Глава 2

КАК МЫСЛИТЬ ОБЪЕКТНО

В главе 1 вы изучили фундаментальные объектно-ориентированные концепции. В остальной части этой книги мы тщательнее разберем эти концепции и познакомимся с некоторыми другими. Для грамотного подхода к проектированию необходимо учитывать много факторов, независимо от того, идет ли речь об объектно-ориентированном проектировании или каком-то другом. В качестве основной единицы при объектно-ориентированном проектировании выступает класс. Желаемым конечным результатом такого проектирования является надежная и функциональная объектная модель, другими словами, полная система.

Как и в случае с большинством вещей в жизни, нет какого-то одного правильного или ошибочного подхода к устранению проблем. Обычно бывает много путей решения одной и той же проблемы. Поэтому, пытаясь выработать объектно-ориентированное решение, не зацикливайтесь на том, чтобы постараться с первого раза все идеально спроектировать (кое-что всегда можно будет усовершенствовать). В действительности вам потребуется прибегнуть к мозговому штурму и позволить мыслительному процессу пойти в разных направлениях. Не старайтесь соответствовать каким-либо стандартам или соглашениям, пытаясь решить проблему, поскольку важно лишь быть креативным.

Фактически на самом старте процесса не стоит даже начинать задумываться о конкретном языке программирования. Первым пунктом повестки дня должно быть определение и решение бизнес-проблем. Займитесь сперва концептуальным анализом и проектированием. Задумывайтесь о конкретных технологиях, только если они будут существенны для решения бизнес-проблем. Например, нельзя спроектировать беспроводную сеть без беспроводной технологии. Однако часто будет случаться так, что вам придется обдумывать сразу несколько программных решений.

Таким образом, перед тем как приступать к проектированию системы или даже класса, следует поразмыслить над соответствующей задачей и повеселиться! В этой главе мы рассмотрим изящное искусство и науку объектно-ориентированного мышления.

Любое фундаментальное изменение в мышлении не является тривиальным. Например, ранее много говорилось о переходе со структурной разработки на

объектно-ориентированную. Один из побочных эффектов ведущихся при этом дебатов заключается в ошибочном представлении, что структурная и объектно-ориентированная разработки являются взаимоисключающими. Однако это не так. Как мы уже знаем из нашего исследования оберток, структурная и объектно-ориентированная разработки сосуществуют. Фактически, создавая объектно-ориентированное приложение, вы повсеместно используете структурные конструкции. Мне никогда не доводилось видеть программу, объектно-ориентированную или любую другую, которая не задействует циклы, операторы `if` и т. д. Кроме того, переход на объектно-ориентированное проектирование не потребует каких-либо затрат.

Чтобы перейти с FORTRAN на COBOL или даже C, вам потребуется изучить новый язык программирования, однако для перехода с COBOL на C++, C# .NET, Visual Basic .NET, Objective-C или Java вам придется освоить новое мышление. Здесь всплывает избитое выражение *объектно-ориентированная парадигма*. При переходе на объектно-ориентированный язык вам сначала потребуется потратить время на изучение объектно-ориентированных концепций и освоение соответствующего мышления. Если такая смена парадигмы не произойдет, то случится одна из двух вещей: либо проект не окажется по-настоящему объектно-ориентированным по своей природе (например, он будет задействовать C++ без использования объектно-ориентированных конструкций), либо он окажется полной объектно-неориентированной неразберихой.

В этой главе рассматриваются три важные вещи, которые вы можете сделать для того, чтобы хорошо освоить объектно-ориентированное мышление:

- знать разницу между интерфейсом и реализацией;
- мыслить более абстрактно;
- обеспечивать для пользователей минимальный интерфейс из возможных.

Мы уже затронули некоторые из этих концепций в главе 1, а теперь разберемся в них более подробно.

Разница между интерфейсом и реализацией

Как мы уже видели в главе 1, один из ключей к грамотному проектированию — понимание разницы между интерфейсом и реализацией. Таким образом, при проектировании класса важно определить, что пользователю требуется знать, а что нет. Механизм сокрытия данных, присущий инкапсуляции, представляет собой инструмент, позволяющий скрывать от пользователей несущественные данные.

Помните пример с тостером из главы 1? Тостер или любой электроприбор, если на то пошло, подключается к интерфейсу, которым является электрическая розетка (рис. 2.1). Все электроприборы получают доступ к необходимому

электричеству через электрическую розетку, которая соответствует нужному интерфейсу. Тостеру не нужно что-либо знать о реализации или о том, как вырабатывается электричество. Для него важно лишь то, чтобы работающая на угле или атомная электростанция могла вырабатывать электричество, — этому электроприбору все равно, какая из станций будет делать это, при условии, что интерфейс работает соответствующим образом, то есть корректно и надежно.

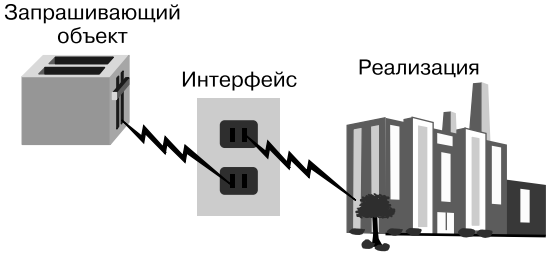

Рис. 2.1. Повторно приведенный пример с электростанцией

ПРЕДОСТЕРЕЖЕНИЕ

Не путайте концепцию интерфейса с терминами вроде «графический интерфейс пользователя» (GUI — Graphical User Interface). Несмотря на то что графический интерфейс пользователя, как видно из его названия, представляет собой интерфейс, используемый здесь термин является более общим по своей природе и не ограничивается понятием графического интерфейса.

В качестве другого примера рассмотрим автомобиль. Интерфейс между вами и автомобилем включает такие компоненты, как руль, педаль газа, педаль тормоза и переключатель зажигания. Когда речь идет об управлении автомобилем, для большинства людей, если отбросить вопросы эстетики, главным является то, как он заводится, разгоняется, останавливается и т. д. Реализация, чем, по сути, является то, чего вы не видите, мало интересует среднестатистического водителя. Фактически большинство людей даже не способно идентифицировать определенные компоненты, например каталитический преобразователь или сальник. Однако любой водитель узнает руль и будет в курсе, как его использовать, поскольку это общий интерфейс. Устанавливая стандартный руль в автомобилях, производители могут быть уверены в том, что люди из их потенциального круга покупателей смогут использовать выпускаемую ими продукцию.

Однако если какой-нибудь производитель решит установить вместо руля джойстик, то большинство водителей будут разочарованы, а продажи таких автомобилей могут оказаться низкими (подобная замена устроит разве что отдельных эклектиков, которым нравится «двигаться против течения»). С другой стороны, если мощность и эстетика не изменятся, то среднестатистический водитель

ничего не заметит, даже если производитель изменит двигатель (часть реализации) выпускаемых автомобилей.

Отметим, что заменяемые двигатели должны быть идентичны, насколько это возможно во всех отношениях. Замена четырехцилиндрового двигателя на восьмицилиндровый изменит правила и, вероятно, не будет работать с другими компонентами, которые взаимодействуют с двигателем, точно так же как изменение тока с переменного на постоянный будет влиять на правила в примере силовой установки.

Двигатель является частью реализации, а руль — частью интерфейса. Изменения в реализации не должны оказывать влияния на водителя, в то время как изменения в интерфейсе могут это делать. Водитель заметил бы эстетические изменения руля, даже если бы тот функционировал так же, как и раньше. Необходимо подчеркнуть, что изменения в двигателе, *заметные* для водителя, нарушают это правило. Например, изменение, которое приведет к заметной потери мощности, в действительности будет изменением интерфейса.

ЧТО ВИДЯТ ПОЛЬЗОВАТЕЛИ

Когда в этой главе речь идет о пользователях, под ними подразумеваются проектировщики и разработчики, а не обязательно конечные пользователи. Таким образом, когда мы говорим об интерфейсах в этом контексте, речь идет об интерфейсах классов, а не о графических интерфейсах пользователей.

Должным образом сконструированные классы состоят из двух частей — интерфейса и реализации.

Интерфейс

Услуги, предоставляемые конечным пользователям, образуют интерфейс. В наиболее благоприятном случае конечным пользователям предоставляются *только* те услуги, которые им необходимы. Разумеется, то, какие услуги требуются определенному конечному пользователю, может оказаться спорным вопросом. Если вы поместите десять человек в одну комнату и попросите каждого из них спроектировать что-то независимо от других, то получите десять абсолютно разных результатов проектирования — и в этом не будет ничего плохого. Однако, как правило, интерфейс класса должен содержать то, что нужно знать пользователям. Если говорить о примере с тостером, то им необходимо знать только то, как подключить прибор к интерфейсу (которым в данном случае является электрическая розетка) и как эксплуатировать его.

ОПРЕДЕЛЕНИЕ ПОЛЬЗОВАТЕЛЕЙ

Пожалуй, наиболее важный момент при проектировании класса — определение его аудитории, или пользователей.

Реализация

Детали реализации скрыты от пользователей. Один из аспектов, касающихся реализации, которые нужно помнить, заключается в следующем: изменения в реализации *не должны* требовать внесения изменений в пользовательский код. В какой-то мере это может показаться сложным, однако в выполнении этого условия заключается суть соответствующей задачи проектирования.

КАЧЕСТВО ИНТЕРФЕЙСА

Если интерфейс спроектирован правильно, то изменения в реализации не должны требовать изменений пользовательского кода.

Помните, что интерфейс включает синтаксис для вызова методов и возврата значений. Если интерфейс не претерпит изменений, то пользователям будет все равно, изменится ли реализация. Важно лишь то, чтобы программисты смогли использовать аналогичный синтаксис и извлечь аналогичное значение.

Мы сталкиваемся с этим постоянно, когда пользуемся сотовыми телефонами. Интерфейс, применяемый для звонка, прост: мы либо набираем номер, либо выбираем тот, что имеется в адресной книге. Кроме того, если оператор связи обновит программное обеспечение, то это не изменит способ, которым вы совершаете звонки. Интерфейс останется прежним независимо от того, как изменится реализация. Однако я могу представить себе ситуацию, что оператор связи изменил интерфейс: это произойдет, если изменится код города. При изменении основного интерфейса, как и кода города, пользователям придется действовать уже по-другому. Бизнес старается сводить к минимуму изменения такого рода, поскольку некоторым клиентам они не понравятся или, возможно, эти люди не захотят мириться с трудностями.

Напомню пример с тостером: хотя интерфейсом всегда является электрическая розетка, реализация может измениться с работающей на угле электростанции на атомную, никак не повлияв на тостер. Здесь следует сделать одну важную оговорку: работающая на угле или атомная электростанция тоже должна соответствовать спецификации интерфейса. Если работающая на угле электростанция обеспечивает питание переменного тока, а атомная — питание постоянного тока, то возникнет проблема. Основной момент здесь заключается в том, что и потребляющее электроэнергию устройство, и реализация должны соответствовать спецификации интерфейса.

Пример интерфейса/реализации

Создадим простой (пусть и не очень функциональный) класс `DataBaseReader`. Мы напишем Java-код, который будет извлекать записи из базы данных. Как уже говорилось ранее, знание своих конечных пользователей всегда является

наиболее важным аспектом при проектировании любого рода. Вам следует провести анализ ситуации и побеседовать с конечными пользователями, а затем составить список требований к проекту. Далее приведены требования, которые нам придется учитывать при создании класса `DataBaseReader`. У нас должна быть возможность:

- открывать соединение с базой данных;
- закрывать соединение с базой данных;
- устанавливать указатель над первой записью в базе данных;
- устанавливать указатель над последней записью в базе данных;
- узнавать количество записей в базе данных;
- определять, есть ли еще записи в базе данных (если мы достигнем конца);
- устанавливать указатель над определенной записью путем обеспечения ключа;
- извлекать ту или иную запись путем обеспечения ключа;
- извлекать следующую запись исходя из позиции курсора.

Учитывая все эти требования, можно предпринять первую попытку спроектировать класс `DataBaseReader`, написав возможные интерфейсы для наших конечных пользователей.

В данном случае класс `DataBaseReader` предназначается для программистов, которым требуется использовать ту или иную базу данных. Таким образом, интерфейс, в сущности, будет представлять собой интерфейс программирования приложений (API — Application Programming Interface), который станут использовать программисты. Соответствующие методы в действительности будут обертками, в которых окажется заключена функциональность, обеспечиваемая системой баз данных. Зачем нам все это нужно? Мы намного подробнее исследуем этот вопрос позднее в текущей главе, а короткий ответ звучит так: нам необходимо сконфигурировать кое-какую функциональность, связанную с базой данных. Например, нам может потребоваться обработать объекты для того, чтобы мы смогли записать их в реляционную базу данных. Создание соответствующего *промежуточного программного обеспечения* — непростая задача, поскольку предполагает проектирование и написание кода, однако представляет собой реальный пример обертывания функциональности. Более важно то, что нам может потребоваться изменить само ядро базы данных, но при этом не придется вносить изменения в код.

На рис. 2.2 показана диаграмма класса, представляющая возможный интерфейс для класса `DataBaseReader`.

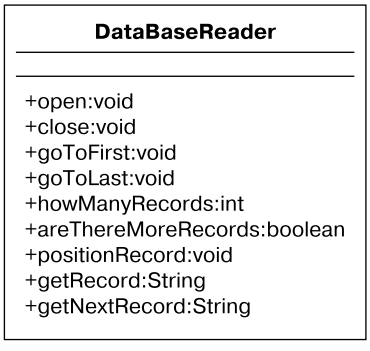

Рис. 2.2. UML-диаграмма класса DataBaseReader

Обратите внимание, что методы в этом классе открытые (помните, что рядом с именем методов, являющихся открытыми интерфейсами, присутствует знак плюс). Следует также отметить, что представлен только интерфейс, а реализация не показана. Уделите минуту на то, чтобы выяснить, отвечает ли в целом эта диаграмма класса требованиям к проекту, намеченным ранее. Если впоследствии вы обнаружите, что эта диаграмма отвечает не всем требованиям, то в этом не будет ничего плохого; помните, что объектно-ориентированное проектирование — итеративный процесс, поэтому вам необязательно стараться сделать все именно с первой попытки.

ОТКРЫТЫЙ ИНТЕРФЕЙС

Помните, что если метод является открытым, то прикладные программисты смогут получить к нему доступ, и, таким образом, он будет считаться частью интерфейса класса. Не путайте термин «интерфейс» с ключевым словом `interface`, используемым в Java и .NET. На эту тему мы поговорим в последующих главах.

Для каждого из приведенных ранее требований нам необходим метод, обеспечивающий желаемую функциональность. Теперь нам нужно задать несколько вопросов.

- Чтобы эффективно использовать этот класс, нужно ли вам как программисту еще что-нибудь знать о нем?
- Нужно ли знать о том, как внутренний код базы данных открывает ее?
- Требуется ли знать о том, как внутренний код базы данных физически выбирает определенную запись?
- Нужно ли знать о том, как внутренний код базы определяет то, остались ли еще записи?

Ответом на все эти вопросы будет громогласное «нет»! Вам не нужно знать что-либо из этой информации. Важно лишь получить соответствующие возвращаемые значения, а также то, что операции выполняются корректно. Кроме того, прикладные программисты, скорее всего, «будут на отдалении» как минимум еще одного абстрактного уровня от реализации. Приложение воспользуется вашими классами для открытия базы данных, что, в свою очередь, приведет к вызову соответствующего API-интерфейса для доступа к этой базе данных.

МИНИМАЛЬНЫЙ ИНТЕРФЕЙС

Один из способов, пусть даже экстремальных, определить минимальный интерфейс заключается в том, чтобы изначально не предоставлять пользователю никаких открытых интерфейсов. Разумеется, соответствующий класс будет бесполезным; однако это заставит пользователя вернуться к вам и сказать: «Эй, мне нужна эта функциональность». Тогда вы сможете начать переговоры. Таким образом, у вас есть возможность добавлять интерфейсы, только когда они запрашиваются. Никогда не предполагайте, что определенному пользователю что-то требуется.

Может показаться, что создание оберток — это перебор, однако их написание несет в себе множество преимуществ. Например, на рынке сегодня присутствует большое количество промежуточных продуктов. Рассмотрим проблему отображения объектов в реляционную базу данных. Некоторые объектно-ориентированные базы данных могут идеально подходить для объектно-ориентированных приложений. Однако есть маленькая проблема: у большинства компаний имеется множество данных в унаследованных системах управления реляционными базами. Как та или иная компания может использовать объектно-ориентированные технологии и быть передовой, сохраняя при этом свою информацию в реляционной базе данных?

Во-первых, вы можете преобразовать все свои унаследованные реляционные базы данных в совершенно новые объектно-ориентированные базы данных. Однако любому, кто испытывает острую боль от преобразования каких-либо данных, известно, что этого следует избегать любой ценой. Хотя на такие преобразования может уйти много времени и сил, зачастую они вообще не приводят к требуемым результатам.

Во-вторых, вы можете воспользоваться промежуточным продуктом для того, чтобы без проблем отобразить объекты, содержащиеся в коде вашего приложения, в реляционную модель. Это более верное решение. Некоторые могут утверждать, что объектно-ориентированные базы данных намного эффективнее подходят для обеспечения постоянства объектов, чем реляционные базы данных. Фактически многие системы разработки без проблем обеспечивают такую функциональность.

ПОСТОЯНСТВО ОБЪЕКТОВ

Постоянство объектов относится к концепции сохранения состояния того или иного объекта для того, чтобы его можно было восстановить и использовать позднее. Объект, не являющийся постоянным, по сути, «умирает», когда оказывается вне области видимости. Состояние объекта, к примеру, можно сохранить в базе данных.

В современной бизнес-среде отображение из реляционных баз данных в объектно-ориентированные — отличное решение. Многие компании интегрировали эти технологии. Компании используют распространенный подход, при котором внешний интерфейс сайта вместе с данными располагается на мейнфрейме.

Если вы создаете полностью объектно-ориентированную систему, то практичным (и более производительным) вариантом может оказаться объектно-ориентированная база данных. Вместе с тем объектно-ориентированные базы данных даже близко нельзя назвать такими же распространенными, как объектно-ориентированные языки программирования.

АВТОНОМНОЕ ПРИЛОЖЕНИЕ

Даже при создании нового объектно-ориентированного приложения с нуля может оказаться нелегко избежать унаследованных данных. Даже новое созданное объектно-ориентированное приложение, скорее всего, не будет автономным, и ему, возможно, потребуется обмениваться информацией, располагающейся в реляционных базах данных (или, собственно говоря, на любом другом устройстве накопления данных).

Вернемся к примеру с базой данных. На рис. 2.2 показан открытый интерфейс, который касается соответствующего класса, и ничего больше. Когда этот класс будет готов, в нем, вероятно, окажется больше методов, при этом он, несомненно, будет включать атрибуты. Однако вам как программисту, который будет использовать этот класс, не потребуется что-либо знать о его закрытых методах и атрибутах. Вам, безусловно, не нужно знать, как выглядит код внутри его открытых методов. Вам просто понадобится быть в курсе, как взаимодействовать с интерфейсами.

Как выглядел бы код этого открытого интерфейса (допустим, мы начнем с примера базы данных Oracle)? Взглянем на метод `open()`:

```
public void open(String Name){

        /* Некая специфичная для приложения обработка */

        /* Вызов API-интерфейса Oracle для открытия базы данных */

        /* Еще некая специфичная для приложения обработка */

};
```

В данной ситуации вы, выступая в роли программиста, понимаете, что методу `open` в качестве параметра требуется `String`. Объект `Name`, который представляет файл базы данных, передается, однако пояснение того, как `Name` отображается в определенную базу данных в случае с этим примером, не является важным. Это все, что нам нужно знать. А теперь переходим к самому увлекательному — что в действительности делает интерфейсы такими замечательными!

Чтобы досадить нашим пользователям, изменим реализацию базы данных. Представим, что вчера вечером мы преобразовали всю информацию из базы Oracle в информацию базы SQL Anywhere (при этом нам пришлось вынести острую боль). Эта операция заняла у нас несколько часов, но мы справились с ней.

Теперь код выглядит так:

```
public void open(String Name){

        /* Некая специфичная для приложения обработка */

        /* Вызов API-интерфейса SQL Anywhere для открытия базы данных */

        /* Еще некая специфичная для приложения обработка */

};
```

К нашему великому разочарованию, этим утром не поступило ни одной жалобы от пользователей. Причина заключается в том, что, хотя реализация изменилась, интерфейс не претерпел изменений! Что касается пользователей, то совершаемые ими вызовы остались такими же, как и раньше. Изменение кода реализации могло бы потребовать довольно много сил (а модуль с однострочным изменением кода пришлось бы перестраивать), однако не понадобилось бы изменять ни одной строки кода приложения, который задействует класс `DataBaseReader`.

ПЕРЕКОМПИЛЯЦИЯ КОДА

Динамически загружаемые классы загружаются во время выполнения, а не являются статически связанными с исполняемым файлом. При использовании динамически загружаемых классов, как, например, в случае с Java и .NET, не придется перекомпилировать ни один из пользовательских классов. Однако в языках программирования со статическим связыванием, например C++, для добавления нового класса потребуется связь.

Разделяя интерфейс пользователя и реализацию, мы сможем избежать головной боли в будущем. На рис. 2.3 реализации баз данных прозрачны для конечных пользователей, видящих только интерфейс.

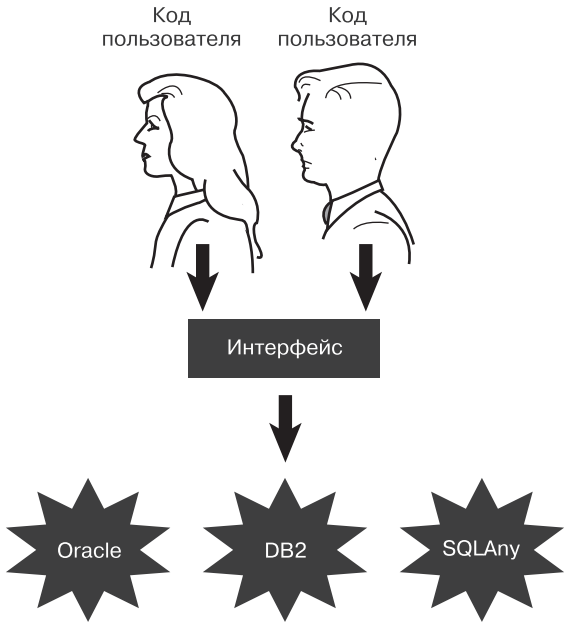

Рис. 2.3. Интерфейс

Использование абстрактного мышления при проектировании классов

Одно из основных преимуществ объектно-ориентированного программирования состоит в том, что классы можно использовать повторно. Пригодные для повторного применения классы обычно располагают интерфейсами, которые больше абстрактны, нежели конкретны. Конкретные интерфейсы склонны быть весьма специфичными, в то время как абстрактные являются более общими. Однако не всегда можно утверждать, что очень абстрактный интерфейс более полезен, чем очень конкретный, пусть это часто и является верным.

Можно создать очень полезный конкретный класс, который окажется вообще непригодным для повторного использования. Это случается постоянно, но в некоторых ситуациях в этом нет ничего плохого. Однако мы сейчас ведем речь о проектировании и хотим воспользоваться преимуществами того, что предлагает нам объектно-ориентированный подход. Поэтому наша цель заключается в том, чтобы спроектировать абстрактные, в высокой степени пригодные для повторного применения классы, а для этого мы спроектируем очень абстрактные интерфейсы пользователя.

Для того чтобы вы смогли наглядно увидеть разницу между абстрактным и конкретным интерфейсами, создадим такой объект, как такси. Намного полезнее располагать интерфейсом, например «отвезите меня в аэропорт», чем отдельными интерфейсами, например «поверните направо», «поверните налево», «поехали», «остановитесь» и т. д., поскольку, как показано на рис. 2.4, клиенту нужно лишь одно — добраться до аэропорта.

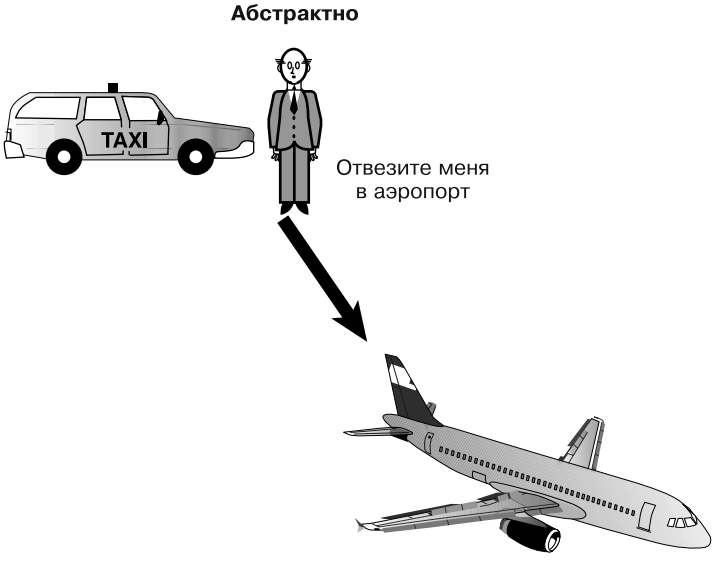

Рис. 2.4. Абстрактный интерфейс

Когда вы выйдете из отеля, в котором жили, бросите свои чемоданы на заднее сиденье такси и сядете в машину, таксист повернется к вам и спросит: «Куда вас отвезти?» Вы ответите: «Пожалуйста, отвезите меня в аэропорт» (при этом, естественно, предполагается, что в городе есть только один крупный аэропорт. Например, в Чикаго вы сказали бы: «Пожалуйста, отвезите меня в аэропорт “Мидуэй”» или «Пожалуйста, отвезите меня в аэропорт “О’Хара”»). Вы сами, возможно, не будете знать, как добраться до аэропорта, но даже если и будете, то вам не придется рассказывать таксисту о том, когда и в какую сторону нужно повернуть, как показано на рис. 2.5. Каким в действительности путем таксист поедет, для вас как пассажира не будет иметь значения (однако плата за проезд в какой-то момент может стать предметом разногласий, если таксист решит сжульничать и повезти вас в аэропорт длинным путем).

В чем же проявляется связь между абстрактностью и повторным использованием? Задайте себе вопрос насчет того, какой из этих двух сценариев более пригоден для повторного использования — абстрактный или не такой абстрактный? Проще говоря, какая фраза более подходит для того, чтобы использовать

Рис. 2.5. Не такой абстрактный интерфейс

ее снова: «Отвезите меня в аэропорт» или «Поверните направо, затем направо, затем налево, затем направо, затем налево»? Очевидно, что первая фраза является более подходящей. Вы можете сказать ее в любом городе всякий раз, когда садитесь в такси и хотите добраться до аэропорта. Вторая фраза подойдет только в отдельных случаях. Таким образом, абстрактный интерфейс «Отвезите меня в аэропорт» в целом является отличным вариантом грамотного подхода к объектно-ориентированному проектированию, результат которого окажется пригоден для повторного использования, а его реализация будет разной в Чикаго, Нью-Йорке и Кливленде.

Обеспечение минимального интерфейса пользователя

При проектировании класса общее правило заключается в том, чтобы всегда обеспечивать для пользователей как можно меньше информации о внутреннем устройстве этого класса. Чтобы сделать это, придерживайтесь следующих простых правил.

- Предоставляйте пользователям только то, что им обязательно потребуется. По сути, это означает, что у класса должно быть как можно меньше интерфейсов. Приступая к проектированию класса, начинайте с минимального интерфейса. Проектирование класса итеративно, поэтому вскоре вы обнаружите, что минимального набора интерфейсов недостаточно. И это будет нормально.

- Лучше в дальнейшем добавить интерфейсы из-за того, что они окажутся действительно нужны пользователям, чем сразу давать этим людям больше интерфейсов, нежели им требуется. Иногда доступ конкретного пользователя к определенным интерфейсам создает проблемы. Например, вам не понадобится интерфейс, предоставляющий информацию о заработной плате всем пользователям, — такие сведения должны быть доступны только тем, кому их положено знать.
- Сейчас рассмотрим наш программный пример. Представьте себе пользователя, несущего системный блок персонального компьютера без монитора или клавиатуры. Очевидно, что от этого персонального компьютера будет мало толку. Здесь для пользователя просто предусматривается минимальный набор интерфейсов, относящийся к персональному компьютеру. Однако этого минимального набора недостаточно, и сразу же возникает необходимость добавить другие интерфейсы.
- Открытые интерфейсы определяют, что у пользователей имеется доступ. Если вы изначально скроете весь класс от пользователей, сделав интерфейсы закрытыми, то когда программисты начнут применять этот класс, вам придется сделать определенные методы открытыми, и они, таким образом, станут частью открытого интерфейса.
- Жизненно важно проектировать классы с точки зрения пользователя, а не с позиции информационных систем. Слишком часто бывает так, что проектировщики классов (не говоря уже о программном обеспечении всех прочих видов) создают тот или иной класс таким образом, чтобы он вписывался в конкретную технологическую модель. Даже если проектировщик станет все делать с точки зрения пользователя, то такой пользователь все равно окажется скорее техническим специалистом, а класс будет проектироваться с таким расчетом, чтобы он работал с технологической точки зрения, а не был удобным в применении пользователями.
- Занимаясь проектированием класса, перечитывайте соответствующие требования и проектируйте его, общаясь с людьми, которые станут использовать этот класс, а не только с разработчиками. Класс, скорее всего, будет эволюционировать, и его потребуется обновить при создании прототипа системы.

Определение пользователей

Снова обратимся к примеру с такси. Мы уже решили, что пользователи в данном случае — это те люди, которые фактически будут применять соответствующую систему. При этом сам собой напрашивается вопрос: что это за люди?

Первым порывом будет сказать, что ими окажутся *клиенты*. Однако это окажется правильным только примерно наполовину. Хотя клиенты, конечно же, являются пользователями, таксист должен быть способен успешно оказать клиентам

соответствующую услугу. Другими словами, обеспечение интерфейса, который, несомненно, понравился бы клиенту, например «Отвезите меня в аэропорт бесплатно», не устроит таксиста. Таким образом, для того чтобы создать реалистичный и пригодный к использованию интерфейс, следует считать пользователями *как* клиента, *так и* таксиста.

Коротко говоря, любой объект, который отправляет сообщение другому объекту, представляющему такси, считается пользователем (и да, пользователи тоже являются объектами). На рис. 2.6 показано, как таксист оказывает услугу.

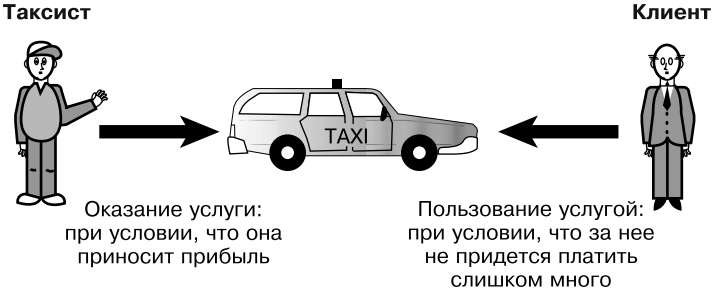

Рис. 2.6. Оказание услуги

ЗАБЕГАЯ ВПЕРЕД

Таксист, скорее всего, тоже будет объектом.

Поведения объектов

Определение пользователей — это лишь часть того, что нужно сделать. После того как пользователи будет определены, вам потребуется определить поведения объектов. Исходя из точки зрения всех пользователей, начинайте определение назначения каждого объекта и того, что он должен делать, чтобы работать должным образом. Следует отметить, что многое из того, что будет выбрано первоначально, не приживется в окончательном варианте открытого интерфейса. Нужно определяться с выбором во время сбора требований с применением различных методик, например на основе вариантов использования UML.

Ограничения, налагаемые средой

В книге «Объектно-ориентированное проектирование на Java» Гилберт и Маккарти отмечают, что среда часто налагает ограничения на возможности объекта. Фактически почти всегда имеют место ограничения, налагаемые средой.

Компьютерное аппаратное обеспечение может ограничивать функциональность программного обеспечения. Например, система может быть не подключена к сети или в компании может использоваться принтер специфического типа. В примере с такси оно не сможет продолжить путь по дороге, если в соответствующем месте не окажется моста, даже если это будет более короткий путь к аэропорту.

Определение открытых интерфейсов

После того как будет собрана вся информация о пользователях, поведениях объектов и среде, вам потребуется определить открытые интерфейсы для каждого пользовательского объекта. Поэтому подумайте о том, как бы вы использовали такой объект, как такси.

- Сесть в такси.
- Сказать таксисту о том, куда вас отвезти.
- Заплатить таксисту.
- Дать таксисту «на чай».
- Выйти из такси.

Что необходимо для того, чтобы использовать такой объект, как такси?

Располагать местом, до которого нужно добраться.

Подозвать такси.

Заплатить таксисту.

Сначала вы задумаетесь о том, как объект используется, а не о том, как он создается. Вы, возможно, обнаружите, что объекту требуется больше интерфейсов, например «Положить чемодан в багажник» или «Вступить в пустой разговор с таксистом». На рис. 2.7 показана диаграмма класса, отражающая возможные методы для класса `Cabbie`.

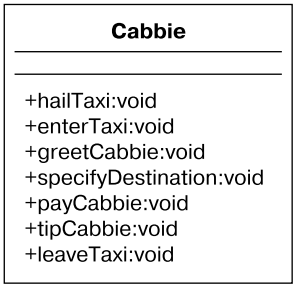

Рис. 2.7. Методы в классе Cabbie

Как и всегда, «шлифовка» финального интерфейса — итеративный процесс. Для каждого интерфейса вам потребуется определить, вносит ли он свой вклад в эксплуатацию объекта. Если нет, то, возможно, он не нужен. Во многих учебниках по объектно-ориентированному программированию рекомендуется, чтобы каждый интерфейс моделировал только одно поведение. Это возвращает нас к вопросу о том, какого абстрагирования мы хотим добиться при проектировании. Если у нас будет интерфейс с именем `enterTaxi()`, то, безусловно, нам не потребуется, чтобы в нем содержалась логика «заплатить таксисту». Если все так и будет, то наша конструкция окажется несколько нелогичной. Кроме того, фактически пользователи класса никак не смогут узнать, что нужно сделать для того, чтобы «заплатить таксисту».

Определение реализации

Выбрав открытые интерфейсы, вы должны будете определить реализацию. После того как вы спроектируете класс, а все методы, необходимые для эксплуатации класса, окажутся на своих местах, потребуется заставить этот класс работать.

Технически все, что не является открытым интерфейсом, можно считать реализацией. Это означает, что пользователи никогда не увидят каких-либо методов, считающихся частью реализации, в том числе подпись конкретного метода (которая включает имя метода и список параметров), а также фактический код внутри этого метода.

Можно располагать закрытым методом, который применяется внутри класса. Любой закрытый метод считается частью реализации при условии, что пользователи никогда не увидят его и, таким образом, не будут иметь к нему доступ. Например, в классе может содержаться метод `changePassword()`; однако этот же класс может включать закрытый метод для шифрования пароля. Этот метод был бы скрыт от пользователей и вызывался бы только изнутри метода `changePassword()`.

Реализация полностью скрыта от пользователей. Код внутри открытых методов является частью реализации, поскольку пользователи не могут увидеть его (они должны видеть только вызывающую структуру интерфейса, а не код, что находится в нем).

Это означает, что теоретически все считающееся реализацией может изменяться, не влияя на то, как пользователи взаимодействуют с классом. При этом, естественно, предполагается, что реализация дает ответы, ожидаемые пользователями.

Хотя интерфейс представляет то, как пользователи видят определенный объект, в действительности реализация — это основная составляющая этого объекта. Реализация содержит код, который описывает состояние объекта.

Резюме

В этой главе мы исследовали три области, которые помогут вам начать мыслить объектно-ориентированным образом. Помните, что не существует какого-либо постоянного списка вопросов, касающихся объектно-ориентированного мышления. Работа в объектно-ориентированном стиле больше представляет собой искусство, нежели науку. Попробуйте сами придумать, как можно было бы описать объектно-ориентированное мышление.

В главе 3 мы поговорим о жизненном цикле объектов: как они «рождаются», «живут» и «умирают». Пока объект «жив», он может переходить из одного состояния в другое, и этих состояний много. Например, объект `DataBaseReader` будет пребывать в одном состоянии, если база данных окажется открыта, и в другом состоянии — если будет закрыта. Способ, которым все это будет представлено, зависит от того, как окажется спроектирован соответствующий класс.

Ссылки

- Мартин Фаулер, «UML. Основы» (UML Distilled). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2003.
- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Скотт Майерс, «Эффективное использование C++» (Effective C++). — 3-е изд. Addison-Wesley Professional, 2005.