3

Проектирование с использованием объектов

Теперь, когда (на основе содержимого главы 2) вы воспитали в себе "вкус" к хорошему проекту, пора объединить понятие объекта с идеей "хорошего проекта". Различие между программистами, которые применяют объекты в своих программах, и теми, кто в действительности понимает суть объектно-ориентированного программирования, можно оценить на основании их отношения к связыванию объектов друг с другом и к общему проекту программы.

Начнем с перехода от процедурного программирования к объектно-ориентированному. Даже если вы использовали объекты в течение ряда лет, вам все же стоит прочитать эту главу, чтобы уловить несколько новых идей в отношении применения объектов. При рассмотрении различных видов взаимоотношений между объектами особое место занимают ловушки, жертвами которых часто становятся программисты при построении объектно-ориентированных программ. Вы также узнаете о том, как связан с объектами принцип абстракции.

Объектно-ориентированный взгляд на мир

Переходя от процедурного кодирования (в C-стиле) к объектно-ориентированному, важно помнить, что в основе объектно-ориентированного программирования (ООП) лежит другой способ представления о том, что происходит в программе. Слишком часто программисты увязают в новом синтаксисе и терминологии ООП, прежде чем начнут в достаточной мере понимать, что представляет собой объект. В этой главе, не углубляясь в детали кодирования, мы делаем основной акцент на принципах и идеях. Об особенностях же синтаксиса C++-объектов читайте в главах 8–10.

О процедурном мышлении

При использовании такого процедурного языка, как C, код делится на маленькие участки, каждый из которых, как правило, выполняет одну-единственную задачу. Без этих C-процедур весь код программы был бы сосредоточен в функции `main()`. В таком коде было бы трудно разобраться, и если бы этим пришлось заниматься вашим коллегам, они, мягко говоря, были бы не в восторге.

Компьютеру безразлично, как организована ваша программа: то ли целиком включена в функцию `main()`, то ли разделена на "микроскопические" кусочки с описательными именами и комментариями. Процедуры представляют собой абстракцию, которая существует, чтобы помочь программисту, а также тем, кто вынужден разбираться в его программе и поддерживать ее. Все зависит от того, как вы отвечаете на основной вопрос: "*Что делает эта программа?*". Отвечая на своем родном языке, вы мыслите процедурно. Например, вы могли бы начать проектирование программы выбора акций в следующей последовательности. Сначала программа получает информацию об акциях из Internet, затем сортирует принятые данные по специальным атрибутам, после чего анализирует отсортированные данные и, наконец, выводит список рекомендаций по покупке и продаже. Приступая к кодированию, эту мысленную модель можно напрямую превратить в C-функции: `retrieveQuotes()`, `sortQuotes()`, `analyzeQuotes()` и `outputRecommendations()`.

> **Несмотря на то что C-процедуры называются "функциями", C не является функциональным языком. Термин "функциональный" кардинально отличается от термина "процедурный" и относится к таким языкам, как Lisp (язык обработки списков Лисп), который использует совершенно другой вид абстракции.**

Процедурный подход оправдывает себя, если программа выполняет действия по заданному списку. Однако современные крупные приложения зачастую опираются не на линейную последовательность событий. Ведь пользователь может выполнить любую команду в любой момент времени. Кроме того, процедурное мышление не охватывает представления данных. В предыдущем примере и речи не было о том, как должна быть организована информация об акциях.

Если процедурный способ мышления напоминает ваше нынешнее отношение к написанию программ, не расстраивайтесь. Сейчас просто важно понять, что ООП — это просто альтернативный и более гибкий способ мышления о программе.

Объектно-ориентированный подход к проектированию

В отличие от процедурного подхода, который опирается на вопрос: "Что программа делает?", объектно-ориентированный подход требует ответа на другой вопрос: "Какие объекты реального мира я моделирую?". В основе ООП лежит мнение, что программист должен делить свою программу не на задачи, а на модели физических объектов. Пусть на первый взгляд это утверждение кажется абстрактным, все прояснится после рассмотрения физических объектов в терминах *классов*, *компонентов*, *свойств* и *поведенческих характеристик*.

Классы

Класс позволяет охарактеризовать объект с помощью некоторой формулировки и одновременно выделить его определенным образом. Возьмем апельсин. Подумайте, существует ли разница между апельсинами вообще, если иметь в виду вкусные фрукты, растущие на деревьях, и конкретным экземпляром, лежащим сейчас передо мной на столе.

Отвечая на вопрос: "Что такое апельсин?", мы будем говорить о *классе* предметов, именуемых апельсинами. Все апельсины — фрукты. Все апельсины растут на деревьях. Все апельсины оранжевого цвета (возможно, отличаются в оттенках). Все апельсины имеют специфический вкус. Класс — это просто инкапсуляция всех признаков, определяющих некоторую категорию объектов.

Описывая конкретный апельсин, мы говорим об *объекте*. Все объекты принадлежат конкретному классу. Поскольку тот объект на моем столе является апельсином, я знаю, что он принадлежит к классу апельсинов. Следовательно, я могу утверждать, что это фрукт, который растет на деревьях. Я могу добавить, что *мой* апельсин обычного (средне-) оранжевого цвета и чрезвычайно приятен на вкус. Объект — это *экземпляр* класса, т.е. конкретный элемент с характеристиками, которые отличают его от других экземпляров того же класса.

В качестве более конкретного примера рассмотрим упоминаемое выше приложение выбора акций. В ООП понятие "котировка акций" можно выразить в виде класса, поскольку оно определяет абстрактное представление о том, что составляет любую котировку. Конкретная же котировка, например "текущая котировка акций компании Microsoft", послужила бы примером объекта, так как она является конкретным экземпляром класса котировок.

Если вы — С-программист, то представьте себе классы и объекты в виде аналогов типов и переменных. В главе 8 вы убедитесь в том, что синтаксис для классов подобен синтаксису для С-структур. Объекты синтаксически очень сходны с С-переменными.

Компоненты

Если взять такой сложный реальный объект, как самолет, то нетрудно убедиться в том, что он состоит из более мелких компонентов: фюзеляжа, рычага управления, шасси, моторов и многих других частей. Если для ООП вполне естественно рассматривать объекты в предположении, что они состоят из более мелких компонентов, то для процедурного программирования характерно разбиение сложных задач на более простые процедуры.

Компонент, по сути, можно сравнить с классом. Он отличается от последнего только более мелкими размерами и более специфичными характеристиками. Например, объектно-ориентированная программа включает класс `Airplane`. Вы только представьте себе, каким огромным был бы этот класс, содержащий полное описание

самолета. Но, к счастью, класс `Airplane` составлен из более мелких и более управляемых компонентов. Каждый из таких компонентов в свою очередь может иметь еще более мелкие компоненты. Например, шасси, являясь компонентом самолета, содержит в качестве своего компонента колесо.

Свойства

Свойства представляют собой характеристики, которыми один объект отличается от другого. Давайте вернемся к упомянутому выше "апельсиновому" классу `Orange` и вспомним, что все апельсины имеют характерный вкус и один из оттенков оранжевого цвета. Эти две характеристики и являются свойствами. Все апельсины имеют одинаковые свойства, но с различными значениями. Помните, я уверял вас, что мой апельсин чрезвычайно приятен на вкус, но ваш ведь может быть "ужасно кислым".

Свойства можно рассматривать на уровне класса. Например, мы уже отмечали, что все апельсины — фрукты и растут на деревьях. Эти характеристики присущи классу фруктов, в то время как специфический оттенок оранжевого цвета характерен для конкретного "фруктового" объекта. Свойства класса присущи всем членам класса, в то время как свойства объекта могут иметь для разных объектов разные значения.

В примере с выбором акций класс котировок акций может иметь такие свойства, как имя компании, аббревиатура ценной бумаги, текущая цена и другие статистические данные.

Свойства — это характеристики, которые описывают объект. Они отвечают на вопрос: "Что отличает этот объект от других?".

Поведенческие характеристики

Поведенческие характеристики отвечают на один из двух вопросов: "Что делает этот объект?" или "Что я могу сделать с этим объектом?". В примере с апельсином легче ответить на второй вопрос: например, мы можем его съесть. Как и о свойствах, мы можем думать о поведенческих характеристиках на уровне класса или на уровне объекта. Все апельсины можно съесть примерно одинаковым способом. Но в какой-нибудь другой сфере их "деятельности" можно зафиксировать определенные различия. Например, при скатывании их вниз по наклонной плоскости поведение совершенно круглых апельсинов будет отличаться от поведения более сплющенных.

В примере с выбором акций попробуем найти более практичные поведенческие характеристики. Рассуждая процедурно, мы решили, что наша программа должна анализировать информацию о котировках акций в форме одной из ее функций. Если же мыслить категориями ООП, то напрашивается решение, согласно которому объект котировок акций вполне может заняться "самоанализом"! Другими словами, анализ может стать одной из главных характеристик поведения объекта котировок акций.

В объектно-ориентированном программировании бо́льшая часть функционального кода выведена из процедур и внесена в объекты. Создавая объекты, которым присущи определенные поведенческие характеристики, и определяя характер их взаимодействия, ООП предлагает более богатый механизм для связывания кода с данными, которые он обрабатывает.

Итак, если собрать все в кучу...

Теперь, учитывая все выше сказанное, вы можете по-другому взглянуть на программу выбора акций и перепроектировать ее на объектно-ориентированный "манер".

Как мы говорили выше, все, что входит в понятие "котировки акций", можно определить в виде класса. Для получения списка котировок программа должна оперировать понятием группы классов, которую часто называют *коллекцией*. Поэтому неплохо бы в нашем проекте определить класс для представления "коллекции котировок акций", который должен состоять из более мелких компонентов, т.е. одиночных "котировок акций".

Теперь о свойствах. Класс коллекции должен обладать по крайней мере одним свойством: реальным списком полученных из Internet котировок. Он может иметь и другие свойства, например, точную дату и время самой последней выборки данных и количество принятых котировок. Что касается поведенческих характеристик, то "коллекция котировок акций" должна обладать способностью обращаться к серверу за получением новой информации и составлять отсортированный список котировок. Такую поведенческую характеристику можно назвать "получением котировок".

Класс котировок акций должен иметь свойства, о которых упоминалось выше, т.е. имя, символ, текущую цену и пр. Его поведение, как минимум, должно характеризоваться способностью анализировать собственные данные. Можно рассмотреть включение в класс и других поведенческих характеристик, например, покупку и продажу акций.

Неплохо бы бегло набросать схемы, отображающие отношения между компонентами. Использование нескольких линий (рис. 3.1) означает, что одна "коллекция котировок акций" содержит много объектов "котировок акций".

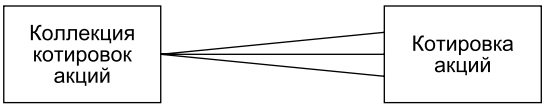


Рис. 3.1

Существует еще один способ отображения классов. Он состоит в перечислении их свойств и поведенческих характеристик (см. следующие две таблицы).

Класс	Компоненты	Свойства	Поведение
`Orange`	Нет	Цвет Вкус	Употреблять в пищу Катить Подбрасывать

Класс	Компоненты	Свойства	Поведение
"Коллекция котировок акций"	Составлена из отдельных объектов "Котировка акций"	Отдельные котировки Время создания Количество котировок	Считывать котировки Сортировать котировки по различным критериям
"Котировка акций"	Нет (пока)	Имя компании Аббревиатура ценной бумаги Текущая цена	Анализировать Покупать акции Продавать акции

Жизнь в мире объектов

При переходе от процедурного мышления к объектно-ориентированной парадигме программисты по-разному относятся к формированию свойств и поведенческих характеристик объектов. Одни при этом частично пересматривают проекты своих

программ и переписывают некоторые фрагменты кода объектов. Другие готовы перечеркнуть свой предыдущий опыт и начать проект как совершенное новое объектно-ориентированное приложение.

Существует два основных подхода к разработке программ с использованием объектов. Некоторые программисты считают объекты просто прекрасным средством инкапсуляции данных и действий над ними. В результате ради улучшения читабельности кода и упрощения его поддержки они стараются как можно гуще "заселить" объектами свои приложения. Для этого они берут отдельные фрагменты кода и заменяют их объектами подобно тому, как хирург вживляет больному кардиостимулятор. В этом, по сути, нет ничего неправильного. Такие программисты видят в объектах инструмент, который обладает во многих ситуациях большими достоинствами. Если определенные части программы "ведут себя подобно объектам" (например, как в случае с котировками акций), то их можно изолировать и описать "человеческим языком".

Другие программисты полностью принимают парадигму ООП и *все* превращают в объекты. По их мнению, некоторые объекты вполне соответствуют реальным вещам (например, апельсин или котировка акций), в то время как другие инкапсулируют более абстрактные идеи (например, механизм сортировки или отмена предыдущего действия). Идеальный подход, как и следует ожидать, лежит где-то посередине. Ваша первая объектно-ориентированная программа может в действительности быть традиционной процедурной программой с некоторыми "объектными вкраплениями". Но, возможно, ваша натура не приемлет половинчатых решений, и вы (гулять так гулять!) превратите в объект все: от класса, представляющего `int`-значения, до класса главного приложения. Со временем вы обязательно найдете *свою* золотую середину.

Избыточность объектов

При создании объектно-ориентированной системы всегда стоит хорошо подумать, прежде чем превращать мельчайшую деталь в объект. Перефразируя Фрейда, можно сказать, что иногда переменная — это только переменная.

Предположим, вы проектируете очередной бестселлер — новую версию игры в крестики-нолики и собираетесь сделать ее полностью объектно-ориентированной. Итак, вы сидите с чашечкой кофе и блокнотом и описываете классы и объекты. В подобных играх зачастую используется объект, который отслеживает ход партии и определяет победителя. Для представления игровой доски вы предполагаете, что объект `Grid` будет отслеживать символьные знаки и их позиции. Компонентом доски (сетки) может быть объект `Piece`, который представляет "крестик" (X) либо "нолик" (O).

Подождите, уж больно резво вы тронулись с места! Для представления "крестика" и "нолика" в таком проекте должен быть специальный класс. Это значит, что у нас уже налицо избыточность объектов. И потом, нельзя ли для представления знаков "X" и "O" использовать просто `char`-переменную? Кроме того, почему бы объекту `Grid` не использовать просто двумерный массив перечислимого типа? А что, объект `Piece` так усложнит код? Рассмотрим следующую таблицу, представляющую предложенный вами (нами) класс `Piece`.

Класс	Компоненты	Свойства	Поведенческие характеристики
`Piece`	Нет	"X" или "O"	Нет

Эта таблица выглядит скудно, но именно ее лаконичность дает нам понять, что вряд ли здесь стоит создавать полноценный объект.

Однако дальновидный программист может поспорить с нами: несмотря на то, что класс `Piece` на данный момент довольно беден по содержанию, преобразование его в объект без особых проблем позволит в будущем расширить программу. Возможно, когда-нибудь это будет графическое приложение, и для поддержки визуального отображения перечеркивания трех одинаковых знаков появится смысл и в создании класса `Piece`. Дополнительным свойством в этом случае мог бы быть цвет фигуры или признак последнего (по времени использования) хода.

Очевидно, правильно ответа не существует вовсе. Главное, чтобы вы при проектировании своего приложения рассмотрели все аргументы "за" и "против". Помните, что объекты призваны помогать программистам справляться с поставленной перед ними задачей. Если объекты используются без веского на то основания, а лишь для того, чтобы сделать код "более объектно-ориентированным", значит, не все благополучно в "королевстве датском".

Слишком общие объекты

Возможно, еще большей неприятностью, чем объекты, которым не стоит быть таковыми, являются слишком общие объекты. Программисты, приступающие к освоению ООП, начинают с примеров, подобных "апельсину", т.е. с реальных объектов. Но в программировании мы сталкиваемся с более абстрактными вещами. Во многих ООП-программах существует так называемый "объект приложения", и это несмотря на то, что приложение в действительности не является тем, что мы можем представить в реальном мире. Тем не менее представление приложения в качестве объекта может оказаться весьма полезным, поскольку приложение само по себе обладает определенными свойствами и поведенческими характеристиками.

Слишком общий объект — это объект, который вообще не представляет никакого конкретного предмета. Программист (из самых лучших побуждений) пытается создать объект, чтобы он был гибким и многократно используемым, но иногда такое намерение может закончиться плачевно. Например, представим себе программу, которая организует и отображает средства аудиовизуальной информации. Она может каталогизировать ваши фотографии, формировать музыкальную коллекцию и служить в качестве дневника. Слишком общий подход — рассматривать все эти вещи объектами и строить один класс, который подойдет под все форматы. Этот класс может иметь свойство, именуемое "данные", которое будет содержать биты изображений, песен или записей дневника (формат зависит от типа средства аудиовизуальной информации). Класс может обладать поведенческой функцией, назовем ее "выполнить" (при ее вызове она отобразит фотографию, воспроизведет песню или выведет на экран запись дневника для редактирования).

Доказательство того, что этот класс слишком общий, содержится в самих именах свойств и поведенческих характеристик. Слово "данные" несет в себе мало информации: мы вынуждены использовать общий термин, поскольку этот класс перенапряжен тремя различными способами применения. Аналогично слово "выполнить" в каждом из трех наших случаев имеет уж слишком различные значения. Наконец, этот проект слишком общий еще и потому, что "средства аудиовизуальной информации" не подразумевают конкретный объект (ни в интерфейсе пользователя, ни в реальной жизни, ни даже в сознании программиста). Основная причина перебора в обобщении заключается в том, что наш гипотетический программист попытался объединить в одном объекте совершенно разные идеи (рис. 3.2).

Рис. 3.2

Отношения между объектами

Как программисту, вам наверняка придется встретиться с ситуациями, когда различные классы будут иметь общие характеристики или по крайней мере каким-то образом связаны друг с другом. Например, хотя создание "аудиовизуального" объекта для представления изображений, песен и текста в программе "цифрового каталога" можно обвинить в преувеличенном обобщении, эти объекты действительно обладают общими характеристиками. Для них всех существует дата и время последней модификации, и все они могут поддерживать функцию удаления.

Объектно-ориентированные языки программирования предоставляют ряд механизмов, в которых учитываются такие взаимоотношения между объектами. Труднее всего убедиться в реальности взаимоотношений. Различают два основных типа отношений между объектами: *HAS-A* и *IS-A*.

Отношения типа *HAS-A*

Объекты, связанные отношением типа *has-a*, или *агрегированием*, соответствуют следующему шаблону: объект A имеет (по-английски *has a*) объект B, или объект A содержит (или включает) объект B. При таком типе отношений один объект можно рассматривать как часть другого. Компоненты, как упоминалось выше, представляют отношение типа *has-a*, поскольку они описывают объекты, которые состоят из других объектов.

Примером такого типа отношений может служить отношение между зоопарком и обезьяной. Вы могли бы сказать, что в зоопарке есть обезьяна или, что то же самое, зоопарк включает обезьяну. При моделировании зоопарка с помощью кода мы создали бы объект "зоопарк", который бы включал компонент "обезьяна".

Зачастую при обдумывании сценариев пользовательского интерфейса полезно понимать, какие отношения существуют между объектами. Дело в том, что даже если не все пользовательские интерфейсы реализованы с использованием ООП (хотя, вероятно, таки большинство из них), визуальные элементы на экране нетрудно преобразовать в объекты. В качестве примера отношения агрегирования (типа *has-a*) для пользовательского интерфейса может служить окно, которое содержит кнопку. Кнопка и окно — это два совершенно различных объекта, но вполне очевидно, что они каким-то образом связаны между собой. Поскольку кнопка находится внутри окна, мы можем сказать, что окно включает кнопку.

Примеры отношений типа *has-a* показаны на рис. 3.3.

Отношение типа *IS-A* (наследование)

Отношение типа *is-a* представляет фундаментальный принцип объектно-ориентированного программирования, который обычно называется *наследованием* (inheriting). Классы позволяют смоделировать тот факт, что реальный мир содержит объекты, которые характеризуются свойствами и поведением. С помощью же наследования можно смоделировать тот факт, что эти объекты организованы иерархически. Сущность иерархий как раз и выражается в отношениях типа *is-a*[1].

По существу, наследование можно описать таким шаблоном: объект A — это вид объекта B, или объект A практически подобен объекту B. Для примера вернемся в наш "зоопарк", но предположим, что в нем, помимо обезьян, находятся и другие животные.

---

[1] *Обозначение* is-a *происходит от английской фразы* is a kind of *(это вид), конкретизирующей категорию отношений между двумя предметами. Например, бабочка — это вид насекомого, а автомобиль — это вид транспортного средства.* — Примеч. ред.

Уже одно только это предложение выражает смысл отношения: обезьяна — *это вид* животного, или даже проще: обезьяна — это животное. Аналогично жираф — это животное, кенгуру — это животное и пингвин — это животное. Ну и что из этого? Так вот, магия наследования проявляется как раз в тот момент, когда вы понимаете, что обезьян, жирафов, кенгуру и пингвинов объединяет нечно *общее*. Этим общим фактом для них являются характеристики животных в целом.

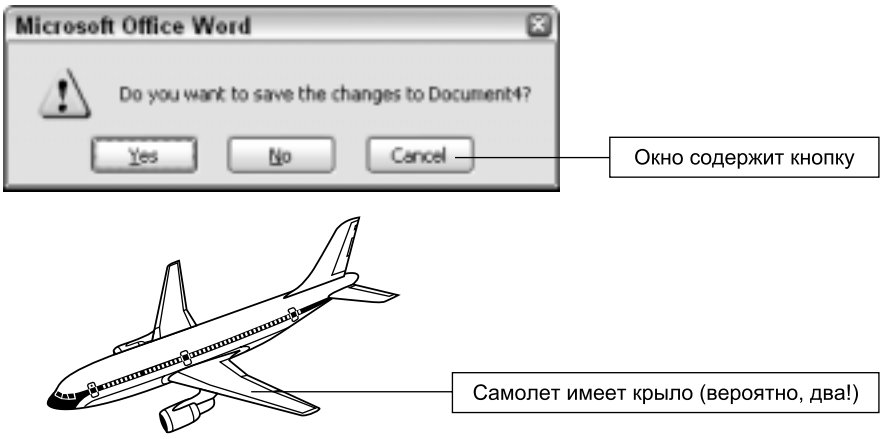


Рис. 3.3

Для программиста вышесказанное означает, что он может определить класс животных (`Animal`), который инкапсулирует все свойства (размер, обитание, питание и т.д.) и поведенческие характеристики (двигаться, есть, спать), присущие каждому животному. Конкретные животные, например обезьяны, образуют подклассы класса `Animal`, поскольку обезьяне свойственны все характеристики животного (вспомните: обезьяна — это животное с дополнительными характеристиками, которые выделяют ее из животных вообще). Схема наследования для класса животных показана на рис. 3.4. Стрелки обозначают направление отношения типа *is-a*.

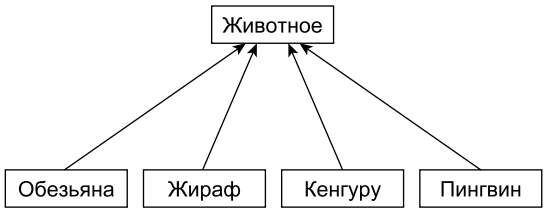


Рис. 3.4

Подобно тому как обезьяны и жирафы — различные виды животных, пользовательский интерфейс тоже может иметь различные типы кнопок. Например, кнопка-флажок (checkbox) — это вид кнопки (button). Предположим, что кнопка — это элемент пользовательского интерфейса, на котором можно щелкнуть и тем самым выполнить некоторое действие, тогда можно сказать, что класс `Checkbox` расширяет класс `Button` путем добавления в него состояния кнопки (с пометкой или нет).

Если предполагается, что между классами существует отношение наследования (типа *is-a*), то необходимо выявить "общую функциональность" *суперкласса*, т.е класса,

который позже планируется расширить за счет других классов (*подклассов*). Если вы считаете, что все ваши подклассы имеют код, подобный или в точности совпадающий с кодом суперкласса, подумайте, как с максимальной эффективностью использовать код суперкласса. В этом случае любые изменения придется вносить только в одном месте, и будущие подклассы "общую функциональность" получат "совершенно бесплатно".

Методы наследования

В предыдущих примерах мы продемонстрировали ряд методов наследования, но не формализовали их. Чтобы при создании подклассов определить, чем объект отличается от *родительского объекта* (или *суперкласса*), программист может использовать разные способы. Иногда, например, для этого достаточно завершить предложение: "A — это (вид) B, который...".

Расширение функциональных возможностей

Подкласс может "превосходить" своего родителя по дополнительным функциональным возможностям. Например, обезьяна — это животное, которое может качаться на ветвях деревьев. Помимо того, что класс `Monkey` обладает всеми функциональными возможностями класса `Animal`, классу `Monkey` также присуща такая поведенческая характеристика, как "умение качаться на ветвях деревьев".

Замена функций

В любом подклассе можно заменить одну из функций или полностью переопределить поведение его родителя. Например, большинство животных перемещаются в пространстве посредством ходьбы, поэтому вы могли бы внести в класс `Animal` поведенческую функцию движения и определить ее как ходьбу. Но как в таком случае быть с кенгуру? Ведь кенгуру — это животное, которое передвигается в пространстве не ходьбой, а "вприпрыжку". Все другие свойства и поведенческие характеристики суперкласса `Animal` применимы к подклассу `Kangaroo`. Поэтому для соответствия истине в подклассе иногда достаточно изменить одну из функций суперкласса (в данном случае функцию движения). Конечно же, если при создании подкласса окажется, что в нем необходимо заменить все функции суперкласса, то это, скорей всего, верный признак того, что вы создаете подкласс не от *того* суперкласса.

Добавление свойств

В подкласс также можно добавлять новые свойства (к тем, которые определены в суперклассе и унаследованы от него). Пингвин имеет все свойства животного, но для него также характерно такое свойство, как размер клюва.

Замена свойств

В C++ предусмотрена возможность переопределения свойств, подобная возможности переопределения поведенческих характеристик. Однако это следует делать только в редких случаях. Важно не путать понятие замены свойства с понятием присваивания свойствам подклассов различных значений. Например, все животные обладают свойством "питание", по которому можно судить о том, что они едят. Обезьяны едят бананы, пингвины — рыбу, однако здесь не идет речь о замене свойства "питание", а лишь о различных значениях, присваиваемых одному и тому же свойству в разных подклассах.

Полиморфизм в сравнении с многократным использованием кода

Полиморфизм означает, что объекты, которые имеют стандартный набор свойств и поведенческих характеристик, можно использовать взаимозаменяемо. Определение класса можно сравнить с контрактом между объектами и кодом, который взаимодействует

с ними. Так, по определению любой объект класса `Monkey` должен поддерживать свойства и поведенческие характеристики класса `Monkey`.

Понятие полиморфизма распространяется и на суперклассы. Поскольку все обезьяны — животные, то все объекты класса `Monkey` поддерживают свойства и функции класса `Animal`.

Полиморфизм — довольно привлекательная часть объектно-ориентированного программирования, поскольку здесь в действительности используются преимущества механизма наследования. При моделировании зоопарка мы могли бы программным путем "обойти" в цикле всех животных и каждого "привести в движение". Так как все животные являются членами класса `Animal`, то они знают, как им двигаться. У некоторых животных функция движения переопределена, но главное здесь то, что наша программа просто дает каждому животному команду двигаться, не заботясь о его типе, и каждое животное двигается известным ему одному способом.

Помимо полиморфизма, есть еще одна причина для создания подклассов. Речь идет об эффективном использовании существующего кода. Например, если вам нужен класс, который воспроизводит музыкальные произведения с эхо-эффектом, а ваш коллега уже написал класс для воспроизведения музыки, но без каких-либо эффектов, вы могли бы расширить возможности уже существующего класса, добавив в него новую функцию. Здесь по-прежнему применяется отношение типа *is-a* (ведь музыкальный проигрыватель с эхо-эффектом — это проигрыватель, оснащенный дополнительной функцией эхо-эффекта), однако эти классы не предназначены для взаимозаменяемого использования. В итоге вы получите два отдельных класса, которые могут использовать совершенно различные части программы (или даже совершенно различные программы), и тем самым счастливо избежите очередного изобретения колеса.

Как различить *HAS-A*- и *IS-A*-отношения

В реальном мире довольно просто классифицировать *has-a*- и *is-a*-отношения между объектами. Ведь никто не скажет, что апельсин включает фрукт, поскольку *апельсин — это фрукт*. Однако в программировании иногда нет такой ясности.

Рассмотрим гипотетический класс, который представляет некоторую хеш-таблицу. Хеш-таблица — это такая структура данных, которая позволяет преобразовать ключ в значение. Например, страховая компания могла бы использовать класс `Hashtable` для преобразования ID-номеров в имена своих клиентов, т.е. по заданному идентификационному номеру (ID) можно найти соответствующее имя. Таким образом, в терминах хеш-таблицы ID является *ключом*, а имя — *значением*.

В стандартной реализации хеш-таблицы каждому ключу соответствует единственное значение. Если ID-номер 14534 соответствует имени "Kleper, Scott", он не может также соответствовать имени "Kleper, Marni". Если вы попытаетесь для ключа, который уже имеет одно значение, добавить второе, то первое значение будет "стерто" (так реализовано по крайней мере в большинстве хеш-таблиц). Другими словами, если ID-номер 14534 преобразуется в имя "Kleper, Scott", а затем вы присваиваете ID-номеру 14534 имя "Kleper, Marni", то мистер Скотт (Scott) окажется незастрахованным (см. следующую последовательность инструкций, состоящую из двух обращений к функции гипотетической хеш-таблицы `enter()`, с отображением результата их выполнения в виде содержимого хеш-таблицы). Если обозначение `hash.enter` вызывает вопросы, то, не забегая вперед, т.е. в объектный синтаксис C++, скажем пока, что эту запись можно прочитать как "использование функции `enter` объекта `hash`".

```
hash.enter(14534, "Kleper, Scott");
```

Ключ	Значение
14534	“Kleper, Scott” [строка]

```
hash.enter(14534, "Kleper, Marni");
```

Ключ	Значение
14534	“Kleper, Marni” [строка]

Нетрудно представить использование структур данных, подобных хеш-таблице, но разрешающих существование нескольких значений для каждого ключа. В примере со страховой компанией некоторая семья могла бы иметь несколько имен, соответствующих одному и тому же ID-номеру. Поскольку такая структура данных очень напоминает хеш-таблицу, неплохо было бы воспользоваться этим фактом. Хеш-таблица может иметь только одно значение для ключа, но это значение может быть любым. Вместо строки значение могло бы представлять собой коллекцию (например, массив или список), содержащую несколько (!) значений для ключа. При каждом добавлении нового клиента для существующего ID-номера новое имя просто добавляется в такую коллекцию. Такое поведение демонстрируется на примере следующей последовательности.

```
Collection collection; // Создаем новую коллекцию.
collection.insert("Kleper, Scott"); // Добавляем
                    // в коллекцию новый элемент.
hash.enter(14534, collection); // Вводим коллекцию в таблицу.
```

Ключ	Значение
14534	{“Kleper, Scott”} [коллекция]

```
Collection collection = hash.get(14534);// Считываем
                                  // существующую коллекцию.
collection.insert("Kleper, Marni"); // Добавляем
                                 // в коллекцию новый элемент.
hash.enter(14534, collection); // Заменяем коллекцию
                               // ее обновленной версией.
```

Ключ	Значение
14534	{“Kleper, Scott”, “Kleper, Marni”} [коллекция]

Использование коллекции вместо строки для программиста довольно утомительно и требует включения повторяющихся фрагментов кода. Было бы лучше оформить поддержку ввода нескольких значений для одного ключа в отдельный класс, скажем, `MultiHash`. Класс `MultiHash` должен работать подобно классу `Hashtable`, за исключением того, что он мог бы негласно сохранять каждое значение как коллекцию строк, а не как одну строку. Очевидно, класс `MultiHash` каким-то образом связан с классом `Hashtable`, поскольку он по-прежнему использует для хранения данных хеш-таблицу. Неясным остается лишь то, отношением какого рода будут связаны эти классы: *is-a* или *has-a*?

Рассмотрим сначала отношение типа *is-a* и представим, что класс `MultiHash` является подклассом класса `Hashtable`. В нем, по-видимому, необходимо переопределить функцию, которая “отвечает” за добавление элементов в таблицу, чтобы она могла либо создать коллекцию и добавить новый элемент, либо извлечь существующую коллекцию и затем добавить в нее новый элемент. Потребуется также переопределить функцию, которая считывает значение по ключу. Она могла бы, например, сцепить все

значения для данного ключа в одну строку. Такой вариант проекта кажется вполне приемлемым. Несмотря на то что здесь налицо переопределение всех функций суперкласса, подкласс по-прежнему использует оригиналы функций своего родителя. Этот подход схематически показан на рис. 3.5.

А теперь рассмотрим отношение типа *has-a*. Класс `MultiHash` — это отдельный класс, но он *содержит* объект класса `Hashtable`. Вероятно, его интерфейс очень похож на интерфейс класса `Hashtable`, но он не должен быть одинаковым. Когда пользователь добавляет что-либо в класс `MultiHash`, это *что-то* в действительности попадает в коллекцию и помещается в объект класса `Hashtable`. Такой вариант проекта тоже кажется вполне приемлемым. Данный подход схематически показан на рис. 3.6.

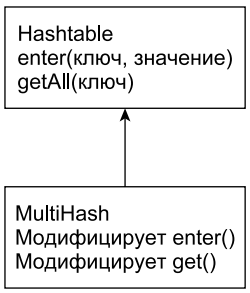


Рис. 3.5

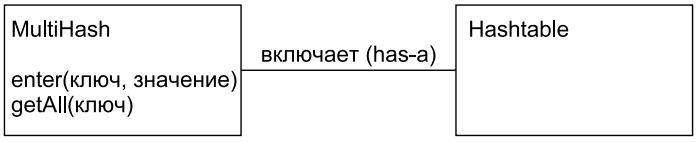


Рис. 3.6

Итак, какое же решение правильно? Здесь не существует однозначного ответа, хотя один из авторов этой книги (который написал класс `MultiHash` для коммерческого применения) видит в этой ситуации отношение типа *has-a*. Основной аргумент — позволить модификациям доступ к открытым интерфейсам, не беспокоясь о поддержке функционирования хеш-таблиц. Например, функция `get` была заменена функцией `getAll`, и теперь стало ясно, что она должна считывать все значения для конкретного ключа в классе `MultiHash`. Кроме того, при использовании отношения типа *has-a* вы вообще не должны волноваться о функционировании каких бы то ни было хеш-таблиц. Например, допустим, что класс хеш-таблицы поддерживает поведение, которое соответствует считыванию общего количества значений. В этом случае, если в классе `MultiHash` нет переопределения функций, он просто сообщит размер коллекций.

При этом можно найти убедительный аргумент в пользу того, что класс `MultiHash` в действительности является классом `Hashtable` с некоторыми новыми функциями, и, следовательно, должен находиться с ним в отношении типа *is-a*. Дело в том, что грань между двумя этими типами отношений иногда едва заметна, и вам придется хорошо подумать о том, как вы собираетесь использовать тот или иной класс, а также решить: то, построением чего вы занимаетесь, является простым использованием функций другого класса или классом с модифицированными или даже совсем новыми функциями.

В следующей таблице представлены аргументы "за" и "против" выбора каждого из этих подходов для класса `MultiHash`.

Отношение типа *NOT-A*

Обсуждая тип отношений между классами, подумайте о том, а существуют ли вообще между ними какие-либо отношения. Создание объектно-ориентированного проекта еще не означает "притягивания за уши" не нужных, по сути, отношений между классом и его под классами.

	Is-A	Has-A
Причины "за"	• По существу, эти два класса представляют собой одну и ту же абстракцию с различными характеристиками. • В классе `MultiHash` определяются (почти) те же функции, что и в классе `Hashtable`	• Класс `MultiHash` может иметь любые поведенческие характеристики, не заботясь о том, какие функции определены в классе `Hashtable`. Их реализацию можно было бы сделать отличной от функций класса `Hashtable`, не изменяя открытых интерфейсов
Причины "против"	• Хеш-таблица (класс `Hashtable`) по определению имеет однозначное соответствие между ключом и значением. Поэтому назвать класс `MultiHash` хеш-таблицей — значит грешить против истины. Класс `MultiHash` переопределяет как функции класса `Hashtable`, так сам признак, по которому можно судить о некорректности проекта. • Неизвестные или неподходящие свойства либо функции класса `Hashtable` могут "обескровить" класс `MultiHash`	• В известном смысле, класс `MultiHash` (со своими новыми функциями) служит примером еще одного изобретения колеса. • Некоторые дополнительные свойства и функции класса `Hashtable` могут оказаться полезными

Иногда может ввести в заблуждение очевидная связь, существующая между некоторыми вещами в реальном мире. Тот факт, что в реальной жизни "Мустанг" является моделью автомобильной марки "Форд" (компании Ford Motors), еще не означает, что при программном моделировании автомобиля класс `Mustang` обязательно должен быть подклассом класса `Ford`. При построении объектно-ориентированных иерархий необходимо моделировать *функциональные* отношения, а не искусственные. На рис. 3.7 показаны отношения, которые представляют интерес с точки зрения онтологии или иерархии, но вряд ли заслуживают выражения в коде.

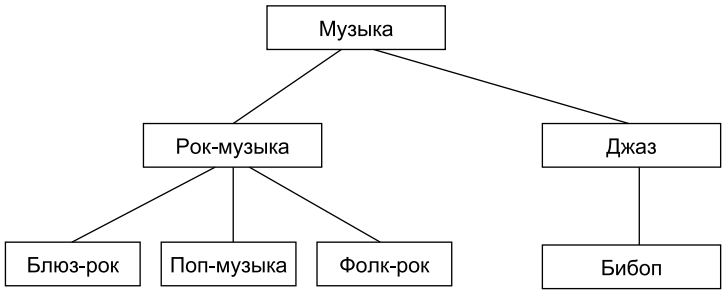
Лучший способ не допустить бесполезного образования подклассов — схематически изобразить проект. Для каждого класса и подкласса следует в этом случае указать, какие свойства и поведенческие характеристики вы планируете в них определить. Если вы увидите, что класс не имеет собственных конкретных свойств или функций, или все эти свойства и функции полностью переопределяются его подклассами, велика вероятность того, что в таком классе нет необходимости вообще.

Иерархии

Объектно-ориентированные иерархии позволяют моделировать многоуровневые отношения: если класс A может быть суперклассом класса B, то класс B при этом может быть суперклассом для класса C. Например, при моделировании зоопарка каждое животное можно представить как подкласс общего класса животных `Animal` (рис. 3.8).

При кодировании каждого из этих подклассов можно заметить, что многие из них чем-то сходны. В этом случае следует рассмотреть их "вхождение" в общего родителя. Если лев и пантера питаются и двигаются одинаково, это может означать, что для них

имеет смысл создать общий класс "больших кошек" `BigCat` (к нему можно отнести и других диких животных семейства кошачьих). При дальнейшем подразделении класса `Animal` можно создать такие подклассы, как `WaterAnimal` (морские животные) и `Marsupial` (сумчатые). Более развернутый иерархический проект модели зоопарка с учетом замеченной общности показан на рис. 3.9.



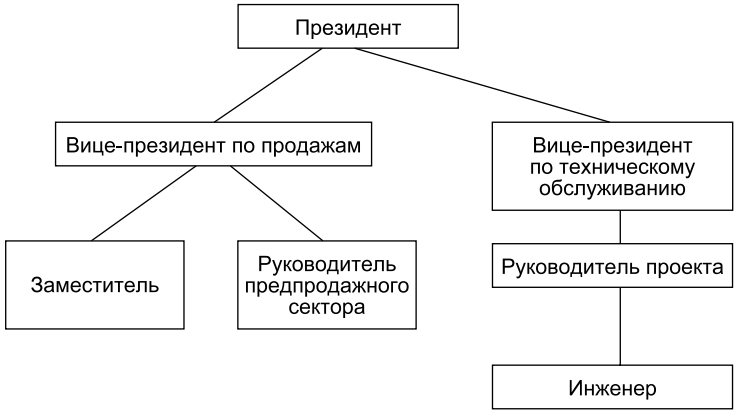


Рис. 3.7

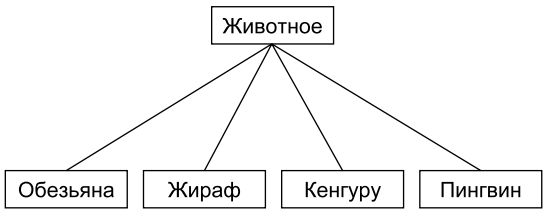


Рис. 3.8

Биолог, глядя на эту иерархию, может испытать чувство разочарования, поскольку пингвин в действительности не относится к тому же семейству, что и дельфин. Но это лишь подчеркивает тот факт, что в коде необходимо найти баланс

между реальными и общими функциональными отношениями. Даже если две вещи тесно связаны в реальном мире, они могут не иметь никаких "классовых" отношений в коде программы, поскольку в действительности они не обнаруживают общих поведенческих черт. Вы могли бы просто разделить животных на млекопитающихся и рыб, но такой подход не позволил бы выделить общие характеристики суперкласса.

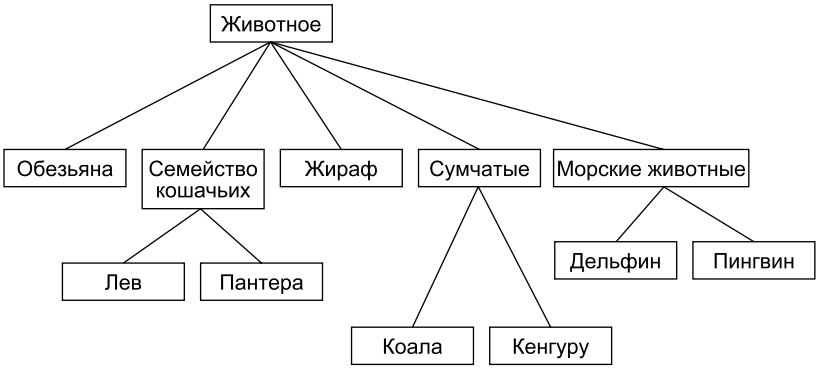


Рис. 3.9

Важно отметить, что существуют и другие способы организации иерархии. Предыдущий проект организован по большей части на основании характера движения животных. Если бы в основу иерархии положить способ питания животных или их рост, то иерархия выглядела бы по-другому. В конце концов важно то, как будут использоваться наши классы. Другими словами, способ построения объектной иерархии продиктован нашими потребностями.

Корректно построенная объектно-ориентированная иерархия позволяет достичь следующих результатов:

- классы организованы на основании существенных функциональных отношений;
- многократное использование кода достигается посредством выделения общих функций в суперкласс;
- при создании подклассов удается избежать переопределения многих родительских функций.

Множественное наследование

Во всех приводимых до сих пор примерах мы имели дело с единичным наследованием. Это означает, что класс имел, самое большее, один непосредственный родительский класс. Но благодаря множественному наследованию класс может иметь не один, а несколько суперклассов.

Если вы считаете, что в мире животных не существует надлежащей объектной иерархии, поскольку виды животных отличаются слишком по многим параметрам, можно обратиться к механизму множественного наследования. При множественном наследовании возможно создание нескольких отдельных иерархий. Для мира животных создайте три иерархии: размера, питания и движения, а затем для каждого животного определите его место в этой иерархической системе.

На рис. 3.10 показан проект иерархии с использованием множественного наследования. В нем по-прежнему существует суперкласс животных `Animal`, который затем подвергается делению по размеру. Другие две отдельные иерархии основаны на характере питания и движения животных. Каждый вид животного затем образует свой подкласс на базе всех этих трех суперклассов.

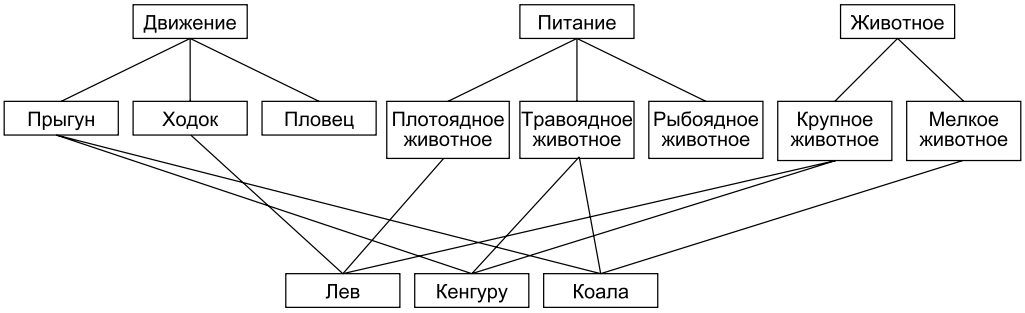


Рис. 3.10

В контексте пользовательского интерфейса представьте себе изображение, на котором пользователь может щелкнуть мышью. В этом случае данный объект должен быть одновременно как кнопкой, так и рисунком, чтобы при его реализации как подкласса у него было два родителя: класс `Image` и класс `Button` (рис. 3.11).

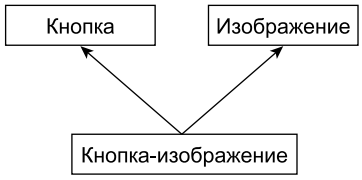


Рис. 3.11

Недостатки множественного наследования

Многие программисты не долюбливают множественное наследование, несмотря на то, что в C++ предусмотрена явная поддержка отношений такого вида (чего не скажешь о языке Java). И на это есть ряд причин.

Во-первых, возникают проблемы с визуализацией множественного наследования. Как показано на рис. 3.10, даже схема простого класса может оказаться очень сложной при наличии нескольких иерархий и пересекающихся линий, обозначающих отношения. Иерархии классов призваны помочь программисту понять отношения между различными частями кода. При использовании же множественного наследования класс может иметь нескольких родителей, которые никак не связаны друг с другом. И такое положение вещей может значительно усложнить представление программиста о том, что происходит с его объектом. В реальном мире мы, как правило, не рассматриваем существование между объектами множественных отношений типа *is-a*.

Во-вторых, множественное наследование может разрушить другие "четкие" иерархии. В примере с миром животных переход к использованию множественного наследования означает, что суперкласс `Animal` стал менее значительным, поскольку код, который описывает животных, теперь разделен на три отдельные иерархии. Несмотря на то что проект, показанный на рис. 3.10, демонстрирует три отчетливые иерархии, нетрудно

представить себе, как они смешаются в своих потомках. Например, как поступить, если окажется, что все прыгуны не только двигаются одинаково, но и употребляют одинаковую пищу? Поскольку мы имеем дело с тремя отдельными иерархиями, то не существует способа объединить принципы движения и питания, не добавив еще один подкласс.

В-третьих, сама реализация множественного наследования довольно сложна. А что если два из ваших суперклассов реализуют одинаковое поведение различными способами? Можно ли использовать два суперкласса, которые сами являются подклассами некоторого общего суперкласса? Эти ситуации усложняют реализацию, поскольку структурирование таких запутанных отношений в коде довольно затруднительно как для автора, так и для читателя.

Отсутствие встроенного механизма для реализации множественного наследования в других языках программирования объясняется тем, что программисты обычно избегают иметь с ним дело. Если вы решите пересмотреть нашу зоологическую иерархию или воспользоваться шаблонами проектирования, описанными в главе 26, вы сможете избежать применения множественного наследования при создании проекта программы.

"Смешанные" классы

"Смешанные" классы представляют еще один тип отношений. В C++ смешанный класс реализуется синтаксически подобно множественному наследованию, но с иной семантикой. Смешанный класс отвечает на вопрос: "Что еще данный класс способен сделать?". Смешанные классы представляют собой способ добавления функций в класс, не затрагивая отношение типа *is-a*.

Вернемся к примеру с зоопарком. Для некоторых животных мы можем ввести понятие "любимчик". Таких животных посетители могут баловать без вреда для себя. Функцию поведения для всех животных-любимчиков можно обозначить как "позволяет ласку". Поскольку "ласковые" животные не имеют общих черт, и вы не хотите разрушить уже созданную вами иерархию, то этот класс "любимчиков" (`Pettable`) можно отнести к категории смешанных.

Смешанные классы часто становятся членами пользовательских интерфейсов. Вместо применения множественного наследования к классу `PictureButton` (кнопка-изображение), чтобы наделить его признаками и кнопки (`Button`), и простого изображения (`Image`), можно заявить, что мы имеем дело с изображением, на котором можно щелкнуть (`Clickable`). А пиктограмма папки на вашем экране будет соответствовать изображению (`Image`), которое можно перетаскивать (`Draggable`). При разработке программных продуктов мы используем множество забавных прилагательных.

Разница между смешанным и суперклассом больше относится к тому, как мы думаем о классе, а не к различию в коде. В общем случае смешанные классы проще систематизировать, чем классы, созданные с использованием множественного наследования, поскольку их области видимости довольно ограничены. Создавая смешанный класс `Pettable`, к существующему классу мы просто добавили одну поведенческую функцию. Смешанный класс `Clickable` отличается от существующего только добавлением функций перетаскивания изображения "вниз" и "вверх". Кроме того, смешанные классы редко могут "похвастаться" развернутой иерархией, поэтому здесь вряд ли стоит опасаться перекрестного функционального "опыления".

Абстракция

В главе 2 вы узнали о принципе абстракции, т.е. понятии отделения реализации от средств, используемых для доступа к ней. Абстракция также является основной частью объектно-ориентированного проектирования.

Сравнение интерфейса с реализацией

Основное для абстракции — эффективно отделить *интерфейс* от *реализации*. Реализация — это код, который пишет программист для решения поставленной перед ним задачи. Интерфейс — это средство использования вашего кода другими людьми. В языке C в качестве примера интерфейса можно привести заголовочный файл, который описывает функции в созданной вами библиотеке. В объектно-ориентированном программировании интерфейс класса может иметь вид коллекции доступных для всех (т.е. открытых) свойств и функций.

Принятие решения по открытому интерфейсу

Как другие программисты будут взаимодействовать с вашими объектами — эта тема подлежит обсуждению при проектировании класса. В C++ свойства и функции (методы) класса могут быть открытыми (`public`), защищенными (`protected`) или закрытыми (`private`). Спецификатор `public` означает, что доступ к свойству или функции может получить любой другой (внешний для данной иерархии классов) код программы. Спецификатор `protected` означает невозможность получения доступа со стороны внешнего (для данной иерархии классов) кода. Спецификатор `private` обеспечивает еще более строгую защиту, чем спецификатор `protected`, поскольку в этом случае доступ к свойствам и функциям класса не могут получить даже подклассы данной иерархии.

Проектирование открытого интерфейса означает выбор спецификаторов для свойств и функций класса, точнее, какие члены класса сделать `public`-членами. Работая над большим проектом совместно с другими программистами, следует рассматривать проектирование открытого интерфейса как процесс.

Кто конечный пользователь интерфейса

Первый шаг в процессе проектирования открытого интерфейса — обсудить целевую аудиторию. Кто конечный пользователь интерфейса? Другой член вашей команды? Вы сами? Программист не из вашей компании? А, может быть, массовый потребитель или оффшорная фирма-исполнитель? Ответ на вопрос о конечном пользователе интерфейса позволит определить не только того, кто поможет вам в его проектировании, но также прольет свет на некоторые цели его разработки.

Если вы сами собираетесь использовать свой интерфейс, у вас есть полная свобода в работе над ним. Это значит, что вы вольны многократно изменять его до тех пор, пока он не станет отвечать вашим требованиям. И если вы — не кустарь-одиночка, а работаете в коллективе, то вполне вероятно, что в один прекрасный день вашим интерфейсом начнут пользоваться и другие члены вашей команды.

Проектирование интерфейса для других "внутренних" программистов несколько отлично от первого варианта. В известном смысле ваш интерфейс становится контрактом, "подписанным" между вами и ними. Например, если вы реализуете компонент хранения данных, остальные находятся в зависимости от того, как ваш интерфейс поддерживает определенные операции. Вам придется выяснить все аспекты использования вашего класса остальными членами вашей команды. Нужно ли при этом организовать управление версиями? Хранение данных какого типа должен обеспечить ваш компонент? В соответствии с контрактом вам не стоит предполагать слишком большую потенциальную гибкость для своего интерфейса. Если проект согласован до начала кодирования, но вы (после написания кода) вдруг решите внести в него изменения, вам не миновать жалоб от других программистов.

Если заказчиком интерфейса является внешний потребитель, вам придется его проектировать с учетом большого пакета различных требований. Привлечь конечного пользователя к выяснению деталей интерфейса было бы идеальным вариантом. При этом желательно рассмотреть конкретные потребности как сегодняшнего, так и завтрашнего дня. Терминология, используемая в интерфейсе, должна соответствовать терминам, с которыми знаком потребитель, да и документацию необходимо писать в расчете на конечного пользователя. А вот шуточные названия и программистский сленг в проекте использовать не следует.

Назначение интерфейса

Для написания интерфейса есть множество причин. Прежде чем доверить код бумаге или хотя бы решить, какие функции сделать открытыми, необходимо понять назначение интерфейса.

Программный интерфейс приложения (API)

Программный интерфейс приложения (application programming interface —API) это внешне видимый механизм, который позволяет распространить продукт или использовать его функции в другом контексте. Если внутренний интерфейс можно сравнить с контрактом, то API-интерфейс ближе к "высеченному из камня" закону. Если кто-то уже использует ваш API-интерфейс (пусть даже не работая в вашей компании), его не обрадуют изменения, если, конечно, вы не добавите новые средства, которые смогут ему быть полезны. Поэтому тщательно распланируйте API-характеристики и обсудите их с потенциальными пользователями прежде, чем делать интерфейс доступным для них.

При проектировании API-интерфейса обычно пытаются найти компромисс между простотой применения и гибкостью. Поскольку конечные пользователи интерфейса не знакомы с внутренней работой вашего продукта, то нужно учитывать трудности его освоения. Прежде всего, ваша компания создает этот API-интерфейс для потребителей, поскольку желает, чтобы он был востребован. И если с ним будет трудно работать, то он обречен на неудачу. Гибкость же часто находится в противоречии с требованием простоты применения. Ваш продукт может иметь множество различных режимов работы, и вполне закономерно, что вы захотите, чтобы потребитель мог использовать все заложенные в нем функции. Но API-интерфейс, который позволяет потребителю делать все, на что способен ваш продукт, будет довольно сложным в обращении.

У программистов есть такая поговорка: "Хороший интерфейс делает простой случай простым, а трудный — возможным". Это значит, что API-интерфейсы, прежде всего, должны быть простыми в освоении. Все варианты использования продукта должны быть доступными для пользователя. При этом любой API должен быть рассчитан и на более интеллектуальное использование, поэтому всегда нужно стремиться к компромиссу между сложностью его применения для редко используемых режимов работы и простотой для обычных и частых.

Вспомогательный класс или библиотека

Нередко ваша задача заключается в разработке некоторых конкретных функций приложения, например, библиотеки случайных чисел или класса регистрации. В таких случаях с интерфейсом вопросы решаются проще, поскольку вы стремитесь сделать открытыми для пользователей большинство или даже все функции, не раскрывая деталей реализации. Здесь важно обсудить вопрос уровня обобщения. Поскольку класс или библиотека имеют уровень общего назначения, то при создании проекта приложения вам придется рассмотреть весь возможный набор вариантов применения вашего продукта.

Интерфейс между подсистемами

Возможно, вам придется проектировать интерфейс между двумя основными подсистемами приложения, например механизм доступа к базе данных. В подобных случаях отделение интерфейса от реализации первостепенно, поскольку, вполне вероятно, что другие программисты начнут заниматься реализацией их части еще до того, как вы "разделаетесь" со своей. Работая над подсистемой, прежде всего, необходимо думать о том, каково ее основное назначение. Идентифицировав главную задачу подсистемы, подумайте о конкретных вариантах ее применения, а также о том, как она должна быть представлена другим частям кода. Попытайтесь взглянуть на нее "глазами" других задач, не углубляясь в детали реализации.

Интерфейс между компонентами

Большинство интерфейсов по своим функциональным возможностям не превышают подсистемный или API-интерфейс. Другими словами, большинство интерфейсов представляют собой объекты, которые используются внутри вашей же программы. В таких случаях основную опасность таит в себе тот момент, когда ваш интерфейс, развиваясь постепенно, вдруг становится неуправляемым. Несмотря на то что подобные интерфейсы предназначены для "внутреннего употребления", лучше думать о них как об "экспортном продукте". Другими словами, рассмотрите (как и при использовании подсистемного интерфейса) по одной основной цели существования каждого класса и осторожно отнеситесь к тому, чтобы функции, которые не связаны с этой целью, делать открытыми.

Думая о будущем

Проектируя интерфейс, не нужно забывать о том, что "день грядущий нам готовит", т.е. о будущих потребностях. Будет ли этот проект годами оставаться неприкосновенным? А, может быть, стоит предусмотреть возможность расширения, ориентируясь на сменную архитектуру? Возможно, вы уверены в том, что ваш интерфейс можно использовать не только по назначению, но и в других целях? Если существует вероятность его альтернативной переделки впоследствии или, что еще хуже, добавления новых функций вразбивку, то из вашего интерфейса может получиться чудовищный коктейль! В этом случае будьте особенно бдительны! Предполагаемое обобщение — это еще одна ловушка. Не стоит проектировать "всеядный" класс токмо для пущей важности, если его будущее назначение пока не ясно.

Проектирование абстракции

Настоящий успех в проектировании хороших во всех отношениях интерфейсов приходит лишь после нескольких лет получения опыта в написании своих и использовании чужих абстракций. Изучая абстракции, написанные другими программистами, старайтесь запомнить, что в них было работоспособным, а что — нет. Чего вам недоставало в API файловой системы Windows, которую вы использовали на прошлой неделе? Что бы вы сделали по-другому, если бы не вашему коллеге, а именно вам пришлось писать сетевой упаковщик? С первого раза интерфейс вряд ли получится превосходным, поэтому дерзайте еще и еще раз. Не бойтесь получить отзыв о своей работе у более опытных коллег. Не бойтесь также что-либо изменить в своей абстракции после начала кодирования, даже если другим программистам придется адаптироваться к новому варианту. В конце концов они поймут, что хорошая абстракция выгодна не только для вас, но и для них.

Порой не обойдется без нервов, когда вы будете связывать свою программу с кодом других программистов. Возможно, ваши коллеги по команде не понимают проблемы,

которая обнаружилась в предыдущем варианте интерфейса, или не осознают, что ваш подход требует дополнительных усилий. В подобных ситуациях будьте готовы как защищать свою работу, так и воспользоваться их идеями (если они того заслуживают). В критических ситуациях может помочь хорошо составленная документация и примеры программ.

Остерегайтесь использования абстракций, состоящих из одного класса. Если глубина создаваемого вами кода довольно значительна, рассмотрите возможность создания классов-попутчиков, которые могут составить "компанию" основному интерфейсу. Например, если ваш интерфейс ориентирован на некоторую обработку данных, то почему бы вам не предусмотреть также существование результирующего объекта, который бы позволял просматривать и интерпретировать полученные результаты?

Там, где это возможно, преобразуйте свойства в функции. Другими словами, не позволяйте внешнему коду напрямую манипулировать данными "за спиной" вашего класса. Вы ведь не хотели бы, чтобы кто-нибудь из программистов (по неаккуратности или в силу своей испорченности) установил высоту объекта визуализации равной отрицательному числу? Лучше для установки подобных значений использовать функции, которые выполняют граничную проверку (на отсутствие нарушения границ).

Не бойтесь возвращаться к "пройденному". Стремитесь получить отзывы на свой проект, учитывайте замечания и учитесь на ошибках (необязательно на своих).

Более конкретные рекомендации по проектированию интерфейсов и возможностям многократного использования кода можно найти в главе 5.

Резюме

В этой главе вы получили представление о проектировании объектно-ориентированных программ (хотя и без рассмотрения соответствующих примеров). Изложенные здесь принципы реализуемы практически во всех объектно-ориентированных языках программирования. Надеемся, вы узнали для себя новые способы формализации уже известных идей и познакомились с новыми подходами к решению старых проблем или с новыми аргументами в пользу некоторых концепций командной работы. Даже если вам никогда не приходилось прежде использовать объекты в своих программах или вы делали это фрагментарно, то теперь о том, как спроектировать объектно-ориентированную программу, вы знаете больше, чем многие уже опытные C++-программисты.

Очень важно досконально изучить отношения между объектами, причем не только потому, что хорошо связанные объекты позволяют использовать код многократно и "навести порядок" в коде, но и потому, что вам обязательно придется работать в команде. Опыт показывает, что связанные объекты проще поддерживать и модифицировать. Поэтому, приступая к проектированию своих программ, используйте раздел "Отношения между объектами" в качестве руководства к действию.

Наконец, вы познакомились с теоретическими основами успешного построения абстракций и поняли, что, принимая решение по созданию открытого интерфейса, необходимо до конца уяснить его назначение и определиться с тем, кто будет его конечным пользователем. В главе 4 вы получите более расширенное представление о разработке абстракций, а именно о создании многократно используемого кода, о повторном использовании идей, а также о применении некоторых библиотек.