Глава 11 | О непогрешимости авторитетов

Вообще говоря, стандартизация на уровне коллективов и сообществ разработчиков — это хорошо. Благодаря ей легче читать код, написанный другими людьми, ухватывать смысл идиом и избегать чрезмерно идиоматического кодирования (последнее не относится к сообществу программистов на Perl[1]). Но слепое следование стандартам ничем не лучше их полного отсутствия. Иногда стандарт препятствует полезному отклонению. Что бы вы ни делали, разрабатывая программу, убедитесь что знаете, зачем вы это делаете. В противном случае вы можете стать жертвой разъяренных обезьян.

Разъяренные обезьяны

Впервые я услышал эту историю на конференции, во время доклада «Разъяренные обезьяны и культ Даров небесных» Дэйва Томаса. Не знаю, правда это или нет (хотя посвятил некоторое время изучению вопроса), тем не менее, мою мысль этот рассказ иллюстрирует прекрасно.

В 1960-х (когда ученым разрешалось делать разные безумные вещи) исследователи поведения животных поставили эксперимент: поместили пять обезьян в комнату, где была стремянка, а с потолка свисала гроздь бананов. Обезьяны быстро сообразили, что до бананов можно добраться по стремянке, но как только обезьяны приближались к стремянке, ученые орошали всю комнату ледяной водой. Результат легко угадать: обезьяны пришли в ярость. Скоро ни одна обезьяна не рисковала подойти к стремянке.

---

[1] Шучу, честное слово. На самом деле я вас люблю, ребята! Не надо заваливать меня письмами.

Тогда ученые заменили одну обезьяну новой, еще не знакомой с ледяным душем. Первым делом она прямиком направилась к стремянке, но остальные обезьяны накинулись на нее с побоями. Она не знала, за что ее бьют, но быстро усвоила: к стремянке приближаться нельзя. Постепенно ученые заменили всех обезьян, так что образовалась группа животных, которых никогда не окатывали холодной водой. Тем не менее, они продолжали нападать на любую обезьяну, подошедшую близко к стремянке.

Мораль? В разработке ПО многие подходы применяются только потому, что «мы всегда так делали». Иными словами, из-за разъяренных обезьян.

Приведу пример из проекта, над которым мне когда-то довелось работать. Все знают, что имена методов в Java принято начинать с маленькой буквы и далее следовать ВерблюжьейНотации, то есть начинать каждое слово с заглавной буквы. При обычном кодировании это нормально, но имена тестовых методов – совсем другое дело. Для блочных тестов желательные длинные описательные имена, из которых легко понять, что именно тестируется. К сожалению, ДлинныеИменаВВерблюжьейНотацииТрудноЧитать. В том конкретном проекте я предложил разделять слова символом подчеркивания:

```
public void testUpdateCacheAndVerifyItemExists() {
}

public void test_Update_cache_and_verify_item_exists() {
}
```

Лично мне имена с символами подчеркивания казались более удобными для восприятия. Интересно было наблюдать за реакцией коллектива на мое предложение. Одним программистам идея сразу пришлась по душе, другие повели себя, как разъяренные обезьяны. Мы все-таки решили принять этот стиль (иногда техническому руководителю приходится быть великодушным диктатором), и выяснилось, что имена стали гораздо более понятными, особенно когда требовалось прочитать длинный список имен в окне прогона тестов в IDE (рис. 11.1).

«Мы всегда так делали» – недостаточное обоснование привычной программистской практики. Если вы *понимаете*, почему всегда делали именно так, и видите в этом смысл, то, разумеется, продолжайте. Но все же подвергайте сомнению допущения и проверяйте их правильность.

Цепные интерфейсы

Цепной интерфейс (*fluent interface*) – это один из модных нынче стилей, применяемый в предметно-ориентированных языках (DSL). Вы пытаетесь разбить длинную последовательность кода на предложения, мотивируя

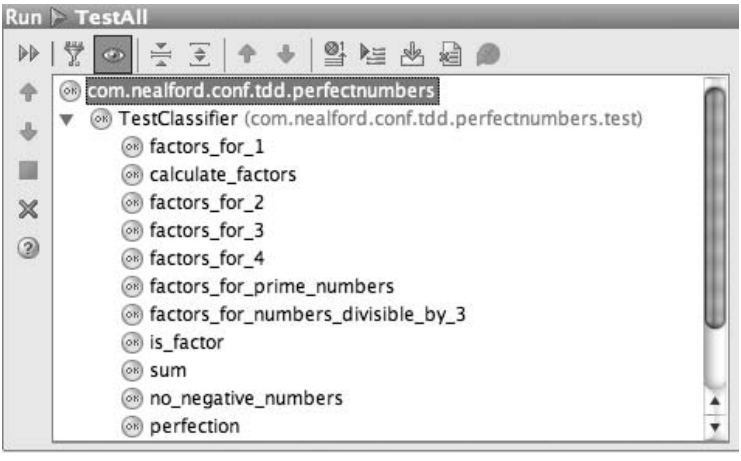

Рис. 11.1. *Имена тестов с символами подчеркивания удобнее для восприятия*

это тем, что в разговорных языках именно так оформляются законченные мысли. Такой код легче читать, поскольку, как и в английском языке, вы сразу видите, где кончается одна мысль и начинается следующая.

Приведу пример, взятый из одного моего проекта. Мы разрабатывали приложение, связанное с железнодорожными вагонами, причем у каждого вагона было маркетинговое описание. Для вагонов действуют многочисленные правила и инструкции, поэтому правильно написать тестовые сценарии оказалось нелегко. Мы постоянно спрашивали бизнес-аналитиков о нюансах определения того типа вагонов, который собирались протестировать. Вот упрощенная версия того, что мы им показывали:

```
Car car = new CarImpl();
MarketingDescription desc = new MarketingDescriptionImpl();
desc.setType("Box");
desc.setSubType("Insulated");
desc.setAttribute("length", "50.5");
desc.setAttribute("ladder", "yes");
desc.setAttribute("lining type", "cork");
car.setDescription(desc);
```

Для программиста на Java это выглядит вполне нормально, но бизнес-аналитики приходили в ярость. «Какого черта ты мне суешь этот код? Просто скажи, что ты хочешь!» Разумеется, при переводе не исключены ошибки. Чтобы как-то сгладить проблему, мы разработали цепной интерфейс для выражения той же информации, но в другом виде:

```
Car car = Car.describedAs()
              .box()
              .length(50.5)
              .type(Type.INSULATED)
              .includes(Equipment.LADDER)
              .lining(Lining.CORK);
```

Это бизнес-аналитикам понравилось куда больше. Заодно нам удалось избавиться от значительной части сомнительной избыточности, свойственной «нормальному» стилю программирования на Java. Реализация же была на удивление проста. Все методы установки свойств возвращают `this` вместо `void`, что позволяет связывать вызовы методов. Например, вот реализация класса `Car`:

```
public class Car {
  private MarketingDescription _desc;

  public Car() {
   _desc = new MarketingDescriptionImpl();
  }

  public static Car describedAs() {
    return new Car();
  }

  public Car box() {
   _desc.setType("box");
    return this;
  }

  public Car length(double length) {
   _desc.setLength(length);
    return this;
  }

  public Car type(Type type) {
   _desc.setType(type);
    return this;
  }

  public Car includes(Equipment equip) {
   _desc.setAttribute("equipment", equip.toString());
    return this;
  }

  public Car lining(Lining lining) {
   _desc.setLining(lining);
    return this;
  }
}
```

Это также может служить примером DSL-паттерна, известного как *построитель выражений* (*expression builder*). Класс `Car` скрывает тот факт, что конструирует внутри себя объект `MarketingDescription`. Построители выражений публикуют свой интерфейс к инкапсулированным выражениям, чтобы упростить реализацию цепного интерфейса. Чтобы можно было связывать вызовы методов, каждый метод установки свойства в классе `Car` должен возвращать `this`.

Почему я поместил этот пример в главу о непогрешимости авторитетов? Чтобы написать цепной интерфейс, как у класса `Car`, необходимо пожертвовать одной из священных коров Java: класс `Car` более не является `JavaBean`. Вроде бы мелочь, но значительная часть инфраструктуры Java опирается на следование этой спецификации. Однако, внимательно изучив спецификацию JavaBeans, вы обнаружите, что в ней есть ряд положений, негативно отражающихся на общем качестве кода.

Спецификация JavaBeans требует, чтобы у каждого объекта был конструктор по умолчанию (см. раздел «Конструкторы» главы 8), хотя где вы найдете корректный объект, не имеющий состояния? Спецификация также постулирует уродливый синтаксис свойств в Java, требуя называть методы чтения свойств `getXXX()`, а методы установки – `setXXX()`, причем последние обязаны иметь тип `void`. Я понимаю, для чего введены эти ограничения (например, наличие конструктора по умолчанию упрощает сериализацию), но никто в мире Java не задается вопросом, а так ли необходимо, чтобы каждый объект был `JavaBean`. По умолчанию принято слушаться разъяренных обезьян. Подвергайте авторитеты сомнению! Согласившись с тем, что объект обязан быть `JavaBean`, невозможно реализовать цепной интерфейс.

Нужно знать, что хочешь создать, понимать, для чего этого будет использоваться, и принимать решения осознанно. «Потому что все говорят, что так должно быть» редко оказывается правильным ответом.

Антиобъекты

Иногда авторитет, который следует подвергнуть сомнению, – это ваша собственная склонность решать задачу определенным способом. На конференции OOPSLA 2006 года была представлена замечательная работа «Collaborative Diffusion: Programming Anti-Objects»[1]. По мысли ее авторов, объекты и их иерархии действительно образуют великолепный механизм абстрагирования для решения большинства задач, но порой те же самые абстракции только усложняют проблему. Идея антиобъектов в том,

[1] *http://www.cs.colorado.edu/~ralex/papers/PDF/OOPSLA06antiobjects.pdf*

чтобы поменять передний и задний план задачи местами и решать более простую, но менее очевидную задачу. Что понимается под «передним и задним планом»? Рассмотрим на примере. (Внимание! Если вы все еще любите играть в PacMan, не читайте следующие абзацы – они навсегда отобьют у вас охоту к этой игре! За знания иногда приходится платить.)

Рассмотрим консольную игру PacMan. Она появилась в 1970 году, когда вычислительная мощность компьютеров была меньше, чем у современных дешевеньких мобильников. Тем не менее, требовалось решить сложную математическую задачу: как привидения должны преследовать игрока в лабиринте? Иными словами: каков кратчайший путь к движущейся цели в лабиринте? Это трудная задача, особенно когда памяти очень мало, а процессор слабый. Поэтому авторы PacMan не стали ее решать, а применили идею антиобъектов, встроив интеллект в сам лабиринт.

Лабиринт в PacMan работает как клеточный автомат (аналогично игре «Жизнь» Конвея). С каждой клеткой ассоциированы простые правила. На каждом шаге исполняются правила из одной клетки, начиная с левой верхней и продвигаясь к правой нижней. Каждая клетка помнит «запах игрока». Когда игрок находится в некоторой клетке, запах игрока в ней максимален. Если он только что покинул клетку, то запах уменьшается на 1. На протяжении нескольких поворотов запах уменьшается, а потом обращается в 0. Сами привидения абсолютно тупые: они просто принюхиваются и, уловив запах игрока, направляются к той клетке, где он максимален.

«Очевидное» решение задачи – встроить интеллект в привидения. Но гораздо проще встроить интеллект в лабиринт. В этом и состоит суть антиобъектов: поменять местами передний и задний план вычислений. Не поддавайтесь мысли о том, что «традиционная» модель всегда дает нужное решение. Иногда конкретную задачу проще решить совсем на другом языке программирования. (Обоснования подхода на основе антиобъектов приведены в главе 14.)