ЧАСТЬ I

Введение в разработку объектно-ориентированного программного обеспечения

В этой части

В главах этой части рассматривается метод разработки объектно-ориентированного программного обеспечения, основанный на использовании шаблонов, представляющих собой обобщение теоретических исследований и лучших практических решений, найденных за многие годы деятельности разработчиков и пользователей. В качестве средства поддержки будет использоваться унифицированный язык моделирования UML.

Этот подход отличается от положений объектно-ориентированной парадигмы, сформулированной в 1980-х годах, согласно которым разработчику достаточно было отыскать в предоставленных ему исходных требованиях к программному обеспечению отдельные существительные и создать на их основе объекты. В этой парадигме инкапсуляция рассматривалась как сокрытие внутренней структуры данных, а объекты должны были содержать как данные, так и методы для работы с ними. Это весьма ограниченное представление, поскольку внимание фокусируется только на принципах реализации объектов. Однако этого мало.

В данной части обсуждается другая версия объектно-ориентированной парадигмы, построенная на расширенных определениях основных концепций. Эти определения являются результатом обобщения тех методов и принципов, которые были найдены в процессе разработки и реализации шаблонов проектирования. Новая схема более полно раскрывает суть объектной технологии.

Глава	**Предмет обсуждения**
1	Современное представление о понятии объекта
2	Знакомство с унифицированным языком моделирования UML, представляющим собой средство наглядного и понятного описания объектно-ориентированных проектов

Глава 1

Объектно-ориентированная парадигма

Введение

Данная глава предназначена для ознакомления с объектно-ориентированной парадигмой, которое будет проведено посредством сравнения и сопоставления ее положений с уже знакомыми читателю стандартными положениями структурного программирования.

Разработка объектно-ориентированной парадигмы была вызвана трудностями, имевшими место в установившейся ранее практике применения положений стандартного структурного программирования. Подробное обсуждение этих трудностей позволит нам более полно раскрыть преимущества объектно-ориентированного программирования, а так же углубить понимание его механизма.

Изучение материала этой главы не сделает из вас специалиста по объектно-ориентированным методам. Более того, здесь даже не предусматривается ознакомление читателя со всеми фундаментальными концепциями объектно-ориентированного подхода. Тем не менее, данная глава является необходимым вступлением к остальной части книги, в которой раскрываются методы правильного объектно-ориентированного проектирования, применяемые экспертами на практике.

В этой главе...

- Обсуждается общий метод анализа, называемый *функциональной декомпозицией.*
- Рассматривается проблема формирования требований к создаваемому программному обеспечению и необходимость постоянного внесения в них изменений (главный бич программирования!).
- Описывается объектно-ориентированная парадигма и демонстрируется ее практическое применение.
- Выявляются некоторые специфические методы объектов.
- Приводится сводная таблица важнейших терминов объектно-ориентированной технологии, используемых в этой главе.

Функциональная декомпозиция: в преддверии появления объектно-ориентированной парадигмы

Давайте начнем с рассмотрения типичного подхода к разработке программного обеспечения. Предположим, что задание заключается в написании программы, которая должна выбирать из базы данных хранящиеся в ней описания фигур, а затем отображать их. В этом случае вполне уместно было бы начать с определения основных этапов решения поставленной задачи. Например, возможное решение может выглядеть следующим образом.

1. Получить перечень фигур, описания которых сохранены в базе данных.
2. Открыть найденный список.
3. Отсортировать список в соответствии с некоторыми правилами.
4. Последовательно отобразить отдельные фигуры на экране.

Любой из перечисленных выше этапов можно разбить на более мелкие этапы, необходимые для его реализации. Например, этап 4 можно разделить следующим образом.

Для каждой из перечисленных в списке фигур выполнить следующее.

4.1. Определить тип фигуры.

4.2. Получить информацию о расположении фигуры.

4.3. Вызвать соответствующую функцию, которая отобразит фигуру исходя из информации о ее расположении.

Этот подход называется методом *функциональной декомпозиции*, поскольку разработчик разбивает (подвергает декомпозиции) проблему на несколько функциональных этапов, обеспечивающих ее решение. Мы поступаем так потому, что проще иметь дело с небольшими частями, чем со всей проблемой в целом. Тот же подход можно было бы использовать при написании инструкции по скалолазанию или по сборке велосипеда. Мы используем этот подход настолько часто и естественно, что у нас редко возникает вопрос о возможных альтернативах.

Основная проблема метода функциональной декомпозиции заключается в том, что он не позволяет каким-либо образом подготовить текст программы к последующим изменениям, появляющимся по мере ее постепенной эволюции. Чаще всего подобные изменения связаны с необходимостью учесть новый вариант поведения программы в уже существующей системе. В нашем примере это может быть создание новой фигуры или реализация нового способа отображения фигур. Если всю логику реализации выбранной поэтапной схемы поместить в одну большую функцию или единственный модуль, то каждое изменение любого из этапов потребует внесения изменений в эту функцию или модуль.

С другой стороны, внесение изменений обычно повышает опасность появления ошибок или других нежелательных последствий. Иначе это можно сформулировать следующим образом.

Множество ошибок появляется при внесении в текст программы изменений.

Попробуйте обосновать это утверждение исходя из собственного опыта. Вспомните, что всякий раз, когда в текст программы необходимо внести изменения, возникает опасение, что изменение текста программы в одном месте может привести к сбою в другом. Почему же это происходит? Должен ли программист анализировать все имеющиеся функции и обращать внимание на способ их использования? Следует ли ему учитывать, как функции могут взаимодействовать друг с другом? Обязан ли он проверять все множество различных подробностей, связанных с каждой функцией, — реализуемую этой функцией логику, компоненты с которыми эта функция взаимодействует, данные, которые она использует? Как это часто бывает с людьми, необходимость учесть при внесении изменений слишком много различных деталей обычно приводит к появлению ошибок.

И не имеет значения, сколько усилий было приложено, насколько тщательно был проведен анализ — пользователь просто не может сформулировать все необходимые требования сразу. Слишком много неизвестного несет в себе будущее. Времена меняются. И так было всегда...

Ничто не может предотвратить наступление перемен. Однако это не значит, что к их приходу нельзя подготовиться.

Проблема формулирования требований к создаваемому программному обеспечению

Любой опрос среди разработчиков программного обеспечения по качеству предоставленных им заказчиком требований к создаваемому программному продукту едва ли даст большое разнообразие ответов. Скорее всего они будут следующими.

- Требования являются неполными.
- Требования по большей части ошибочны.
- Требования (и пользователи) противоречивы.
- Требования не описывают поставленную задачу подробно.

Практически нет шансов получить ответ, подобный следующему: “предоставленные требования не только исчерпывающе полные, ясные и понятные, но также включают описание всех функциональных возможностей, которые потребуются заказчику в течение пяти последующих лет!”

За тридцать лет практической деятельности в области разработки программного обеспечения главный вывод, который мне удалось сделать относительно предоставляемых заказчиком требований, можно сформулировать следующим образом.

Требования к программному обеспечению всегда изменяются.

Я также понял, что большинство разработчиков знают это обстоятельство и считают его весьма неприятным. Но лишь немногие из них действительно учитывают возможность изменения существующих требований при написании программ.

Изменение требований к программному обеспечению происходит вследствие ряда простых причин.

- Взгляды пользователей на их нужды изменяются как в результате обсуждений с разработчиками, так и при ознакомлении с новыми возможностями в области программного обеспечения.
- Представление разработчиков о предметной области меняется по мере создания ими программного обеспечения, предназначенного для ее автоматизации, поскольку им становятся известны дополнительные особенности этой области.
- Появляются все новые вычислительные среды, предназначенные для разработки программного обеспечения. (Кто мог пять лет назад предположить, что Web-дизайн когда-либо достигнет таких высот, как в настоящее время?)

Все это вовсе не значит, что можно опустить руки и отказаться от сбора полноценных требований пользователей к создаваемой системе. Напротив, это значит, что необходимо научиться приспосабливать создаваемый программный код к неизбежному внесению изменений. Это также значит, что необходимо прекратить обвинять себя (или заказчиков) в том, что является совершенно закономерным.

Требуется внести изменения? Так сделайте это!

- За исключением самых простых случаев, требования изменяются всегда и везде, вне зависимости от того, насколько хорошо был сделан первичный анализ!
- Вместо того чтобы жаловаться на изменение исходно предоставленных требований, необходимо так модифицировать процесс разработки, чтобы обеспечить эффективную обработку любых возникающих изменений.

Внесение изменений при использовании метода функциональной декомпозиции

Вернемся к задаче отображения фигур. Как написать текст программы таким образом, чтобы было проще отражать в нем возможные изменения исходных требований? Вместо того чтобы реализовать всю функциональность программы в одной большой функции, можно разбить ее текст на несколько модулей.

Например, для реализации этапа 4.3, назначение которого можно сформулировать так: "Вызвать соответствующую функцию, которая отобразит фигуру исходя из информации о ее расположении", можно написать модуль, текст которого представлен в листинге 1.1.

Листинг 1.1. Применение модульности для уменьшения влияния изменений

```
function: display shape
input: тип фигуры, описание фигуры
action:
   switch (тип фигуры)
      case квадрат: (текст функции отображения квадрата)
      case круг: (текст функции отображения круга)
```

Теперь при поступлении требований обеспечить отображение фигур нового типа — например, треугольников — потребуется изменить только этот модуль (звучит обнадеживающе!).

Однако данный подход имеет свои недостатки. Например, в тексте модуля указано, что входными данными для него являются тип и описание фигуры. Однако организация их единообразного описания, подходящего для всех фигур, может быть как возможна, так и не возможна — в зависимости от того, как реализовано сохранение сведений о фигурах. Что, если описание фигуры в некоторых случаях будет сохраняться в виде массива точек? Будет ли предложенная схема применима и в этой ситуации?

Несомненно, применение модульного подхода позволяет сделать программу более понятной, что обусловливает и простоту ее сопровождения. Но модульность не всегда гарантирует легкость *любого* требуемого изменения программы.

Подход, который я ранее использовал, имеет существенные недостатки, связанные с понятиями *слабой связности* и *сильной связанности*. Стив Мак-Коннел (Steve McConnel) в своей книге *Code Complete* дает замечательное определение как понятия *связности* (cohesion), так и понятия *связанности* (coupling).

- Понятие связности относится к тому, насколько "сильна взаимозависимость операций подпрограммы".[1]

В других случаях связность иногда определяют как *прозрачность* (clarity), поскольку чем больше взаимозависимость операций подпрограммы (или класса), тем понятнее ее структура и назначение.

- Понятие *связанности* относится к тому, насколько "сильна связь между двумя подпрограммами. Связанность является дополнением понятия связности. Связность определяет, насколько тесно внутренние элементы подпрограммы взаимодействуют между собой. Связанность характеризует, насколько тесно подпрограмма связана с другими подпрограммами. При этом целью разработчика является создание подпрограмм, обладающих внутренней целостностью (сильной связностью) и слабой, прямой, явной и гибкой зависимостью от других подпрограмм (слабая связанность)".

Практический опыт большинства программистов показывает, что изменение функции или части кода программы в одном месте зачастую приводит к неожиданным последствиям совсем в другом месте кода. Этот тип ошибки получил название "нежелательный побочный эффект". Он объясняется тем, что, оказывая необходимое влияние (внося изменение), мы одновременно совершаем совсем ненужное воздействие — т.е. вносим ошибку! Хуже всего то, что подобные ошибки часто очень трудно найти, поскольку, делая первоначальное изменение, мы обычно не замечаем той связи, которая приводит к нежелательным побочным эффектам (если бы эта связь была замечена, то вносимые изменения имели бы совсем другой характер).

В действительности, по поводу ошибок подобного типа можно сделать один поразительный вывод.

На практике ошибки исправляются относительно быстро.

Я считаю, что на исправление ошибок затрачивается лишь незначительная часть времени всего процесса внесения изменений и их отладки. Основная часть времени при внесении изменений и их отладке тратиться на *обнаружение* неполадок и выявле-

---

[1] *McConnel, S. Code Complete:* A Practical Handbook of Software Construction*, Redmond, Microsoft Press, 1993.*

ние нежелательных побочных эффектов. Сам же процесс исправления ошибок относительно непродолжителен.

Поскольку нежелательные побочные эффекты относятся к тому типу ошибок, которые труднее всего обнаружить, внесение любого изменения в функцию, обрабатывающую большое количество разнообразных данных, с высокой степенью вероятности приведет к возникновению проблем.

Устранять побочные эффекты очень трудно

- Применение функциональной декомпозиции способствует появлению побочных эффектов, которые очень трудно обнаружить.
- Бо́льшая часть времени, затрачиваемого на сопровождение и отладку, расходуется не на исправление ошибок, а на их обнаружение, а также на поиск решений, позволяющих избежать появления нежелательных побочных эффектов от внесения требуемых исправлений.

При использовании метода функциональной декомпозиции постоянное изменение требований сводило все мои усилия по разработке и сопровождению программного обеспечения на нет. Главные проблемы были связаны с функциями. Внесение изменений в одну группу функций или данных неизбежно приводило к необходимости внесения изменений в другую группу функций или данных, что, в свою очередь, затрагивало следующую группу функций и т.д. Функциональная декомпозиция приложения приводит к каскадным изменениям, подобным снежному кому, скатывающемуся с высокой горы, избежать которых достаточно сложно.

Обработка изменяющихся требований

Чтобы найти способ справиться с проблемой изменяющихся требований, а заодно подыскать альтернативу методу функциональной декомпозиции, давайте рассмотрим, как люди обычно подходят к решению стоящих перед ними задач. Представим себя на месте преподавателя. После окончания лекции слушателям предстоит посетить занятия по другим предметам, но они не знают, где расположены соответствующие аудитории. В ваши обязанности входит информирование слушателей относительно места проведения их следующих занятий.

Если следовать принципам структурного программирования, можно поступить, например, следующим образом.

1. Составить список присутствующих.
2. Для каждого человека в списке выполнить следующее:
   а) определить предмет, по которому проводится следующее занятие;
   б) выяснить расположение соответствующей аудитории;
   в) определить путь от данной аудитории к той, в которой будет проводиться следующее занятие;
   г) объяснить слушателю, как пройти в искомую аудиторию.

Для реализации перечисленных выше действий потребуется уточнить следующее.

1. Способ получения списка присутствующих.
2. Метод получения расписания занятий для каждого слушателя в аудитории.
3. Алгоритм определения пути следования из вашей аудитории в любую указанную аудиторию.
4. Алгоритм управляющей программы, которая работала бы с каждым слушателем в аудитории и выполняла все необходимые этапы.

Я сомневаюсь в том, что кто-либо действительно выберет подобный подход. Скорее всего, преподаватель просто вывесит на стене схему, поясняющую, как пройти в другие аудитории, и сделает вначале занятий следующее объявление: "На стене аудитории вывешено расписание последующих занятий и схема расположения соответствующих аудиторий. С их помощью каждый из вас сможет определить свои дальнейшие действия". При этом преподаватель будет уверен в том, что все присутствующие смогут получить необходимую информацию о своих последующих занятиях и самостоятельно определить по схеме маршрут своего перемещения.

Какая же разница между этими двумя подходами?

- В первом случае, составляя подробные инструкции для каждого слушателя, потребуется не упустить из виду множество мелких деталей. И никто, кроме вас, не несет ответственности за информирование слушателей. Нагрузка просто чрезмерна!
- Во втором случае просто составляются общие инструкции, и каждому из присутствующих предоставляется возможность самостоятельно найти решение стоящей перед ним задачи.

Главное различие заключается в **передаче ответственности**. В первом случае ответственность за все несет только преподаватель, во втором — каждый из слушателей сам отвечает за свои действия. В любом из вариантов предусматривается достижение одного и того же результата, различаются только способы его достижения.

Какой же вывод можно сделать из сказанного?

Чтобы оценить эффект перераспределения ответственности, давайте посмотрим, что произойдет при введении нового требования.

Предположим, было получено указание давать специальный инструктаж для аспирантов, принимающих участие в занятиях. Допустим, им поручили узнать оценки слушателей по итогам занятия и передать эти сведения в канцелярию по пути в следующую аудиторию. В первом варианте потребуется изменить алгоритм управляющей программы, чтобы она смогла отличать аспирантов от обычных слушателей и давать аспирантам специальные инструкции. Вполне вероятно, что подобная корректировка потребует значительных усилий.

Однако во втором случае, когда слушатели несут ответственность за себя сами, потребуется лишь вывесить дополнительные указания для аспирантов. Завершающая фраза лектора будет той же самой: "Каждый слушатель может идти в свою следующую аудиторию". При этом все слушатели будут следовать этой инструкции в соответствии со своим индивидуальным расписанием.

Различие в требуемых действиях весьма значительное. С одной стороны, каждый раз при появлении новой категории слушателей со специальными инструкциями поведения

потребуется модифицировать алгоритм управляющей программы. С другой стороны, новые категории слушателей по-прежнему будут действовать самостоятельно.

Можно выделить три основных отличия второй схемы от первой.

- Каждый слушатель сам несет ответственность за свои действия, тогда как в первом случае за все отвечает управляющая программа. (Для этого лишь необходимо, чтобы каждый слушатель знал, к какому разряду он относиться.)
- К различным категориям слушателей применяется одна и та же схема взаимодействия (как к аспирантам, так и к обыкновенным слушателям).
- Нет необходимости иметь сведения об особых обязанностях отдельных слушателей.

Чтобы полностью понять смысл всего вышесказанного, введем некоторую специальную терминологию. Мартин Фулер (Martin Fowler) в книге *UML Distilled* описывает три различных подхода к процессу разработки программного обеспечения.[2] Они представлены в табл. 1.1.

Таблица 1.1. Различные подходы к процессу разработки программного обеспечения

Название	Описание
На концептуальном уровне	При этом подходе "выявляются основные концепции изучаемой области задач... концептуальная модель может незначительно или даже вообще не касаться вопросов ее реализации в создаваемом программном обеспечении..."
На уровне спецификаций	"В этом случае проектируется само программное обеспечение, но лишь с точки зрения его интерфейсной части, а не полной реализации"
На уровне реализации	На этом этапе создается собственно текст программ. "В большинстве случаев именно этот подход воспринимается как основной, но зачастую оказывается оправданным предварительное обращение к подходам предыдущих типов"

Еще раз вернемся к примеру с посещением следующих занятий. Обратите внимание на то, что преподаватель взаимодействует со слушателями на *концептуальном уровне*, т.е. он говорит другим, что необходимо сделать, но не объясняет как это делается. Однако тот путь, который слушатель должен пройти, чтобы попасть в следующую аудиторию, вполне конкретен. Каждый из них следует определенным инструкциям, а это значит, что они действуют на *уровне реализации*.

Взаимодействуя со слушателями на одном уровне (концептуальном), преподаватель закладывает основу их будущих действий на другом уровне (реализации), при этом он не вникает в подробности происходящего и определяет лишь общую схему необходимых действий. Этот механизм может быть весьма эффективным. Давайте попробуем применить его к процессу создания программных продуктов.

---

[2] *Fowler, M., Scott, K.,* UML Distilled: A brief Guide to the Standard Object Modeling Language. 2nd edition. *Reading Mass., Addison-Wesley, 1999.*

Объектно-ориентированная парадигма

Объектно-ориентированная парадигма строится на основе концепции объекта. Все в ней сосредоточено на объектах, поэтому текст создаваемых программ также строится на использовании объектов, а не функций.

Что же такое объект? Традиционно объект понимается как совокупность данных и *методов* (объектно-ориентированный термин для функций). К сожалению, такое определение объекта очень ограничено. Ниже будет дано более емкое определение (оно повторно приводится в главе 8, *Расширение горизонтов*). Затем мы обсудим данные объекта, которыми могут являться как обычные числа и строки символов, так и другие объекты.

Преимущество использования объектов заключается в том, что на них возлагается ответственность за их собственное поведение (табл. 1.2). Объекты изначально относятся к определенному типу, известному им. Данные объекта позволяют ему определять свое состояние, а программный код объекта обеспечивает его корректное функционирование (т.е. выполнение тех действий, для которых он, собственно, предназначен).

Таблица 1.2. Объекты и возложенные на них обязанности

Тип объекта	Обязанности объекта
`Student` (Слушатель)	Знать, в какой аудитории он сейчас находится. Знать, в какую аудиторию необходимо перейти. Перейти из данной аудитории в следующую
`Instructor` (Преподаватель)	Выдать сообщение о том, что необходимо перейти в следующую аудиторию
`Classroom` (Аудитория)	Предоставить информацию о местонахождении
`Direction_giver` (Путеводитель)	Определить пары аудиторий. Указать маршрут перехода из одной аудитории в другую

В нашем примере объекты были определены для каждой сущности в предметной области. Обязанности объектов (т.е. методы) были установлены согласно действиям, выполняемым соответствующими сущностями. Подобный подход согласуется с правилом выделения объектов посредством выборки всех присутствующих в требованиях существительных, а необходимые этим объектам методы устанавливаются при обнаружении всех связанных с данными существительными глаголов. Я нахожу этот метод несколько ограниченным и далее в книге предлагаю лучший вариант. Тем не менее, этот подход позволяет нам начать работу.

Лучшим способом восприятия концепции объекта является представление его как некоей сущности, имеющей определенные обязанности. Хороший стиль проектирования предполагает создание объектов, обладающих всей полнотой ответственности за свое собственное поведение, и эта ответственность должна быть четко определена. Именно поэтому на объект, представляющий слушателя, возложена обязанность знать, как пройти из одной аудитории в другую.

Рассмотрим объекты с точки зрения тех подходов, которые выделены в работе Фулера.

- На *концептуальном уровне* объекты выступают как совокупность обязательств.
- На *уровне определения спецификаций* объекты понимаются как набор методов, которые могут вызываться другими объектами или этим же объектом.
- На *уровне реализации* объекты рассматриваются как совокупность программного кода и данных.

К сожалению, изучение и обсуждение методов объектно-ориентированного проектирования чаще всего ведется только с точки зрения уровня реализации — в терминах программного кода и данных, а не с позиций концептуального уровня или уровня определения спецификаций. Однако потенциал последних двух вариантов представления объектов также достаточно высок!

Поскольку объекты обладают обязанностями и несут полную ответственность за свои действия, обязательно должен существовать способ управлять ими — т.е. указывать, что необходимо сделать. Не забывайте, что объект содержит данные, описывающие его текущее состояние, и методы, реализующие его функциональность. Некоторые методы объекта идентифицируются как методы, доступные для других объектов. Набор этих методов определяет открытый интерфейс объекта (public interface).

Например, в предыдущем примере объект **`Student`** может включать метод `gotoNextClassroom()`. Нет необходимости передавать этому методу какие-либо параметры, поскольку каждый объект **`Student`** несет полную ответственность за свое поведение. В частности, ему должно быть известно следующее.

- Что ему необходимо, чтобы перейти в следующую аудиторию.
- Как получить дополнительную информацию, необходимую для такого перехода.

Первоначально в задаче была определена только одна категория — обыкновенные слушатели, посещающие различные занятия. Обратите внимание на то, что в аудитории (в моделируемой системе) может присутствовать множество обыкновенных слушателей. Но что делать, если потребуется учитывать существование дополнительных *категорий* слушателей? Очевидно, что решение, при котором каждый объект `student` будет обладать уникальным набором методов, определяющих его функциональность, является неэффективным. Это тем более справедливо при наличии задач, общих для всех слушателей.

Более эффективно реализовать ряд методов, общих для всех объектов `student`, которые каждый из них мог бы использовать или приспосабливать для своих нужд. Целесообразно будет определить объект "типичный слушатель", содержащий все общеупотребительные методы. Затем на его основе можно будет создавать индивидуальные объекты слушателей, каждый из которых будет поддерживать собственную, необходимую ему информацию.

В объектно-ориентированной терминологии подобный обобщенный объект получил название *класса*. Класс является определением линии поведения объекта. Он содержит полное описание следующих элементов:

- всех элементов данных, входящих в состав объекта;
- всех методов, которые объект способен выполнять;
- необходимых способов доступа к имеющимся элементам данных и методам.

Поскольку содержащимся в объекте элементам данных можно присваивать различные значения, объекты одного и того же класса будут содержать различные набо-

ры конкретных значений данных, но, в то же время, обладать одной и той же функциональностью (реализуемой методами).

Чтобы получить доступ к объекту, предварительно следует сообщить программе, что необходим новый объект такого-то типа (т.е. того класса, к которому относится этот объект). Создаваемый новый объект получил название *экземпляра* класса. Создание экземпляра класса называется его *реализацией*.

При использовании объектно-ориентированного подхода создание приложения имитации перехода слушателей в различные аудитории существенно упрощается. Требуемый алгоритм будет выглядеть следующим образом.

1. Запустить управляющую программу.
2. Создать коллекцию экземпляров объектов, представляющих находящихся в аудитории слушателей.
3. Послать объекту коллекции сообщение о необходимости перехода в следующую аудиторию.
4. Объект коллекции уведомляет каждый объект слушателя о необходимости перехода в следующую аудиторию.
5. Каждый объект слушателя выполняет такие действия:
   а) определяет, в какой аудитории будет проходить следующее занятие;
   б) выясняет, как пройти в эту аудиторию;
   в) выполняет переход;
   г) завершает свою работу.

Указанная схема хорошо работает до тех пор, пока не возникнет необходимость добавить новый тип слушателей — например аспирантов.

В этом случае возникает дилемма. Ясно, что коллекция должна включать любые категории слушателей (как студентов, так и аспирантов). Проблема заключается в том, как коллекция должна будет обращаться к собственным составляющим. Поскольку речь идет о написании программы, то коллекция должна иметь вид массива или являться каким-либо другим объектом подобного типа. Если коллекция будет иметь определенный тип, например `RegularStudent`, то в нее нельзя будет поместить объекты типа `GraduateStudent`. Если же коллекция будет представлять собой некоторый произвольный набор объектов, то нельзя гарантировать, что в нее никогда не будет помещен недопустимый тип объекта (который не способен выполнять операцию "переход в следующую аудиторию").

Решить проблему очень просто. Необходимо иметь общий тип, который содержал бы более одного конкретного типа. В нашем случае это класс `Student`, включающий в себя как класс `RegularStudent`, так и класс `GraduateStudent`. В объектно-ориентированной терминологии подобный класс называется *абстрактным классом*.

Абстрактные классы определяют поведение других, родственных им классов. Эти "другие" классы представляют специфический тип родственного поведения. Такой класс обычно называют *конкретным классом*, поскольку он представляет особую определенную, фиксированную реализацию общей концепции.

В нашем примере абстрактным классом является класс `Student`, а два производных класса представлены конкретными классами `RegularStudent` и `GraduateStudent`.

Здесь класс **RegularStudent** представляет один, а **GraduateStudent** — другой вариант класса **Student**.

Такой тип отношений между объектами называется "отношением *is-a*" (является), более формальное название которого — *наследование*. Таким образом, можно сказать, что класс **RegularStudent** *наследует* классу **Student**. В другом варианте это же отношение выражают, говоря, что класс **GraduateStudent** *порожден*, или является *подклассом* класса **Student**. С другой стороны, о классе **Student** говорят, что это *базовый класс*, *генерализация*, или *суперкласс*, для классов **RegularStudent** и **GraduateStudent**.

Абстрактные классы для других классов выступают в роли шаблонов. Я использую их для определения методов, реализуемых в их классах-потомках. Абстрактные классы могут также содержать общие методы, которые будут использоваться всеми классами-потомками. Использует класс-потомок методы своего родителя или реализует свои собственные, в любом случае в иерархии классов он занимает следующую, более низкую, позицию (что отвечает общему требованию, по которому объекты должны сами отвечать за свое поведение).

Все это означает, что теперь можно определить коллекцию, объединяющую любые объекты класса **Student**. Используемый тип ссылки будет при этом иметь значение **Student**, и компилятор сможет проверить, являются ли ссылающиеся на указатель **Student** объекты на самом деле экземплярами суперкласса **Student**. Достигаемые при этом преимущества очевидны.

- Коллекция будет содержать только экземпляры класса **Student**.
- Обеспечивается контроль согласования типов (в коллекцию включаются только экземпляры класса **Student**, которые поддерживают метод "переход в следующую аудиторию").
- Каждый подкласс класса **Student** волен реализовать свою функциональность по собственному усмотрению.

Абстрактные классы

Абстрактные классы часто описываются как классы, не предусматривающие создания своих экземпляров. Это определение правильно, но только на уровне реализации. В остальных случаях оно оказывается слишком ограниченным. Полезнее будет определить понятие абстрактного класса на концептуальном уровне. На концептуальном уровне абстрактные классы – это просто шаблоны для создания других классов.

Следовательно, с помощью абстрактного класса мы просто присваиваем собственное название некоторому множеству родственных классов. Это позволит нам в дальнейшем рассматривать данное множество как единую концепцию.

При использовании объектно-ориентированной парадигмы любую проблему следует рассматривать со всех трех точек зрения.

Поскольку объекты несут полную ответственность за собственное поведение, существует множество элементов, доступ к которым со стороны других объектов они разрешать не заинтересованы. Ранее уже упоминалось понятие *открытый интерфейс* (public interface) — оно охватывает все методы объекта, доступные для любых других объектов. В объектно-ориентированных системах существует несколько уровней доступа к элементам объектов.

- *Public* (открытый) — доступ разрешен для всех.

- *Protected* (защищенный) — доступ разрешен только объектам этого же класса или его классам-потомкам.
- *Private* (закрытый) — доступ предоставляется только объектам этого класса.

Теперь мы подошли к понятию *инкапсуляции*. Инкапсуляцию чаще всего определяют как сокрытие данных. Объекты обычно не разрешают доступ извне к своим внутренним данным-членам (т.е. для этих элементов указывается уровень доступа *защищенный* или *закрытый*).

Однако инкапсуляция представляет собой нечто большее, чем просто сокрытие данных. В общем случае понятие инкапсуляции охватывает *любой тип сокрытия*.

В нашем примере преподаватель не делает различия между студентами и аспирантами. Сведения о категории слушателя скрыты от него (т.е. конкретный тип объекта `Student` оказывается инкапсулированным). Как будет показано позже, этот принцип очень важен.

Следующее понятие, которое подлежит обсуждению, — *полиморфизм*.

В объектно-ориентированных языках к объектам часто обращаются по ссылкам определенного типа, представляющего собой тип соответствующего абстрактного класса. Однако в действительности обращение выполняется к экземплярам конкретных классов, порожденных от данного абстрактного класса.

Таким образом, если объекту с помощью абстрактной ссылки дается указание о реализации какого-либо концептуального действия, то реальное поведение этого объекта будет зависеть от его конкретного производного типа. Термин *полиморфизм* происходит от слов *poly* (что значит "много") и *morph* (что значит "форма"), т.е. дословно оно означает "много форм". Это очень точное название, поскольку одно и тоже обращение может иметь следствием множество различных форм поведения.

В нашем примере преподаватель говорит слушателям, что они "должны перейти в следующую аудиторию". Однако реальное поведение слушателя будет зависеть от того, к какой категории он относится (яркий пример полиморфизма).

Обзор объектно-ориентированной терминологии

Термин	Описание
Объект	Сущность, обладающая всей полнотой ответственности за свое поведение. Реализуется посредством включения в текст программы описания класса, содержащего требуемые компоненты данных (переменные, входящие в состав объекта) и методы (связанные с объектом функции).
Класс	Репозиторий методов. Описывает данные-члены объектов. Текст программы организуется вокруг классов.
Инкапсуляция	Обычно определяется как сокрытие данных, но более полное определение подразумевает сокрытие информации любого рода.
Наследование	Позволяет описать класс как специализированный подтип другого класса. Подобные специализированные классы называются производными от базового (или исходного) класса. Базовый класс иногда называют суперклассом, тогда как классы-потомки иногда называют подклассами.
Экземпляр	Конкретный представитель класса (он всегда является объектом).
Реализация	Процесс создания экземпляра класса.

Полиморфизм	Возможность одним и тем же способом обращаться к различным подклассам некоторого родительского класса. При этом достигаемый результат будет зависеть от конкретного типа того класса-потомка, к которому производится обращение.
Подход	Существует три различных подхода к анализу объектной модели: на *концептуальном уровне*, на *уровне спецификаций* и на *уровне реализации*. Выделение этих подходов позволяет лучше понять механизм взаимодействия между абстрактными классами и их потомками. Абстрактные классы помогают найти решение задачи на концептуальном уровне, а также определить способ взаимодействия объектов любых порожденных от них классов. Каждый класс-потомок предназначен для реализации каких-либо специфических потребностей.

Объектно-ориентированное программирование в действии

Давайте еще раз вернемся к примеру с рисованием фигур, обсуждавшемуся в начале этой главы. Как можно реализовать поставленную задачу, используя объектно-ориентированный подход? Вспомним основные этапы, выделенные для ее решения.

1. Найти перечень фигур, описания которых сохранены в базе данных.
2. Открыть найденный список.
3. Отсортировать список в соответствии с некоторыми правилами.
4. Последовательно отобразить отдельные фигуры на экране.

Для решения поставленной задачи с использованием объектно-ориентированного подхода потребуется определить необходимые объекты и распределить между ними обязанности.

Будем считать, что для решения задачи необходимы объекты, перечисленные в табл. 1.3.

Таблица 1.3. Объекты, необходимые для решения задачи рисования фигур

Класс	Обязанности (методы)
`ShapeDataBase`	`getCollection` — извлекает заданный набор фигур
`Shape` (абстрактный класс)	`display` — описывает интерфейс для фигур `getX` — возвращает координату X фигуры (используется для сортировки) `getY` — возвращает координату Y фигуры (используется для сортировки)
`Square` (производный от класса `Shape`)	`display` — отображает квадрат (представленный этим объектом)
`Circle` (производный от класса `Shape`)	`display` — отображает окружность (представленную этим объектом)

Окончание таблицы

Класс	Обязанности (методы)
`Collection`	`display` — посылает всем входящим в коллекцию объектам сообщение о необходимости отображения представляемых ими фигур `sort` — сортирует коллекцию объектов
`Display`	`drawLine` — выводит на экран линию `drawCircle` — выводит на экран окружность

В этом случае алгоритм основной программы может быть следующим.

1. Основная программа создает экземпляр объекта, представляющего базу данных.
2. Основная программа посылает объекту базы данных указание отыскать сведения о всех затребованных фигурах и создать коллекцию объектов, содержащих сведения по отдельным фигурам (в действительности в коллекцию будут помещены только два типа объектов: `circle` и `square`, поскольку наш объект коллекции может содержать только эти типы объектов).
3. Основная программа посылает объекту коллекции указание упорядочить помещенные в нее объекты.
4. Основная программа посылает объекту коллекции указание вывести изображения фигур на экран.
5. Объект коллекции посылает каждому входящему в него объекту указание вывести на экран изображение представляемой им фигуры.
6. Каждый объект фигуры самостоятельно осуществляет вывод на экран той фигуры, которую он представляет, посылая соответствующие сообщения объекту `Display`.

Давайте посмотрим, как предложенная выше схема позволяет учитывать изменение существующих требований (не забывайте, что требования к программному обеспечению постоянно изменяются). Допустим, что изменение требований заключается в следующем.

- **Добавить новый тип фигуры** (например треугольник). Для введения нового типа фигуры необходимо выполнить только следующие два действия:
  - создать новый класс (производный от класса **Shape**), предназначенный для описания вновь добавляемой фигуры;
  - в этом новом классе реализовать такую версию метода отображения, которая будет соответствовать добавляемой фигуре.
- **Изменить алгоритм сортировки**. Для изменения метода, отвечающего за сортировку фигур, необходимо выполнить только одно действие:
  - внести требуемые изменения в метод сортировки объекта **Collection**, при этом новый алгоритм будет использоваться для упорядочения фигур всех типов.

Можно сделать вывод, что применение объектно-ориентированного подхода снижает влияние изменения требований на систему.

Кроме того, определенные преимущества достигаются и за счет инкапсуляции. Тот факт, что инкапсуляция позволяет скрывать информацию от пользователя, имеет следующие следствия.

- Упрощается использование отдельных объектов, поскольку пользователь не вникает в подробности способа их реализации.
- Способ реализации объекта может быть изменен без необходимости дополнительной проверки обращающихся к нему объектов. (Поскольку изначально вызывающие объекты не имели никаких сведений о реализации того объекта, к которому они обращаются, то изменение его реализации не будет иметь для них никакого значения.)
- Внутренние компоненты объекта неизвестны другим объектам — они используются только этим объектом с целью реализации функциональности, определяемой его интерфейсом.

И наконец, рассмотрим проблему нежелательных побочных эффектов, которая часто возникает при изменении функций. Устранение ошибок такого рода весьма эффективно обеспечивается инкапсуляцией. Внутренние компоненты объектов неизвестны другим объектам. Если использовать принцип инкапсуляции и следовать правилу, согласно которому объект должен сам отвечать за свое поведение, то оказать на объект некоторое воздействие можно будет только посредством вызова его методов. В результате данные объекта и механизмы реализации его функциональности оказываются полностью защищенными от изменений со стороны других объектов.

Инкапсуляция — эффективное средство защиты

- Чем больше на объект возлагается ответственности за его поведение, тем меньше ответственности остается на долю управляющей программы.
- Инкапсуляция позволяет сделать внесение изменений во внутренние механизмы объекта совершенно прозрачным для других объектов.
- Инкапсуляция помогает предотвратить нежелательные побочные эффекты.

Специальные методы объектов

Выше обсуждались методы, которые либо используются самим объектом, либо вызываются другими объектами. Но как происходит создание объектов или их удаление? Поскольку объекты являются самодостаточными единицами, вероятно, полезно было бы иметь методы, обрабатывающие эти специфические ситуации.

Действительно, такие специальные методы существуют и называются, соответственно, *конструкторами* (constructor) и *деструкторами* (destructor).

Конструктор — это специальный метод, который вызывается при создании объекта. Его назначение — обеспечить создание данного объекта. Это одна из важных сторон ответственности объекта за свое собственное поведение. Конструктор — это самое подходящее место для инициализации элементов объекта, установки значений данных-членов, принимаемых по умолчанию, определения отношений с другими объектами и выполнения любых других действий, необходимых для создания объек-

тов, полностью подготовленных к работе. Все объектно-ориентированные языки поддерживают методы-конструкторы и организуют их вызов при создании объектов.

Правильное применение конструкторов позволяет устранить (или по крайней мере сократить) использование неинициализированных переменных. Этот тип ошибок является обычным следствием проявленной разработчиком небрежности. Когда существует единая схема и некоторое фиксированное место (в данном случае это конструкторы объектов) инициализации всех элементов текста программы, можно быть уверенным в том, что эта инициализация действительно произойдет. Ошибки, вызванные обращением к неинициализированным переменным, легко исправить, но трудно найти. Поэтому подобное соглашение (автоматический вызов метода-конструктора) позволяет повысить эффективность работы программистов.

Деструктор — это специальный метод, обеспечивающий необходимую очистку используемых ресурсов перед тем, как объект перестанет существовать; т.е. перед уничтожением объекта. Все объектно-ориентированные языки поддерживают методы-деструкторы и выполняют их вызов при удалении объекта.

Деструктор обычно используется для освобождения ресурсов, использовавшихся объектом при выполнении возложенных на него задач. Поскольку язык Java включает встроенные средства автоматической сборки мусора (автоматического удаления объектов, уже выполнивших свою задачу), в языке Java функции-деструкторы не так важны, как в языке C++. В языке C++ деструкторы объектов обычно включают действия по уничтожению других объектов, использовавшихся только данным объектом.

Резюме

В этой главе было показано, как применение объектной технологии может ослабить влияние частых изменений требований к системе. Кроме того, мы выяснили, чем объектная технология отличается от метода функциональной декомпозиции.

Были рассмотрены важнейшие концепции объектно-ориентированного программирования и представлены основные используемые термины с пояснением их смысла и назначения. Усвоение этого материала очень важно для успешного восприятия концепций, обсуждаемых в остальной части этой книги (табл. 1.4 и 1.5).

Таблица 1.4. Основные концепции объектно-ориентированного программирования

Концепция	Краткое описание
Функциональная декомпозиция	Программисты, использующие структурированный подход, обычно начинают работу над проектом с выполнения функциональной декомпозиции. Метод функциональной декомпозиции заключается в разбиении задачи на более мелкие подзадачи (функции). Каждая функция, в свою очередь, разбивается на более мелкие до тех пор, пока каждая полученная функция не станет легко реализуемой
Изменение требований	Изменение требований пользователя к системе — это неотъемлемая часть общего жизненного цикла программного обеспечения. Вместо того, чтобы обвинять пользователя (или себя) по поводу мнимой невозможности выработки корректных и полных требований к системе, следует использовать такие методы проектирования, которые позволят эффективно модернизировать систему в соответствии с изменяющимися требованиями

Окончание таблицы

Концепция	Краткое описание
Объекты	Объекты определяются по кругу выполняемых ими обязанностей. Использование объектов упрощает алгоритм работы программы, поскольку каждый объект сам отвечает за свое поведение
Конструкторы и деструкторы	Объект включает следующие специальные методы, которые вызываются при его создании и удалении: • конструкторы, которые выполняют инициализацию или исходную настройку объекта; • деструкторы, которые уничтожают объект при его удалении. Все объектно-ориентированные языки поддерживают методы конструкторы и деструкторы, позволяющие упростить управление объектами

Таблица 1.5. Объектно-ориентированная терминология

Термин	Определение
Абстрактный класс	Определяет методы и общие свойства некоторого множества классов, подобных друг другу на концептуальном уровне. Реализация абстрактных классов невозможна
Свойство	Данные, принадлежащие объекту (также называются данными-членами класса)
Класс	"Проект" объекта содержит описание методов и данных объектов соответствующего типа
Конструктор	Специальный метод, который вызывается при создании экземпляра объекта
Инкапсуляция	Некоторый вид сокрытия информации. Объекты инкапсулируют свои данные. Абстрактные классы инкапсулируют свои порожденные конкретные классы
Порожденный класс	Класс, который является специализацией своего суперкласса. Включает все свойства и методы суперкласса, но может также включать собственные свойства и иную реализацию методов суперкласса
Деструктор	Специальный метод, который вызывается при удалении объекта
Функциональная декомпозиция	Один из методов анализа, при котором поставленная задача разбивается на более мелкие подзадачи-функции
Наследование	Механизм получения классом специальных признаков, устанавливающий связь между классами-потомками и классами-родителями (возможно, абстрактными)
Экземпляр	Конкретный объект, относящийся к некоторому классу
Реализация	Процесс создания экземпляра класса

Окончание таблицы

Концепция	**Краткое описание**
Компонент	Отдельный элемент данных или метод класса
Метод	Функция, входящая в состав объекта
Объект	Сущность, наделенная определенными обязанностями. Особая, самодостаточная структура, содержащая как данные, так и методы, манипулирующие этими данными. Данные объекта защищены от доступа со стороны внешних объектов
Полиморфизм	Способность родственных объектов различным образом реализовывать методы, присущие их типу
Суперкласс	Класс, от которого происходят другие классы. Содержит базовые определения свойств и методов, которые будут использоваться всеми классами-потомками (возможно, с переопределением)

Глава 2

UML — унифицированный язык моделирования

Введение

Эта глава содержит краткий обзор основных понятий языка UML (Unified Modeling Language — унифицированный язык моделирования). UML — это специализированный язык моделирования, который используют специалисты, работающие с объектно-ориентированными технологиями. Если вы еще не знакомы с языком UML, то, прочитав эту главу, вы получите те минимальные знания, которые необходимы для ясного понимания всех рисунков и диаграмм, содержащихся в книге.

В этой главе представлено следующее.

- Общее описание языка UML и его назначения.
- Диаграммы языка UML, широко используемые в этой книге:
  - диаграмма классов;
  - диаграмма взаимодействий.

Что такое язык UML

Язык UML является визуальным языком (использующим графические элементы с буквенными обозначениями), который предназначен для создания моделей программ. Под моделями программ понимается графическое представление программ в виде различных диаграмм, отражающих связи между объектами в программном коде.

Язык UML включает несколько типов диаграмм: одни из них предназначены для проведения анализа, другие — для проектирования, третьи — для целей реализации (или, выражаясь более точно, для распределения программного кода между объектами). Назначение различных диаграмм описано в табл. 2.1. Каждая диаграмма отражает те или иные связи между различными наборами сущностей в зависимости от ее типа.

Таблица 2.1. Диаграммы языка UML и их назначение

Когда используются	Наименование
На стадии анализа	• **Диаграммы прецедентов** (Use Case Diagrams). Представляют различные сущности, взаимодействующие с системой (например, пользователей или другие программные системы), и указывают функции, которые необходимо реализовать в создаваемой системе.

Окончание таблицы

Когда используются	Наименование
	• **Диаграммы видов деятельности** (Activity Diagrams). Предназначены для представления последовательностей бизнес-операций, существующих в проблемном домене. Под последним подразумевается та реальная среда, в которой работают люди и функционируют другие агенты — т.е. предметная область программной системы. *Примечание.* Поскольку в книге обсуждаются преимущественно вопросы проектирования, то диаграммы прецедентов и диаграммы видов деятельности в ней не используются
Изучение взаимодействия *объектов*	• **Диаграммы взаимодействий** (Interaction Diagrams). Представляют способы взаимодействия между собой конкретных объектов. Поскольку они чаще используются для рассмотрения конкретных случаев, нежели общих ситуаций, то представляют наибольший интерес при контроле исходных требований к проектам или при проверке самих проектов. Наиболее распространенным типом диаграмм взаимодействий является **Диаграмма последовательностей** (Sequence Diagram)
На этапе проектирования	• **Диаграммы классов** (Class Diagrams). Детально представляют различные виды взаимодействия отдельных классов системы
Изучение *поведения* объектов в зависимости от того состояния, в котором они находятся	• **Диаграммы состояний** (State Diagrams). Подробно представляют различные состояния, в которых могут находиться объекты, а также возможные переходы между этими состояниями
На этапе реализации	• **Диаграммы развертывания** (Deployment Diagrams). Показывают, как разворачиваются различные модули. В данной книге диаграммы этого типа обсуждаться не будут

Для чего используется язык UML

Прежде всего язык UML — это инструмент общения с самим собой, членами команды и клиентами. Некорректно сформулированные требования к программному продукту (неполные или неточные) — широко распространенное явление в области разработки программного обеспечения. Язык UML позволяет разработать более качественные исходные требования.

С помощью языка UML можно определить, является ли достигнутое при анализе понимание системы адекватным. Поскольку большинство систем обладает сложной структурой и включает различного рода информацию, которую требуется обрабатывать так или иначе, разработчику предлагается набор диаграмм, различающихся по типу представляемой на них информации.

Проще всего оценить достоинства языка UML, если вспомнить несколько последних обсуждений проектов, в которых вам приходилось принимать участие. Если на подобных обсуждениях кто-либо сообщает о написанном им коде и пытается охарактеризовать его без использования языка моделирования, подобного UML, то, вероятнее всего, разговор окажется путанным и бесконечным. Язык UML предлагает не только оптимальный путь описания проектов, созданных с применением объектных технологий, но также вынуждает разработчика более четко формулировать используемые им принципы (поскольку их требуется изложить в письменном виде).

Диаграмма классов

Одной из основных диаграмм языка UML является диаграмма классов. Она описывает классы и отражает отношения, существующие между ними. Возможны следующие типы отношений.

- **Отношение *is-a*** — в этом случае один класс является подвидом другого класса.
- Когда существует взаимосвязь между двумя классами:
  - **отношение *has-a*** — т.е. один из классов "содержит" другой класс;
  - **отношение *uses*** — т.е. один класс "использует" другой класс.

Обсуждение этой темы можно продолжить. Например, выражение "одно содержит другое" может означать следующее.

- Один объект является частью другого объекта, который его содержит (как двигатель в автомобиле).
- Имеется набор (коллекция) объектов, которые могут существовать сами по себе (как самолеты в аэропорту).

Первый пример получил название *объединение* (composition), а второй — *агрегация* (aggregation).[1]

На рис. 2.1 представлено несколько важнейших моментов. Прежде всего, каждый прямоугольник здесь соответствует определенному классу. В языке UML представление класса может включать до трех элементов.

- Имя класса.
- Имена данных-членов класса.
- Имена методов (функций) класса.

---

[1] Гамма (Gamma), Хелм (Helm), Джонсон (Johnson) и Влиссайдес (Vlissides) ("банда четырех") первый пример назвали "aggregation" (агрегация), а второй – "composition" (соединение) — как раз наоборот по отношению к тому, как это принято в языке UML. Однако книга "банды четырех" была написана раньше завершения разработки языка UML. Тем не менее, представленные определения сейчас чаще ассоциируются именно с языком UML. Это иллюстрирует один из важнейших мотивов создания языка UML — до его появления существовало несколько различных языков моделирования и каждый имел собственные понятия и термины.

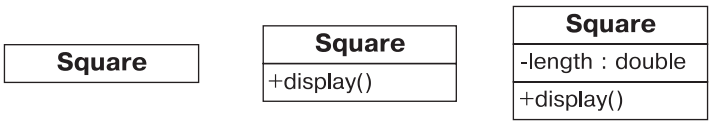

РИС. 2.1. *Диаграмма классов – три возможных варианта представления класса*

На диаграмме представлены три различных способа описания класса.

- *Прямоугольник слева* содержит лишь имя класса. Этот тип представления класса используется, если отсутствует необходимость в указании более детальной информации о данном классе.
- *Средний прямоугольник* содержит имя как класса, так и его метода. В данном случае можно сделать вывод, что класс **`Square`**[2] имеет метод `display()`. Символ плюс (+) перед словом `display` (именем метода) означает, что этот метод является открытым — т.е. его могут вызывать не только объекты данного класса.
- *Прямоугольник справа*, кроме предыдущих двух понятий (имя и методы класса), представляет и данное-член класса. В этом случае знак минус (-) перед данным-членом `length` (которое имеет тип `double`) означает, что оно является закрытым — т.е. оно недоступно никаким другим объектам, кроме того, которому оно принадлежит.[3]

Условные обозначения языка UML, описывающие возможности доступа

Средства языка UML позволяют контролировать доступ к данным-членам и методам классов. В частности, для каждого существующего члена класса можно указать присвоенный ему уровень доступа. В большинстве языков программирования приняты три уровня доступа.

- **Public** (открытый) – обозначается знаком *плюс* (+).

  Означает, что эти данные или методы могут использовать любые объекты.

- **Protected** (защищенный) – обозначается знаком *номер* (#).

  Означает, что только данный класс и все его потомки (а также все классы-потомки, порожденные от его потомков) могут получать доступ к этим данным или методам.

- **Private** (закрытый) – обозначается знаком *минус* (-).

  Означает, что только методы данного класса могут получать доступ к этим данным или методам. (Примечание: в некоторых языках программирования это ограничение усиливается до конкретного экземпляра объекта.)

На диаграмме классов можно также указать отношения между различными классами. На рис. 2.2 показано отношение, существующее между классом **`Shape`** и несколькими порожденными от него классами.

---

[2] *Здесь и далее при указании имени класса оно будет выделяться полужирным шрифтом.*

[3] *В некоторых языках программирования объекты одного типа могут совместно использовать закрытые данные-члены друг друга.*

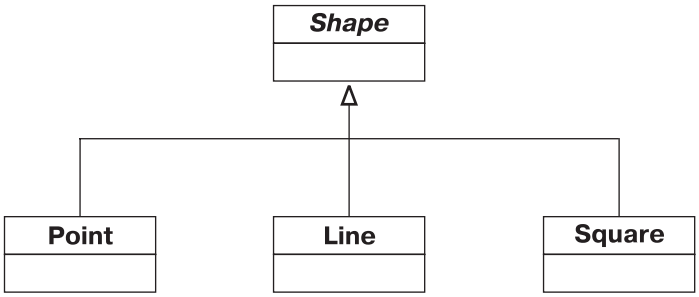

РИС. 2.2. *Диаграмма классов, представляющая отношение* **is-a**

На рис. 2.2 представлено несколько моментов. Во-первых, стрелка под классом ***`Shape`*** означает, что другие классы, указывающие на класс ***`Shape`***, являются его потомками. Далее, дополнительное выделение имени класса ***`Shape`*** курсивом означает, что этот класс является абстрактным. Абстрактный класс — это такой класс, который используется только для определения интерфейса своих классов-потомков.

Фактически существует два различных вида отношения ***has-a***. Один объект может содержать другой объект, который либо является его частью, либо нет. На рис. 2.3 показан класс **`Airport`** (аэропорт), содержащий класс ***`Aircraft`*** (летательный аппарат). Класс ***`Aircraft`*** не является частью класса **`Airport`**, но можно сказать, что класс **`Airport`** содержит его. Этот тип отношений получил название *агрегации*.

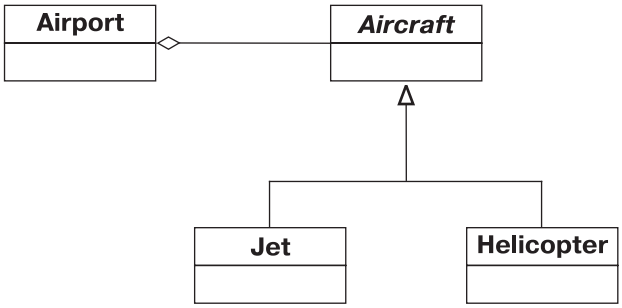

РИС. 2.3. *Диаграмма классов, представляющая отношение* **has-a**

На этой диаграмме также показано, что от класса ***`Aircraft`*** порождены классы **`Jet`** (реактивный самолет) и **`Helicopter`** (вертолет). Можно сделать вывод, что класс ***`Aircraft`*** является абстрактным классом, поскольку его имя выделено курсивом. А это значит, что класс **`Airport`** будет включать либо объект класса **`Jet`**, либо объект класса **`Helicopter`**, но обращаться с этими объектами он будет одинаково — как с объектом класса ***`Aircraft`***. Пустой (незакрашенный) ромб с правой стороны класса **`Airport`** указывает на отношение агрегации между ним и классом ***`Aircraft`***.

Второй тип отношений ***has-a*** состоит в том, что вложенный объект является частью того объекта, в котором он содержится. Такой тип отношений получил название *объединение*.

На рис. 2.4 показано, что класс **`Tires`** (шины) является частью класса **`Car`** (машина) — т.е. машина состоит из шин и, вероятно, других деталей. Подобный тип отношений ***has-a***, называемый *объединением*, представляется закрашенным ромбом. На этой диаграмме также показано, что класс **`Car`** *использует* класс **`GasStation`** (автозаправочная станция). Отношение использования представлено пунктирной линией со стрелкой. Иначе его иногда называют отношением взаимосвязи.

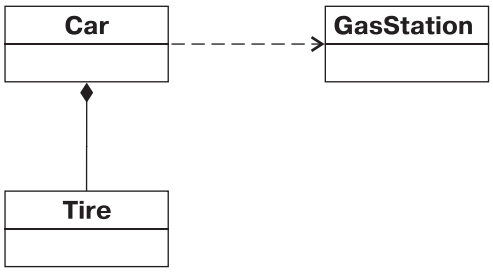

РИС. 2.4. ***Диаграмма классов, представляющая отношения объединения и использования***

Как объединение, так и агрегация предполагают, что объект содержит один или более других объектов. Но объединение подразумевает, что один объект является частью другого объекта, тогда как агрегация означает, что вложенные объекты представляют собой скорее некоторый набор или коллекцию самостоятельных элементов. Объединение можно рассматривать как нераздельную взаимосвязь, когда время существования вложенных объектов определяется содержащим их объектом. В данном случае функции создания и удаления вложенных объектов целесообразно возложить на методы конструктора и деструктора включающего их объекта.

Примечания в языке UML

На рис. 2.5 представлено новое обозначение – примечание. Примечанием здесь является элемент, в котором содержится текст "незакрашенный ромб означает агрегацию". На диаграммах примечания выглядят как листок бумаги с отвернутым правым углом. Часто они изображаются с линией, соединяющей их с конкретным классом, – таким образом выражается их принадлежность к данному классу.

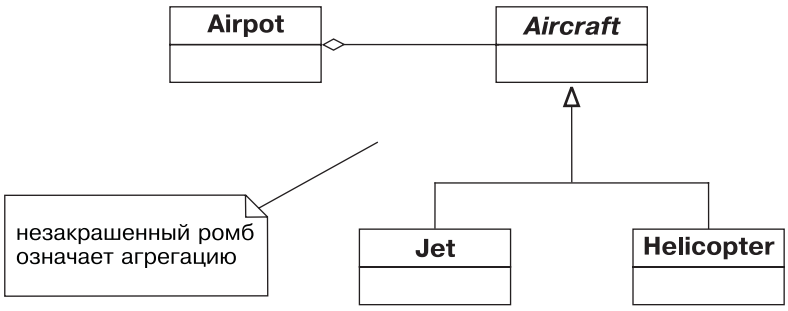

РИС. 2.5. ***Диаграмма классов с примечанием***

Диаграммы классов отображают отношения между классами. Однако отношения объединения и агрегации могут потребовать указать дополнительные сведения об объектах какого-либо конкретного типа. Например, верно, что класс **Airport** содержит класс ***Aircraft***, но точнее будет сказать, что конкретные аэропорты содержат конкретные самолеты. Может возникнуть вопрос: "Сколько самолетов может принять аэропорт?". С помощью этого вопроса мы вводим такое понятие, как *кардинальность* элементов отношения. Способы представления кардинальности на диаграммах классов показаны на рис. 2.6 и 2.7.

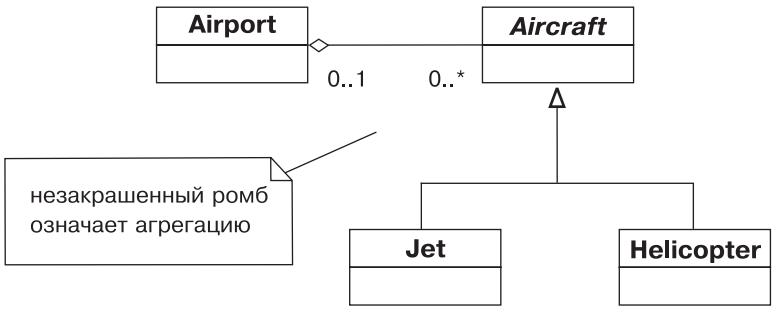

РИС. 2.6. *На диаграмме классов указана кардинальность отношения* ***Airport-Aircraft***

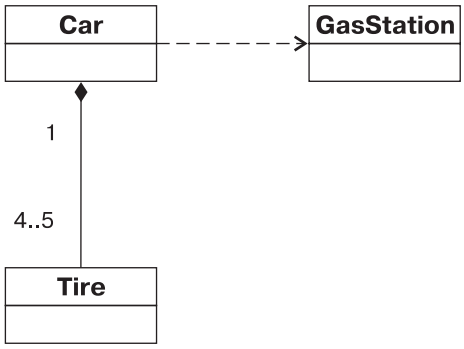

РИС. 2.7. *На диаграмме указаны классы кардинальности отношения* ***Car-Tire***

На рис. 2.6 показано, что объект класса **Airport** может включать от `0` до произвольного числа (здесь оно обозначено символом звездочки, но иногда может быть представлено буквой `n`) объектов класса ***Aircraft***. Цифры `0..1` со стороны класса **Airport** указывают, что объект класса ***Aircraft*** может принадлежать или только одному объекту класса **Airport**, или вовсе не принадлежать никакому объекту (например, самолет может находиться в полете).

На рис. 2.7 показано, что объект класса **Car** может включать либо `4`, либо `5` шин (машина может иметь запасную шину или не иметь ее). Шины всегда принадлежат только одной машине. Некоторые полагают, что отсутствие спецификации кардинальности отношения указывает на то, что в ней принимает участие только один объ-

ект. Однако это неправильно. В действительности, если спецификации кардинальности отношения на диаграмме нет, мы не имеем права делать какие-либо выводы относительно количества участвующих в отношении объектов.

Как и ранее, на рис. 2.7 пунктирная линия между классами **Car** и **GasStation** указывает на существующее между ними отношение использования. В языке UML пунктирной линией принято обозначать семантические (смысловые) отношения между двумя элементами модели.

Диаграммы взаимодействий

Диаграммы классов отражают статические отношения между классами. Другими словами, на них не отображаются никакие действия. Тем не менее, иногда бывает очень полезно показать действительное взаимодействие отдельных экземпляров объектов, полученных при реализации различных классов.

Диаграммы языка UML, отражающие взаимодействие объектов друг с другом, получили название *диаграмм взаимодействий*. Наиболее распространенным видом диаграмм взаимодействий является диаграмма последовательностей, представленная на рис. 2.8.

Диаграммы последовательностей следует читать сверху вниз.

- Каждый прямоугольник в верхней части диаграммы представляет конкретный объект. Большинство прямоугольников содержит имена классов, причем обратите внимание, что в некоторых случаях перед именем класса стоит двоеточие. Другие прямоугольники содержат такие имена, как `shape1:Square`.
- Прямоугольники в верхней части диаграммы содержат имя класса (справа от двоеточия) и, возможно, имя объекта (указывается перед двоеточием).
- Вертикальные линии представляют жизненный путь объектов. К сожалению, большинство программ автоматизированной подготовки UML-диаграмм не поддерживают этот аспект и рисуют линии от верхнего края и до конца листа, оставляя неясными действительные сроки существования объекта.
- Обмен сообщениями между объектами отображается с помощью горизонтальных линий, проведенных между соответствующими вертикальными линиями.
- Иногда возвращаемые значения и/или объекты должны быть указаны подробно, а иногда и так понятно, что они возвращаются.

Например, еще раз обратимся к рис. 2.8.

- Слева вверху показана подпрограмма `Main`, которая посылает объекту **ShapeDB** (который не поименован) сообщение с требованием извлечь сведения о фигурах (`getShapes`). Получив этот запрос, объект **ShapeDB** выполняет следующее.
  - Создает экземпляр класса **Collection**.
  - Создает экземпляр класса **Square**.
  - Добавляет объект класса **Square** в объект класса **Collection**.
  - Создает экземпляр класса **Circle**.
  - Добавляет объект класса **Circle** в объект класса **Collection**.
  - Возвращает указатель на объект класса **Collection** в вызывающую подпрограмму (`Main`).

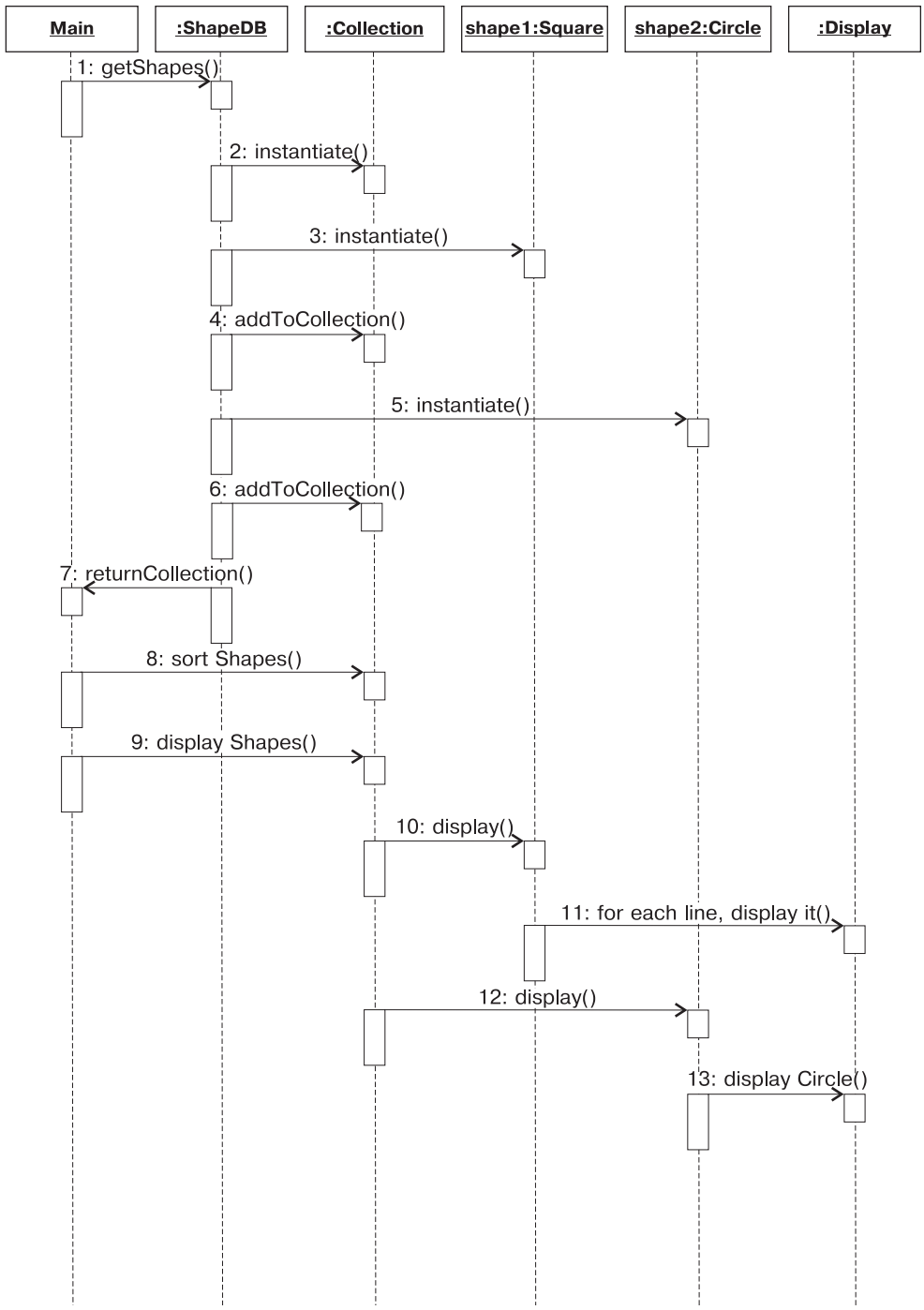

РИС. 2.8. *Диаграмма последовательностей для программы обработки фигур*

Чтобы выяснить дальнейшие действия программы, достаточно прочитать в указанной выше манере оставшуюся часть диаграммы. Данная диаграмма называется диаграммой последовательностей, поскольку она отражает последовательность выполняемых операций.

Нотация `Object:Class`

В некоторых диаграммах языка UML необходимо представлять обращения к объектам, указывая класс, от которого они порождены. В этом случае имена объекта и класса записываются через двоеточие. На рис. 2.8 запись **`shape1:Square`** указывает, что выполняется обращение к объекту **`shape1`**, являющемуся экземпляром класса **`Square`**.

Резюме

Язык UML предоставляет разработчику средства как демонстрации, так и анализа проектов. Не стоит особо беспокоиться относительно строгости начертания создаваемых диаграмм. Лучше сосредоточить внимание на тщательной проработке как основных концепций, так и всех деталей создаваемого проекта. Главные рекомендации заключаются в следующем.

- Если появляется необходимость выразить что-либо на диаграмме обычными словами, воспользуйтесь примечанием.
- Если у вас были сомнения по поводу назначения некоторой пиктограммы или символа, и для уточнения их значения потребовалось отыскать некоторые дополнительные источники, целесообразно поместить на диаграмму примечание с соответствующими разъяснениями, поскольку и другие могут испытывать затруднения в их истолковании.
- Всегда стремитесь к максимальной ясности создаваемых диаграмм.

Не вызывает сомнений, что следует избегать использования языка UML в какой-либо нестандартной манере — это лишь осложняет работу. Просто постарайтесь четко представить себе, что именно вы стремитесь передать при построении диаграмм.