Глава 5

Класс

Идея классов возникла задолго до Платона. Его идеальные тела были классами, воплощения которых существовали в реальном мире. Сфера Платона была абсолютно идеальна, но нематериальна. Вокруг нас есть много сфер, но все они не идеальны в какой-то мере.

Объектно-ориентированное программирование взяло эту идею в изложении поздних западных философов. Программы поделили на классы, которые являются общим описанием целого множества похожих вещей, или объектов.

Классы важны для взаимодействия, поскольку они могут описывать многие специфические вещи. Они охватывают широкий диапазон шаблонов реализации, в то время как шаблоны проектирования обычно относятся к взаимоотношениям классов.

В этой главе появятся следующие шаблоны.

- Класс. Используется, чтобы сказать: “Эти данные и логика есть одно целое”.
- Простое имя суперкласса. Наименование корневого класса иерархии простым именем, производным от метафоры.
- Специальное имя субкласса. Наименование субклассов для обозначения сходств и различий с суперклассом.
- Абстрактный интерфейс. Разделяет интерфейс и имплементацию.
- Интерфейс. Определяет абстрактный, редко меняющийся интерфейс с помощью Java-интерфейса.
- Интерфейс версии. Безопасно расширяет интерфейс до субинтерфейса.
- Абстрактный класс. Описывает абстрактный интерфейс (который, возможно, будет меняться) с помощью абстрактного класса.
- Объект-значение. Объект, который ведет себя как математическое значение.
- Специализация. Разграничивает сходства и различия взаимосвязанных вычислений.
- Субкласс. Выражает одномерную вариацию в субклассе.
- Имплементор. Переопределяет метод для другого варианта вычислений.
- Вложенный класс. Заключает локально используемый код в приватный класс.

- Контекстно-зависимое поведение. Варьирует логику в зависимости от воплощения.
- Условный шаблон. Меняет логику в соответствии с определенными условиями.
- Делегирование. Логика изменяется путем делегирования одному из нескольких типов объектов.
- Сменный селектор. Изменения логики отражаются при выполнении метода.
- Анонимный вложенный класс. Меняет логику, переопределяя один или два метода при создании объекта.
- Библиотечный класс. Представляет функциональный пакет, не подходящий ни одному объекту, в качестве набора статических методов.

Класс

Данные более подвержены изменениям, чем логика. Классы работают именно благодаря этому. Каждый из них есть декларация: "Логика — одно целое и меняется медленнее, чем данные, которыми она оперирует. Данные тоже объединены, меняются с одной частотой и используются соответствующей логикой". Строгое разделение между изменчивыми данными и не меняющейся логикой не является абсолютным. Иногда логика слегка модифицируется в зависимости от данных; иногда изменения очень глубоки. Иногда данные не меняются в процессе вычислений. Изучение того, как объединять логику в классы и выражать вариации в логике, — это часть эффективного объектного программирования.

Организация иерархии классов — это способ сжатия кода путем объявления суперкласса и текстового включения его в субклассы. Как и любая технология сжатия, она делает код плохо читаемым. Приходится вникать в контекст суперкласса, чтобы понять субклассы.

Другой аспект эффективного программирования объектов — разумное использование наследования. Создание субкласса выражается фразой: "Я такой же, как и суперкласс, только другой". (Не странно ли то, что мы говорим о *переопределении* метода в *субклассе*? Насколько лучшими программистами мы можем стать, если будем внимательно выбирать метафоры?)

Классы — относительно дорогой элемент проектирования объектно-ориентированных программ. Класс должен делать что-то значительное. Уменьшение числа классов в системе — это улучшение, если только оставшиеся классы не раздуваются от этого.

Приведенные далее шаблоны поясняют, как взаимодействовать с помощью классов.

Простое имя суперкласса

Само нахождение правильных имен — один из самых приятных элементов программирования. Вы боретесь с идеей. Часто код получается сложным и выглядит не так, как должно. А иногда, напротив, кто-то видит программу, округляет глаза и говорит: "Да, точно! Это действительно `Scheduler`!" Правильное имя является результатом последовательных упрощений и улучшений.

Имена классов являются наиболее важными. Класс — это центральная концепция, "якорь" проектирования. Как только у класса возникает имя, появляются имена и у всех остальных операций. Редко случается обратное, разве что класс был изначально плохо назван.

При именовании возникает конфликт между краткостью и выразительностью. Например, в беседе можно выразить имя класса так: "Вы помните, что нужно повернуть `Figure` перед ее смещением?" Имена должны быть короткими и точными. Однако часто приходится использовать несколько слов, чтобы дать соответствующее имя.

Разрешить эту дилемму можно, выбирая подходящую характеру вычислений метафору. С этим мощным средством каждое слово приносит богатую сеть ассоциаций, взаимосвязей и смысловых линий. Например, в инфраструктуре HotDraw мое первое имя для объекта рисования было `DrawingObject`. Вард Куннингэм (Ward Cunningham) предложил типографскую метафору: рисование — это как печать страницы. Графические элементы на странице — фигуры, так что класс стал называться `Figure`. В контексте метафоры `Figure` — сравнительно более короткое, точное и богатое имя, чем `DrawingObject`.

Порой поиск хорошего имени занимает много времени. Законченный код может работать недели, месяцы или (как было в одном случае со мной) годы, пока не отыщется лучшее имя для класса. Иногда приходится порядочно потрудиться для поиска имени: достать словарь, сделать список наиболее подходящих имен, выйти на прогулку. В некоторых случаях нужно продвигаться вперед, доверяя времени, и ваше подсознание подскажет лучшее имя.

Общение — инструмент, постоянно помогающий мне в поиске имен. Разъяснение сути объекта другому человеку приводит меня к ряду богатых и запоминающихся образов, которые могут быть преобразованы в новые имена.

Для наименования важных классов лучше использовать одно слово.

Специальное имя субкласса

Имена субклассов имеют два назначения. Они должны рассказывать, на что *похож* класс и от чего *отличен*. Опять же необходимо соблюдение баланса между длиной и выразительностью. В отличие от корневых классов иерархии, имена субклассов используются не так часто, поэтому их можно делать длиннее в ущерб лаконичности.

Для формирования имени субкласса можно добавить приставку к названию суперкласса.

Здесь есть единственное исключение: если субклассы используются строго для реализации механизма разделения, имеют в основе свою собственную, отдельную концепцию и находятся во главе собственной иерархии, то они могут получать более короткие имена. Например, HotDraw содержит класс `Handle`, предоставляющий операции редактирования фигуры, когда она выбрана. Называя класс `Handle`, а не расширяя имя от `Figure`, мы подчеркиваем, что может быть целое семейство классов `Handle` — например, `StretchyHandle` или `TransparencyHandle`. Таким образом, `Handle` стоит во главе собственной иерархии классов и поэтому имеет простое название суперкласса, а не специальное имя субкласса.

Другое препятствие в наименовании субклассов — многоуровневые иерархии. Часто они только ждут своего появления, но когда это случается, классам требуются хорошие имена. Вместо того чтобы слепо модифицировать непосредственный суперкласс приставками, подумайте об этом с точки зрения читателя. Какие классы он должен счесть похожими? Используйте соответствующий суперкласс в качестве основы для имени.

Назначение имен классов — во взаимодействии с людьми. Для нужд компьютера классы могут быть просто пронумерованы. Имена классов слишком длинны и сложны для чтения и форматирования. Слишком короткие имена утомляют краткосрочную память читателя. Кластеры классов с не относящимися друг к другу именами трудно осмыслить и вспомнить. Используйте имена классов, чтобы поведать историю вашего кода.

Абстрактный интерфейс

Старая поговорка о разработке ПО гласит: программируют для интерфейса, а не ради затеи. Это другая сторона предположения о том, что проектные решения не должны зримо присутствовать там, где в этом нет нужды. Если бо́льшая часть кода знает только лишь, что я работаю с коллекцией, то мне легко будет сменить конкретный класс позже. Однако на каком-то этапе вам действительно понадобится подключить конкретный класс, чтобы выполнились вычисления.

Под интерфейсом я подразумеваю здесь набор операций без реализации. Это может быть представлено в Java как интерфейсом, так и суперклассом. К различным случаям подходят соответствующие шаблоны.

Каждый слой интерфейса имеет стоимость, так как является еще одной вещью, которую нужно выучить, понять, документировать, отладить, организовать, осмотреть и наименовать. Максимизация числа интерфейсов не снижает стоимости ПО. Платите за интерфейсы, только когда вам нужна присущая им гибкость. В то время как зачастую вы можете не знать наперед, когда появится нужда в интерфейсах, комбинируйте предположения о том, где гибкость не нужна, а где ее следует ввести.

Мы очень много жалуемся на негибкость ПО, но существует множество вещей, где мы не нуждаемся ни в какой системе гибкости. Со времени фундаментальных изменений, таких как количество битов в integer, до сегодняшних широкомасштабных начинаний в виде новых моделей бизнеса большинство ПО, как правило, не нуждается в гибкости.

Другой экономический фактор, говорящий в пользу интерфейсов, — это непредсказуемость рынка ПО. Наша индустрия кажется вовлеченной в идею о том, что если мы правильно спроектировали программу, то аппаратная часть систем не должна меняться. Я недавно прочитал список причин модификации ПО. В нем были программисты, плохо делающие свою работу по предъявленным требованиям, изменчивость спонсоров и т.п. Единственный фактор, отсутствовавший в списке, — это закономерные изменения. Список предполагает, что изменение — всегда ошибка. Почему бы не сделать предсказание погоды на все времена? Потому что она меняется непредсказуемо. В таком случае мы можем условиться раз и навсегда, что система должна быть гибкой, так как изменения требований и технологии непредсказуемы. Это не освобождает нас от обязанности наилучшим образом удовлетворять нужды сегодняшних потребителей, но устанавливает какие-то умственные ограничения на ценность "защищенного от будущего" ПО.

Сведение всех этих факторов вместе — необходимость гибкости, ее цена, непредсказуемость того, где она понадобится, — заставляют меня верить, что вводить ее надо только в случае явной нужды. При этом вам придется заплатить за изменения в существующем ПО. Если же вы не можете этого сделать лично, то стоимость возрастет пропорционально, как описывается ниже в главе, посвященной развертыванию инфраструктур.

Механизмы Java — интерфейсы и суперклассы — имеют неодинаковую себестоимость использования.

Интерфейс

Один из способов сказать: "Эту часть кода я хотел бы завершить, хотя детали пока что меня не волнуют", — объявить интерфейс в Java. Интерфейсы — важное нововведение, впервые появившееся на массовом рынке программных языков именно в Java. Они имеют хороший баланс, предлагая гибкость множественного наследования, не отягощенную сложностью и неопределенностью. Класс может быть объявлен как разделяющий множество интерфейсов. Также они раскрывают только операции, а не поля, что эффективно защищает пользователей интерфейса от изменений реализации.

Когда же изменение касается реализации, то сам по себе интерфейс остается прежним. Любая добавка или модификация в интерфейсе требует изменения всех воплощений. Если вы не можете менять воплощения, то широкое применение интерфейсов становится существенным тормозом в развитии проекта.

Одна особенность интерфейсов, ограничивающая их способность к взаимодействию, — это объявление всех методов публичными. Я часто хотел, чтобы область видимости операций интерфейса оставалась в рамках пакета. Создание чуть более, чем необходимо, публичных элементов проектирования не так важно для личного использования, но если интерфейс предполагается широко обнародовать, то лучше быть точным сначала, чем встраивать в код инерцию к будущим изменениям.

В зависимости от способа мышления есть два стиля наименования интерфейсов. Если это классы без имплементации, то их так и следует именовать: простое имя суперкласса, специальное имя субкласса. Единственная проблема в этом — использование хороших имен до того, как вы перейдете к определению собственно классов. Интерфейс `File` должен иметь имплементации, названные примерно как `ActualFile`, `ConcreteFile` или (внимание!) `FileImpl` (одновременно и суффикс, и сокращение). Вообще-то, существенно знать, с чем происходит взаимодействие — с абстрактным или конкретным объектом, — тогда как способ реализации объекта в виде интерфейса или суперкласса не столь важен. Данный способ именования вносит различия в названия интерфейсов и суперклассов, оставляя вам выбор в будущем на случай необходимости.

Иногда скрыть использование интерфейса не так важно для взаимодействия, как назвать конкретный класс. В таком случае к имени интерфейса добавляйте префикс I. Если интерфейс будет назван `IFile`, то класс можно назвать просто `File`.

Абстрактный класс

Другой способ выразить различие между абстрактным интерфейсом и конкретной реализацией в Java — это использовать суперклассы. Они абстрактны в том смысле, что имплементация всегда может быть заменена субклассом, независимо от применения ключевого слова `abstract`.

Решение об использовании абстрактного класса вместо интерфейса сводится к двум вопросам: изменения в интерфейсе и нужда в едином классе для одновременной поддержки множества интерфейсов. Абстрактные интерфейсы могут изменяться дважды — в имплементации и сами по себе как интерфейсы. Последнее — не лучшая черта Java. Любая модификация интерфейса требует изменения имплементаций. С повсеместно реализованным интерфейсом это запросто приводит к "параличу" существующих проектов, в дальнейшем развивающихся только в версиях интерфейсов.

Абстрактные классы не страдают от этого ограничения. Когда обозначена базовая реализация, в абстрактный класс могут быть добавлены новые операции без разрушения имплементаций.

Единственный недостаток этого метода состоит в том, что классы могут принадлежать только одному суперклассу. Если необходим другой вид тех же классов, то следует использовать интерфейсы.

Появление слова `abstract` в объявлении класса говорит читателю о том, что ему предстоит создать реализацию, прежде чем он сможет применить данный класс. Если только есть шанс сделать корневой класс полезным и завершенным самим по себе — делайте это. Однажды шагнув на путь абстракций, можно зайти слишком далеко и создать кучу бесполезного кода. Борьба за воплощение корневых классов приведет к устранению массивных абстракций.

Интерфейсы и классы не взаимоисключают друг друга. Можно создать интерфейс, сказав: "Вот путь доступа к этому виду функциональности", и суперкласс — как способ воплощения той же функциональности. В этом случае переменные должны быть объявлены как интерфейсы, чтобы разработчики в будущем могли менять имплементации как им заблагорассудится.

Интерфейс версий

Что вы делаете, когда возникает необходимость изменить интерфейс, хотя это невозможно? Обычно такая ситуация возникает при добавлении новых операций. Так как внесение новой операции поломает все существующие имплементации, это сложно реализовать. Однако можно объявить дополнительный интерфейс, расширяющий текущий. Пользователи, нуждающиеся в новой функциональности, могут использовать новый интерфейс, тогда как остальные просто не обратят внимания на его существование. Тогда везде, при обращении к новой операции, необходимо проверять тип объекта и приводить к новому типу.

Например, обсудим конструкцию

```
interface Command {
void run();
}
```

Допустим, что этот интерфейс используется тысячи раз и его изменение довольно дорого. Однако вам понадобилась операция отмены команд. Тогда получится следующая версия интерфейса.

```
interface ReversibleCommand extends Command {
void undo();
}
```

Существующее воплощение `Command` работает, как и прежде. Воплощения `ReversibleCommand` так же функциональны, как и `Command`. Для использования новой операции нужно привести тип.

```
...
Command recent= ...;
if (recent instanceof ReversibleCommand) {
ReversibleCommand downcasted= (ReversibleCommand) recent;
```

```
downcasted.undo();
]
...
```

Использование конструкции `instanceof` в целом снижает гибкость за счет привязывания кода к определенным классам. Однако в этом случае она может быть оправдана, так как применяется с целью развития интерфейса. Если у вас появляется несколько альтернативных интерфейсов, то клиентам требуется много работы, чтобы учесть все вариации. Это сигнализирует о том, что надо подумать над конструкцией в целом.

Альтернативные интерфейсы — неправильное решение неверно поставленных проблем. Интерфейсы не приспосабливаются к изменениям в своей структуре с той же легкостью, как это делают их имплементации, хотя и стремятся к модификации, как все конструкторские решения. Мы изучили все о проектировании имплементаций и поддержке. Альтернативные интерфейсы создают новый язык, похожий на Java, но с другими правилами. Написание своего языка — игра с более жесткими ограничениями, чем создание приложений. Однако, если вы столкнетесь с ситуацией, когда необходимо будет расширить интерфейс, лучше знать, как это сделать.

Объект-значение

Объекты с меняющимся состоянием — не единственный способ размышлять о вычислениях. Математика развивалась более тысячелетия как способ осмысления ситуаций, сводящихся к абстрактному миру абсолютной правды и определенности, где состояния могут быть выражены через вечные истины.

Наш текущий язык программирования — смесь обоих стилей. Так называемые примитивы в Java (в большинстве случаев) принадлежат к миру математики. Когда я прибавляю единицу к числу, я произвожу математическое сложение (кроме того случая, когда кто-то решает, что компьютер может считать только до 2^{32} или до 2^{64}, и все приходится начинать заново). Я не меняю значения переменной, когда прибавляю 1, — я создаю новое значение. Не существует способа изменить 0, как это возможно с большинством объектов.

Этот функциональный стиль программирования никогда не меняет состояний — он создает новые значения. Когда появляется (может быть, на секунду) статичная ситуация, которую нужно сформулировать или задать ей вопросы, то функциональный стиль отлично подходит. Если ситуация нестабильна во времени, то и состояние меняется соответственно. Некоторые вещи могут быть описаны обоими способами. Как решить, какой из них больше подходит конкретному случаю?

Например, рисование картины можно представить как изменение состояния некоего графического средства, такого, как точечный рисунок. Одновременно ту же картину можно описать статическим методом (рис. 5.1).

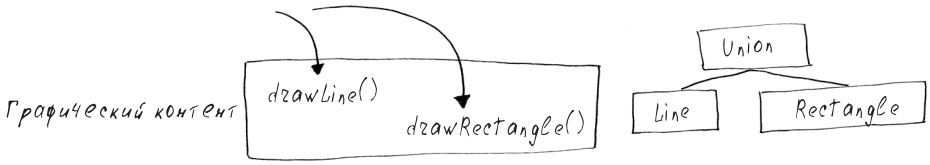

Рис. 5.1. Графика, представленная процедурами и объектами

Любая репрезентация применима в какой-то степени на основании собственного опыта, но также имеет значение и сложность картины, и скорость ее изменения.

Процедурные интерфейсы более универсальны, чем функциональные. Их единственный недостаток — необходимость выполнять вызовы процедур в определенной последовательности, что становится частично сутью интерфейса. Изменение таких программ часто рискованно и затруднено, так как сравнительно небольшие модификации имеют нежелательные последствия, если меняют внутреннюю сущность последовательности.

Математическое представление более красиво, оно заставляет последовательность что-то значить саму по себе и позволяет создавать мир, в котором будут только абсолютные, вневременные состояния. Вводите математические микромиры где только возможно. Управление может осуществляться в объекте с переменным состоянием.

Например, реализуем систему расчетов, используя базовые транзакции без изменения математических значений (рис. 5.2).

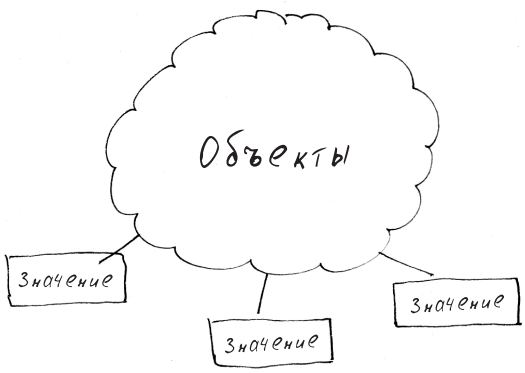

Рис. 5.2. Объекты с переменным состоянием, ссылающиеся на неизменные объекты по краям

```
class Transaction {
int value;
Transaction(int value, Account credit, Account debit) {
this.value= value;
credit.addCredit(this);
debit.addDebit(this);
```

```
}
int getValue() {
return value;
}
}
```

С момента создания объекта `Transaction` не существует способа изменить его значение. Более того, конструктор делает предположение, что все транзакции относятся к двум счетам. Во время прочтения этого кода не возникнет беспокойства о "зависших", неучтенных транзакциях, или о транзакциях, изменивших значение после объявления.

Для реализации объектов в стиле "значение" (т.е. объектов, ведущих себя как integer, в отличие от объектов-контейнеров, содержащих текущее состояние) проведите для начала границу между миром состояний и миром значений. В приведенном примере `Transaction` является примером значения, тогда как `Accounts` — контейнер состояния. В объектах-значениях все состояния устанавливаются в конструкторах, не оставляя возможности изменить их после. Операции на таких объектах всегда возвращают новые объекты, которые должны быть сохранены вызывающей стороной.

```
bounds.translateBy(10, 20); // mutable Rectangle
bounds=bounds.translateBy(10, 20); // value-style Rectangle
```

Самым существенным возражением против этого подхода всегда была производительность. Необходимость создавать все эти промежуточные объекты может создать нагрузку на память управляющей системы. В программировании в целом этот факт обычно не учитывается, поскольку большинство частей программы не являются узкими местами. Другая трудность использования значений — это непонимание стиля и сложность проведения четких границ между системами с изменяющимся состоянием и статичными системами. Объекты-значения часто плохи для обоих случаев, тогда как интерфейсы более совершенны, хотя бывает сложно делать предположения об их состоянии.

Забираясь в такие дебри, я лишь хочу сказать, что гораздо больше должно быть написано об этих трех стилях программирования — объектах, функциях и процедурах — и об их эффективном совместном употреблении. Но, учитывая назначение этой книги, я закругляюсь, еще раз напомнив, что программы лучше всего выражать как комбинацию объектов-значений и объектов-состояний.

Специализация

Выражение в программе сходств и различий вычислений делает ваши программы более легкими для чтения, использования и модификации. Практически не существует уникальных программ. Многие из них выражают одни идеи, как делают зачастую и разные части одной программы. Передача пользователю этих сходств и различий

помогает понять код, найти соответствие между замыслами пользователя и существующими вариациями, или, если соответствий нет, приспособить написанный код к своим нуждам или написать что-то совершенно новое.

Простейшие вариации — отличия состояний. Строка “abc” отличается от “def”. Алгоритмы, оперирующие на этих строках, идентичны. Например, длина строк вычисляется одним и тем же способом.

Наиболее сложные вариации — различия в логике. Процедура символического интегрирования не имеет ничего общего с процедурой вывода математического текста, хотя они могут принимать одни и те же входные данные.

Между двумя крайностями — идентичной логикой с различающимися данными и различной логикой с одинаковыми данными — лежит пространство обычного программирования. Данные могут быть в основном похожими, хотя и немного различными. То же самое относится и к логике. (Я думаю, процедуры символического интегрирования и вывода математического текста имеют все же небольшую общую часть.) Размыта даже линия между логикой и данными. Булев флаг — это данные, но он может управлять исполнением. Вспомогательный объект может храниться в поле, но влиять на вычисления.

Нижеприведенные шаблоны являют собой техники, указывающие преимущественно на логические сходства и различия. Вариации данных не кажутся сложными или запутанными. Эффективное выражение сходств и различий в логике открывает новые возможности для дальнейшего расширения кода.

Субкласс

Объявляя субкласс, мы говорим: “Эти объекты совсем как те, за исключением...” Если у вас есть правильный суперкласс, то создание субкласса — это мощное средство программирования. Переопределяя нужный метод, вы можете создавать вариации всего в нескольких линиях кода.

Когда объекты только стали популярными, субклассы казались волшебной пилюлей. Вначале они использовались для классификации — `Train` наследовался от `Vehicle` независимо от реального разделения имплементации. Со временем некоторые люди заметили, что наследование позволяет разделять имплементацию и может быть использовано, чтобы вынести за скобки ее общие части. Однако вскоре стали заметны ограничения наследования. Во-первых, этой картой можно сыграть только раз. Если обнаруживается, что какой-то набор вариаций недостаточно хорошо выражается субклассами, то приходится поработать над распутыванием кода перед его реструктурированием. Во-вторых, необходимо изучить суперкласс до того, как вы сможете понять субкласс. Чем более сложен суперкласс, тем больше неудобств это доставляет. В-третьих, изменения суперкласса рискованны, так как его использование в субклассах может быть незаметным. Наконец, все эти проблемы усугубляются в глубоких иерархиях наследования.

Частично разрушительный эффект наследования проявляется в параллельных иерархиях, когда каждому субклассу в *этой* иерархии соответствует класс в *той* иерархии. Это одна из форм дублирования, подразумевающая взаимное копирование. Для успешного внедрения новой вариации приходится менять обе иерархии. Хотя не всегда есть способ немедленно устранить такую ситуацию, попытки сделать это улучшают конструкцию.

Рис. 5.3. Параллельные иерархии

Примером может служить страховая система (рис. 5.3). В этой картине что-то определенно не так, потому что `InsuranceContract` никак не касается `PensionContract`, как бы ни было привлекательно сместить поле результата вниз в субклассы. Потребовался год, чтобы прийти к решению (так и не реализованному). Оно заключалось в смещении вариации таким образом, чтобы `Contract` работал одинаково и для страховки, и для пенсии. Это потребовало создания нового объекта для представления ожидаемых движений наличности (рис. 5.4).

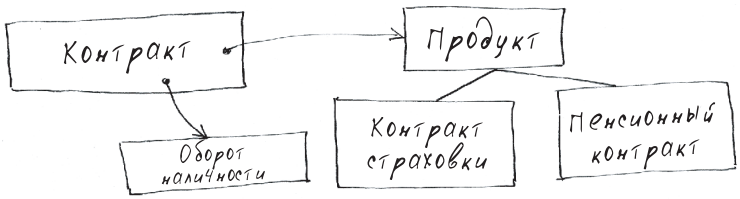

Рис. 5.4. Иерархия без дубликатов

Не упуская из виду все эти предосторожности, субклассы можно сделать мощным средством для выражения тематических и вариативных вычислений. Правильный субкласс помогает описать в точности то, что надо, в одном-двух методах. Один из аспектов использования субклассов — тщательное планирование логики методов суперкласса, делающих одну работу. Впоследствии должна быть возможность переопределить хотя бы один метод. Если методы суперкласса слишком велики, можно скопировать код и отредактировать его (рис. 5.5).

Скопированный код представляет нежелательное, но подразумеваемое дублирование между двумя классами. Невозможно безопасно поменять код в суперклассе без проверки и потенциального изменения всех скопированных мест в субклассе.

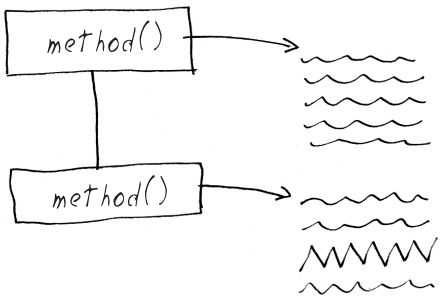

Рис. 5.5. Код, скопированный и модифицированный в субклассе

Своей целью в проектировании я ставлю способность переключаться между стратегиями по желанию, в зависимости от нужд кода, если он существует к тому моменту. Визуализируйте код с помощью условий, субклассов и делегирования. Разве вам не кажется, что есть преимущества и в другой стратегии, отличной от используемой вами сейчас? Сделайте несколько шагов в этом направлении и посмотрите, не улучшился ли код.

Последнее ограничение субклассов — невозможность передать меняющуюся логику. Вариация должна быть известна вам до создания объекта и не может быть изменена потом. Чтобы выразить такую ситуацию, используйте условия и делегирование.

Имплементор

Полиморфное сообщение — это фундаментальный способ выразить выбор в программе, построенной из объектов. Чтобы сообщение сделало свою работу выбора, должно быть более одного потенциального объекта-получателя.

Имплементация одного протокола несколько раз с помощью Java-интерфейса, или объявления `implements`, или субкласса, выраженного через `extends`, — это способ сказать: "Если произошло что-то, соответствующее замыслу данного кода, то с его точки зрения детали происшедшего несущественны".

Сила полиморфных сообщений в том, что они открывают путь вариациям. Если часть программы передает несколько байтов в другую систему, то внедрение абстракции типа `Socket` заставляет имплементацию сокета варьировать поведение, не влияя на вызывающий код. В сравнении с процедурной реализацией того же замысла, с ее однозначной и закрытой условной логикой, версия объект/сообщение более ясна, она отделяет выражение замысла (записать несколько байтов) от имплементации (вызов TCP/IP стека с определенными параметрами). В то же время выражение вычислений через объекты и сообщения открывает систему к таким будущим вариациям, о каких даже не мечтали первоначально программисты. Эта неожиданная комбинация ясности выражения и гибкости и делает объектные языки доминирующей парадигмой программирования.

Этот превосходный ресурс легко истощить, делая процедурные программы на Java. Шаблоны созданы для того, чтобы помочь вам выражать одновременно ясную и способную к расширению логику.

Вложенный класс

Иногда требуется упаковать часть вычислений, но не хочется делать этого ценой создания нового класса с отдельным файлом. Объявление маленьких, приватных (вложенных) классов дает дешевый способ использовать многие преимущества отдельного класса.

Иногда вложенный класс расширяет только `Object`. Некоторые вложенные классы расширяют другие суперклассы, что целесообразно для выражения усовершенствований, полезных лишь локально.

Одна из отличительных черт вложенного класса заключается в том, что при создании воплощения ему негласно передается копия создающего объекта. Это удобно, если по какой-либо причине нужно получить доступ к внешнему классу без явной регламентации взаимоотношений двух объектов.

```
public class InnerClassExample {
  private String field;

  public class Inner {
    public String example() {
    return field; // Используется переменная field
                  //  экземпляра внешнего класса
  }

@Test public void passes() {
  field= "abc";
  Inner bar= new Inner();
  assertEquals("abc", bar.example());
}
}
```

Таким образом, конструктор в приведенном примере все-таки принимает аргумент, хотя в объявлении его и нет. Это становится проблемой при создании воплощений вложенных классов по образу внешнего.

```
public class InnerClassExample {
  public class Inner {
   public Inner() {
   }
  }

 @Test(expected=NoSuchMethodException.class)
  public void innerHasNoNoArgConstructor() throws Exception {
```

```
  Inner.class.getConstructor(new Class[0]);
}
}
```

Чтобы получить вложенный класс, полностью независимый от обрамляющего воплощения, объявляйте его как `static`.

Контекстно-зависимое поведение

Теоретически все воплощения класса оперируют одной логикой. Устранение этого ограничения дает нам новые стили выражения. Однако все они недешево стоят. Когда логика кода полностью определяется классом, читатели могут посмотреть на класс и понять, что произойдет. Как только появляются воплощения с разным поведением, становится необходимо изучать работающие примеры или анализировать поток данных, чтобы понять, как поведет себя конкретный объект.

Следующий шаг обходится еще дороже. Речь идет об изменении логики по ходу работы программы. Для облегчения понимания старайтесь устанавливать контекстно-зависимое поведение сразу же после создания объекта и после уже не менять его.

Условия

Простейшими формами контекстно-зависимого поведения являются операторы `if/then` и `switch`. Используя условия, различные объекты выполняют разные задачи, основываясь на входных данных. Условия имеют то преимущество, что вся логика остается в одном классе. Читателям не приходится искать возможные пути исполнения программы. Однако есть и недостатки этого метода. Один из них — невозможность модификации поведения без изменений кода проблемного объекта.

Каждый путь выполнения программы может быть правильным с какой-то вероятностью. Таким образом, чем больше независимых путей, тем более ошибочна может быть программа в целом. Конечно, ошибки не полностью независимы, но все же в достаточной степени, чтобы программа с большим количеством путей имела больше дефектов. Умножение условий уменьшает надежность.

Эта проблема усложняется, когда условия дублируются. Обсудим простой графический редактор. Фигуры должны иметь метод `display()`.

```
public void display() {
  switch (getType()) {
  case RECTANGLE :
    //...
    break;
  case OVAL :
    //...
    break;
```

```
  case TEXT :
    //...
    break;
  default :
    break;
  }
}
```

Также нужен метод, определяющий принадлежность точки фигуре.

```
public boolean contains(Point p) {
  switch (getType()) {
    case RECTANGLE :
      //...
      break;
    case OVAL :
      //...
      break;
    case TEXT :
      //...
      break;
    default :
      break;
  }
}
```

Теперь предположим, что в программе добавился еще один вид фигур. Во-первых, вам нужно ввести новый пункт во всех блоках `switch`. Во-вторых, чтобы произвести это изменение, придется затронуть класс `Figure`, рискуя существующей функциональностью. Наконец, каждый, кто захочет добавлять новые фигуры, должен координировать изменения в единственном классе.

Все эти проблемы могут быть устранены путем преобразования условной логики в сообщения, используя субклассы или делегирование (какую технику лучше применить, зависит от кода). Дублированная условная логика, или же логика, в которой обработка сильно отличается в разных ветвях условий, лучше выражается сообщениями, чем однозначным указанием. То же относится к часто меняющейся условной логике — использование сообщений упрощает изменение одной ветви, сводя к минимуму влияние на другие (рис. 5.6).

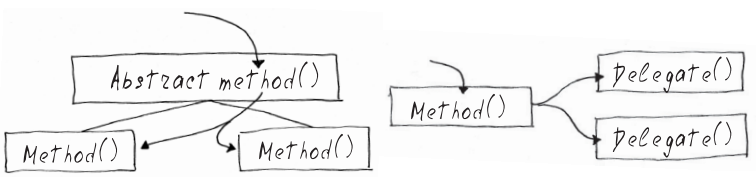

Рис. 5.6. Условная логика, представленная субклассами и делегированием

Таким образом, преимущества условий — простота и локальность — обращаются в недостатки при слишком широком использовании.

Делегирование

Другой способ исполнять различную логику в различных воплощениях — передача работы нескольким возможным видам объектов. Общая логика содержится в направляющем классе, вариации делегируются.

Пример использования делегирования для перехвата вариации — обработка пользовательского ввода в графическом редакторе. Иногда щелчок на кнопке означает создание прямоугольника, иногда — перемещение фигуры и т.п.

Один из способов сделать это заключается в применении условной логики.

```
public void mouseDown() {
  switch (getTool()) {
    case SELECTING :
      //...
      break;
    case CREATING_RECTANGLE :
      //...
      break;
    case EDITING_TEXT :
      //...
      break;
    default :
      break;
  }
}
```

Этот подход имеет все недостатки, обсуждаемые ранее: добавление нового инструмента требует модификации кода, а дублирование условий (в `mouseUp()`, `mouseMove()` и т.д.) усложняет эту задачу.

Субклассы также не являются непосредственным ответом, так как редактор должен менять инструменты во время работы. Делегирование предоставляет необходимую возможность.

```
public void mouseDown() {
  getTool().mouseDown();
}
```

Код, который существовал в пунктах блока `switch`, был перемещен в различные инструменты. Теперь новые инструменты могут быть представлены без модификации кода редактора или существующих инструментов, хотя чтение кода требует больше перемещений, так как логика нажатия кнопки распределена между несколькими классами. Понимание того, как ведет себя редактор в данном случае, требует понимания используемого инструмента.

Делегаты могут храниться в полях (“сменный объект”), но могут и вычисляться на лету. JUnit 4 динамически вычисляет объекты, использующиеся для проведения теста над данным классом. В зависимости от стиля теста (старый или новый) создаются различные делегаты. Это смесь условной логики (для создания делегатов) и собственно делегирования.

Тот же подход может использоваться для разделения кода в качестве контекстно-зависимого поведения. Объект, обращающийся к `Stream`, может быть вовлечен в контекстно-зависимое поведение, если тип `Stream` меняется во время исполнения, или может разделять имплементацию `Stream` со всеми остальными пользователями.

Обычный трюк — передача делегирующего объекта в качестве параметра делегируемому.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown(this);
}
RectangleTool
public void mouseDown(GraphicEditor editor) {
  editor.add(new RectangleFigure());
}
```

Если делегат хочет послать сообщение самому себе, то это допускает двоякое толкование. Иногда сообщение должно быть послано делегирующему объекту, иногда — делегируемому. В приведенном примере `RectangleTool` добавляет фигуру, но к делегирующему `GraphicsEditor`, а не к самому себе. `GraphicsEditor` может быть послан как параметр делегируемому элементу `mouseDown()`, но в этом случае проще сохранить постоянную ссылку в инструменте. Передача `GraphicsEditor` параметром делает возможным использование одного инструмента в различных редакторах, но, если это не критично, проще использовать код по обратной ссылке.

```
GraphicEditor
public void mouseDown() {
  tool.mouseDown();
}
RectangleTool
private GraphicEditor editor;
  public RectangleTool(GraphicEditor editor) {
  this.editor= editor;
}
public void mouseDown() {
  editor.add(new RectangleFigure());
}
```

Сменный селектор

Давайте предположим, что нам нужно контекстно-зависимое поведение, но только в одном-двух методах, и включение всех вариаций кода в один класс не требуется. В этом случае сохраняйте имя необходимого метода в поле и вызывайте его по мере надобности.

Изначально каждый тест из JUnit сохраняется в своем собственном классе (рис. 5.7). Каждый субкласс имеет только один метод. Такой подход смотрится концептуально тяжелым для представления единственного класса.

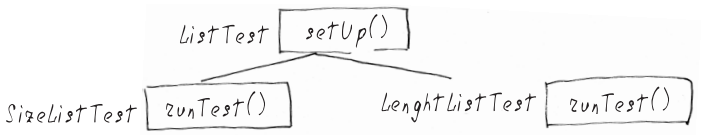

Рис. 5.7. Небольшие субклассы для представления одного теста

Имплементируя базовый метод `runTest()`, `ListTests` запускает разные методы тестов с различными именами. Предполагается, что имя теста совпадает с именем вызываемого метода во время работы теста. Ниже приведена простая версия кода, воплощающего сменный селектор исполняемого теста.

```
String name;
public void runTest() throws Exception {
   Class[] noArguments= new Class[0];
   Method method= getClass().getMethod(name, noArguments);
   method.invoke(this, new Object[0]);
}
```

Упрощенная иерархия использует единственный класс (рис. 5.8). Как происходит со всеми техниками сжатия кода, этот модифицированный код легко читается, только если вы понимаете “фишку”.

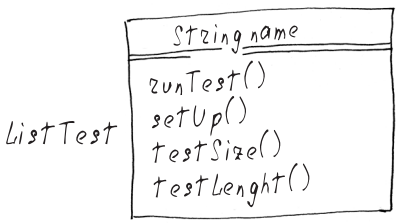

Рис. 5.8. Сменный селектор помогает упаковать тесты в один класс

Когда сменные селекторы появились в обращении, их стали использовать слишком часто. Вы будете смотреть на какой-то код, решив, что он не может быть испол-

нен, и получите останов системы, так как этот фрагмент был где-то вызван сменным селектором. Цена использования этой технологии может быть огромной, но ограниченное использование для решения сложных проблем оправдывает эту цену.

Анонимный вложенный класс

Java предлагает еще одну альтернативу контекстно-специфическому поведению — анонимные вложенные классы. Идея состоит в том, чтобы создать используемый только в одном месте класс, который может переопределять один или более методов строго для локальных целей. По этой причине ссылки на класс могут неявными, в отличие от ссылок по имени.

Эффективное использование анонимных вложенных классов предполагает применение сверхупрощенного API — как имплементация `Runnable` в единственном методе `run()` — или внедрение суперкласса, который обеспечит всю необходимую имплементацию. Код анонимного вложенного класса разрушает репрезентацию обрамляющего кода, поэтому он должен быть коротким, чтобы не отвлекать читателя.

Анонимные вложенные классы имеют ограничение — код воплощения должен быть известен заранее (в отличие от делегатов, которые могут добавляться позже) и не может изменяться после его создания. Такие классы трудно протестировать напрямую, поэтому они не должны вмещать сложную логику. Так как они безымянны, вы не имеете возможности выразить замысел с помощью хорошо подобранного имени.

Библиотечный класс

Куда вы поместите функциональность, не подходящую ни одному объекту? Можно создать статические методы в классе, не содержащем ничего, кроме этих методов. Никто никогда и не будет намереваться создавать воплощения этого класса — он послужит лишь контейнером для библиотечных статических функций.

Хотя такие решения довольно распространены, они плохо масштабируемы. Приходится расплачиваться за помещение всей логики в статические методы наибольшим преимуществом объектного программирования: упрощение кода путем разделения данных приватного пространства имен. Старайтесь превратить библиотечные классы в объекты, где это только возможно.

Иногда это легко, если очевиден собственник метода. Например, библиотечный класс `Collections` имеет метод `sort(List)`. Специфика параметра подсказывает нам, что этот метод, вероятно, лучше приписать классу `List`.

Постепенный путь преобразования библиотечного класса в объект — конвертирование статических методов в методы воплощения. Для начала оставьте тот же интерфейс, используя статические методы для передачи управления воплощению, как делается, например, в классе

```
public static void method(...params...) {
...some logic...
}
```

и преобразуйте в

```
public static void method(...params...) {
  new Library().instanceMethod(...params...);
}
private void instanceMethod(...params...) {
...some logic...
}
```

Теперь, если несколько методов имеют одинаковый список параметров (и если их принадлежность одному классу явно выражена), преобразуйте параметры методов в параметры конструктора.

```
public static void method(...params...) {
  new Library(...params...).instanceMethod();
}
private void instanceMethod() {
...some logic...
}
```

Потом измените интерфейс, переместив создание воплощения в клиентскую часть и уничтожив статические методы.

```
public void instanceMethod(...params...) {
...some logic...
}
```

Этот опыт дает представление о том, как переименовывать классы и методы, чтобы код был хорошо читаемым.

Выводы

Класс хранит вместе связанные между собой состояния. Следующая глава представляет шаблоны, передающие информацию о состоянии.