Краткое содержание

Содержание

1 Знакомство с объектами

Компьютерная революция начиналась с машин, поэтому многие языки программирования ориентированы на машины.

Однако компьютер — не столько машина, сколько инструмент, расширяющий возможности человеческого разума, — «велосипед для разума», как выражается Стив Джобс (Steve Jobs), и новая разновидность выразительных средств. В результате этот инструмент все меньше напоминает машины и все больше — новые стороны нашего разума и такие выразительные средства, как живопись, скульптура, мультипликация или кино. Объектно-ориентированное программирование стало одним из направлений на пути превращения компьютера в выразительное средство.

В этой главе читатель познакомится с основными концепциями объектно-ориентированного программирования (ООП), включая краткий обзор методов ООП-разработки. Настоящая глава (как и книга в целом) предполагает, что у читателя имеется опыт работы на процедурных языках программирования — не обязательно на языке C.

Приведенный материал носит общий, подготовительный характер. Некоторые читатели чувствуют себя неуверенно, если перед погружением в мир объектно-ориентированного программирования они не увидят «общую картину». Для таких читателей и был подготовлен содержательный обзор ООП. С другой стороны, многие читатели просто не воспринимают «общую картину», пока не узнают, как «это» работает на практике; они вязнут в общих описаниях, пока не увидят конкретный программный код. Если вы относитесь ко второй категории и желаете как можно скорее начать практическое знакомство с языком, спокойно пропустите эту главу — это не помешает изучать язык или писать собственные программы. Впрочем, когда-нибудь в будущем к этой главе было бы полезно вернуться. Так вы сможете понять, почему объекты настолько важны и как проектировать программы на объектной основе.

Развитие абстрактных представлений

Любой язык программирования основан на абстрактных представлениях. Более того, сложность решаемых задач напрямую зависит от типа и качества этой абстракции. Под «типом» подразумевается ответ на вопрос: «Что представляет данная абстракция?» Например, ассемблер можно рассматривать как маленькую абстракцию того компьютера, на котором он работает. Многие так называемые «массовые» языки, появившиеся позднее (например, Fortran, BASIC и C), могли рассматриваться как абстракции языка ассемблера. Они были гораздо совершеннее, но их первичная абстракция по-прежнему заставляла программиста мыслить в контексте структуры компьютера, а не структуры решаемой задачи. Программисту приходилось сопоставлять машинную модель («пространство решения», то есть контекст, в котором моделируется задача, — в данном случае компьютер) и модель решаемой задачи («пространство задачи», в котором определяется поставленная задача). Такое сопоставление требовало определенных усилий и было чужеродным по отношению к языку программирования. Это затрудняло разработку программ и повышало затраты на их сопровождение; в результате возникла целая отрасль «методологии программирования».

Наряду с моделированием компьютера также существует другое решение — моделирование решаемой задачи. В ранних языках типа LISP и APL было выбрано особое представление мира («Любая задача сводится к обработке списков» или «Любая задача алгоритмизируется»). В языке PROLOG задача интерпретировалась как цепочка принимаемых решений. Создавались специальные языки программирования, основанные на системах ограничений или на манипуляциях с графическими условными обозначениями (впрочем, последние слишком сильно ограничивали свободу действий программиста). Все эти подходы хорошо соответствовали конкретному классу задач, для которых они разрабатывались, но за пределами этой области были весьма неудобными.

Объектно-ориентированный подход не ограничивается узкой специализацией. Он дает программисту средства представления элементов в пространстве задачи. Представление носит достаточно обобщенный характер, чтобы программист не ограничивался конкретным типом задач. Элементы пространства задачи и их представления в пространстве решения называются *объектами* (конечно, наряду с ними существуют другие объекты, не имеющие аналогов в пространстве задачи). Идея заключается в том, чтобы программа могла адаптироваться к формулировке задачи за счет добавления новых типов объектов. Таким образом, читая программный код с описанием решения, вы также будете читать словесную формулировку задачи. По своей гибкости и мощи эта языковая абстракция превосходит все, что существовало до нее. Итак, ООП позволяет описать задачу в контексте задачи, а не в контексте того компьютера, на котором она будет решаться. Впрочем, определенная связь с компьютерами все же сохранилась. Каждый объект отдаленно напоминает маленький компьютер — он тоже обладает состоянием и поддерживает набор операций, которые вы можете выполнять. С другой стороны, нетрудно провести аналогию с объектами реального мира, которые также обладают характеристиками и определенным поведением.

Некоторые проектировщики языков посчитали, что самого по себе объектно-ориентированного программирования недостаточно для простого решения всех задач программирования, и выступили за объединение разных подходов в *мультипарадигменных* языках программирования.

Алан Кей (Alan Kay) кратко сформулировал пять основных концепций SmallTalk — первого успешного объектно-ориентированного языка, который стал одним из прототипов C++. Эти концепции представляют «чистый» подход к объектно-ориентированному программированию.

- *Любая сущность представляется объектом.* Объект можно рассматривать как усовершенствованную переменную: он содержит данные, но ему также можно направлять запросы на выполнение операций «с самим собой». Теоретически любой концептуальный компонент решаемой задачи (собака, здание, некоторая операция и т. д.) может быть представлен в программе в виде объекта.
- *Программа представляет собой совокупность объектов, которые общаются друг с другом посредством отправки сообщений.* Чтобы обратиться с запросом к объекту, следует отправить ему сообщение. Точнее, сообщение может рассматриваться как запрос на вызов функции, принадлежащей конкретному объекту.
- *Каждый объект владеет собственным блоком памяти, состоящим из других объектов.* Иначе говоря, новые объекты в программе создаются как «оболочки» для существующих объектов. Это позволяет усложнять программу, скрывая ее за простотой отдельного объекта.
- *Каждый объект обладает типом.* В терминологии ООП каждый объект является *экземпляром класса*, где термин «класс» является синонимом термина «тип». Важнейшая отличительная характеристика класса формулируется так: «Какие сообщения принимает данный класс?»
- *Все объекты одного типа могут получать одни и те же сообщения.* Как вы вскоре убедитесь, это утверждение не совсем корректно. Поскольку объект типа «круг» также является объектом типа «геометрическая фигура», объект-круг заведомо принимает сообщения, предназначенные для объектов-фигур. Следовательно, программа для работы с геометрическими фигурами может автоматически обрабатывать все объекты, которые являются частными случаями геометрической фигуры. Данная особенность является одной из самых сильных сторон ООП.

Интерфейс объекта

Вероятно, Аристотель был первым, кто серьезно изучал концепцию *типов*: он говорил о «классе птиц и классе рыб». Принцип, в соответствии с которым все объекты, несмотря на свою уникальность, также принадлежат к некоторому классу, обладающему и общими характеристиками и поведением, был напрямую воплощен в первом объектно-ориентированном языке Simula-67 — именно там появилось основополагающее ключевое слово `class`, которое вводило новый тип в программу.

Язык Simula, как нетрудно догадаться по его названию, создавался для имитации — например, для решения знаменитой «задачи кассира». В этой задаче задействовано множество кассиров, клиентов, счетов, операций и денежных единиц — короче говоря, множество «объектов». Объекты, различающиеся только по текущему состоянию во время выполнения программы, группируются в «классы»; отсюда и произошло ключевое слово class. Создание абстрактных типов данных (классов) является одной из основополагающих концепций объектно-ориентированного программирования. Абстрактные типы данных работают почти так же, как встроенные типы: программист определяет переменные этого типа (в терминологии ООП такие переменные называются *объектами*, или *экземплярами*) и работает с ними при помощи *сообщений*, или *запросов*. Программист отправляет запрос, а объект сам решает, как следует обработать этот запрос. Объекты, относящиеся к одному классу, обладают общими характеристиками: так, у каждого счета имеется баланс, каждый кассир может принимать деньги от клиентов и т. д. В то же время каждый объект обладает собственным состоянием: на счетах хранятся разные суммы, кассиры обладают разными именами и т. д. Таким образом, любой объект — кассир, клиент, счет, операция — представляется в программе отдельной сущностью (объектом), причем каждый объект принадлежит к конкретному классу, определяющему его характеристики и поведение.

Итак, хотя в объектно-ориентированном программировании фактически определяются новые типы данных, практически во всех объектно-ориентированных языках используется термин «класс». Если вы видите термин «тип», читайте «класс» — и наоборот[1].

Поскольку класс определяет совокупность объектов с одинаковыми характеристиками (элементы данных) и поведением (функциональность), встроенные типы данных тоже можно рассматривать как классы — скажем, вещественные числа тоже обладают набором характеристик и поведением. Различие состоит в том, что программист определяет класс в соответствии со спецификой задачи, а не использует готовый тип данных, представляющий машинную единицу хранения данных. Он расширяет язык программирования и включает в него новые типы, соответствующие его потребностям. Система программирования принимает новые классы, обеспечивает для них всю обработку и проверку типов наравне со встроенными типами.

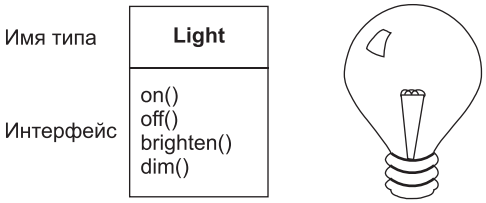
Область применения объектно-ориентированного подхода не ограничивается имитацией. Согласны вы с тем, что любая программа имитирует проектируемую систему, или нет, применение методов ООП помогает сводить трудные задачи к простым решениям.

После определения класса вы можете создать сколько угодно объектов этого класса и работать с ними как с элементами пространства решения задачи. Одна из основных трудностей объектно-ориентированного программирования как раз и заключается в создании однозначного соответствия между элементами пространства задачи и объектами в пространстве решения.

Но как заставить объект сделать нечто полезное? Необходимо передать объекту запрос на выполнение некоторой операции, например зачислить деньги на счет,

[1] Некоторые авторы различают эти два понятия. Они утверждают, что тип определяет интерфейс, а класс — конкретную реализацию этого интерфейса.

нарисовать что-нибудь на экране или повернуть выключатель. Каждый объект обрабатывает только конкретные разновидности запросов. Совокупность этих запросов образует *интерфейс*, который определяется типом объекта. Рассмотрим простой пример — электрическую лампочку:



Интерфейс определяет лишь то, *какие* запросы обрабатываются данным объектом. Где-то внутри объекта должен находиться программный код, который эти запросы обрабатывает. Этот код вместе со скрытыми данными образует *реализацию*. С позиций процедурного программирования в этой схеме нет ничего сложного. В типе с каждым возможным запросом ассоциируется некоторая функция, которая вызывается при получении запроса. При описании этого процесса обычно говорят, что программист «отправляет сообщение» («обращается с запросом») к объекту, а объект решает, что делать с этим сообщением (какой фрагмент кода выполнить).

В приведенном примере имеется тип/класс с именем `Light` и конкретный объект класса `Light` с именем `It`. К объекту `Light` можно обратиться со следующими запросами: включиться (`on()`), выключиться (`off()`), прибавить или убавить яркость (`brighten()` и `dim()`). При создании объекта класса `Light` указывается имя этого объекта (`It`). Чтобы отправить сообщение объекту, вы указываете имя объекта и отделяете его от сообщения точкой (`.`). С точки зрения пользователя готового класса, в этом и заключается все программирование с применением объектов.

Приведенная выше диаграмма оформлена по стандарту *UML* (Unified Modeling Language). Каждый класс представлен отдельным блоком; в верхней части блока находится имя класса, в средней — все переменные класса, которые вы желаете документировать, а в нижней части — *функции класса* (функции, которые принадлежат данному объекту и обрабатывают получаемые сообщения). Довольно часто на UML-диаграммах отображаются только имя и открытые функции класса, в этом случае средняя часть блока отсутствует. А если важно только имя класса, то и нижняя часть блока становится необязательной.

Скрытая реализация

Для удобства всех программистов можно условно разделить на *создателей классов* (тех, кто определяет новые типы данных) и *прикладных программистов* («потребителей», использующих типы данных в своих приложениях). Прикладные программисты собирают библиотеки классов и используют их для ускорения разработки приложения. Создатель класса стремится построить класс, который предоставляет прикладному программисту доступ лишь к тому, что необходимо для его работы, и скрывает все остальное. Почему? Скрытые аспекты класса не могут использоваться прикладными программистами, поэтому создатель класса может свободно

изменять их, не беспокоясь о том, как эти изменения отразятся на других. Скрытая часть класса обычно представляет чувствительные «внутренности» объекта, которые могут быть легко испорчены прикладным программистом по неосторожности или от недостатка информации, поэтому скрытие реализации уменьшает количество ошибок в программе. Скрытие реализации — одна из важнейших концепций ООП, ее важность трудно переоценить.

В любых отношениях важно установить границы, которые должны соблюдаться всеми участниками этих отношений. При создании библиотеки вы устанавливаете отношения с прикладным программистом, который использует вашу библиотеку в своих приложениях — возможно, для построения более крупной библиотеки.

Если все члены класса будут доступны всем желающим, то прикладной программист сможет делать с классом все что угодно и обеспечить соблюдение правил будет в принципе невозможно. Даже если прямое обращение к некоторым членам классов нежелательно, его не удастся предотвратить без ограничения доступа.

Итак, существует несколько причин для ограничения доступа. Во-первых, некоторые аспекты класса должны быть защищены от вмешательства прикладных программистов. Речь идет об аспектах, задействованных во внутренней работе типа данных, но не входящих в его интерфейс, применяемый для решения практических задач пользователей. На самом деле ограничение доступа не мешает, а помогает прикладному программисту выделить то, что действительно существенно для его работы.

Во-вторых, ограничение доступа позволяет проектировщику библиотеки изменить внутренний механизм работы класса, не беспокоясь о том, как эти изменения отразятся на прикладных программистах. Допустим, в первой версии для ускорения разработки использовалась упрощенная реализация, а потом вдруг выяснилось, что класс необходимо переписать заново для ускорения его работы. Четкое отделение интерфейса от реализации позволяет легко решить эту задачу, после чего пользователю останется лишь перекомпоновать программу.

Для определения уровней доступа в C++ используются три ключевых слова: `public`, `private` и `protected`. Их смысл весьма прямолинеен — эти *спецификаторы доступа* указывают, кто может работать с объявлениями членов класса, следующими за ними. Спецификатор `public` означает, что следующие объявления доступны для всех. С другой стороны, спецификатор `private` означает, что доступ к следующим объявлениям возможен только из функций данного типа. Фактически он разделяет «зоны ответственности» создателя класса и прикладного программиста. При попытке обратиться извне к закрытому (`private`) члену класса происходит ошибка компиляции. Спецификатор `protected` аналогичен `private` с одним отличием: производные классы могут обращаться к защищенным (`protected`) членам класса, но доступ к закрытым (`private`) членам для них запрещен. О наследовании и производных классах мы поговорим чуть позже.

Повторное использование реализации

Созданный и протестированный класс должен (в идеале) быть достаточно самостоятельным, чтобы его можно было использовать повторно в других программах. Впрочем, на практике добиться этого сложнее, чем хотелось бы; качественное проектирование классов требует опыта и хорошей интуиции. Но если такой класс

будет создан, было бы обидно не задействовать его. Возможность повторного использования кода относится к числу важнейших преимуществ объектно-ориентированного программирования.

В простейшем случае вы просто напрямую создаете объект соответствующего класса, однако существует и другой способ — включить объект этого класса внутрь нового класса. Это называется «созданием вложенного объекта». Новый класс может содержать сколько угодно объектов произвольных типов в любой комбинации, необходимой для достижения желаемого поведения нового класса. Методика создания нового класса из существующих классов называется *композицией* (также встречается более общий термин — *агрегирование*). О композиции также часто говорят как об «отношении принадлежности» по принципу «у машины есть двигатель».

(На этой UML-диаграмме композиция обозначена закрашенным ромбом, который указывает, что двигатель принадлежит только одной машине. Автор обычно изображает такие связи в упрощенном виде — только линией без ромба.)

Композиция обладает чрезвычайно гибкими возможностями. Вложенные объекты нового класса обычно объявляются закрытыми, что делает их недоступными для прикладных программистов, работающих с классом. С другой стороны, создатель класса может изменять эти объекты, не нарушая работы существующего клиентского кода. Кроме того, замена вложенных объектов на стадии выполнения программы позволяет динамически изменять ее поведение. Механизм наследования (см. ниже) такой гибкостью не обладает, поскольку для производных классов устанавливаются ограничения, проверяемые на стадии компиляции.

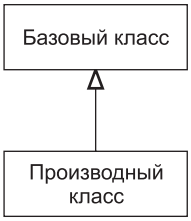
Наследование играет особую роль в объектно-ориентированном программировании. Многие авторы особо подчеркивают его важность, и у неопытных программистов может сложиться впечатление, что наследование требуется в любых ситуациях. Это часто приводит к появлению громоздких, чрезмерно усложненных архитектур. При создании новых классов сначала подумайте, нельзя ли воспользоваться более простым и гибким механизмом композиции. А когда у вас появится практический опыт, станет вполне очевидно, в каких случаях лучше подходит наследование.

Наследование и повторное использование интерфейса

Сама по себе идея объекта чрезвычайно удобна. Она позволяет объединить данные и функциональность на *концептуальном* уровне, чтобы программист работал в контексте пространства решения задачи и не ограничивался идиоматическими представлениями базового компьютера. Эти концепции выражаются на уровне базового синтаксиса языка программирования ключевым словом class.

Но было бы обидно потрудиться над созданием класса, а потом заново определять новый класс, обладающий похожими возможностями. Гораздо удобнее взять

существующий класс, скопировать его, а затем внести дополнения и изменения в копию. В сущности, именно это происходит при наследовании, но с одним исключением: изменения, вносимые в исходный класс (также называемый *базовым классом*, *родительским классом* или *суперклассом*), будут отражены и в его «копии» (называемой *производным классом*, *дочерним классом* или *субклассом*).

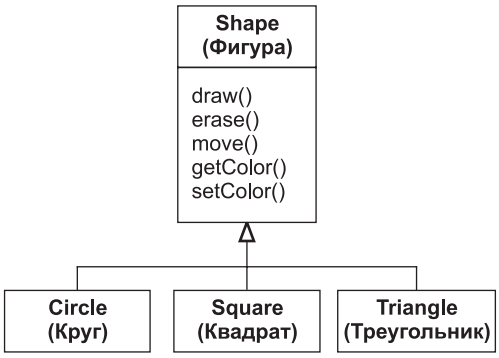


(Стрелка на приведенной UML-диаграмме направлена от производного класса к базовому. Как будет показано ниже, один базовый класс может иметь несколько производных классов.)

Тип не ограничивается определением общих ограничений для набора объектов; он также участвует в системе отношений с другими типами. Два типа могут обладать общими характеристиками и поведением, но один из них может обладать дополнительными характеристиками или обрабатывать дополнительные сообщения (или обрабатывать их иным образом). В механизме наследования сходство между типами выражается концепциями базовых и производных типов. Базовый тип имеет все характеристики и аспекты поведения всех типов, производных от него. Программист создает базовый тип для представления важнейших особенностей объектов системы. Затем от базового типа наследуются производные типы, выражающие разные варианты реализации этих важнейших особенностей.

Предположим, вы программируете мусороуборочную машину для сортировки разных типов мусора. В этом случае базовый тип «мусор» определяет общие характеристики (вес, ценность и т. д.) и общие операции (измельчение, переработка и т. д.). Из базового типа (мусора) производятся конкретные типы отходов, обладающие дополнительными характеристиками (например, цветом стекла у бутылки) или аспектами поведения (скажем, возможностью прессовки алюминиевых банок). Кроме того, некоторые аспекты производного типа могут отличаться от аспектов базового прототипа (например, ценность бумаги определяется в зависимости от ее плотности и состояния). Путем наследования программист строит иерархию типов, которая должна выражать решаемую задачу в контексте этих типов.

Рассмотрим другой, классический пример (возможно, взятый из системы автоматизированного проектирования или компьютерной игры). Базовый тип «геометрическая фигура» определяет свойства, общие для всех фигур: размер, цвет, положение и т. д. С фигурами выполняются различные операции: вывод, стирание, перемещение, закраска и т. д. Из этого базового типа производятся типы конкретных видов фигур (кругов, квадратов, треугольников и т. д.), обладающих дополнительными характеристиками и поведением. Например, к некоторым фигурам может применяться операция зеркального отражения. В иерархии типов воплощаются сходства и различия между разными фигурами.

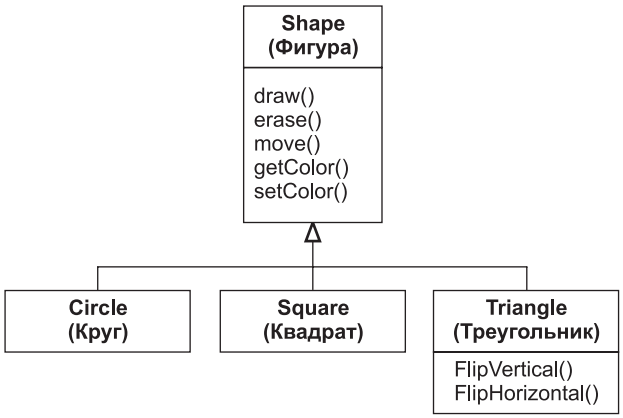


Формулировка решения в контексте задачи приносит огромную пользу, поскольку она избавляет от применения промежуточных моделей для перехода от описания задачи к описанию решения. При использовании объектов иерархия типов становится основной моделью, что позволяет перейти от описания системы в реальном мире к описанию системы на уровне программного кода. Оказывается, концептуальная простота объектно-ориентированного проектирования даже сбивает с толку некоторых программистов. Разум, привыкший к поиску сложных решений, приходит в замешательство от этой простоты.

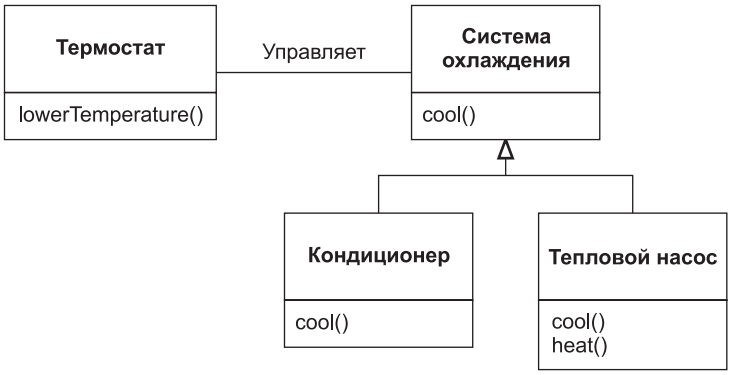
Объявляя новый тип производным от существующего типа, вы создаете новый тип. Этот новый тип не только содержит все члены существующего типа (хотя члены со спецификатором private скрыты и недоступны для производного класса), но что еще важнее — он также воспроизводит интерфейс базового класса. Иначе говоря, все сообщения, которые принимались объектами базового класса, также будут приниматься объектами производного класса. Поскольку мы знаем, что класс определяется теми сообщениями, которые он принимает, можно сказать, что *производный класс является частным случаем базового класса*. Так, в предыдущем примере это означает, что «круг является частным случаем геометрической фигуры». Эквивалентность типов на базе наследования — одно из важнейших положений, необходимых для понимания сути объектно-ориентированного программирования.

Базовый и производный классы обладают общим интерфейсом, поэтому было бы естественно предположить, что вместе с интерфейсом совместно используется некоторая реализация. При получении сообщения объект выполняет некоторый фрагмент кода. Если просто объявить класс производным и не делать ничего более, методы интерфейса базового класса перейдут в производный класс. В этом случае объекты производного класса не только являются частным случаем базового класса, но и ведут себя точно так же, что не особенно интересно.

Существуют два способа модификации производных классов по отношению к базовым. Первый способ тривиален: в производный класс просто добавляются новые функции, не входящие в интерфейс базового класса. Если базовый класс не обладает возможностями, необходимыми для некоторого частного случая, он расширяется в производном классе. Такое простое и даже примитивное применение наследования часто оказывается идеальным решением проблемы, но при расширении следует хорошо подумать, не понадобятся ли эти дополнительные функции в базовом классе. При объектно-ориентированном программировании рекомендуется периодически производить итеративный анализ и переработку структур классов.



Хотя при наследовании интерфейсы часто дополняются новыми функциями, это не обязательно. Второй, более важный, способ модификации производных классов основан на *переопределении*, то есть *изменении поведения* существующих функций базового класса.



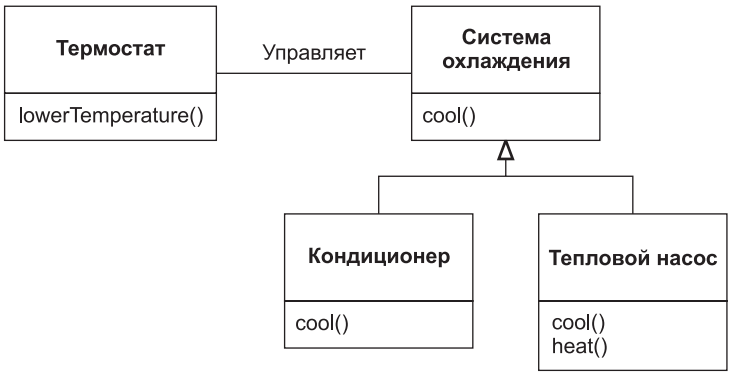
Чтобы переопределить функцию, достаточно создать в производном классе новое определение функции. В сущности, это означает: «Мы используем ту же интерфейсную функцию, но в производном типе она будет делать нечто иное».

Точное и приблизительное подобие

В отношении наследования возникает один важный вопрос: должно ли наследование ограничиваться переопределением функций базового класса (без добавления новых функций, отсутствующих в базовом классе)? В этом случае производный тип в *точности* эквивалентен базовому классу, поскольку он обладает точно таким же интерфейсом. Следовательно, объекты производного класса могут свободно использоваться вместо объектов базового класса. Иногда это называется *чистой подстановкой*, или *принципом подстановки*. В каком-то смысле такой подход к наследованию может считаться идеальным. Базовый и производный классы связаны *отношением точного подобия*, когда вы можете сказать, что круг *является* геометрической фигурой. На этом факте основан один из способов проверки иерархии

наследования: попробуйте сформулировать отношение точного подобия между классами и посмотрите, насколько осмысленно оно выглядит.

Иногда в производные типы приходится добавлять новые элементы интерфейса, что приводит к расширению интерфейса и созданию нового типа. Новый тип по-прежнему может использоваться вместо базового, однако эта подстановка не идеальна, потому что новые функции остаются недоступными для базового типа. Такая ситуация описывается как *отношение приблизительного подобия*; новый тип поддерживает интерфейс старого типа, но он также содержит другие функции, поэтому говорить о точном совпадении интерфейсов не приходится. Представьте, что в вашем доме установлена аппаратура (интерфейс) для управления охлаждающими системами. Допустим, кондиционер сломался и был заменен тепловым насосом, который может как охлаждать, так и подогревать воздух. Тепловой насос *является аналогом* кондиционера, но он способен на большее. Поскольку аппаратура рассчитана только на охлаждение, она сможет взаимодействовать только с подсистемой охлаждения нового объекта. Интерфейс объекта расширился, но существующая система «знает» только об исходном интерфейсе.



Конечно, при первом взгляде на эту иерархию становится ясно, что базовый класс «система охлаждения» ограничен и его следовало бы заменить «системой терморегулирования» с поддержкой подогрева, — в этом случае принцип подстановки заработает нормально. Тем не менее приведенная выше диаграмма дает пример того, что нередко происходит при проектировании иерархий классов и в реальном мире.

При первом знакомстве с принципом подстановки часто кажется, что этот подход (чистая подстановка) является единственно верным. Если вам удастся ограничиться этим способом — что ж, превосходно. Но на практике включение новых функций в интерфейс производного класса часто оказывается вынужденной мерой. В результате тщательного анализа оба случая выявляются достаточно очевидно.

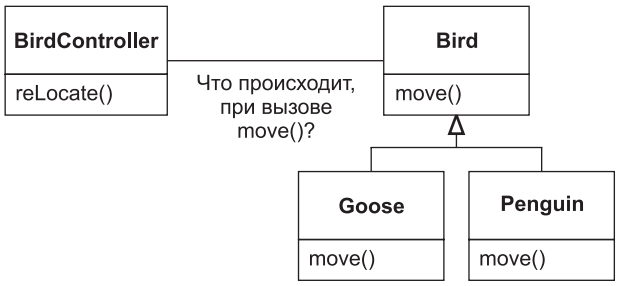
Взаимозаменяемость объектов и полиморфизм

При использовании иерархий типов часто требуется интерпретировать объект не с фактическим, а с базовым типом. Это позволяет программировать независимо от конкретных типов. Так, в примере с геометрическими фигурами функции работа-

ют с обобщенными фигурами независимо от того, являются ли они кругами, квадратами, треугольниками и т. д. Нарисовать, стереть или переместить можно любую фигуру, поэтому эти функции просто передают соответствующее сообщение объекту геометрической фигуры; они не заботятся о том, как объект обработает это сообщение.

Такой код не зависит от введения новых типов — самого распространенного способа расширения объектно-ориентированных программ для обработки новых ситуаций. Например, определение новой разновидности геометрической фигуры (пятиугольник) не повлияет на работу функций, работающих с обобщенными фигурами. Возможность простого расширения программ посредством объявления новых производных типов играет важную роль, поскольку она существенно улучшает архитектуру программы с одновременным сокращением затрат на ее сопровождение.

Тем не менее интерпретация объектов производных типов в контексте базового типа (круг как геометрическая фигура, велосипед как средство передвижения, ворона как птица и т. д.) не обходится без проблем. Если функция должна приказать обобщенной геометрической фигуре нарисовать себя, обобщенному средству передвижения — повернуть, обобщенной птице — лететь и т. д., на стадии компиляции еще неизвестно, какой фрагмент кода должен при этом выполняться. Собственно, в этом и заключается вся соль — при отправке сообщения программист *не желает* знать, какой фрагмент кода будет выполняться. Функция вывода может применяться к кругам, квадратам и треугольникам, а объект должен выполнить правильный код в зависимости от фактического типа. Если вам не нужно знать, какой именно фрагмент кода будет выполняться, то при добавлении нового подтипа выполняемый код будет выбран без изменения вызова функции. Итак, если компилятор не может заранее узнать, какой фрагмент кода будет выполняться, что же ему делать? Например, на следующей диаграмме объект `BirdController` работает с обобщенными объектами `Bird` (птица) и не знает их точного типа. С точки зрения `BirdController` это удобно, поскольку этому классу не нужно определять конкретный подтип `Bird`, с которым он работает, или проверять поведение этого подтипа. Так как же при вызове функции `move()` без определения конкретного типа `Bird` выбирается правильное поведение? Ведь гусь (`Goose`) летает, плавает и бегает, а пингвин (`Penguin`) только плавает и бегает.



Ответ на этот вопрос кроется в одном из трюков объектно-ориентированного программирования. Компилятор не может сгенерировать вызов функции в традиционном смысле. В вызовах функций, сгенерированных обычным компилятором,

применяется так называемое *раннее связывание*. Возможно, многим читателям этот термин еще не встречался, потому что они и не подозревали о существовании другой схемы связывания. Термин означает, что компилятор генерирует вызов функции с конкретным именем, а компоновщик преобразует этот вызов в абсолютный адрес выполняемой функции. В ООП программа не может определить адрес функции до выполнения программы, поэтому при отправке сообщения обобщенному объекту необходима другая схема.

Для решения этой проблемы в объектно-ориентированных языках применяется *позднее*, или *динамическое*, *связывание*. При отправке сообщения объекту функция, которой должно быть передано управление, не определяется до времени выполнения программы. Компилятор всего лишь убеждается в том, что функция существует, проверяет типы аргументов и возвращаемого значения (языки, в которых эта проверка не выполняется, называются *языками со слабой типизацией*), но не знает, куда именно будет передано управление.

Для выполнения позднего связывания компилятор C++ заменяет абсолютный вызов функции специальным фрагментом кода, который вычисляет адрес тела функции по информации, хранящейся в объекте (эта тема подробно рассматривается в главе 15). Таким образом, разные объекты по-разному ведут себя в зависимости от этого фрагмента. При получении сообщения объект сам решает, как его обработать.

Виртуальные функции (то есть функции, наделенные гибкими возможностями позднего связывания) объявляются с ключевым словом `virtual`. Чтобы использовать виртуальные функции в программах, не обязательно понимать механику их вызова, но без виртуальных функций объектно-ориентированное программирование на C++ невозможно. В C++ необходимо помнить о добавлении ключевого слова `virtual`, потому что по умолчанию функции классов *не используют* динамическое связывание. В виртуальных функциях проявляются различия в поведении классов, принадлежащих к одному семейству. Эти различия заложены в основу полиморфизма.

Вернемся к примеру с фигурами. Диаграмма с семейством классов, использующих единый интерфейс, была приведена выше в этой главе. Для демонстрации полиморфизма мы напишем фрагмент кода, который будет игнорировать особенности конкретных типов и взаимодействовать только с базовым классом. Такой код изолируется от информации конкретных типов, он проще в программировании и сопровождении. А если в иерархию будет добавлен новый производный тип, например многоугольник (`Hexagon`), то этот фрагмент будет работать с новой разновидностью типа `Shape` так же, как он работал с существующими типами. Тем самым обеспечивается *расширяемость* программы.

На C++ эта функция выглядит примерно так:

```
void doStuff(Shape& s) {
  s.erase();
  // ...
  s.draw();
}
```

Функция работает с любыми объектами типа `Shape` и не зависит от конкретного типа выводимых и стираемых объектов (суффикс `&` означает «Получить адрес объекта, передаваемого функции `doStuff()`», но вам пока не обязательно разбираться в этих тонкостях). Вызов функции `doStuff()` в другой части программы выглядит так:

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

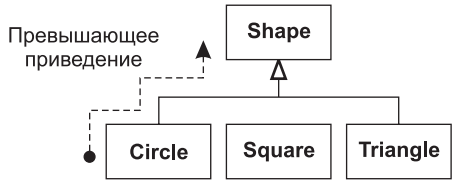
Вызовы doStuff() автоматически работают правильно независимо от фактического типа объекта.

На самом деле это очень интересный факт. Рассмотрим следующую строку:

```
doStuff(c);
```

Здесь происходит следующее: функции, которая ожидает получить объект типа Shape, передается объект типа Circle. Поскольку класс Circle является частным случаем Shape, он может интерпретироваться как таковой функцией doStuff(). Иначе говоря, любое сообщение, которое функция doStuff() может отправить объекту Shape, будет успешно принято также и объектом Circle. Все предельно безопасно и логично.

Интерпретация объекта производного типа так, как если бы он относился к базовому типу, называется *повышающим приведением типа*. Термин «повышающее» происходит от типичной структуры диаграмм наследования, при которой базовый тип расположен сверху, а производные классы выстроены внизу. Таким образом, приведение к базовому типу соответствует перемещению вверх на диаграмме наследования.



В объектно-ориентированных программах обычно в том или ином виде задействовано повышающее приведение типа, потому что оно избавляет программиста от необходимости знать конкретный тип объектов, с которыми он работает. Вспомните функцию doStuff():

```
s.erase();
// ...
s.draw();
```

Обратите внимание: мы не говорим «для объекта Circle сделать то, для объекта Square — сделать это». Программа, анализирующая все возможные разновидности Shape, получится слишком громоздкой, вдобавок ее придется изменять при каждом добавлении новой разновидности Shape. В данном случае мы говорим: «Фигура, я знаю, что ты умеешь самостоятельно обрабатывать вызовы erase() и draw(); сделай это и позаботься обо всех деталях».

Интересно, что код функции doStuff() каким-то образом делает именно то, что требуется. При вызове draw() для Circle будет выполнен совсем не тот код, который выполняется при вызове draw() для Square или Line, а при отправке сообщения draw() неизвестной разновидности Shape правильное поведение выбирается автоматически в соответствии с фактическим типом объекта. И это выглядит совершенно

удивительно, потому что, как упоминалось выше, при компиляции функции doStuff() компилятор не обладает информацией о типах, с которыми он работает. Следовательно, в обычной ситуации можно было бы предположить, что будут вызваны версии функций erase() и draw() класса Shape, а не конкретных подтипов Circle, Square и Line. Тем не менее благодаря полиморфизму все работает правильно. Обо всех деталях позаботятся компилятор и система времени выполнения. Программист должен знать лишь то, что этот механизм работает, и уметь применять его при проектировании. Если функция класса объявлена виртуальной, то при получении сообщения объект автоматически выполнит правильные действия, даже если при этом потребуется повышающее приведение типа.

Создание и уничтожение объектов

С технической точки зрения основными проявлениями ООП считаются абстрактные типы данных, наследование и полиморфизм, но существуют и другие, не менее важные аспекты. В этом разделе приводится их краткий обзор.

Особого внимания заслуживают операции создания и уничтожения объектов. Где хранятся данные объекта и как определяется продолжительность его жизненного цикла? В разных языках программирования используются разные решения. В подходе, принятом в C++, важнейшим фактором считается контроль над эффективностью, поэтому программисту предоставляется выбор. Если он стремится добиться максимальной скорости выполнения, то место хранения и продолжительность жизненного цикла объектов определяются на стадии написания программы; для этого объекты размещаются в стеке или в статической памяти. *Стеком* называется область памяти, которая напрямую используется процессором во время выполнения программы. Переменные, хранящиеся в стеке, также называются *автоматическими* переменными. *Статической памятью* называется фиксированный блок памяти, выделяемый перед началом выполнения программы. При хранении данных в стеке или статической памяти на первое место ставится скорость выделения и освобождения памяти, очень существенная в некоторых ситуациях. Однако за нее приходится расплачиваться гибкостью, так как программист должен точно знать количество, жизненный цикл и тип объектов *во время* написания программы. Если задача плохо поддается прогнозированию (системы управления складом, авиадиспетчерские системы и т. д.), это приводит к чрезмерным ограничениям.

Во втором подходе объекты создаются динамически в специальном пуле памяти, называемом *кучей*. В этом случае программист до момента выполнения программы не знает, сколько объектов потребуется, каким будет их жизненный цикл или фактический тип. Эти решения принимаются «на месте» во время выполнения программы. Если программисту понадобится новый объект, он просто создает его в куче при помощи ключевого слова new. Когда объект становится ненужным, он ликвидируется при помощи ключевого слова delete.

Поскольку операции с кучей производятся динамически на стадии выполнения, выделение памяти занимает гораздо больше времени, чем при создании объектов в стеке (выделение памяти в стеке часто сводится к одной команде процессора для смещения указателя стека). Динамический подход основан на разумном допущении, что для достаточно сложных объектов дополнительные затраты на по-

иск области памяти и ее освобождение не оказывают заметного влияния на общую продолжительность операции. Кроме того, более гибкий характер динамического выделения играет важную роль при решении общих задач программирования.

Однако наряду с этими факторами существует и другой — продолжительность жизни объекта. Если объект создается в стеке или в статической памяти, продолжительность его существования определяет компилятор, автоматически уничтожая объект. Но если объект создается в куче, компилятор не располагает информацией о сроке его существования. В C++ программист должен самостоятельно уничтожить объект в своей программе при помощи ключевого слова delete. Впрочем, существует и другой вариант — в среде выполнения может поддерживаться механизм *уборки мусора*, который автоматически обнаруживает неиспользуемые объекты и уничтожает их. Конечно, уборка мусора существенно упрощает программирование, но, с другой стороны, она приводит к дополнительным затратам ресурсов. Этот факт не соответствовал требованиям, принятым при проектировании C++, поэтому механизм уборки мусора не стал частью языка (хотя для C++ существуют уборщики мусора, разработанные сторонними производителями).

Обработка исключений

Со времен появления первых языков программирования обработка ошибок вызывала массу проблем. Разработать хорошую схему обработки ошибок чрезвычайно трудно, поэтому многие языки просто игнорировали эту проблему и перепоручали ее проектировщикам библиотек; в свою очередь, те предлагали компромиссные решения, которые часто работали, но во многих ситуациях легко обходились (как правило, их можно было просто игнорировать). Главным недостатком большинства схем обработки ошибок была их зависимость от того, насколько прилежно программистом выполнялись определенные правила, соблюдение которых не обеспечивалось на уровне языка. Если программист не проявлял должного усердия (как это часто бывает в спешке), все эти схемы становились бесполезными.

Механизм *обработки исключений* переводит обработку ошибок на уровень языка программирования, а иногда даже на уровень операционной системы. Исключение представляет собой объект, который «генерируется» в месте ошибки и «перехватывается» соответствующим *обработчиком исключения*, предназначенным специально для обработки этого конкретного типа ошибки. Со стороны все выглядит так, словно при возникновении ошибки выполнение программы идет по другой, параллельной ветви. А это означает, что код обработки ошибок может быть отделен от кода нормального выполнения. Подобная изоляция упрощает программу, поскольку программисту не приходится постоянно проверять все возможные ошибки. Кроме того, сгенерированное исключение принципиально отличается от возвращаемого численного кода или флага, устанавливаемого функцией для обозначения ошибки, — эти признаки могут просто игнорироваться. Исключения не игнорируются, поэтому рано или поздно они заведомо будут обработаны. Наконец, исключения обеспечивают восстановление после аварийных ситуаций. Вместо завершения программы нередко удается исправить ошибку и продолжить работу, что значительно повышает устойчивость системы.

Учтите, что обработка исключений не является объектно-ориентированным средством, хотя в объектно-ориентированных языках исключения обычно представляются объектами. Механизм обработки исключений появился раньше первых объектно-ориентированных языков.

Анализ и проектирование

Объектно-ориентированная парадигма требует принципиально нового подхода к программированию, и многие программисты поначалу даже не представляют, с чего начать работу над ООП-проектом. Свыкнувшись с мыслью, что все данные должны быть объектами, и научившись мыслить в объектно-ориентированном ключе, вы постепенно начнете создавать «хорошие» архитектуры, в полной мере использующие все преимущества ООП.

Методологией называется совокупность процессов и эвристических подходов, применяемых для упрощения сложных задач программирования. С момента появления объектно-ориентированного программирования было предложено немало разных методологий. В этом разделе вы получите представление о том, чего же мы пытаемся добиться при использовании методологий в своей работе.

В области методологии (и ООП в частности) проводится множество экспериментов, поэтому вы должны хорошо понять, какие проблемы призван решить тот или иной метод, прежде чем переходить на него. Это особенно справедливо по отношению к C++ — языку, который должен был упростить выражение программных концепций по сравнению с C. Иногда даже можно обойтись без более сложных методологий. Простыми средствами в C++ решается гораздо более широкий круг задач, чем с помощью простых методологий в процедурных языках.

Также следует понимать, что термин «методология» нередко оказывается слишком пышным и обещает слишком много. Все, что вы делаете во время проектирования и написания программы, является методом. Возможно, вы сами изобрели этот метод и даже не отдаете себе отчета в его применении, однако это и есть тот самый процесс, через который вы проходите в процессе творения. Если процесс получился эффективным, возможно, для его воплощения на C++ достаточно минимальной доработки. Если вас не устраивает производительность вашего труда и то, какими получаются ваши программы, подумайте о переходе на другую формальную методику или поищите нужные составляющие из нескольких формальных методов.

В процессе разработки самое важное правило звучит так: «Не заблудитесь». Такое происходит очень часто. Многие методы из области аналитики и проектирования ориентированы на решение глобальных задач. Помните, что большинство проектов не относится к этой категории, поэтому обычно успешный анализ и проектирование удается провести на базе небольшого подмножества того, что рекомендует некоторый метод. И все же некий упорядоченный подход, сколь бы ограниченным он ни был, обычно помогает взяться за дело гораздо лучше, чем если вы просто начнете программировать.

Другая опасность — «аналитический паралич», когда программист считает, что двигаться дальше можно только после досконального анализа всех деталей на текущей стадии. Помните: даже при самом тщательном анализе в системе непременно отыщутся какие-нибудь факторы, которые проявятся лишь на стадии проектирования. Еще больше непредвиденных факторов обнаружится во время програм-

мирования, а то и тогда, когда программа будет запущена в эксплуатацию. Из-за этого желательно побыстрее пройти стадии анализа и проектирования и перейти к тестированию предполагаемой системы.

Последнее обстоятельство стоит подчеркнуть особо. Опыт использования процедурных языков подсказывает, что тщательное планирование и понимание мельчайших деталей перед выходом на проектирование и реализацию весьма похвальны. Конечно, при создании СУБД необходимо добросовестно проанализировать все потребности клиента. Однако СУБД принадлежит к классу хорошо формализуемых и понятных задач; во многих программах такого рода основной проблемой является *выбор структуры* базы данных. В этой главе речь идет о совершенно ином круге задач, решение которых не может быть получено простой переработкой хорошо известного решения, так как содержит непредсказуемые факторы — элементы, не имеющие очевидных готовых решений и требующие исследований[1]. Попытки всестороннего анализа перед переходом к проектированию и реализации в задачах такого рода вызывают «аналитический паралич», потому что на стадии анализа вы не располагаете достаточной информацией. Решение таких задач требует итераций по всему рабочему циклу, а это связано с определенными рисками (впрочем, это логично — вы пытаетесь сделать нечто новое, поэтому потенциальный выигрыш тоже увеличивается). Может показаться, что ускоренный переход к предварительной реализации лишь увеличивает этот риск, но на самом деле происходит обратное: риск уменьшается, потому что вы на ранней стадии узнаете о неприемлемости тех или иных подходов. Разработка программных продуктов требует грамотного принятия решений в условиях риска.

Часто предлагается «создать первый образец, выкинуть его и начать заново». ООП позволяет выкинуть *часть* первого образца, но поскольку код инкапсулируется в классах, даже на первой итерации вы неизбежно создадите некоторые полезные структуры классов и концепции системной архитектуры, которые стоит оставить. Таким образом, первый, ускоренный «заход на цель» не только предоставляет бесценную информацию для последующего анализа, проектирования и реализации, но и закладывает кодовую базу для дальнейших итераций.

Если методология, с которой вы знакомитесь, содержит огромное количество деталей, состоит из множества этапов и поясняется массой документов, вам все равно будет трудно понять, в какой момент следует остановиться. Помните, что вы ищете ответы всего на два вопроса.

- Какие объекты будут задействованы в проекте? (Как разбивать проект на составляющие?)
- Какими интерфейсами они будут обладать? (Какие сообщения должны отправляться каждому объекту?)

Даже если у вас не будет ничего, кроме представлений об объектах и их интерфейсах, можно переходить к написанию программы. По различным причинам вам могут потребоваться другие описания и документы, но меньшим обойтись невозможно.

---

[1] Эмпирическое правило оценки проектов такого рода, придуманное автором, гласит: если проект содержит более одного непредсказуемого фактора, даже не пытайтесь планировать затраты или сроки завершения проекта до создания рабочего прототипа. Слишком большое количество возможных вариантов развития событий все равно не позволит этого сделать.

Процесс условно делится на пять фаз с дополнительной нулевой фазой, на которой просто принимается решение об использовании той или иной структуры.

Фаза 0 — составление плана

Сначала следует решить, из каких этапов будет состоять процесс. На первый взгляд звучит просто (собственно, *все* формулировки такого рода будут звучать просто), но нередко подобные решения принимаются уже после начала программирования. Если вы руководствуетесь планом «Поскорее взяться за программирование, а дальше ориентироваться по обстановке» — что ж, замечательно (при решении очевидных проблем иногда достаточно и этого). По крайней мере, осознайте, что ваш план звучит именно так.

На данном этапе также можно решить, что процесс нуждается в дополнительном структурировании, но увлекаться тоже не стоит. Вполне естественно, что некоторые программисты любят работать в «отпускном режиме», когда процесс разработки не подчиняется никаким структурным ограничениям: «Когда сделаем, тогда и сделаем». Но на собственном опыте автор убедился, что наличие промежуточных контрольных точек помогает сосредоточиться и направляет усилия программиста к этим контрольным точкам, не позволяя ему зациклиться на одной цели — «завершении проекта». Кроме того, проект делится на части, с которыми удобнее работать и которые выглядят не столь устрашающе (не говоря уже о том, что прохождение контрольной точки является хорошим поводом для застолья).

Когда автор решил изучить структуру литературного произведения (чтобы когда-нибудь написать роман), поначалу не воспринималась сама идея структурирования — казалось, нужно просто переносить на страницу те мысли, которые рождаются в голове. Позднее стало ясно, что по крайней мере в книгах по программированию структура достаточно очевидна и о ней можно не думать. Тем не менее свои работы автор все равно структурирует, пусть даже неформально и полусознательно. Следовательно, даже если ваш план сводится к тому, чтобы поскорее взяться за программирование, все равно стоит пройти следующие фазы, задать и ответить на некоторые вопросы.

Формулировка задачи

Любая система, над которой вы работаете, независимо от сложности имеет некоторую фундаментальную цель — ту основную потребность, которую она должна удовлетворять. Если заглянуть за пользовательский интерфейс, за аппаратно- или системно-зависимые детали, за алгоритмы и проблемы эффективности, вы в итоге придете к истинной сути задачи, простой и прямолинейной. Она, как и «сверхконцепции» голливудских фильмов, формулируется в одной-двух фразах. Именно это «отфильтрованное» описание должно стать отправной точкой для решения задачи.

Сверхконцепция, то есть формулировка задачи, весьма важна, потому что она задает тон для всего проекта. Не всегда удается правильно сформулировать задачу с первого раза (возможно, она станет окончательно ясной лишь на более поздней стадии проекта), но, по крайней мере, продолжайте попытки до тех пор, пока результат вас не удовлетворит. Например, в системе управления воздушным движением можно начать с формулировки, ориентированной на конкретную систему, над которой вы работаете: «Диспетчерская вышка отслеживает перемещения самолетов». Но подумайте, что произойдет, если уменьшить систему до очень маленького аэропорта,

где никакой вышки может и не быть (а то и диспетчера). Более полезная модель ориентирована не столько на конкретные условия, сколько на суть задачи: «Самолеты прибывают, разгружаются, проходят техобслуживание, загружаются и улетают».

Фаза 1 — что делать

В предыдущем поколении программных архитектур (называемых *процедурными архитектурами*) данная фаза называлась «*анализом требований и разработкой системных спецификаций*». В результате на свет появлялись документы с устрашающими названиями, каждый из которых вполне годился для отдельного крупного проекта. Конечно, в таких дебрях было легко заблудиться, хотя сами по себе эти подзадачи были весьма полезными. Под «анализом требований» понималось следующее: «Составить список условий, по которому мы узнаем, когда работа завершена и ее нужно передавать заказчику». Под разработкой системных спецификаций понимается: «Описание того, *что* (но не *как*) должна делать наша программа, чтобы удовлетворять поставленным требованиям». Фактически анализ требований определяет контракт между вами и заказчиком (даже если заказчик работает внутри вашей компании либо является каким-то другим объектом или системой). Системная спецификация представляет собой высокоуровневое исследование проблемы, в ходе которого вы определяете, можно ли решить поставленную задачу и сколько времени это займет. Поскольку обе подзадачи должны быть согласованы между разными людьми (а также из-за того, что они обычно изменяются со временем), автор считает, что в целях экономии времени их следует сократить до абсолютного минимума (в идеальном случае — до списка с несложными диаграммами). Возможно, в вашей ситуации действуют другие ограничения, из-за которых объем документов придется увеличить, но небольшие и лаконичные исходные документы могут быть созданы за несколько сеансов «мозгового штурма» с руководителем. Такая процедура не только требует вклада со стороны всех участников, но и способствует единству взглядов и взаимопониманию в группе на первоначальной стадии проекта. Существует еще одно, возможно, более важное обстоятельство: группа возьмется за работу с большим энтузиазмом.

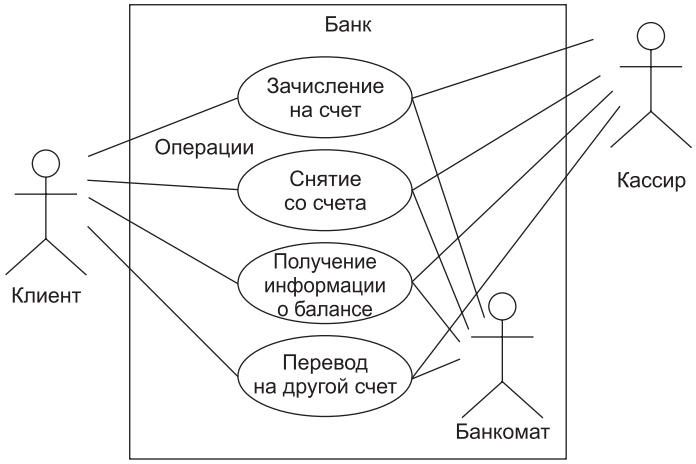
В этой фазе необходимо сосредоточиться на сути решаемой проблемы: определении того, что должна делать система. Самым ценным инструментом для этой цели является *набор функциональных требований*. Функциональные требования определяют ключевые особенности системы и дают представление о некоторых основных классах, которые будут использоваться в вашей программе. В сущности, набор функциональных требований состоит из содержательных ответов на некоторые вопросы[1].

- Кто будет использовать эту систему?
- Какие операции смогут выполнять операторы?
- Как оператор будет выполнять ту или иную операцию?
- Какие еще возможны варианты, если та же операция будет выполняться кем-то другим или у того же оператора появится другая цель (анализ вариантов)?
- Какие проблемы могут возникнуть при выполнении операции с системой (анализ исключений)?

[1] Спасибо за помощь Джеймсу Джаррету (James H. Jarrett).

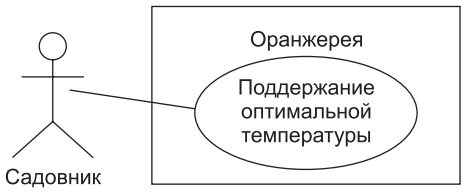
Например, если вы проектируете банкомат, то функциональные требования будут описывать действия банкомата в любой возможной ситуации. Каждая из таких ситуаций называется *сценарием*. Сценарий можно рассматривать как вопрос, начинающийся со слов: «Как поступит система, если...?» Например: «Как поступит банкомат, если клиент только что сделал вклад с 24-часовым оформлением, а без зачисления чека на счету не хватает средств для снятия желаемой суммы?»

Диаграммы функциональных требований должны быть как можно проще, чтобы вы раньше времени не увязли в подробностях реализации:



Фигурки на этом рисунке изображают «операторов» — обычно это люди или какие-либо независимые агенты (например, другая компьютерная система, как в случае с банкоматом). Прямоугольник изображает границы системы, а овалы — описания отдельных операций, выполняемых системой. Линии, соединяющие операторов с операциями, изображают взаимодействия.

В данном случае важна не конкретная реализация системы, а ее структура с точки зрения пользователя, поэтому даже для сложной системы функциональные требования не обязаны быть излишне сложными. Пример:



Функциональные требования определяются всеми взаимодействиями пользователя с системой. В результате выработки полного набора функциональных требований у вас появляется описание того, как должна работать система на самом общем уровне. Основным достоинством наборов функциональных требований является то, что они всегда ориентируются на суть происходящего и не дают отвлекаться на вопросы, не имеющие прямого отношения к решаемой задаче. Иначе говоря, если у вас появился полный набор функциональных требований, вы може-

те описать свою систему и перейти на следующую фазу. Возможно, вам не удастся правильно определить эти требования с первой попытки — ничего страшного. Все прояснится со временем, а любые попытки создания идеальной системной спецификации на этой стадии обычно заводят в тупик.

Если первые попытки завершатся неудачей, попробуйте воспользоваться методом грубого приближения: опишите систему в нескольких абзацах, а затем выделите в этом описании существительные и глаголы. Существительные могут определять операторов, контекст или место действия, а также объекты, с которыми выполняются операции. Глаголы обычно подразумевают взаимодействие между операторами и операциями и определяют этапы внутри функциональных требований. Помните, что глаголы и существительные часто соответствуют объектам и сообщениям на стадии проектирования (а также что функциональные требования описывают взаимодействие между подсистемами, а методика «существительных и глаголов» может использоваться только как вспомогательный инструмент «мозгового штурма» и ее результаты не преобразуются напрямую в функциональные требования).

Хотя весь этот процесс больше напоминает шаманство, нежели научный метод, на какой-то стадии важно провести начальное планирование. Вы уже имеете некоторое представление о системе и можете хотя бы приблизительно оценить, сколько времени займет ее разработка. При этом приходится учитывать множество факторов. Если срок окажется слишком длинным, ваша компания может отказаться от разработки системы и потратить свои ресурсы на что-нибудь более разумное. А может, ваш начальник уже решил, сколько времени должна занять работа над проектом, и попытается повлиять на вашу оценку. Но лучше с самого начала составить честное расписание и принять все ответственные решения. Известно много способов планирования (как и способов прогнозирования биржевых котировок), и все же лучше всего положиться в этом вопросе на собственный опыт и интуицию. Прикиньте, сколько времени потребует ваш проект, затем удвойте это число и прибавьте еще 10 %. Скорее всего, ваше внутреннее ощущение вас не подводит; вам действительно *удастся* получить нечто работоспособное за это время. Еще столько же времени уйдет на то, чтобы привести это «нечто» к более или менее приличному виду, а последние 10 % уйдут на шлифовку и всякие мелочи[1]. Однако все это еще нужно объяснить руководству, и несмотря на все стоны и попытки давления, которыми сопровождаются такие объяснения, дело обстоит именно так.

Фаза 2 — как делать

В этой фазе создается архитектура, описывающая классы и их взаимодействия. Прекрасная методика определения классов и их взаимодействий основана на использовании карточек CRC (Class-Responsibility-Collaboration — Класс-Ответственность-Взаимодействие»). Ценность данного инструмента отчасти обусловлена его предельной простотой. Вам понадобятся лишь пустые карточки 8 × 12 см. Каждая карточка представляет отдельный класс. На карточках записывается следующая информация.

[1] В последнее время мнение автора по этому вопросу изменилось. Удвоение и прибавление 10 % дает довольно точную оценку (при небольшом количестве непредсказуемых факторов), но чтобы уложиться в это время, необходимо прилежно работать. Если вы захотите сделать программу элегантной и получить удовольствие от своей работы, лучше умножать на 3 или на 4.

- Имя класса. Имя должно отражать сущность класса и быть понятным с первого взгляда.
- Ответственность класса — то, что он должен делать. Обычно задается простым перечислением имен функций классов (поскольку при хорошем проектировании эти имена должны быть содержательными), однако может содержать и другую информацию. Если понадобится с чего-то начать, взгляните на проблему с точки зрения ленивого программиста: какие объекты должны появиться, чтобы ваша проблема сама собой решилась?
- Взаимодействия класса — с какими другими классами должен взаимодействовать данный класс? Термин «взаимодействие» намеренно сделан предельно широким: под ним могут пониматься агрегирование или просто существование другого объекта, который выполняет работу по требованию объекта данного класса. Во взаимодействиях также должен учитываться круг пользователей класса. Например, если вы создаете класс Firecracker (фейерверк), какой класс будет им пользоваться: Chemist (химик) или Spectator (зритель)? Первый должен знать, какие реактивы требуются для составления смеси, а второй будет реагировать на яркие краски и узоры после взрыва.

Возможно, вам кажется, что карточки должны быть больше, чтобы на них поместилась вся нужная информация. Однако небольшие размеры выбраны намеренно — не только для того, чтобы классы были более компактными, но и чтобы предотвратить чрезмерную детализацию на слишком ранней стадии. Если все, что необходимо знать о классе, не помещается на маленькой карточке, значит, этот класс слишком сложен (вы либо увлеклись детализацией, либо класс нужно разбить на несколько классов). Идеальный класс должен быть понятен с первого взгляда. Карточки помогают выработать начальное представление об архитектуре системы, чтобы вы могли представить себе «общую картину», а уже потом занимались уточнением проекта.

Одно из самых выдающихся достоинств карточек проявляется в общении между людьми. Такое общение желательно проводить «в реальном времени» в группах и без компьютеров. Каждый участник отвечает за несколько классов, которые поначалу не имеют имен или иной информации. Группа имитирует работу системы, последовательно рассматривая разные сценарии, и решает, какие сообщения должны пересылаться различным объектам для выполнения текущего сценария. По ходу дела выявляются необходимые классы, их ответственности и взаимодействия, а карточки постепенно заполняются. После перебора всех сценариев у вас появится относительно полный набросок архитектуры системы.

Прежде чем автор начал пользоваться карточками, наибольшего успеха при разработке предварительной архитектуры удавалось добиваться рисованием объектов фломастером на белой доске перед группами, которые еще не создали ни одного ООП-проекта. Таким образом обсуждались взаимодействия между объектами, некоторые объекты стирались и заменялись другими. В сущности, автору просто приходилось рисовать все «карточки» на доске. Группа (которая хорошо знала, что должен делать проект) реально создавала архитектуру проекта, а не получала ее откуда-то сверху. Автору оставалось лишь направлять этот процесс: задавать нужные вопросы, выдвигать предположения и изменять их на основании мнения участников. Истинная красота процесса состояла в том, что группа училась объект-

но-ориентированному проектированию не на абстрактных примерах, а на примере той архитектуры, которая больше всего интересовала ее в тот момент.

После того как карточки будут заполнены, можно подумать о создании более формального описания архитектуры на языке UML. Использовать UML не обязательно, но это может быть полезно, особенно если вы хотите вывесить диаграмму на стену для всеобщего обозрения. Альтернативой UML может стать текстовое описание объектов и их интерфейсов или, в зависимости от вашего языка программирования, — программный код[1].

UML также обладает дополнительными средствами построения диаграмм для описания динамической модели системы. Диаграммы удобны при доминирующей роли переходов между состояниями системы или подсистемы (например, в системах управления). Возможно, вам также потребуется описать структуры данных в тех системах или подсистемах, в которых доминирующая роль отводится данным (например, при работе с базами данных).

Признаком завершения фазы 2 является законченное описание объектов и их интерфейсов... или по крайней мере их большей части — некоторые из них остаются неучтенными и выявляются только во время фазы 3. Это вполне нормальное явление. Важно лишь, чтобы все объекты были рано или поздно выявлены. Хорошо, когда они обнаруживаются на ранней стадии процесса, но ООП обладает достаточной структурной устойчивостью, так что более позднее обнаружение не принесет особого вреда. Процесс проектирования объектов можно разделить на пять стадий.

Пять стадий проектирования объектов

Проектирование объекта не ограничивается непосредственным написанием программы. В действительности процесс проектирования объектов состоит из нескольких последовательных стадий. Рассматривая его с этой точки зрения, вы не будете ждать, что вам удастся немедленно достичь совершенства. Достаточно осознавать, что понимание структуры объекта и его работы приходит со временем.

1. *Выявление объектов*. Эта стадия происходит во время первоначального анализа программы. Выявление объектов основано на анализе внешних факторов и ограничений, дублирования элементов в системе и наименьших концептуальных единиц. При использовании библиотек классов выбор некоторых объектов оказывается очевидным. Анализ сходства между классами, которое может быть положено в основу выбора базовых классов и иерархии наследования, может происходить как на этой стадии, так и позднее в процессе проектирования.
2. *Доработка объектов*. В процессе разработки объекта иногда выясняется, что объект должен содержать новые члены, отсутствующие в первоначальном проекте. Изменение внутреннего устройства объекта может потребовать поддержки со стороны других классов.
3. *Конструирование системы*. Как уже отмечалось, на этой стадии иногда выявляются дополнительные требования к объектам. По мере углубленного изучения задачи происходит совершенствование ваших объектов. Необходимость взаимодействия с другими объектами в системе может повлиять на

[1] В качестве языка для «исполняемого псевдокода» часто используется язык Python (www.python.org).

первоначальную структуру классов или потребовать введения новых классов. Например, может выясниться, что в системе нужны вспомогательные классы (вроде классов связанных списков), которые содержат минимум информации о состоянии (или вовсе не содержат ее) и просто облегчают работу других классов.

4. *Расширение системы.* По мере расширения возможностей системы иногда выясняется, что это расширение упирается в несовершенство предыдущей архитектуры. На основании полученной информации вы можете реструктурировать часть системы, возможно, добавить новые классы или иерархии.
5. *Повторное использование объектов.* На этой стадии класс проходит настоящее «испытание на прочность». Если кто-то попытается заново использовать его в принципиально новой ситуации, могут обнаружиться какие-то недостатки и недочеты. По мере адаптации класса к новым условиям все четче проявляются общие принципы работы класса, пока вы не получите тип, действительно подходящий для повторного использования. Но не стоит ожидать, что многие объекты в системной архитектуре должны иметь универсальный характер — вполне нормально, если большинство объектов ориентировано на конкретную систему. Типы, подходящие для повторного использования, встречаются реже и ориентируются на решение более общих задач.

Рекомендации по разработке объектов

Ниже приведены некоторые рекомендации, которыми следует руководствоваться при разработке классов.

- Создайте класс для решения конкретной задачи, обеспечьте его рост и развитие по мере решения других задач.
- Помните, выявление необходимых классов (и их интерфейсов) является важнейшей составляющей системного проектирования. Если у вас уже есть описания классов, работа значительно упрощается.
- Не пытайтесь узнать все с самого начала; изучайте проблему по мере надобности. От этого все равно никуда не деться.
- Переходите к программированию. Создайте рабочий прототип, который подтвердит или опровергнет первоначальную архитектуру. Не бойтесь, что у вас получится «спагетти-код» в стиле процедурного программирования, — классы обеспечивают логическое деление проблемы и помогают справиться с анархией и энтропией. Плохо спроектированные классы не нарушат работу хорошо спроектированных классов.
- Стремитесь к простоте. Небольшие понятные объекты с очевидными задачами лучше больших, запутанных интерфейсов. Принимая решения, руководствуйтесь принципом «бритвы Оккама»: рассмотрите все возможные варианты и выберите самый простой из них, потому что простые классы почти всегда оказываются лучшими. Начните с простых и компактных решений, и вы сможете расширить интерфейс класса, когда начнете его лучше понимать. С другой стороны, чем больше времени пройдет, тем сложнее будет удалять элементы из класса.

Фаза 3 — построение ядра

В этой фазе первоначальный набросок архитектуры преобразуется в компилируемый исполняемый код, который можно протестировать. Но самое главное, что прототип должен либо подтвердить, либо опровергнуть исходную архитектуру. Как будет показано при описании фазы 4, этот процесс выполняется не за один раз, а представляет собой последовательность этапов интерактивного построения системы.

Ваша задача — выявить ядро системной архитектуры, реализация которого абсолютно необходима для построения работоспособной системы, какой бы неполной ни была эта система в первом приближении. Фактически вы строите каркас, который будет наращиваться при последующих итерациях. Также при этом выполняется первая попытка системной интеграции и первое тестирование, а заказчик получает представление о том, как будет выглядеть его система и как идет работа. В идеальном случае при этом выявляются важнейшие риски, связанные с разработкой системы. Возможно, также обнаружатся некоторые изменения и усовершенствования в исходной архитектуре, о которых вы бы не узнали без реализации системы.

Одной из составляющих разработки системы является сравнение промежуточного результата с анализом требований и системными спецификациями (в той форме, в которой они существуют). Убедитесь в том, что результаты тестирования соответствуют всем требованиям и сценариям. Когда ядро системы заработает стабильно, можно двигаться дальше и добавлять новые возможности.

Фаза 4 — итеративный перебор сценариев

Получив работоспособное ядро, вы начинаете добавлять к нему новые возможности. Каждая группа возможностей может рассматриваться как мини-проект, а ее реализация занимает в цикле разработки относительно короткий период, который называется *итерацией*.

Какова продолжительность итераций? В идеале каждая итерация продолжается от одной до трех недель (срок зависит от языка реализации). В конце этого периода вы получаете интегрированную протестированную систему, которая обладает большими возможностями, чем раньше. Но особенно интересен тот факт, что итерации формируются на основе отдельных функциональных требований. Каждое функциональное требование представляет собой совокупность связанных функциональных возможностей, которые встраиваются в систему как единое целое во время одной итерации. Это не только дает лучшее представление о практических масштабах одной итерации, но и дополнительно подкрепляет саму идею функциональных требований, которые не теряют смысла после анализа и проектирования, а становятся фундаментальными единицами в процессе разработки программного обеспечения.

Итерация прекращается либо при достижении поставленной цели, либо в том случае, если на данном этапе заказчик удовлетворен текущей версией. Поскольку процесс разработки имеет итеративную природу, вместо единой конечной точки существует много возможностей поставки готового продукта; скажем, продукты с открытыми текстами создаются исключительно в итеративных условиях с хорошей обратной связью, чем и объясняется их успех.

Итеративный процесс разработки полезен по многим причинам. Итеративность позволяет обнаружить и решить критически важные проблемы на более ранней стадии; у заказчика появляется возможность откорректировать начальную постановку задачи; работа программиста становится более творческой, а сам проект лучше управляется. Не стоит забывать и о таком важном факторе, как обратная связь с заказчиком, который точно представляет себе текущее состояние продукта. В результате сокращается или вовсе исчезает необходимость в утомительных совещаниях, а разработчик получает большую поддержку и уверенность со стороны заказчика.

Фаза 5 — эволюция

Эта стадия цикла разработки традиционно называлась «сопровождением». Под этим универсальным термином может пониматься все что угодно: от «заставить систему работать так, как было задумано с самого начала», до «включить новые возможности, о которых забыл упомянуть заказчик» и более традиционных «исправить найденные ошибки» или «добавить новые возможности в случае необходимости». Термин «сопровождение» отягощен таким количеством ошибочных толкований, что теперь он уже воспринимается как лукавство. Он наводит на мысль, что вы создали безупречную программу, так что дальше остается время от времени менять запчасти, смазывать ее и вытирать насухо, чтобы не ржавела. Возможно, происходящее правильнее было бы назвать другим термином.

Автор предпочитает термин «эволюция». Он означает следующее: «С первого раза идеально все равно не получится, поэтому мы предусматриваем возможность для дополнительного анализа, возврата и внесения изменений». По мере изучения и более глубокого понимания проблемы иногда приходится вносить множество изменений. Элегантность, достигнутая в результате доработки программы, окупается как в краткосрочном, так и в долгосрочном плане. Эволюция обеспечивает тот промежуток времени, когда программа превращается из хорошей в замечательную, а проблемы, которые оставались непонятными при первой попытке, внезапно проясняются. Тогда же классы могут превратиться из разовых разработок в ресурсы многократного использования.

Под «доработкой» здесь понимается не только то, что программа должна начать функционировать в соответствии со всеми требованиями и сценариями. Также это означает, что внутренняя структура программы выглядит разумной с вашей точки зрения, программа не содержит неуклюжего синтаксиса, слишком крупных объектов или неловко пристегнутых «заплаток». Кроме того, вы должны быть уверены в том, что структура программы переживет все неизбежные изменения во время ее жизненного цикла и что эти изменения можно будет вносить быстро и просто. Необходимо не только понимать, что именно вы строите, но и в каком направлении будет развиваться программа (то, что автор называет *вектором изменений*). К счастью, объектно-ориентированные языки особенно хорошо подходят для повторных модификаций такого рода — границы объектов предохраняют структуру программы от распада. Благодаря им изменения, которые бы показались радикальными в процедурных программах, вносятся без распространения «ударной волны» по всей программе. В сущности, поддержка эволюции оказывается едва ли не самым важным преимуществом ООП.

В процессе эволюции вы создаете нечто, хотя бы отдаленно напоминающее итоговую цель, а затем с учетом ваших потребностей смотрите, где возникли расхождения. Далее можно вернуться и исправить ошибку переработкой архитектуры и повторной реализацией тех частей программы, которые работают неправильно[1]. Возможно, чтобы добраться до правильного решения, вам придется решать задачу многократно (или разбираться с ее отдельным аспектом). При этом особенно полезны типовые решения, или *эталоны*, описанные во втором томе книги.

Эволюция имеет место и в другом случае — когда вы строите систему, убеждаетесь, что она удовлетворяет начальным требованиям, а потом вдруг выясняется, что это совсем не то. Когда система уже работает, вы понимаете, что на самом деле она должна решать другую проблему. Если вы не исключаете подобную возможность, постарайтесь как можно быстрее построить первую версию и выяснить, соответствует ли она вашим ожиданиям.

Главное, о чем необходимо помнить, — что по умолчанию (а скорее, даже по определению) модификация класса не должна нарушать работу его базовых и производных классов. Не бойтесь модификации (особенно если у вас имеется встроенный набор модульных тестов для проверки их правильности). Модификации не всегда нарушают работу программы, а любые изменения результата будут ограничиваться производными классами и/или отдельными классами, взаимодействующими с измененным классом.

О пользе планирования

Никому не придет в голову строить жилой дом без множества тщательно проработанных планов. Если вы строите сарай или собачью будку, процесс планирования будет не столь тщательным, но, скорее всего, вы все же сделаете хотя бы простейшие наброски и будете руководствоваться ими. Программирование часто бросалось в крайности. В течение долгого времени разработки вообще не структурировались, но это стало приводить к провалам крупных проектов. В ответ появились новые методологии, невероятно детальные и структурированные, ориентированные прежде всего на крупные проекты. Эти методологии были слишком жуткими — казалось, программисту придется тратить все время на написание документации, а на программирование его уже не останется (кстати, на практике так часто и выходило). Все это наводит на мысль о «золотой середине» — подвижной шкале. Используйте тот способ, который лучше соответствует вашим потребностям (и вашим личным предпочтениям). Но каким бы расплывчатым ни был ваш план, помните, что проекты с *хоть каким-нибудь* планом идут гораздо лучше проектов, полностью лишенных всякого плана. Также не стоит забывать, что по статистике свыше 50 % проектов завершаются неудачей (а по некоторым данным — до 70 %)!

Если следовать некоторому плану (желательно простому и понятному) и заранее проработать общую структуру программы, работа пойдет легче, чем если вы просто возьметесь за программирование. Более того, ваша работа будет гораздо

[1] Эта методика напоминает «ускоренную разработку прототипа», когда вы наспех сооружаете простенькую модель для изучения системы, а затем выкидываете прототип и строите ее правильно. Главная проблема «ускоренной разработки прототипа» заключалась в том, что программисты его не выбрасывали, а использовали как базу для построения окончательной версии. В сочетании с недостаточной структурированностью процедурного программирования это часто приводило к созданию хаотичных систем, дорогих в сопровождении.

интереснее. Судя по опыту автора, разработка элегантного решения приносит необычайное удовольствие; такое программирование ближе к искусству, чем к технологии. Помните, что элегантность всегда окупается. Элегантное программирование упрощает не только разработку и отладку, но также чтение и сопровождение программы, а с этим связана прямая финансовая выгода.

Экстремальное программирование

Автор начал изучать методы анализа и проектирования еще в аспирантуре. Из всего, с чем удалось познакомиться, концепция экстремального программирования (eXtreme Programming, XP) была наиболее радикальной и восхитительной. Ее описание можно найти в книге Кента Бека (Kent Beck) «Extreme Programming Explained»[1] и на сайте www.xprogramming.com.

Экстремальное программирование — это одновременно и философия программирования, и набор рекомендаций по ее воплощению в жизнь. Некоторые из этих рекомендаций встречаются в других методологиях последнего времени, но, на взгляд автора, самыми важными и характерными положениями экстремального программирования являются тезисы «начинайте с написания тестов» и «программируйте парами». Хотя Бек настоятельно рекомендует взять на вооружение весь процесс, он указывает, что даже принятие этих двух правил заметно улучшит эффективность вашего труда и надежность программ.

Начинайте с написания тестов

Тестирование традиционно откладывалось до завершающей стадии проекта, когда «все уже работает, но проверить не помешает». Оно рассматривалась как низкоприоритетная задача, а к специалистам по тестированию часто относились пренебрежительно и даже содержали их в каком-нибудь чулане отдельно от «настоящих программистов». Тестеры отвечали окружающему миру взаимностью, носили черное и злорадно хихикали, когда им удавалось что-нибудь сломать (честно говоря, автор и сам ощущал нечто подобное, когда ломал компиляторы C++).

В экстремальном программировании концепция тестирования полностью пересматривается, и ей уделяется такое же (а то и более важное) значение по сравнению с программированием. Более того, тесты пишутся *раньше* тестируемого кода и остаются с ним навечно. Тесты должны успешно выполняться при каждой интеграции проекта (что нередко означает «чаще одного раза в день»).

Написание тестов в начале проекта имеет два исключительно важных последствия.

Во-первых, оно заставляет программиста четко определить интерфейс класса. Автор часто предлагает в процессе проектирования системы «представить себе идеальный класс для решения некоторой проблемы». Стратегия тестирования в XP идет еще дальше — вы должны точно указать, как выглядит класс и каково его поведение с точки зрения пользователя этого класса. Все предельно конкретно, ни малейшей неопределенности. Конечно, можно описать класс в прозе или нарисо-

[1] См. *Бек К., Фаулер М.* Экстремальное программирование: планирование: Библиотека программиста». СПб.: Питер, 2003.— *Примеч. ред.*

вать множество диаграмм, описывающих поведение класса и его структуру, но ничто так не приближено к реальности, как набор тестов. Описания и диаграммы — не более чем пожелания, а тест описывает обязательный контракт между компилятором и выполняемой программой. Трудно представить себе более конкретную спецификацию класса, чем тест.

При создании теста вам придется досконально продумать структуру класса, причем иногда при этом обнаруживаются возможности, упущенные во время мысленных экспериментов с UML-диаграммами, карточками, сценариями и т. д.

Второе важное последствие от опережающего написания тестов объясняется тем, что тесты все равно выполняются при каждом получении программы, причем половину тестирования фактически проводит компилятор. Если взглянуть на эволюцию языков программирования с этой точки зрения, вы увидите, что все подлинные усовершенствования в технологии так или иначе связаны с проверкой и тестированием. В ассемблере проверялся только синтаксис, а в C появились некоторые семантические ограничения, которые предотвращали ошибки определенных типов. В языках ООП установлены еще более жесткие семантические ограничения, которые, если хорошо подумать, тоже являются формами проверки. «Правильно ли используется этот тип? Правильно ли вызывается эта функция?» — все это разные типы проверок, осуществляемых компилятором или системой времени выполнения. Результаты включения этих проверок в язык уже известны: программисты создают более сложные системы и запускают их в работу с гораздо меньшими затратами времени и усилий. Раньше автор не понимал, из-за чего это происходит, но теперь стало ясно, что дело в тестах: если что-то пойдет не так, то встроенные тесты сообщат о наличии проблемы и ее источниках.

Однако на большее встроенные средства тестирования, поддерживаемые на языковом уровне, не способны. На каком-то этапе *вам* придется взяться за дело и добавить остальные тесты, которые завершали бы тестовый комплекс (в сочетании с тестами компилятора и системы времени выполнения) для проверки всей программы. Но если уж компилятор постоянно помогает вам в поиске ошибок, то почему бы не пользоваться тестами с самого начала? Вот почему лучше сразу написать тесты и автоматически включать их во все промежуточные версии системы. Ваши тесты станут логическим расширением проверочных средств, предоставляемых языком.

Используя все более и более мощные языки программирования, автор обнаружил, что они поощряют к смелым экспериментам, поскольку при поиске ошибок сам язык избавляет от лишних трат времени. Стратегия тестирования экстремального программирования делает то же самое с целым проектом. Вы уверены в том, что тесты всегда помогут выявить все проблемы (и регулярно добавляете в проект новые тесты, когда в голову приходит хорошая идея), поэтому в проект можно вносить серьезные изменения, не беспокоясь о том, что он придет в состояние полного хаоса.

Парное программирование

Парное программирование противоречит закоренелому индивидуализму, который нам прививают всю жизнь, начиная со школы (где все успехи и неудачи засчитываются лично нам, а взаимодействие с соседями считается жульничеством) и кончая средствами массовой информации, особенно голливудскими фильмами, в которых герой-одиночка обычно противостоит безликой массе

злодеев[1]. Программисты также считаются воплощением индивидуализма — «ковбоями от программирования», как любит выражаться Ларри Константайн (Larry Constantine). И все же экстремальное программирование, которое ломает сложившиеся стереотипы, утверждает, что программу должны писать двое работников за одним рабочим местом. Более того, рабочие места в группах не должны разделяться перегородками, которые так любимы дизайнерами наших рабочих помещений. Бек говорит, что при переходе на экстремальное программирование нужно прежде всего взять гаечные ключи и разобрать все, что мешает общению[2] (для этого также понадобится начальник, который усмирит гнев хозяйственного отдела).

Главная ценность парного программирования заключается в том, что один программист пишет программу, а второй думает. Этот второй представляет себе общую картину, помня не только о непосредственно решаемой задаче, но и о правилах XP. Когда работают двое, вряд ли кто-нибудь заявит: «Я не хочу писать тесты с самого начала». Если программист окажется в тупике, он может поменяться местами с напарником. А если в тупике окажутся оба, возможно, их разговоры услышит кто-нибудь другой, кто сможет помочь. Парная работа способствует нормальному продвижению проекта, но не менее важно и то, что программирование становится более социальным и интересным занятием.

Автор начал использовать парное программирование на практических занятиях своих учебных курсов. На взгляд автора, оно значительно улучшает восприятие материала учащимися.

Причины успеха C++

Одна из причин успеха C++ заключается в том, что проектировщики языка стремились не просто превратить C в объектно-ориентированный язык (хотя начиналось именно с этого), но также решить ряд других проблем, с которыми и в наше время сталкиваются разработчики, особенно имеющие большой опыт программирования на C.

Традиционно языки ООП требовали, чтобы программист забыл все, что он знает, и начал «с нуля» с новыми концепциями и новым синтаксисом. Это аргументировалось тем, что в долгосрочной перспективе лучше полностью отказаться от старого багажа, унаследованного из процедурных языков. Возможно, в долгосрочной перспективе это действительно так, но с практической точки зрения в этом багаже было немало ценного. Причем самым ценным элементом была даже не существующая база программного кода (которую при наличии хорошего инструментария можно было конвертировать), а скорее *база мышления*. Если действующему программисту C для перехода на новый язык приходилось отказываться от всего, что он знает о C, производительность его труда резко снижалась на многие месяцы, пока его ум не привыкал к новой парадигме. С другой стороны, если программист

[1] Наверное, этот пример больше рассчитан на американцев, хотя голливудские истории проникают повсюду.

[2] Включая (в первую очередь) систему местной громкой связи. Однажды автор работал в компании, которая требовала передавать по громкой связи все телефонные звонки всех руководителей, что основательно мешало нашей работе (однако начальство и представить себе не могло, как можно жить без громкой связи). Пришлось потихоньку резать провода, идущие к динамикам.

мог взять за основу существующие познания в C и расширять их, то при переходе в мир объектно-ориентированного программирования он продолжал эффективно работать. У каждого из нас есть собственная внутренняя модель программирования, и такие переходы достаточно хлопотны и без дополнительного бремени в виде новой языковой модели. Таким образом, успех C++ в основном объяснялся экономическими причинами: переход на ООП вообще требует затрат, но переход с C на C++ может обойдись дешевле[1].

Главной целью разработки C++ является повышение эффективности. Источников этого повышения много, но язык спроектирован так, чтобы по возможности помочь программисту, не отягощая его искусственными правилами или требованиями об использовании некоторого ограниченного набора средств. Язык C++ планировался как практичный язык, а главной целью его разработки была максимальная польза для программиста (по крайней мере, с точки зрения C).

Улучшенный язык C

Даже если вы продолжите писать C-образный код, переход на C++ сразу же принесет пользу, потому что C++ закрыл много «дыр» C, обеспечил более качественную проверку типов и анализ на стадии компиляции. Программист вынужден объявлять функции, чтобы компилятор мог проверить правильность их вызова. В C++ практически исчезла необходимость в использовании препроцессора для подстановки значений и макросов, что раньше приводило к трудноуловимым ошибкам. В C++ появились *ссылки*, обеспечивающие более удобную передачу адресов в аргументах и возвращаемых значениях функций. Механизм *перегрузки функций* существенно упростил работу с именами, поскольку теперь одно и то же имя может быть присвоено разным функциям. *Пространства имен* также улучшают контроль над использованием имен в программе. Существует еще множество мелких усовершенствований, которые делают C++ более надежным языком по сравнению с C.

Быстрое обучение

Главная проблема с изучением нового языка часто связана с производительностью труда программиста. Ни одна компания не может себе позволить вдруг лишиться эффективно работающего программиста только потому, что он изучает новый язык. C++ является расширением C, его синтаксис и модель программирования не будут абсолютно новыми для программиста. Благодаря этой особенности программист по-прежнему пишет полезные программы, используя новые возможности по мере их изучения. Возможно, это одна из главных составляющих успеха C++.

Кроме того, весь существующий код C может компилироваться в C++, но так как компилятор становится более придирчивым, при перекомпиляции в программах C часто обнаруживаются скрытые ошибки.

---

[1] Автор говорит «может», потому что из-за сложностей C++ переход на Java иногда обходится дешевле. Однако решение о выборе языка принимается на основе многих факторов, и в этой книге предполагается, что вы уже выбрали C++.

Эффективность

В некоторых ситуациях скоростью работы программы приходится жертвовать ради ускорения работы программиста. Например, финансовая модель обычно остается актуальной в течение ограниченного периода времени, поэтому ее быстрая реализация важнее быстрой работы программы. Однако большинство приложений требует определенной скорости работы, поэтому C++ всегда отдает предпочтение более эффективным решениям. А так как программисты C особенно внимательно следят за эффективностью, они не будут жаловаться, что новый язык неповоротлив и медлителен. В C++ предусмотрен целый ряд средств оптимизации на случай, если сгенерированный код окажется недостаточно эффективным.

В распоряжении программиста оказываются низкоуровневые средства C вместе с возможностью включения ассемблерного кода в программы C++. Статистика показывает, что быстродействие объектно-ориентированной программы C++ обычно находится в пределах ±10 % от быстродействия аналогичной программы C, а иногда гораздо ближе. С другой стороны, архитектура ООП-программы может быть гораздо эффективнее своего аналога в C.

Простота описания и понимания системы

Классы, спроектированные для решения проблемы, обычно выражают ее лучше обычного кода. Это означает, что при написании программы решение представляется в пространстве задачи («Замкнуть контакт»), а не в пространстве машины («Установить бит, который является признаком замыкания контакта»). Программист работает с концепциями более высокого уровня, а одна строка программы несет гораздо большую смысловую нагрузку.

Другое преимущество простоты выражения проявляется на стадии сопровождения, на которую (если верить статистике) уходит заметная часть расходов в жизненном цикле программы. Естественно, понятные программы проще сопровождать. Кроме того, при этом сокращаются затраты на создание и ведение документации.

Максимальная интеграция с библиотеками

Чтобы написать программу, быстрее всего воспользоваться уже готовым кодом, то есть библиотеками. Одной из главных целей, поставленных при проектировании C++, была простота работы с библиотеками. Для этого библиотеки трансформируются в новые типы данных (классы), поэтому подключение библиотеки означает добавление новых типов в язык. Компилятор C++ сам следит за использованием библиотеки, обеспечивает всю необходимую инициализацию и зачистку, а также проверяет правильность вызова функций, поэтому программист может сосредоточить внимание на том, что делает библиотека, а не на том, как с ней работать.

Поскольку пространства имен ограничивают область действия имен отдельными частями программы, свободное использование библиотек не создает конфликтов имен, характерных для C.

Шаблоны и повторное использование исходных текстов

Существует достаточно обширная категория типов, для повторного использования которых необходима модификация исходных текстов. *Шаблоны* C++ автома-

тически изменяют исходные тексты, что делает их исключительно важным инструментом для повторного использования библиотечного кода. Обобщенный класс, созданный на базе шаблона, легко адаптируется для другого типа-параметра. Шаблоны особенно хороши тем, что полностью скрывают от прикладного программиста все сложности, связанные с повторным использованием кода.

Обработка ошибок

Как известно, обработка ошибок в C порождала немало проблем. Нередко проблема просто игнорировалась и программист полагался «на авось». При построении большого и сложного проекта нет ничего хуже, чем обнаружить ошибку, которая находится неизвестно где. Механизм *обработки исключений* (кратко упомянутый в этом и подробно рассматриваемый во втором томе) гарантирует, что ошибка будет обнаружена и каким-то образом обработана программой.

Масштабное программирование

Во многих традиционных языках существуют встроенные ограничения на размеры и сложность программ. Например, BASIC прекрасно подходит для разработки быстрых решений некоторых классов задач, но если программа занимает больше нескольких страниц или выходит за рамки обычной области применения этого языка, программирование начинает напоминать плавание в непрерывно густеющей жидкости. У языка C тоже есть свои ограничения. Например, когда объем программы превышает 50 000 строк кода или около того, начинаются проблемы с конфликтами имен — фактически у вас просто кончаются удобные имена функций и переменных. Другая неприятная проблема связана с недоработками самого языка C — в большой программе иногда бывает очень трудно найти ошибку.

Не существует четких критериев, определяющих границы применимости языка, но даже если бы такие критерии и были, скорее всего, вы бы их игнорировали. Никто не скажет: «Моя программа на BASIC слишком велика — пожалуй, ее стоит переписать на C!» Вместо этого вы попытаетесь затолкнуть в нее еще несколько строк и добавить «всего одну» новую возможность. В конечном счете это обернется лишними затратами при сопровождении.

Язык C++ проектировался для *масштабного программирования*, при котором стираются грани между мелкими и крупными программами. Конечно, при написании утилиты сложности «Hello, World!» вам не придется использовать ООП, шаблоны, пространства имен и обработку исключений, но эти возможности присутствуют в языке и могут задействоваться при необходимости. С другой стороны, компилятор с одинаковым рвением выискивает ошибки как в больших, так и в мелких программах.

Стратегии перехода

Если вы твердо решили перейти на ООП, возникает следующий вопрос: «Как заставить начальство/коллег/другие отделы перейти на использование объектов?» Подумайте, как бы вы — независимый программист — перешли на изучение нового языка и новой парадигмы программирования. Вам ведь уже приходилось делать

это раньше, не правда ли? Все начинается с обучения и примеров; далее следует экспериментальный проект, который позволяет прочувствовать суть дела без лишних сложностей. Затем наступает черед «реального» проекта, который делает что-то полезное. На протяжении первых проектов вы продолжаете обучение — читаете книги, задаете вопросы знатокам и обмениваетесь полезной информацией с друзьями. Так выглядит рекомендуемая многими опытными программистами процедура перехода с C на C++. Конечно, при переходе целой организации добавится некоторая групповая динамика, но на каждом этапе стоит помнить, как бы эта задача решалась одним человеком.

Рекомендации

Ниже приводятся некоторые рекомендации по переходу на ООП и C++.

Обучение

Первым шагом должна стать некоторая форма обучения. Помните о капиталовложениях компании в существующий код C и постарайтесь избежать хаоса в то время, пока народ будет разбираться в тонкостях множественного наследования. Выберите небольшую группу для обучения; желательно, чтобы ее участники были любознательны, хорошо уживались друг с другом и могли оказывать взаимную помощь в ходе изучения C++.

Иногда предлагается другой подход — одновременное обучение всей компании сразу с обзорными курсами для руководства и занятиями по проектированию и программированию для разработчиков проектов. Подобные решения особенно хороши в небольших компаниях, желающих изменить свой профиль, или на уровне подразделений крупных компаний. Однако из-за возрастающей стоимости проекта в некоторых случаях сначала обучается небольшая группа персонала, которая создает опытный проект (возможно — с внешним наставником), а затем обучает остальных сотрудников.

Проекты с низкой степенью риска

Попробуйте для начала выполнить проект с низкой степенью риска, в котором ошибки не так критичны. Когда участники проекта получат определенный опыт, они смогут инициировать другие проекты или выполнять обязанности службы технической поддержки по ООП. Нельзя исключать, что первый проект провалится, поэтому он не должен быть жизненно важным для организации. Проекту достаточно быть простым, самодостаточным и поучительным; это означает, что разработанные в нем классы должны быть понятны для других программистов, которые будут следующими изучать C++.

Успешные решения

Прежде чем начинать работу «с нуля», стоит поискать примеры хорошего объектно-ориентированного проектирования. Вполне вероятно, что кто-то уже решил вашу задачу, но даже если полностью подходящего решения не найдется, возможно, вам удастся воспользоваться своими познаниями в области абстракции и адаптировать существующую архитектуру к своим потребностям. На этом принципе строится общая концепция *эталонов* (типовых решений), описанная во втором томе книги.

Готовые библиотеки классов

Основным экономическим стимулом для перехода на ООП является возможность простого использования готового кода в виде библиотек классов (и прежде всего стандартной библиотеки C++, подробно описанной во втором томе книги). Кратчайший цикл разработки приложения достигается тогда, когда программисту не приходится писать ничего, кроме функции `main()`, создающей и использующей объекты из готовых библиотек. Тем не менее многие новички не понимают этого, не знают о существовании готовых библиотек или от избытка энтузиазма готовы заново переписать уже существующие классы. Ваш переход на ООП и C++ будет еще эффективнее, если вы на ранней стадии поищете готовый код и задействуете его в своем проекте.

Существующие программы на C++

Компиляция кода C компилятором C++ обычно приносит пользу (причем порой огромную), так как она помогает обнаружить проблемы старых программ. Тем не менее обычно не стоит брать готовые, работоспособные программы и переписывать их на C++ (а если их необходимо привести к объектному виду, можно «завернуть» готовый код C в классы C++). Конечно, это может принести определенную выгоду, особенно если код предназначен для повторного использования, но вряд стоит ждать радикального повышения быстродействия в готовых проектах. Достоинства C++ и ООП наиболее очевидно проявляются при воплощении нового проекта из концепции в реальность.

Организационные трудности

Если вы работаете на руководящей должности, ваша задача — предоставить все необходимые ресурсы, преодолеть все препятствия на пути к успеху и вообще создать наиболее продуктивную и приятную рабочую среду, чтобы вашим подчиненным было легче совершать те чудеса, которых от вас вечно требует начальство. Переход на C++ влияет на все три категории задач, и все было бы замечательно, если бы за него не приходилось платить. Хотя для группы программистов C переход на C++ может обойтись дешевле перехода на другие объектно-ориентированные языки[1], он все же потребует определенных затрат. Существует целый ряд препятствий, о которых необходимо знать, прежде чем вы решитесь перейти на C++ и перевести на него вашу организацию.

Начальные издержки

Начальные издержки перехода на C++ не ограничиваются стоимостью компилятора C++ (кстати, один из лучших компиляторов — компилятор GNU C++ — распространяется бесплатно). Для сведения к минимуму средне- и долгосрочных затрат необходимо потратиться на подготовку персонала (а возможно, и на наем наставников для первого проекта), а также на поиск и приобретение библиотек классов, используемых в решении задачи (вместо самостоятельной разработки этих классов). Это непосредственные затраты, которые должны учитываться в любом реалистичном предложении. Кроме того, существуют скрытые издержки в виде снижения производительности во время изучения нового языка, а возможно,

[1] Также стоит упомянуть язык Java, который в последнее время стал весьма распространенным.

и новой среды программирования. Конечно, правильная организация обучения способна свести эти затраты к минимуму, но участникам группы все равно придется тратить силы на усвоение новой технологии. В это время они будут совершать больше ошибок (что только полезно, поскольку осознание своих ошибок — самый быстрый способ обучения) и работать менее эффективно. Но даже на стадии изучения C++ в определенном классе задач и при правильном выборе классов и среды разработки (даже с учетом большего количества ошибок и меньшего количества строк программы в день) может быть достигнута более высокая производительность, чем при работе на C.

Проблемы эффективности

Стандартный вопрос: «Разве от применения ООП мои программы не становятся больше и не начинают работать медленнее?». Ответ: «Когда как». Большинство объектно-ориентированных языков проектировалось с расчетом на эксперименты и ускоренную разработку прототипов, нежели на быстроту и эффективность. Соответственно, они практически всегда приводили к увеличению объема и снижению быстродействия программ. Однако язык C++ проектировался для реального программирования. Стремясь побыстрее создать прототип, вы как можно скорее связываете готовые компоненты, не обращая внимания на эффективность. Если при этом используются библиотеки сторонних производителей, они обычно уже оптимизированы своими разработчиками; впрочем, в режиме ускоренной разработки это обычно несущественно. Если в результате получается система, которая делает то, что нужно, и достаточно быстро, работа закончена. А если нет, приходится браться за профайлер и начинать оптимизацию. Поиск начинается с тех усовершенствований, которые могут быть реализованы встроенными средствами C++. Если это не приносит успеха, вы переходите к оптимизации внутренней реализации, чтобы сохранить в неприкосновенности код работы с классами. Только если это не помогает, приходится вносить изменения в архитектуру системы. Тот факт, что эффективность играет критически важную роль в архитектуре системы, свидетельствует о необходимости ее включения в исходные критерии проектирования. Средства быстрой разработки позволяют вовремя это понять.

Как уже упоминалось, различия в быстродействии и размерах программ C и C++ чаще всего характеризуются величиной ±10 %, хотя обычно они гораздо ближе друг к другу. В отдельных случаях применение C++ вместо C даже обеспечивает значительный выигрыш в быстродействии и размерах программы, потому что архитектура программ C++ может существенно отличаться от архитектуры программ C.

Результаты сравнения программ C и C++ по быстродействию и размерам не являются абсолютно достоверными и, скорее всего, таковыми останутся. Сколько бы народ ни предлагал реализовать один и тот же проект на C и C++, вряд ли какая-нибудь компания захочет тратить деньги на подобные эксперименты — разве что очень крупная и заинтересованная в таких исследованиях (но даже в этом случае деньги можно потратить лучше). Почти все программисты, переходящие на C++ или другой язык ООП с C (или другого процедурного языка), говорят, что их труд стал гораздо эффективнее, — и это самый убедительный аргумент, который только можно представить.

Распространенные ошибки проектирования

В первое время после перехода на ООП и C++ программисты обычно совершают одни и те же стандартные ошибки. Это часто происходит из-за нехватки общения с экспертами в ходе проектирования и реализации первых проектов, потому что свои эксперты в компании еще не появились, а намерение пригласить платных консультантов может столкнуться с возражениями. Многие новички воображают, что они уже разобрались в ООП, и идут в неверном направлении. То, что очевидно опытному программисту, может вызвать долгую внутреннюю борьбу у новичка. Чтобы обойтись без лишних переживаний, для обучения персонала лучше пригласить эксперта.

С другой стороны, простота, с которой совершаются эти ошибки проектирования, указывает на главный недостаток C++ (который одновременно является главным достоинством): его совместимость с языком C. Чтобы можно было компилировать код C, разработчикам C++ пришлось пойти на некоторые компромиссы, что привело к появлению неких «темных мест» — вполне реальных и требующих немалых усилий при изучении языка. В этой книге автор постарался раскрыть большинство ловушек, часто встречающихся при программировании на C++. Всегда помните, что система страховки от ошибок в C++ не идеальна.

Итоги

В этой главе автор постарался дать представление об общих перспективах объектно-ориентированного программирования и C++. В частности, были описаны отличительные особенности ООП (и C++ в частности), методологии ООП, а также некоторые проблемы, встречающиеся при переходе компании на ООП и C++.

Впрочем, ООП и C++ — не панацея. Очень важно правильно оценить ваши потребности и решить, удовлетворит ли их C++ оптимальным образом или лучше выбрать другую систему программирования (возможно, ту, которую вы используете в данный момент). Если вы знаете, что в обозримом будущем вам придется иметь дело с нестандартными задачами и специфическими ограничениями, на которые язык C++ не рассчитан, вам придется самостоятельно изучить возможные альтернативы[1]. Но даже если в итоге ваш выбор остановится на C++, вы, по крайней мере, будете хорошо знать возможные варианты и четко понимать, что привело вас к этому решению.

Вы знаете, как выглядит процедурная программа: она состоит из определений данных и вызовов функций. Чтобы понять смысл такой программы и построить ее мысленную модель, придется основательно поработать, проанализировать вызовы функций и низкоуровневые операции. Из-за этого при проектировании процедурных программ необходимы промежуточные представления — сами по себе программы получаются слишком запутанными, потому что выразительные средства процедурных языков больше ориентированы на компьютер, нежели на решаемую задачу.

[1] В частности, автор рекомендует познакомиться с языками Java (http://java.sun.com) и Python (http://www.Python.org).

Поскольку C++ дополняет язык C многими новыми концепциями, было бы естественно предположить, что функция `main()` в программе C++ гораздо сложнее своего аналога в эквивалентной программе C. Однако вас ждет приятный сюрприз: хорошо написанная программа C++ обычно гораздо проще и понятнее эквивалентной программы C. Она содержит определения объектов, которые соответствуют концепциям из пространства задачи (вместо аспектов компьютерного представления), и сообщений, пересылаемых объектам для выполнения операций в этом пространстве. Одна из прелестей объектно-ориентированного программирования заключается в том, что хорошо спроектированная программа понятна при простом чтении. Кроме того, программы C++ обычно содержат меньше кода, потому что многие проблемы решаются при помощи готового библиотечного кода.