объектно-ориентированного программирования

1. Инкапсуляция: объекты объединяют данные и операции.
2. Наследование: классы могут наследовать свойства других классов.
3. Полиморфизм: объекты могут выбирать подходящие операции во время выполнения программы.

Классы могут **наследовать** (inherit) свойства других классов. Например, определив класс часов, мы можем разработать класс будильников, наследующий свойства часов, добавив новые операции, свойственные будильникам. Это можно сделать быстро, поскольку класс часов уже разработан. Таким образом, **наследование** (inheritance) позволяет повторно использовать классы, определенные ранее (возможно, для других, но похожих целей), выполняя соответствующие модификации.

Наследование может поставить компилятор в затруднительное положение, поскольку он не сможет определить, какую операцию следует выполнить в конкретной ситуации. Однако полиморфизм (polymorphism) — буквально означающий *изменчивость форм* — позволяет выбрать нужную операцию уже на этапе выполнения программы. Таким образом, результат выполнения конкретной операции зависит от объектов, к которым она применяется.

Перегруженный оператор имеет несколько значений

Например, если в программе используется оператор +, операндами которого являются числа, то выполняется сложение чисел, но если к строкам приме-

няется **перегруженный** (overloaded) оператор +, то выполняется их конкатенация. Хотя в данном случае компилятор может сам определить правильный смысл оператора +, полиморфизм допускает ситуации, когда смысл операции уточняется лишь на этапе выполнения программы.

Наследование и полиморфизм обсуждаются в главе 8.

Проектирование "сверху вниз"

Обычно объектно-ориентированный подход приводит к модульному решению задач, основываясь лишь на анализе данных. При разработке алгоритма для конкретной функции или в ситуациях, когда на первое место выходит алгоритм, а не данные, с которыми он работает, модульное решение можно получить с помощью **проектирования "сверху вниз"** (top-down design). В то время как с помощью объектно-ориентированного подхода можно идентифицировать данные, основываясь на именах существительных, использованных в описании задачи, проектирование "сверху вниз" основано на анализе глаголов.

Структурная схема иллюстрирует отношения между модулями

Стратегия проектирования "сверху вниз" основана на последовательном понижении уровня детализации задачи. Рассмотрим простой пример. Допустим, что нам нужно вычислить среднюю экзаменационную оценку. На рис. 1.4 показана **структурная схема** (structire chart), иллюстрирующая иерархию модулей и взаимодействие между ними. Во-первых, для каждого модуля указывается лишь описание его *предназначения*, лишенное каких-либо деталей. Каждый модуль разбивается на несколько более мелких модулей. В результате возникает иерархия модулей. Каждый модуль уточняется его наследником, решающим более мелкую задачу и содержащим больше информации о *способе* решения задачи, чем его предшественник. Процесс уточнения продолжается, пока модули не окажутся достаточно простыми для представления их в виде функций на языке C++ и изолированных фрагментов кода, решающих очень маленькие, независимые друг от друга задачи.

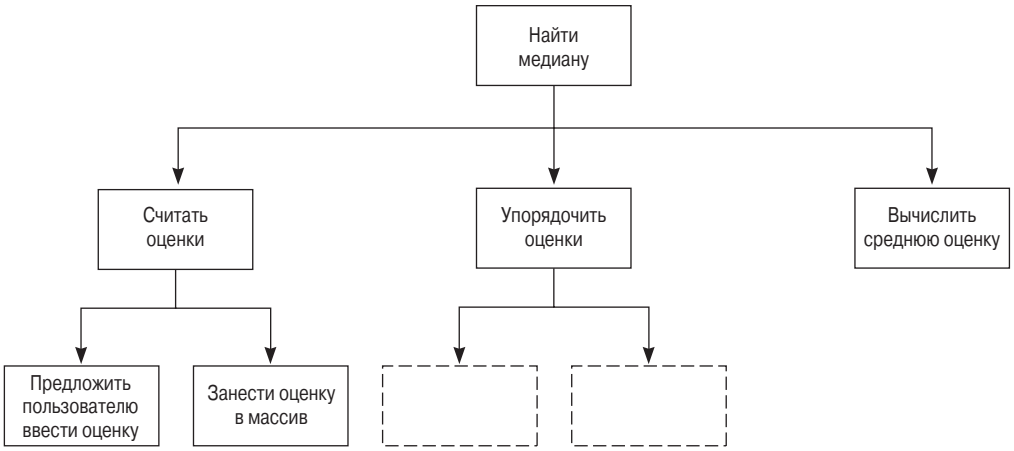

Рис. 1.4. Структурная схема, иллюстрирующая иерархию модулей

Обратите внимание, что на рис. 1.4 задача разбивается на три независимые подзадачи.

Решение состоит из независимых подзадач

- *Считать экзаменационные оценки*
- *Упорядочить оценки*
- *Определить "среднюю" оценку*

Если три эти задачи решаются тремя разными модулями, то, вызывая их, можно найти среднюю оценку, независимо от способов их реализации.

Разработка каждого модуля начинается с разбиения его на подзадачи. Например, задачу считывания оценок можно уточнить с помощью двух модулей.

Подзадачи

- *Предложить пользователю ввести оценку*
- *Записать оценку в массив*

Каждый из этих модулей можно уточнить аналогичным способом. В итоге мы получим псевдокод алгоритма, решающего поставленную задачу.

Общие принципы проектирования

Обычно при решении задачи используются объектно-ориентированное проектирование (ООП), проектирование "сверху вниз" (ПСВ), абстракция и сокрытие информации. Подход, ведущий к модульному решению задачи, описывается следующими принципами проектирования.

ОСНОВНЫЕ ПОНЯТИЯ

Принципы проектирования

1. Для получения модульного решения одновременно используйте объектно-ориентированное проектирование и подход "сверху вниз". Таким образом, абстрактные типы данных и алгоритмы нужно разрабатывать параллельно.
2. Для решения задач обработки данных используйте объектно-ориентированное проектирование.
3. Для разработки алгоритмов используйте подход "сверху вниз".
4. Если главными в решении задачи являются алгоритмы, а не данные, применяйте проектирование "сверху вниз".
5. При разработке абстрактных типов данных и алгоритмов акцентируйте внимание на вопросе *что*, а не *как*.
6. Старайтесь применять готовые компоненты программного обеспечения.

Моделирование объектно-ориентированных проектов с помощью языка UML

Универсальный язык моделирования (UML — Unified Modeling Language) используется для описания объектно-ориентированных проектов. Этот язык содержит спецификации диаграмм и текстовых описаний. Диаграммы особенно полезны для общего описания проектов, включая спецификации классов, и разных способов взаимодействия между ними. Обычно программа состоит из многих классов, поэтому возможность описывать взаимодействия между ними представляет собой ценное свойство языка UML.

В этом разделе мы рассмотрим лишь спецификации классов, поэтому он содержит только диаграммы классов и связанные с ними синтаксические конструкции. В диаграмме класса указывается его имя, данные-члены и операции. На рис. 1.5 показана диаграмма класса `Clock`, описанного выше. Верхний раздел диаграммы содержит имя класса. Средний раздел содержит данные-члены, а нижний — операции класса. Обратите внимание, что диаграмма носит довольно общий характер; она не диктует выбор фактической реализации класса. Это типичное представление концептуальной модели класса, не зависящее от выбора языка его реализации.

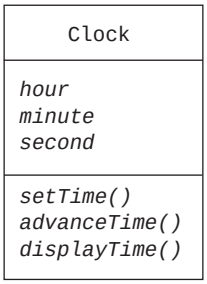

Рис. 1.5. Диаграмма класса Clock на языке UML

Наряду с диаграммами классов язык UML позволяет создавать текстовые описания для представления данных-членов и операций, выполняемых в классе. Эти записи можно включать в диаграммы классов, однако это усложняет диаграммы, снижая степень их общности. В данном разделе мы будем использовать именно текстовые описания классов, поскольку они позволяют создавать более полные спецификации, чем диаграммы.

Синтаксис описания данных-членов на языке UML имеет следующий вид.

модификатор_доступа имя: тип = значение_по_умолчанию

Здесь использованы следующие обозначения.

- Модификатор доступа принимает значение + (`public`) или - (`private`). Третье возможное значение — символ # (`protected`). Эту возможность мы обсудим в главе 8.
- Элемент *имя* означает имя атрибута.
- Элемент *тип* означает тип атрибута.
- Элемент *значение_по_умолчанию* задает начальное значение атрибута.

Как показывает диаграмма класса, нужно задать хотя бы имя класса. Элемент *значение_по_умолчанию* используется лишь в тех ситуациях, когда значение атрибута задается по умолчанию. В некоторых случаях нужно избегать явного указания типа атрибута, отложив решение этого вопроса до этапа реализации. В дальнейшем мы будем использовать следующие названия распространенных типов аргументов: `integer` — для целочисленных значений, `float` — для значений с плавающей точкой, `boolean` — для булевых значений и `string` — для строковых значений. Обратите внимание, что эти имена не совпадают с соответствующими названиями типов данных в языке C++, поскольку текстовое описание класса не должно зависеть от языка его реализации.

Вот как выглядит текстовое описание атрибутов класса `Clock`, показанного на рис. 1.5.

- *–hour: integer*
- *–minute: integer*
- *–second: integer*

Следуя принципу сокрытия информации, данные-члены `hours`, `minute` и `second` объявлены закрытыми.

Синтаксические конструкции языка UML, предназначенные для описания операций, выглядят немного сложнее.

модификатор_доступа имя(список_параметров):
тип_возвращаемого_значения (строка_свойств)

Здесь использованы следующие обозначения.

- Модификатор доступа принимает те же значения, что и в предыдущем случае.
- Элемент *имя* означает имя операции.

- Элемент *список_параметров* содержит параметры, разделенные запятой. Синтаксическая конструкция для описания параметров выглядит следующим образом.

  направление имя: тип = значение_по_умолчанию.
- Здесь элемент *направление* используется для индикации ввода (`in`), вывода (`out`) или ввода-вывода (`inout`) параметра.
- Элемент `name` является именем параметра.
- Элемент `type` задает тип параметра.
- Элемент *значение_по_умолчанию* задает значение, которое следует присвоить параметру, если соответствующий аргумент пропущен.
- Элемент *тип_возвращаемого_значения* задает тип значения, возвращаемого операцией; если операция не возвращает никакого значения, место этого элемента остается пустым.
- Элемент *строка_свойств* перечисляет свойства операции.

Как и для атрибутов, в диаграммах классов нужно указывать хотя бы *имя* операции. Иногда в диаграмму включается *список_параметров*, если это позволяет лучше понять функциональные возможности класса.

Строка_свойств может содержать множество разнообразных значений, однако нас будет интересовать лишь свойство `query`. Это свойство позволяет идентифицировать операции, которые не имеют права модифицировать данные, содержащиеся в классе.

Текстовое описание операций, предусмотренных в классе `Clock`, имеет следующий вид.

- *+setTime(in hr: integer, in min: integer, in sec: integer)*
- *–advanceTime( )*
- *+displayTime( ) (query)*

Здесь операции `setTime` и `displayTime` определены открытыми, а операция `advanceTime` — закрытой. Функция `displayTime` имеет свойство `query`, означающее, что она не изменяет никаких данных. Эта функция лишь выводит данные на экран.

Преимущества объектно-ориентированного подхода

При использовании объектно-ориентированного подхода (ООП) время, затрачиваемое на проектирование программы, увеличивается. Кроме того, решение, к которому приводит этот подход, обычно носит более общий характер, чем это необходимо. Однако дополнительные усилия, потраченные на ООП, обычно компенсируются.

Используя объектно-ориентированное проектирование при решении задач, необходимо идентифицировать возникающие классы. При этом указывается предназначение каждого класса и способ его взаимодействия с другими классами. Таким образом, возникает спецификация каждого класса, в которой указываются его данные и операции. Затем центр внимания перемещается на детали реализации каждого класса, используя подход “сверху вниз” для разработки операций. Классы легче реализовывать по отдельности.

Реализовав класс, необходимо провести его двойное тестирование. Во-первых, нужно проверить операции класса. Для этого обычно создают небольшие программы, вызывающие разные операции и проверяющие результаты в соответствии с их спецификациями. Проверив каждый класс, нужно провести тестирование взаимодействий между классами, возникающих при решении задачи.

При идентификации классов, возникающих при решении задачи, часто обнаруживаются семейства

Семейство связанных классов

связанных друг с другом классов. Этот этап занимает много времени, особенно если классы разрабатываются с нуля. Реализовав один класс (называемый **предком** (ancestor)), можно ускорить создание новых классов (**потомков** (descendant)), поскольку потомки могут наследовать данные и операции предка.

Повторное использование классов

Например, как указывалось выше, определив класс часов, можно разработать класс будильников, наследующий свойства часов, но обладающий дополнительными особенностями. На реализацию класса будильников пришлось бы затратить намного больше времени, если бы класс часов не был разработан раньше. Ранее реализованные классы можно применять в новых программах либо без изменения, либо с модификациями, которые включают в себя новые классы, производные от существующих. Повторное использование классов позволяет сократить время, затрачиваемое на объектно-ориентированное проектирование.

Наследование облегчает эксплуатацию и верификацию программ

Объектно-ориентированное программирование оказывает положительное влияние и на другие этапы жизненного цикла программного обеспечения, в частности на эксплуатацию и верификацию программ. Для изменения всей цепочки потомков достаточно модифицировать их предка. Если бы не было наследования, изменения пришлось бы вносить в каждый класс иерархии. Кроме того, программу можно обогатить новыми свойствами, добавляя новых потомков. Это никак не влияет на их предков, и, следовательно, не порождает новых ошибок в остальной части программы. Можно даже добавлять потомков, которые изменяют поведение их предка, даже если он был написан и скомпилирован очень давно.

Краткий обзор основных понятий программирования

Будем считать хорошим наиболее эффективное решение задачи. Тогда возникают вопросы: чем отличается хорошее решение от плохого и как сконструировать хорошее решение? В этом разделе мы попробуем кратко подытожить ответы на эти очень трудные вопросы.

Темы, которые здесь обсуждались, вам должны быть знакомы. Однако новички обычно не придают этим вопросам большого значения. Освоив первый курс программирования, многие студенты считают достаточным, если программа “просто работает”. Последующее обсуждение должно помочь читателям понять, насколько важны эти вопросы.

Люди тоже читают программы

Одно из наиболее распространенных заблуждений гласит: программы предназначены только для компьютеров. Как следствие, новички считают, что их программы могут “понимать” только компьютеры — ведь это они их компилируют, выполняют и выдают результаты их работы! Однако и другие люди тоже часто вынуждены читать и модифицировать программы. Обычно в программистской среде над программой работают несколько человек. Один программист может написать программу, которую другие люди будут использовать вместе со своими программами, а через несколько лет совсем другие люди станут ее модифицировать. Следовательно, очень важно, чтобы программы можно было легко читать и понимать.

Программист должен постоянно помнить о шести принципах программирования.

ОСНОВНЫЕ ПОНЯТИЯ

Шесть принципов программирования

1. Модульность.
2. Модифицируемость.
3. Легкость использования.
4. Безопасность.

5. Стиль.
6. Отладка.

Модульность

На протяжении всей книги мы будем постоянно напоминать, что на каждом этапе решения задачи необходимо придерживаться принципа модульности, начиная с самого начала. В предыдущих разделах мы уже указывали, что задачи становятся все больше и сложнее. Модульность позволяет понизить уровень сложности программы. Благотворное влияние модульности проявляется в следующих аспектах программирования.

Модульность облегчает программирование

- **Конструирование программ.** Единственное различие между маленькой модульной программой и большой модульной программой заключается в количестве модулей, из которых они состоят. Поскольку модули не зависят друг от друга, создание большой модульной программы не очень отличается от написания многих маленьких независимых модульных программ, хотя взаимодействие между модулями может быть весьма сложным. Работа над большой цельной программой напоминает одновременную работу со множеством маленьких взаимосвязанных программ. Кроме того, модульность позволяет применить командный способ программирования, при котором несколько программистов работают независимо друг от друга, а затем объединяют свои модули в одну программу.

Модульность позволяет изолировать ошибки

- **Отладка программ.** Отладка большой программы может оказаться практически невыполнимой задачей. Представьте себе, что вы набрали 10 000 строк и наконец-то приступили к их компиляции. Ничего не может быть приятнее! Теперь представьте, что при выполнении программы среди нескольких сотен строк вывода вы обнаружили неверное число. Пройдет несколько дней, пока вы продеретесь сквозь переплетения операторов и узнаете причину этой ошибки, которая может оказаться вполне невинной.

  Большое преимущество модульного подхода заключается в том, что задача отладки большой программы сводится к отладке множества маленьких подпрограмм. Начиная кодировать модуль, вы должны быть уверены, что все остальные модули закодированы правильно. Следовательно, закончив программирование модуля, вы должны внимательно проверить его, как отдельно, так и вместе с другими модулями, вызывая его с фактическими аргументами, тщательно подобранными для выявления всех возможных недостатков. Если это тестирование проведено подобающим образом, можно быть уверенным, что любая обнаруженная ошибка содержится только в модуле, который кодировался последним. *Модульность позволяет изолировать ошибки*.

  Теоретически можно применять формальные методы проверки программ. Модульные программы хорошо поддаются такой верификации.

Модульные программы легко читать

- **Чтение программ.** Человек, читающий большую программу, чувствует себя заблудившимся в глухом лесу. Модульный подход не только позволяет программистам справляться со сложностями, возникающими при решении задачи, но и помогает читателям программы понять, как она работает. Модульную программу легко отследить, поскольку читатель хорошо представляет себе, что происходит, не вдаваясь в детали кода. Для того чтобы разобраться в хорошо написанной функции, достаточно лишь прочитать ее имя, начальные комментарии и имена функций, которые вызываются внутри нее. Читатели программы должны вникать в тонкости кода, только если они хотят понять

детали выполняемых операций. Читабельность программ обсуждается в разделе, посвященном стилю программирования.

- **Модификация программ.** Модифицируемость — это тема следующего раздела, однако она тесно связана с модульностью программы, поэтому о ней стоит вспомнить. Небольшое изменение в требованиях, предъявляемых к программе, должно приводить к небольшому изменению ее кода. Если это не так, значит, программа плохо написана и, в частности, не обладает свойством модульности. Чтобы учесть небольшие изменения в исходных требованиях, в модульной программе обычно достаточно изменить лишь несколько модулей, особенно если модули не зависят друг от друга (т.е. слабо связаны) и каждый модуль выполняет отдельную точно поставленную задачу (т.е. высоко координирован).

  Модульность изолирует изменения

  Вносить изменения в программу нужно постепенно. При модульном подходе большие изменения разбиваются на множество маленьких и относительно простых модификаций в изолированных частях программы. *Модульность изолирует модификации*.

- **Исключение избыточного кода.** Другое преимущество модульного подхода проявляется в идентификации вычислений, которые в программе выполняются несколько раз. Такие вычисления следует реализовывать в виде функций. В этом случае код, предназначенный для таких вычислений, в программе встречается только один раз, повышая ее читабельность и модифицируемость. В следующем разделе мы продемонстрируем это на конкретном примере.

  Модульность исключает избыточность

Модифицируемость

Представьте себе, что спецификация программы через какое-то время изменилась. Обычно люди не вполне отчетливо представляют себе, чего они хотят от программы, постепенно уточняя ее спецификацию. В этом разделе указаны три способа, позволяющие облегчить изменение программы: использование функций, именованных констант и операторов `typedef`.

Функции. Допустим, что в некую библиотеку входит большая программа для ведения каталога книг. В некоторых точках программа выводит на экран информацию о заказанной книге. В каждой из этих точек программа может вызывать оператор `cout`, для того чтобы вывести на экран номер, фамилию автора и название книги. Этот оператор можно заменить вызовом функции `displayBook`, которая выводит ту же самую информацию.

Функции облегчают модификацию программ

Функции позволяют не только исключить избыточный код, но и облегчают модификацию программ. Например, чтобы изменить формат вывода, достаточно изменить реализацию функции `displayBook`, а не вносить исправления в многочисленные операторы `cout`, как это предполагалось в исходном варианте. Если бы функции не было, пришлось бы вносить изменения в каждой точке программы, где на экран выводится информация о книгах. Найти каждую такую точку было бы достаточно трудно и, вероятно, некоторые из них остались бы не измененными. Этот простой пример наглядно демонстрирует преимущества использования функций.

В качестве другой иллюстрации напомним пример, рассмотренный нами ранее, в котором упорядочивались данные. Разрабатывая алгоритм сортировки в виде отдельного модуля и реализуя его в виде функции, можно сделать программу легко модифицируемой. Например, если алгоритм сортировки окажется слишком медленным, можно просто заменить соответствующую функцию, оставив неизменной остальную часть программы. Нужно лишь “вырезать” старую функцию и “вставить” новую. Если бы сортировка была интегрирована в программу, понадобилась бы довольно сложная хирургическая операция.

В общем, будьте готовы переписать вашу программу, чтобы учесть небольшие изменения в ее спецификации. Обычно хорошо организованные программы модифицируются легко: поскольку каждый ее модуль решает определенную часть общей задачи, небольшое изменение в постановке задачи влияет лишь на отдельные модули.

Именованные константы облегчают модификацию программ

Именованные константы. Для облегчения модификации программы можно применять именованные константы. Например, если на размер массива, используемого в программе, накладываются ограничения, исправить его довольно сложно. Допустим, что программа использует массив для обработки экзаменационных оценок по компьютерным наукам. В момент написания программы курс компьютерных наук слушали 202 студента, поэтому массив был объявлен следующим образом.

```
int scores[202];
```

Программа обрабатывает массив несколькими способами. Например, она считывает оценки, записывает их и усредняет. Псевдокод решения каждой из этих задач содержит примерно такую конструкцию.

```
for (index = 0 through 201)
   Обработка оценок
```

Если количество студентов изменится, нужно не только изменить объявление массива *scores*, но и модифицировать каждый цикл, чтобы учесть новый размер массива. Кроме того, размер массива может влиять на другие операторы в программе. Здесь 202, а там 201 — что изменять?

Однако можно применить именованную константу.

```
const int NUMBER_OF_MAJORS = 202;
```

Тогда массив можно объявить следующим образом.

```
int scores[NUMBER_OF_MAJORS];
```

Псевдокод соответствующих циклов примет такой вид.

```
for (index = 0 through NUMBER_OF_MAJORS-1)
   Обработка оценок
```

В выражениях, которые включают в себя размер массива, нужно использовать именованную константу *NUMBER_OF_MAJORS* (например, *NUMBER_OF_MAJORS-1*). Тогда размер массива можно изменить, изменив всего лишь определение именованной константы и скомпилировав программу снова.

Операторы typedef облегчают модификацию программ

Оператор typedef. Допустим, что ваша программа выполняет вычисления с переменными, имеющими тип *float*, и вдруг обнаружилось, что точности типа *float* недостаточно. Например, для того чтобы изменить объявление типа *float* на объявление типа *long double*, придется пройтись по всем объявлениям и в каждом из них сделать соответствующее изменение.

Для того чтобы облегчить процесс изменений, используется оператор *typedef*, который переименовывает существующий тип. Например, оператор

```
typedef float RealType;
```

объявляет тип *RealType* синонимом типа *float*, что позволяет использовать их с одинаковым успехом. Если в предыдущей программе все переменные типа *float* объявить как пе-

ременные типа *RealType*, то программу будет легко модифицировать и читать. Для того чтобы изменить точность вычислений, нужно просто изменить оператор *typedef*.

```
typedef long double RealType;
```

Легкость использования

Разрабатывая интерфейс программы, нужно думать о людях, которые будут с ней работать. Пользователи часто вводят в программу входные данные и анализируют полученные результаты. При этом следует учитывать следующие очевидные особенности.

Приглашение к вводу данных

- В интерактивной среде ввод данных должен быть простым и ясным. Например, приглашение "?" невозможно сравнить с предложением "Пожалуйста, введите номер вашего банковского счета." Никогда не следует рассчитывать, что пользователи интуитивно догадаются, какого ответа ждет от них программа.

Эхо ввода

- Программа всегда должна выводить эхо входных данных. Если программа считывает данные, неважно, с клавиатуры или из файла, она должна выводить их на экран. Это необходимо по двум причинам. Во-первых, это позволяет пользователям контролировать входные данные, предотвращая опечатки и ошибки. Эта проверка особенно полезна в интерактивном режиме. Во-вторых, выходные данные более осмысленны и самоочевидны, если они содержат исходные данные, введенные пользователем.

Разметка вывода

- Вывод должен быть хорошо размеченным и понятным. Рассмотрим в качестве примера следующий набор выходных данных.

```
18:00 6 1
Джонс, К. 223 2234.00 1088.19 Н, О Смит, Т. 111
110.23 З, Харрис, У. 44 44000.000 22222.22
```

- Эти данные намного легче интерпретировать, если вывести их в следующем виде.

```
Счета вкладчиков по состоянию на 18:00 1 июня

Состояние счета: Н - новый, О- общий, З - закрыт

Имя         Номер    Снятие     Вклады     Состояние
Джонс, К.   223      $ 2234.00  $ 1088.19  Н, О
Смит, Т.    111      $  110.23  ---------  З
Харрис, У.   44      $44000.00  $22222.22  ------
```

Хороший пользовательский интерфейс имеет большое значение

Это лишь самые общие характеристики хорошего пользовательского интерфейса. В зависимости от более тонких моментов, программы классифицируются от просто пригодных к работе до дружелюбных к пользователю. Обычно студенты стремятся игнорировать необходимость разработки хорошего пользовательского интерфейса. Однако, посвятив этому немного дополнительного времени, они могут обнаружить существенную разницу между хорошей программой и программой, которая просто решает задачу. Рассмотрим, например, программу, предлагающую пользователю ввести строку данных в некотором фиксированном формате, где элементы ввода разделяются только одним пробелом. Свободный формат ввода, допускающий несколько пробелов между данными, был бы более удобен для пользователя. Для создания цикла, игнорирующего пробелы, нужно затратить совсем немного времени, так зачем же навязывать пользователю фиксированный формат? Кроме то-

го, разработав такой интерфейс однажды, вы можете затем постоянно использовать его в своих программах и библиотеках, а пользователь никогда не будет беспокоиться о формате входных данных.

Надежное программирование

Надежная программа всегда работает безотказно, независимо от способов ее применения. К сожалению, эта цель является практически недостижимой. Намного реальнее ограничить возможности неправильного обращения с программой и предотвратить эти ошибки.

Проверка ошибок при вводе данных

Мы рассмотрим два вида ошибок. Первая разновидность — это *ошибки при вводе данных*. Допустим, например, что программа ожидает ввода неотрицательного числа, а на вход поступает число –12. Обнаружив такую ошибку, программа не должна вычислять неверный результат или прекращать работу, выдав непонятное сообщение об ошибке. Вместо этого надежная программа должна вывести на экран сообщение, имеющее приблизительно следующее содержание.

```
-12 — неправильное количество детей.
Пожалуйста, повторите ввод.
```

Проверка логики программы

Вторая разновидность ошибок — *семантические*, т.е. *ошибки в логике программы*. Хотя они тесно связаны с процессом отладки, который будет обсуждаться в конце этой главы, обнаружение семантических ошибок является этапом безопасного программирования. Внешне совершенно правильные программы в некоторых ситуациях начинают вести себя непредсказуемо, даже если введенные данные были абсолютно корректными. Например, программист мог не предусмотреть реакцию программы на конкретные данные, даже если во всем остальном ее логика безупречна. Кроме того, модифицируя часть программы, авторы часто нарушают предположения, которые должны выполняться в отношении остальных ее частей. Программа должна быть организована так, чтобы семантические ошибки такого рода не возникали. Она должна постоянно контролировать себя, обнаруживая отклонения и неверные результаты.

Предотвращение неверного ввода. Допустим, что мы должны вычислить статистические показатели, касающиеся людей, чей годовой доход колеблется от $10000 до $100000. Суммы округляются до тысяч: $10000, $11000 и т.д. Исходные данные хранятся в файле, состоящем из одной или нескольких строк, имеющих следующий вид.

```
G  N
```

Здесь *N* — это количество людей, попадающих в группу с доходом *G* тысяч долларов в год. Если эти данные записывали несколько разных людей, то в файле могут оказаться несколько записей, относящихся к одному и тому же числу *G*. После ввода данных программа должна суммировать их и записать количество людей, соответствующее каждой величине *G*. В этом контексте совершенно ясно, что *G* — это целое число, изменяющееся в диапазоне от 10 до 100 включительно, а *N* — неотрицательное целое число.

Чтобы продемонстрировать, как можно предотвратить ввод неверных данных, рассмотрим функцию, предназначенную для ввода чисел при решении поставленной выше задачи. Первый вариант этой функции иллюстрирует, насколько программа оказывается далекой от идеала. В конце концов, нам все же удастся приблизить функцию ввода данных к желательному эталону.

Первый вариант функции выглядит следующим образом.

Ненадежная функция

```
const int LOW_END = 10;
// Нижняя граница доходов
const int HIGH_END = 10;   // Верхняя граница доходов
const int TABLE_SIZE = HIGH_END - LOW_END + 1;
typedef int TableType[TABLE_SIZE];

int index(int group)
// Возвращает индекс массива, соответствующий номеру группы.
{
   return group - LOW_END;
} // Конец функции index

void readData(TableType incomeData)
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
// Считывает и организовывает статистические данные о доходах.
// Предусловие: вызываемый модуль выдает инструкции и
// предлагает пользователю ввести данные. Входные данные
// не должны содержать ошибки. Каждая строка имеет вид G N,
// где N — количество людей, чей годовой доход равен
// G тысяч долларов, причем LOW_END <= G <= HIGH_END.
// Ввод данных завершается после считывания строки,
// в которой числа G и N равны нулю.
// Постусловие: число incomeData[G-LOW_END] равно общему
// количеству людей, чей доход равен G тысяч долларов для
// каждого считанного значения G. Считанные значения
// выводятся на экран.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   int group, number; // Входные значения

   // Очищаем массив
   for (group = LOW_END; group <= HIGH_END; ++group)
      incomeData[index(group)] = 0;

   for (cin >> group >> number;
      (group != 0) || (number != 0);
      cin >> group >> number)
   {  // Инвариант: переменные group и number не равны нулю
      cout << "Количество людей в группе" << group <<
         " равно " << number << ".\n";
      incomeData[index(group)] += number;
   } // Конец цикла for
} // Конец функции readData
```

Эта функция порождает несколько проблем. Если входная строка содержит неожиданные данные, программа не сможет на них адекватно среагировать. Рассмотрим две конкретные возможности.

- Первое целое число, которое функция присваивает переменной *group*, выходит за пределы допустимого диапазона (от *LOW_END* до *HIGH_END*). В этом случае обращение к элементу массива *income[index(group)]* становится некорректным.
- Второе целое число, которое функция присваивает переменной *number*, является отрицательным. Несмотря на то что отрицательное значение переменной *number* лишено смысла, так как количество людей в группе не может быть меньше нуля, функция

добавит его в массив. Таким образом, массив *incomeData* будет содержать неверные данные.

Проверка неправильных входных данных

После считывания данных нужно проверить, лежит ли значение переменной *group* в допустимом диапазоне (от *LOW_END* до *HIGH_END*) и является ли переменная *number* положительной. Если это не так, необходимо обработать ошибку ввода.

Вместо проверки переменной *number* можно было бы проверить, положителен ли элемент *incomeData[index(group)]* после добавления к нему числа *number*. Однако такой подход неэффективен. Во-первых, добавить отрицательное число к элементу массива *incomeData* можно так, что сам он не станет отрицательным. Например, если число *number* равно –4000, а соответствующий элемент массива *incomeData* равен 10000, то их сумма будет равна 6000. Следовательно, факт, что число *number* отрицательно, останется незамеченным. Это приведет к неправильной работе всей остальной программы.

При обнаружении неправильных входных данных возможны несколько сценариев. Во-первых, функция может установить соответствующий признак ошибки и прекратить работу. Во-вторых, функция может установить соответствующий признак ошибки, проигнорировать ее и продолжить работу. Какой из этих сценариев выбрать, зависит от конкретной ситуации.

Функция *readData*, приведенная ниже, универсальна и максимально облегчает модифицируемость программы, в которой она используется. Обнаружив ошибку при вводе данных, она задает ее признак, игнорирует неправильную строку и продолжает работу. Установив признак ошибки, функция предоставляет вызывающему модулю возможность самому принять решение — прекратить работу или продолжить выполнение программы. Таким образом, эту функцию можно применять в разных контекстах, легко модифицируя реакцию на обнаружение ошибки.

Надежная функция

```
bool readData(TableType incomeData)
// - - - - - - - - - - - - - - - - - - -
// Считывает и организовывает статистические данные.
// Предусловие: вызываемый модуль выдает инструкции и
// предлагает пользователю ввести данные. Каждая строка
// содержит два целых числа в виде G N, где N — количество
// людей, чей годовой доход равен G тысяч долларов, причем
// LOW_END <= G <= HIGH_END. Ввод данных завершается после
// считывания строки, в которой числа G и N равны нулю.
// Постусловие: число incomeData[G-LOW_END] равно общему
// количеству людей, чей доход равен G тысяч долларов для
// каждого считанного значения G. Считанные значения
// выводятся на экран. Если числа G или N неверны
// (не равны нулю и G < LOW_END, G > HIGH_END или N < 0),
// функция игнорирует строку ввода, задает возвращаемое
// значение равным false и продолжает работу.
// Решение о продолжении выполнения программы принимает
// вызывающий модуль. Если входные данные не содержат ошибок,
// функция возвращает значение true.
// - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
{
   int group, number; // Входные значения
   bool dataCorrect = true; // Ошибок пока нет

   for (group = LOW_END; group <= HIGH_END; ++group)
      incomeData[index(group)] = 0;
```

```
    for (cin >> group >> number;
        (group != 0) || (number != 0);
        cin >> group >> number)
    {   // Инвариант: переменные group и number не равны нулю
        cout << "Количество людей в группе" << group <<
           " равно " << number << ".\n";

        if ((group >= LOW_end) && (group <= HIGH_END) &&
           (number >= 0))
           // Входные данные корректны -- добавляем их в счетчик
           incomeData[index(group)] += number;

        else
           // Ошибка при вводе данных:
           // устанавливаем признак ошибки, игнорируя строку
           dataCorrect = false;
    } // Конец цикла for
    return dataCorrect;
} // Конец функции readData
```

Хотя в большинстве случаев эта функция работает отлично, все же она еще недостаточно надежна. Что произойдет, если входная строка будет содержать лишь одно целое число? А если числа в этой строке окажутся нецелыми? Функция была бы более надежной, если бы она считывала данные посимвольно, конвертируя их в целое число и проверяя конец строки. Чаще всего это было бы небольшим излишеством. Однако если люди, вводящие данные, часто делают ошибки, набирая нецелые числа, функцию ввода можно было бы легко изменить, поскольку она реализована в виде изолированного модуля. В любом случае в комментариях, сопровождающих текст функции, нужно формулировать все предположения о входных данных и указывать, как функция реагирует на неправильный ввод.

Предотвращение семантических ошибок. Рассмотрим теперь вторую разновидность: семантические ошибки. Их иногда не удается выловить на этапе отладки и легко внести, модифицируя программу.

К сожалению, сама программа не может сообщить, что в ней кроется ошибка. (Неправильная программа не знает, что она неправильная.) Однако в программу можно включить проверку определенных условий, которые должны выполняться, если она работает правильно. Как уже указывалось, эти условия называются инвариантами.

Функции должны проверять свои инварианты

В качестве простого инварианта рассмотрим предыдущий пример. *Все целые числа в массиве* `incomeData` *должны быть неотрицательными*. Хотя выше мы сказали, что проверка элементов массива `incomeData` неэффективна при анализе числа `number`, ее можно использовать в качестве *дополнительного условия*. Например, если функция `readData` обнаружила, что какой-то элемент массива `incomeData` выходит за пределы допустимого диапазона, это является сигналом о потенциальных проблемах.

Функции должны проверять предусловия

Еще один способ повышения надежности программы заключается в проверке предусловий функций. Рассмотрим, например, функцию `factorial`, вычисляющую факториал целого числа.