Глава 1

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

В разработке программного обеспечения (ПО) проектирование часто упоминается как этап, *предваряющий* программирование. Это не так. В реальности анализ, проектирование и программирование взаимозависимы и связаны друг с другом. В книге будут приведены несколько листингов, где невозможно разделить программирование и проектирование. Одна из особенностей языка Python заключается в том, что он позволяет хорошо структурировать код.

В этой главе мы разберемся, как перейти от хорошей идеи к программированию как таковому. В процессе проектирования мы сформируем наглядные элементы, а именно диаграммы — они помогут продумать структуру кода, прежде чем приступать к его написанию. Итак, рассмотрим следующие темы.

- Что такое объектно-ориентированное проектирование.
- Различие между объектно-ориентированным проектированием и объектно-ориентированным программированием.
- Базовые принципы объектно-ориентированного проектирования.
- Базовые концепции унифицированного языка моделирования (UML) и когда его следует применять.

В тематическом исследовании этой главы рассмотрим архитектурную модель представления 4+1. Нам предстоит:

- кратко рассмотреть классическое приложение на основе машинного обучения, познакомиться с известной задачей классификации ирисов;
- изучить общее окружение (контекст) этого классификатора ирисов;
- набросать два представления иерархии классов, необходимых для решения задачи классификации ирисов.

Введение в объектно-ориентированное программирование

Все знают, что объекты — это предметы, которые можно потрогать, ощутить и использовать. Для детей объекты — игрушки. Деревянные кубики, пластиковые формочки, пазлы — первые объекты, с которыми человек сталкивается в жизни. Некоторые объекты выполняют строго определенные действия: колокольчик звенит, кнопка нажимается, рычаг передвигается.

То же можно сказать про объекты в разработке ПО. Да, их нельзя потрогать, но и эти объекты делают что-то конкретное. Точное определение таково: объект — коллекция (набор) **данных** и **поведения**.

Так что же в таком случае значит «объектно-ориентированное программирование»? *«Ориентированный»* трактуется как *«направленный»*, значит, объектно-ориентированное программирование — программирование, которое моделирует поведение реальных объектов. Это один из способов описать сложную систему. Она состоит из взаимодействующих объектов — каждый со своими данными и поведением.

Если вы знакомы с концепцией ООП, то знаете о существовании *объектно-ориентированного анализа*, *объектно-ориентированного проектирования*, *объектно-ориентированного анализа и проектирования*, а также *объектно-ориентированного программирования*. Это все отдельные части общей концепции объектно-ориентированного подхода. Фактически анализ, проектирование и программирование — различные стадии жизненного цикла разработки программного обеспечения.

В целях упрощения будем называть совокупность этих стадий объектно-ориентированным программированием.

Объектно-ориентированный анализ (ООА) — процесс изучения проблем, систем и задач программного обеспечения, а также определение объектов и взаимодействия между ними. На стадии анализа мы отвечаем на вопрос *«Что мы хотим получить?»*.

Результат этой стадии процесса — *четко сформулированные требования к программному обеспечению*. Например, задачи должны быть сформулированы не в виде пользовательских историй: «Мне как ботанику нужен сайт, который помогает пользователям классифицировать растения. Так я смогу помочь корректно определить вид и род растений». А необходимо указать по шагам действия пользователей на сайте, как это сделано в перечне ниже; здесь курсив

обозначает действия, а полужирный шрифт — объекты. Так, пользователь должен иметь возможность:

- *посмотреть* **предыдущие загрузки**;
- *загрузить* **известный экземпляр**;
- *протестировать* **качество**;
- *посмотреть* **продукты**;
- *рассмотреть* **рекомендации**.

Термин *«анализ»* неточен. Разве ребенок анализирует объекты? Вовсе нет — он исследует, пробует и открывает, что именно может получиться из кубиков и кусочков пазла. И правильнее говорить об «объектно-ориентированном исследовании». В разработке ПО формирование и анализ требований включают беседу с пользователями, изучение их поведения, отбор вариантов.

Объектно-ориентированное проектирование — процесс обработки полученных требований и составление спецификации. Проектировщик выделяет объекты и определяет их поведение, указывает, как одни объекты активизируют поведение других. На этой стадии мы отвечаем на вопрос «*Как* мы сделаем то, что хотим?».

Результат стадии проектирования — получение спецификации. Этот этап считается завершенным тогда, когда каждое требование, указанное в объектно-ориентированном анализе, представлено как класс и интерфейс, которые можно реализовать на любом объектно-ориентированном языке программирования (в идеальном случае).

Объектно-ориентированное программирование — процесс разработки программы, то есть реализация проекта в виде работающей программы, которая нужна заказчику.

Вот и все! И было бы прекрасно, если бы в реальной жизни мы работали строго по порядку, завершали один этап перед тем, как перейти к другому. Так, как нас научили проверенные учебники. На самом деле все гораздо сложнее. Независимо от того, как много усилий мы затратили, чтобы разделить фазы, — когда мы начнем проектировать, мы выясним, что требования нуждаются в дополнительном анализе. А когда начнем программировать, нам понадобится внести изменения в сам проект.

По мнению большинства разработчиков, в наши дни каскадная модель (некоторые называют ее «водопад») едва ли работает так однозначно. Часто приходится использовать итеративную модель разработки ПО. При итеративном подходе моделируются, разрабатываются и программируются небольшие части задач.

Затем продукт в целом пересматривается, вносятся улучшения и корректировки, формулируются новые требования. Так качество продукта постепенно улучшается, и он обретает новые особенности в каждом очередном коротком цикле разработки.

Основная задача книги — разобрать ООП, и в данной главе мы рассмотрим его принципы и их роль в проектировании ПО. Это позволит понять концепцию еще до подробного изучения синтаксиса Python.

Объекты и классы

Объект — это данные с ассоциируемым поведением. Как различать объекты? Да просто: как яблоки и апельсины. Говорят, их нельзя сравнивать, но оба этих предмета — объекты. Яблоки и апельсины нечасто кодируются разработчиками, но, допустим, вы создаете приложение инвентаризации для фруктовой компании. Также предположим, что сбор яблок идет в бочки, а апельсинов в корзины.

Мы установили четыре типа объектов: яблоки, апельсины, бочки и корзины. В ООП *тип объекта* называется **классом**. Говоря технически, у нас имеется четыре класса объектов.

Важно различать понятия. Классы обозначают связанные объекты. Можно сказать, что класс — шаблон создания объектов. Представим, что перед вами на столе три апельсина. Каждый из них — отдельный объект, но все три имеют общие атрибуты и поведение — они представляют один класс, общий класс апельсинов.

Отношения между четырьмя классами в нашем приложении инвентаризации могут быть описаны на диаграмме классов языка **UML** (Unified Modeling Language — унифицированный язык моделирования). Все довольно просто (рис. 1.1).

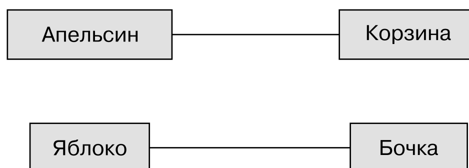

Рис. 1.1. Диаграмма классов

Диаграмма показывает, что экземпляр класса **Апельсин** (проще говоря, апельсины) ассоциируется с **Корзиной**, а образец класса **Яблоко** (яблоки) — с **Бочкой**. *Ассоциация* отражает отношение экземпляров двух классов.

Синтаксис UML очевиден, не нужно читать специальные учебники для понимания языка. UML легко представить наглядно. Довольно часто для описания классов и их отношений мы рисуем квадраты и линии между ними. Разработчики используют подобные интуитивно понятные диаграммы, чтобы общаться с бизнес-аналитиками, менеджерами и между собой.

Обратите внимание, UML-диаграмма отображает классы **Яблоко** и **Бочка**, а не атрибуты объекта. Класс **Яблоко** и класс **Бочка** показывают, что данное яблоко находится в бочке. В UML возможно отобразить индивидуальные объекты, но в этом нет нужды. Достаточно знать, что каждый объект — представитель класса.

Часть разработчиков считает, что на создание UML-диаграмм не стоит тратить силы. По их мнению, формальные спецификации в виде UML-диаграмм до реализации проекта не требуются, а после реализации их официальное сопровождение бесполезно и становится только тратой времени.

Но ведь каждая команда разработчиков обсуждает проект, так что UML — полезный коммуникативный инструмент. Даже в компаниях, которые им пренебрегают, на совещаниях прибегают к упрощенной версии UML.

Кроме того, этот инструмент понадобится в будущем. Рано или поздно разработчику придется вспоминать: «Почему я сделал так, а не иначе?» Диаграммы помогут взглянуть на картину в целом и вспомнить забытое.

В этой главе нет инструкций по UML. Вы можете самостоятельно поискать в Интернете или приобрести книги на эту тему, если возникнет такая необходимость. Тема визуализации обширна: показывать приходится и диаграммы класса, и объекты, и варианты использования, и изменение состояний и действий. Мы остановимся только на диаграммах класса. Вы сможете выбрать подходящие вам конструкции из примеров и затем применить синтаксис UML в своих проектах.

Вернемся к нашей диаграмме с рис. 1.1. Ее задача не в том, чтобы разработчик знал, что яблоки находятся в бочках или сколько в каждой из них яблок. Диаграмма указывает только на связи. Визуализация связей между классами — то, ради чего мы используем подобные изображения. Основная цель — более точно выразить отношения между классами.

UML примечателен возможностью подбирать инструменты языка по мере необходимости. Нужно только понять, что именно необходимо. На планерке мы нарисуем только линии между квадратами, тогда как в формальном документе укажем больше деталей.

В примере с яблоками и бочками известно, что в одной бочке хранится много яблок. Чтобы не допустить ошибки и не посчитать, что в одной бочке находится одно яблоко, можно дополнить диаграмму новой деталью (рис. 1.2).

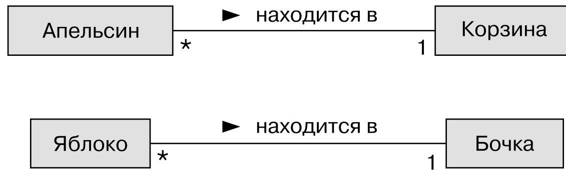

Рис. 1.2. Диаграмма классов детализированная

Диаграмма показывает, что апельсины **находятся в** корзинах, маленькая стрелка обозначает нахождение одного объекта в другом. Также речь идет о количестве объектов. Одна Корзина может содержать неограниченное количество (это обозначено на диаграмме символом *) объектов Апельсин, и каждый Апельсин находится только в одной Корзине. Это количество называют *мощностью ассоциации (multiplicity of the object)*. Вы могли также слышать другое название — *кардинальность (cardinality)*. Полезно, однако, понимать под кардинальностью некоторое число и диапазон, а под мощностью — обобщенное описание «число больше одного экземпляра».

Иногда можно забыть, с какой стороны линии находится число мощности. Мощность, ближайшая к классу, обозначает количество объектов класса, которые ассоциированы с каким-либо объектом с другой стороны линии. Так, для ассоциации яблок, находящихся в бочках: если читать слева направо, экземпляры класса **Яблоко** (объекты **Яблоко**) находятся в одной **Бочке**; и, наоборот, справа налево — только одна **Бочка** может быть ассоциирована с каким-либо числом **Яблок**.

Итак, мы уже знаем основы классов и уточнили взаимоотношения объектов. Теперь нужно разобрать атрибуты объекта, показатели состояния и поведение, которое связано с изменением состояния или взаимодействием с объектами.

Атрибуты и поведение

Еще раз о базовой терминологии ООП. Объекты — экземпляры класса, которые связаны между собой. Экземпляр класса — некоторый объект с определенными данными и поведением; апельсин на столе, перед нами, считается экземпляром общего класса апельсинов.

Апельсин имеет состояние: например, он спелый или неспелый; мы судим о состоянии объекта по его атрибутам. Также апельсин имеет некоторое поведение. Сам по себе он неподвижен, а вот его состояние может изменяться. Давайте разберем подробнее два термина — *«состояние»* и *«поведение»*.

Данные — показатель состояния объекта

Обратимся к данным. Данные обозначают индивидуальные особенности объекта, его состояние, а класс — общие особенности, признаки, свойственные всем объектам класса. При этом каждый конкретный объект имеет свои значения данных для каждого признака. Например, три апельсина на столе (вы же еще не съели их, правда?) могут иметь разный вес. Класс **Апельсин** имеет атрибут **Вес** для представления этих данных. Все экземпляры класса **Апельсин** имеют атрибут **Вес**, но его значение для каждого апельсина индивидуально. Значение атрибутов, кстати, не обязательно уникально, два апельсина могут иметь одинаковый вес.

Атрибуты часто обозначаются как **члены** или **свойства**. Некоторые авторы разграничивают два термина — «атрибуты» и «свойства». Например, говорят, что значения атрибутов можно устанавливать, а свойства доступны только для чтения. Но на языке Python подобное разграничение бессмысленно: свойство можно перевести в режим *«только для чтения»*, но его значения будут основаны на значении, которое в конечном счете доступно для записи. В тексте книги мы используем эти термины как синонимы. Кроме того, в главе 5 будет описан случай, когда ключевое слово «свойство» применяется в узком смысле для обозначения атрибутов специального типа.

В Python атрибут называют также **экземпляром переменной**, что помогает понять, как атрибут работает. Атрибуты — это переменные с уникальными значениями для каждого экземпляра класса. Python содержит и другие виды атрибутов, но для начала ограничимся этим упрощенным описанием.

В нашем приложении инвентаризации фруктов фермер может поинтересоваться: из какого сада апельсин? Когда он сорван? Сколько он весит? Также полезно знать, в какой корзине хранится апельсин. Яблоки могут иметь атрибут цвета, а бочки могут иметь разные размеры.

Некоторые атрибуты могут принадлежать нескольким классам (нам же интересно знать и о яблоках, когда они сорваны). Но пока, для первого примера, добавим в диаграмму только некоторые атрибуты (рис. 1.3).

В зависимости от того, насколько подробно нужно проработать структуру, может понадобиться указать тип каждого значения атрибута. В UML атрибуты типа имеют общепринятые названия, точно такие же, как в основных языках программирования: целое число, число с плавающей точкой, строка, байт или логический (булев) тип. Однако атрибуты могут также обозначать коллекции: списки, деревья и графы — и даже, что очень важно, неуниверсальные, специфические для данного приложения классы. Очевидно, что здесь налицо

пересечение этапов проектирования и программирования. Различные примитивы и встроенные коллекции, доступные на одном языке программирования, могут не использоваться в другом.

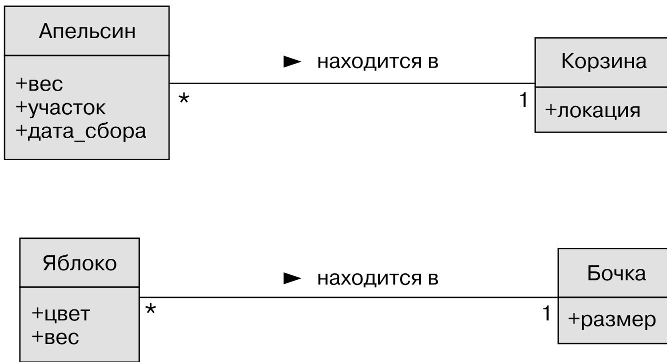

Рис. 1.3. Диаграмма класса с атрибутами

На рис. 1.4 приведен пример диаграммы с использованием подсказок типов на языке Python.

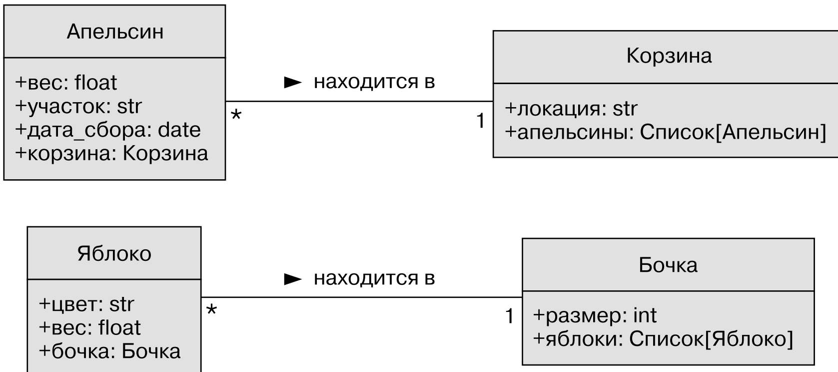

Рис. 1.4. Диаграмма класса с атрибутами и их типом

Обычно на этапе проектирования нет необходимости беспокоиться о точном определении типов данных, конкретная их реализация подбирается на этапе программирования. Достаточно оказывается общих имен. Поэтому можно просто назвать тип `date` (дата) вместо точного обозначения `datetime.datetime`. Если в нашем проекте требуется тип контейнера списка, Java-программисты при

реализации могут выбрать связанный список (`LinkedList`) или списочный массив (`ArrayList`), а программисты на Python (да, это мы!) могут указать `List[Apple]` как подсказку типа и реализовать тип `list`.

В рассматриваемой сейчас фруктовой компании все описанные атрибуты — базовые примитивы. Но есть также неявные атрибуты, и дальше мы сможем их сделать явными — ассоциациями. Конкретный апельсин ссылается на корзину, в которой находится много апельсинов. Ссылка указывает на атрибут `basket` с подсказкой типа `Basket`.

Поведение — это действия

Выяснив, как данные определяют состояние объекта, разберем последний неизвестный термин — *«поведение»*. Поведение — это действия, которые происходят с объектом. Поведение, которое определяется классом объекта, называется **методами** этого класса. На программном уровне методы сравнимы с функциями в структурном программировании, но, в отличие от последних, методы имеют доступ к атрибутам, в частности к переменным с данными, которые связаны с объектом. Так же как функции, методы принимают **параметры** и возвращают **значение**.

Параметры метода предоставляются ему как коллекция (набор) объектов, которые необходимо **передать** в метод. **Аргументы** — фактически переданные экземпляры объекта во время вызова метода. При этом передаваемые переменные связаны с переменными **параметров** в теле метода. Метод использует их независимо от того, для какого типа задач и достижения какой цели он оказывается вызван. Обычно результат выполнения метода — возвращаемые значения. Но возможна и другая ситуация: его действие направлено на изменение внутреннего состояния объекта.

Продолжим развитие нашего базового примера (пусть и несколько искусственного) — приложения инвентаризации. Посмотрим, будет ли работать это приложение как задумано. Одно действие, связанное с апельсинами, — **собрать** их. Это простое действие, реализуемое за два этапа.

1. Поместить апельсин в корзину и обновить атрибут **Корзина** в объекте **Апельсин**.
2. Добавить апельсин в список **Апельсин** в объекте **Корзина**.

При этом методу **собрать** нужно знать, в какую корзину кладется апельсин. Это выполняется, когда методу **собрать** передается параметр **Корзина**. Потом наша фруктовая ферма начнет **продавать** сок, нам нужно будет добавить метод

выжать к классу **Апельсин**. Вызывая метод **выжать**, мы захотим получить обратно количество сока и при этом **Апельсин** удалить из **Корзины**.

Класс **Корзина** содержит метод **продать**. Когда корзина продана, наша система инвентаризации обновляет данные по объектам, которые еще не указаны нами, для расчета дохода. В другом случае корзина с апельсинами может еще до продажи оказаться испорченной, потому нужно добавить метод **выбросить**. Добавим эти методы на диаграмме (рис. 1.5).

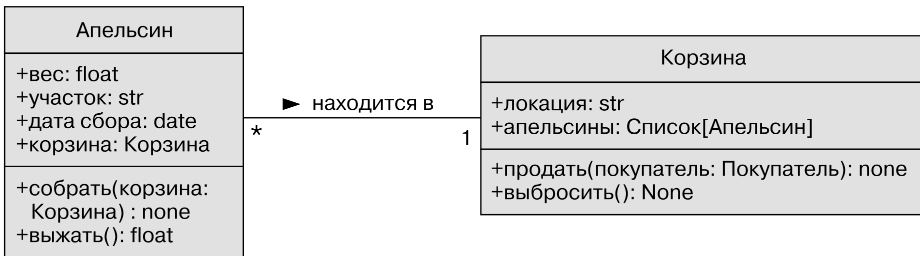

Рис. 1.5. Диаграмма класса с атрибутами и методами

Добавление атрибутов и методов позволяет создать и описать систему взаимодействия между объектами. Каждый объект в системе — представитель определенного класса. Эти классы содержат типы данных объекта и вызываемые методы. Данные каждого объекта могут иметь разные состояния, отличаясь от состояний других экземпляров того же класса. Каждый объект реагирует на вызов метода по-своему, в зависимости от значений состояния.

Объектно-ориентированный анализ и объектно-ориентированное проектирование выявляют, что является объектом и как объекты взаимодействуют между собой. Каждый класс имеет свою зону ответственности и способы взаимодействия. В следующем разделе разберем, какие принципы делают такое взаимодействие простым и понятным.

И обратите внимание, что продажа корзины не является безусловной особенностью класса **Корзина**. Вполне возможно, что другие классы (здесь не указанные) ответственны за разные корзины и их перемещение. В каждом отдельном проекте часто присутствуют свои условия и ограничения. Резонно возникает вопрос, как именно ответственность распределена между разными классами. Не всегда очевидно единственно верное техническое решение разделения ответственности между классами, и часто приходится неоднократно вносить изменения в UML-диаграммы, чтобы исследовать разные варианты.

Сокрытие информации и создание общедоступного интерфейса

Ключевая задача моделирования объекта в объектно-ориентированном проектировании — определить, какой будет внешний **интерфейс** данного объекта. Интерфейс — коллекция, набор атрибутов и методов, доступных для взаимодействия с другими объектами. Особо стоит подчеркнуть, что свободный доступ извне к внутренней работе объекта не нужен, а в некоторых языках и запрещен.

Возьмем пример реального мира — телевизор. Пульт управления — наш интерфейс. Каждая кнопка пульта управления представляет метод, который мы вызываем, чтобы повлиять на объект «телевизор». Когда нажимаем кнопку, вызываем объект и обращаемся к методам, нам неинтересно, как именно телевизор получит сигнал — через кабель, спутник или Интернет. Нас не интересует, как идет электрический сигнал, когда мы увеличиваем громкость, неважно даже, будет ли звук передан в динамики или в наушники. Разбирая телевизор, чтобы что-либо починить внутри, например разделить выход на наушники и динамики, мы лишаемся гарантии.

Сокрытие внутренней реализации объекта называют **сокрытием информации**. Также об этом можно сказать, что информация **инкапсулирована**, хотя это более широкий термин. Инкапсулирование данных не обязательно означает сокрытие. Буквально «инкапсуляция» означает «помещение в капсулу» или «упаковку» атрибутов. Внешний корпус ТВ заключает в себя (инкапсулирует) состояние и поведение телевизора. Мы имеем доступ к внешнему экрану, динамикам и пульту управления. Но у нас нет доступа к связке усилителей или приемников ТВ.

При покупке компонента развлекательной системы мы изменяем уровень инкапсулированности и глубже вникаем во взаимодействие между компонентами. Если же мы внедряем IoT-устройство, нам оказывается важно понять, что внутри, какая именно начинка скрыта производителем.

Различие между инкапсуляцией и сокрытием информации не слишком важно в большинстве случаев, особенно на стадии проектирования. Во многих практических руководствах эти термины используют как взаимозаменяемые. Разработчикам Python не нужно скрывать информацию, создавая полностью приватные и недоступные переменные (мы обсудим это в главе 2), так что мы будем использовать термин «инкапсуляция» в широком смысле.

Общедоступность интерфейса, однако, крайне важна. В него будет трудно внести изменения, ведь именно через него одни классы общаются с другими. Каждое

изменение интерфейса может лишить доступа к клиентским объектам, связанным с ним. Легко поменять то, что внутри, — улучшить производительность приложения или изменить права доступа по сети или локально: на клиентских объектах перемены отразятся несильно, взаимодействие останется неизменным при использовании ими внешнего интерфейса. Но если поменять интерфейс, будь то общедоступные имена атрибутов или порядок и типы аргументов у методов, то все клиентские классы также должны быть изменены. При проектировании внешнего интерфейса класса придерживайтесь однозначного правила: делайте его простым. Важнее простота в использовании интерфейса, чем заложенная в него при программировании замысловатость (это верно и для пользовательского интерфейса).

Здесь стоит отметить, что переменные Python с символом _ в начале имени — это переменные, не принадлежащие к внешнему интерфейсу.

Помните, программные объекты представляют реальные объекты, но не являются ими. Программные объекты — модели. Именно благодаря этому качеству они имеют право игнорировать все незначимые детали. Модель машины, которую один из авторов конструировал в детстве, была и вправду похожа на реальный «Форд Ти-бёрд» 1956 года, но не запускалась. Пока я был ребенком, это и не было так важно, сложные детали я не мог понять. Моя модель была **абстракцией** реального объекта.

Абстракция — еще один термин ООП, связанный с инкапсуляцией и сокрытием информации. Абстракция означает работу с частью информации соответствующей задачи. Так происходит разделение общедоступного интерфейса и внутренних механизмов. Водитель может повернуть руль, ускориться, затормозить. Для него не имеет значения, как работают двигатель, сцепление и тормоз. Зато это важно для механика, который решает свои задачи на другом уровне абстракции — настраивает двигатель или прокачивает тормоза. На рис. 1.6 изображены эти два уровня абстракции.

Теперь мы познакомились с основными терминами изучаемого подхода. Обобщим коротко то, что касается использования профессионального сленга: абстракция — процесс инкапсуляции информации на разных уровнях публичного интерфейса. Каждый частный элемент может быть скрыт. На UML-диаграмме используется знак – вместо +, чтобы показать, что этот элемент не является частью внешнего интерфейса.

Исходя из данных определений, сформулируем вывод: следует обеспечить, чтобы разрабатываемая модель была понятна другим объектам, ведь она должна с ними взаимодействовать. Для этого разработчику необходимо обращать пристальное внимание на детали.

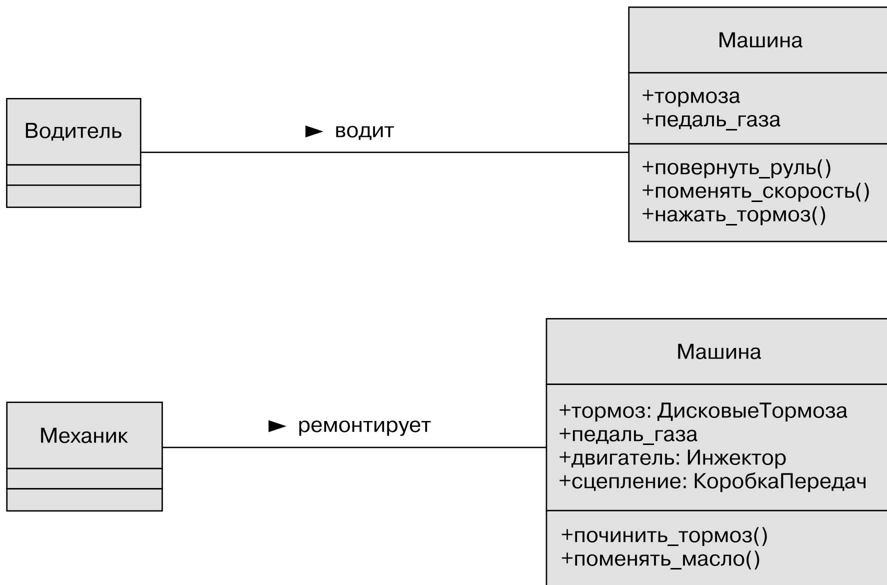

Рис. 1.6. Уровни абстракции автомобиля

Проверяйте, чтобы методы и свойства имели осмысленные названия. На стадии анализа системы объекты по обыкновению существительные, а методы — глаголы. Атрибуты можно выразить как прилагательными, так и существительными. Называйте свои классы, атрибуты и методы в соответствии с сутью задачи.

При разработке полезно представить себя на месте объекта. Вам нужно понимать, за что именно вы отвечаете головой. Не позволяйте другим вмешиваться в работу, избавьтесь от информационного шума и потока данных, если считаете их лишними. Не позволяйте никому навешивать на вас лишние задачи, если вы считаете, что это не ваша обязанность.

Композиция

К этому моменту вы уже научились проектировать структуру своего ПО как систему взаимодействующих объектов с определенным представлением на соответствующем уровне абстракции. Но пока еще неизвестно, как задавать уровни абстракции. И в этом, надо сказать, существуют разные подходы. Паттерны проектирования мы обсудим ниже, в главах 10–12. Пока достаточно отметить, что большинство паттернов проектирования основаны на двух базовых принципах ООП: **композиции** и **наследовании**. Композиция проще, начнем с нее.