Глава 8

Как ввести новое свойство

Этот вопрос относится к самым абстрактным в данной книге, поскольку его решение очень зависит от конкретной предметной области. И по этой причине я поначалу не хотел включать его рассмотрение в данную книгу. Тем не менее, независимо от применяемого подхода к проектированию или конкретных ограничений, накладываемых на него, существует ряд способов, позволяющих упростить задачу ввода нового свойства в программу.

Итак, начнем обсуждение данного вопроса с контекста. Одна из самых больших трудностей работы с унаследованным кодом заключается в отсутствии тестов вокруг большей части такого кода. Более того, разместить их по местам не так-то просто. В связи с этим у многих разработчиков возникает искушение прибегнуть к способам, представленным в главе 6. Эти способы (почкования и охвата) можно использовать для ввода кода без тестов, но такой подход связан с целым рядом опасностей: как явных, так и неявных. Прежде всего при почковании и охвате мы не видоизменяем код существенно, и поэтому он не становится от этого лучше хотя бы на время. Кроме того, существует опасность дублирования кода. Если добавляемый код дублирует код, существующий на непроверенных участках, то он может просто залежаться и испортиться. Более того, мы можем даже не осознавать того, что дублируем код до тех пор, пока не зайдем слишком далеко, внося в него изменения. И еще одна опасность связана с боязнью и отказом от изменения кода. Боязнь возникает в связи с тем, что мы просто не в состоянии изменить конкретный фрагмент кода и упростить его для удобства дальнейшей работы с ним, а отказ вызван тем, что целые фрагменты кода не становятся лучше в результате внесенных изменений. Такая боязнь мешает принятию разумного решения. Об этом напоминают жалкие остатки почкования и охвата в коде.

Как известно, трудностям лучше противостоять смело, чем уклоняться от них. Если код можно подвергнуть тестированию, значит, стоит воспользоваться способами, представленными в этой главе, для удачного продвижения вперед в работе над кодом. О способах размещения тестов по местам речь пойдет в главе 13, а о преодолении трудностей, связанных с зависимостями, — в главах 9 и 10.

Разместив тесты по местам, мы оказываемся в лучшем положении для ввода новых свойств. Ведь мы опираемся на прочное основание.

Разработка посредством тестирования

Самым эффективным способом из тех, что я знаю для ввода новых свойств в программное обеспечение, является *разработка посредством тестирования* (TDD). Суть этого способа такова: мы придумываем сначала метод, который поможет нам решить часть стоящей перед нами задачи, а затем пишем для него контрольный пример непрохождения теста. Сам метод еще не существует, но если мы сумеем написать для него тест, то будем лучше понимать, каким именно должен быть код для этого метода.

Разработка посредством тестирования выполняется в такой последовательности.

1. Написание контрольного примера непрохождения теста.
2. Подготовка к компиляции.

3. Подготовка к прохождению.
4. Исключение дублирования кода.
5. Повтор.

Рассмотрим следующий пример. Допустим, что мы работаем над финансовым приложением и нам требуется класс, в котором нужно применить эффективный математический аппарат для проверки возможности продажи определенных товаров. Для этой цели нам потребуется класс Java, в котором выполняются расчеты так называемого первого статистического момента относительно определенной точки. У нас еще нет метода, который должен это делать, но мы знаем, что можем написать контрольный пример для данного метода. Если нам известен порядок расчетов, то мы знаем, что для данных в коде теста ответ должен быть равен `-0,5`.

Написание контрольного примера непрохождения теста

Ниже приведен тест для проверки нужных нам функций.

```
public void testFirstMoment() {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);

    assertEquals(-0.5, calculator.firstMomentAbout(2.0), TOLERANCE);
}
```

Подготовка к компиляции

Итак, мы написали тест, но он не компилируется, поскольку у нас нет метода `firstMomentAbout` (первый относительный момент) в классе `InstrumentCalculator` (калькулятор инструментов). Но мы можем ввести его как пустой метод. А поскольку нам требуется, чтобы тест не прошел, то мы должны возвратить значение `NaN` двойной длины (типа `double`), которое явно не равно предполагаемому значению `-0,5`.

```
public class InstrumentCalculator
{
    double firstMomentAbout(double point) {
        return Double.NaN;
    }
    ...
}
```

Подготовка к прохождению

Разместив тест на месте, мы готовы написать код, чтобы подготовить тест к прохождению.

```
public double firstMomentAbout(double point) {
    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
```

```
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Это слишком большое количество кода, который требуется написать для прохождения теста при разработке посредством тестирования. Как правило, он пишется более мелкими фрагментами. Но если вы уверены в применяемом алгоритме, то можете писать код и такими крупными фрагментами.

Исключение дублирования кода

Имеется ли в данном коде дублирование? Нет, не имеется. Следовательно, вы можем переходить к следующему контрольному примеру.

Написание контрольного примера непрохождения теста

Благодаря написанному только что коду тест проходит, но он годится не для всех случаев. Так, если в операторе `return` случайно произойдет деление на нуль, то что нам делать в этом случае? Что нужно возвратить, если нет элементов? В таком случае нам нужно сгенерировать исключение. Результаты не будут иметь для нас особого значения до тех пор, пока в нашем списке элементов не появятся данные.

Следующий тест имеет особое назначение. Он не проходит, если исключение в связи с неверным основанием (`InvalidBasisException`) не сгенерировано, и проходит, если не генерируется ни это, ни любое другое исключение. Если выполнить этот тест, то он не пройдет, поскольку при делении на нуль в методе `firstMomentAbout` генерируется исключение в связи с арифметической операцией (`ArithmeticException`).

```
public void testFirstMoment() {
    try {
        new InstrumentCalculator().firstMomentAbout(0.0);
        fail("ожидалось исключение InvalidBasisException");
    }
    catch (InvalidBasisException e) {
    }
}
```

Подготовка к компиляции

Для этой цели нам придется изменить объявление метода `firstMomentAbout`, чтобы в нем генерировалось исключение `InvalidBasisException`.

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
```

```
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Но этот код все равно не компилируется. Ошибки, сообщаемые компилятором, указывают на то, что нам нужно генерировать исключение, если оно перечислено в объявлении. Поэтому мы должны написать код, определяющий условия для генерирования исключения.

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {

    if (element.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Подготовка к прохождению

Теперь все наши тесты проходят.

Исключение дублирования кода

В данном случае дублирование кода отсутствует.

Написание контрольного примера непрохождения теста

Следующий фрагмент кода, который нам требуется написать, относится к методу, рассчитывающему второй статистический момент относительно определенной точки. На самом деле код для расчета этого момента является лишь разновидностью кода для расчета первого статистического момента. Ниже приведен тест, который позволяет нам перейти к написанию такого кода. В данном случае ожидается возврат значения `0,5`, а не `-0,5`, и поэтому нам нужно написать новый тест для метода `secondMomentAbout` (второй относительный момент), который еще не существует.

```
public void testSecondMoment() throws Exception {
    InstrumentCalculator calculator = new InstrumentCalculator();
    calculator.addElement(1.0);
    calculator.addElement(2.0);

    assertEquals(0.5, calculator.secondMomentAbout(2.0), TOLERANCE);
}
```

Подготовка к компиляции

Для подготовки теста к компиляции придется добавить объявление метода `secondMomentAbout`. Это можно было бы сделать таким же образом, как для и метода `firstMomentAbout`, но ведь код для расчета второго статистического момента лишь незначительно отличается от кода для расчета первого статистического момента.

Следующая строка кода в методе `firstMomentAbout`:

```
numerator += element - point;
```

должна быть заменена такой строкой в методе `secondMomentAbout`:

```
numerator += Math.pow(element – point, 2.0);
```

В целом, для такого рода расчетов имеется общий шаблон. Расчеты *n*-го статистического момента проводятся с помощью следующего выражения:

```
numerator += Math.pow(element – point, N);
```

Приведенная выше строка кода в методе `firstMomentAbout` оказывается вполне пригодной для расчетов, поскольку выражение `element – point` в ней равнозначно выражению `Math.pow(element – point, 1.0)`.

В данный момент у нас два варианта выбора. Обратив внимание на общность обоих упомянутых выше методов, мы можем написать сначала общий метод, воспринимающий конкретную точку, относительно которой выполняются расчеты статистического момента, а также значение `N`, определяющее порядок этого момента, и затем заменить каждый пример применения метода `firstMomentAbout(double)` вызовом нового общего метода. Конечно, мы могли бы это сделать, но тогда при вызове данного метода пришлось бы указывать значение `N`, а нам бы не хотелось, чтобы клиенты предоставляли при этом произвольное значение `N`. Но похоже, что мы уклонились немного в сторону, поэтому остановимся пока что на этом, чтобы завершить то, с чего мы начали, а именно: подготовить тест к компиляции. К обобщению методов мы можем вернуться позднее, если сочтем это по-прежнему актуальным.

Для подготовки к компиляции нам достаточно скопировать метод `firstMomentAbout` и переименовать его в метод `secondMomentAbout`.

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += element - point;
    }
    return numerator / elements.size();
}
```

Подготовка к прохождению

Данный код не проходит тестирование. В таком случае мы можем вернуться к подготовке теста к прохождению, внеся в код следующие изменения.

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += Math.pow(element - point, 2.0);
    }
    return numerator / elements.size();
}
```

Возможно, вас несколько смутило такое вырезание, копирование и вставка кода, но в дальнейшем мы постараемся исключить дублирование кода в методе `secondMomentAbout`. Несмотря на то, что мы пишем в данном примере совершенно новый код, копирование нужного фрагмента кода и его видоизменение в новом методе оказывается достаточно эффективным в контексте унаследованного кода. Нередко приходится вводить новые свойства в особенно скверный код, и тогда нам легче понять результаты своих модификаций кода, если мы поместим их в новом месте и сравним их рядом со старым кодом. В дальнейшем можем исключить дублирование кода, чтобы аккуратно свести новый код в отдельный класс, или же просто отказаться от модификации кода и попытаться выполнить ее иначе, зная, что в нашем распоряжении по-прежнему имеется старый код для изучения и анализа.

Исключение дублирования кода

А теперь, когда оба теста проходят, мы можем перейти к следующей стадии: исключению дублирования кода. Как же нам сделать это?

Для этого можно, в частности, извлечь все тело метода `secondMomentAbout`, переименовать его на `nthMomentAbout` и дополнить его параметром `N`.

```
public double secondMomentAbout(double point)
        throws InvalidBasisException {
    return nthMomentAbout(point, 2.0);
}

private double nthMomentAbout(double point, double n)
        throws InvalidBasisException {

    if (elements.size() == 0)
        throw new InvalidBasisException("элементы отсутствуют");

    double numerator = 0.0;
```

```
    for (Iterator it = elements.iterator(); it.hasNext(); ) {
        double element = ((Double)(it.next())).doubleValue();
        numerator += Math.pow(element - point, n);
    }
    return numerator / elements.size();
}
```

Если мы теперь выполним наши тесты, то обнаружим, что они по-прежнему проходят. Далее мы можем вернуться к методу `firstMomentAbout` и заменить его тело вызовом метода `nthMomentAbout`.

```
public double firstMomentAbout(double point)
        throws InvalidBasisException {
    return nthMomentAbout(point, 1.0);
}
```

Эта последняя стадия исключения дублирования очень важна. Ведь мы можем быстро и просто ввести новые свойства в код, копируя целые блоки кода. Но если мы не исключим в дальнейшем дублирование кода, то добавим немало хлопот себе на стадии разработки и модификации программного обеспечения и другим на стадии его сопровождения. С другой стороны, при наличии всех необходимых тестов на местах мы можем без особого труда исключить дублирование кода. Мы убедились в этом на рассматриваемом здесь примере, но в данном случае тесты оказались в нашем распоряжении лишь потому, что мы с самого начала пользовались разработкой посредством тестирования. В унаследованном коде особое значение имеют тесты, которые мы пишем и размещаем вокруг существующего кода по ходу разработки посредством тестирования. Разместив их по местам, мы можем свободно писать любой код, который требуется для ввода нового свойства, зная, что его можно добавить к остальному коду, не усугубляя положение.

Разработка посредством тестирования и унаследованный код

Одно из самых ценных свойств разработки посредством тестирования состоит в том, что она позволяет нам поочередно сосредоточивать основное внимание на решении отдельных задач. Мы либо пишем код, либо реорганизуем его, но никогда не делаем и то, и другое одновременно.

Такое разделение особенно ценно в работе с унаследованным кодом, поскольку оно позволяет нам писать новый код независимо от старого.

Написав новый код, мы можем реорганизовать его, исключив любое дублирование нового и старого кода.

Для унаследованного кода разработка посредством тестирования выполняется в следующей расширенной последовательности.

0. *Подготовка к тестированию класса, в который требуется внести изменения.*

1. Написание контрольного примера непрохождения теста.

2. Подготовка к компиляции.

3. Подготовка к прохождению. (*При этом желательно не изменять существующий код.*)

4. Исключение дублирования кода.

5. Повтор.

Программирование по разности

Разработка посредством тестирования никак не связана с объектной ориентацией. На самом деле в примере из предыдущего раздела приведен фрагмент процедурного кода, заключенный в оболочку класса. В объектно-ориентированном программировании у нас имеется другая возможность: использовать наследование, чтобы вводить новые свойства, не видоизменяя класс непосредственно. После ввода нового свойства мы можем точно определить, как нам его интегрировать.

Основной способ сделать это называется *программированием по разности*. Это довольно старый способ, широко обсуждавшийся и применявшийся еще в 1980-е годы, но в 1990-е годы он утратил свою популярность, как только в сообществе ООП было замечено, что наследование может вызывать серьезные осложнения, если пользоваться им чрезмерно. Но это совсем не означает, что мы должны отказаться от наследования. С помощью тестов мы можем без особого труда переходить к другим структурам, если наследование вызывает какие-то осложнения.

Покажем принцип программирования по разности на конкретном примере. Допустим, что мы протестировали класс Java, называемый `MailForwarder` (пересылка почты) и являющийся составной частью программы на языке Java, управляющей списками рассылки. У этого класса имеется метод под названием `getFromAddress` (получить адрес отправителя). В коде это выглядит следующим образом:

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    Address [] from = message.getFrom ();
    if (from != null && from.length > 0)
        return new InternetAddress (from [0].toString ());
    return new InternetAddress (getDefaultFrom());
}
```

Назначение данного метода — извлечь адрес отправителя полученного почтового сообщения и возвратить его, чтобы затем воспользоваться им в качестве адреса отправителя сообщения, пересылаемого по списку получателей.

Он используется только в одном месте, т.е. в следующих строках кода из метода под названием `forwardMessage` (переслать сообщение):

```
MimeMessage forward = new MimeMessage (session);
forward.setFrom (getFromAddress (message));
```

Как нам поступить далее, если потребуется поддержка анонимных списков рассылки в качестве нового требования к программе? Члены этих списков могут отправлять сообщения, но в качестве адреса отправителя в их сообщениях должен быть задан конкретный адрес электронной почты, исходя из значения переменной экземпляра `domain` класса `MessageFowarder` (пересылка сообщений). Ниже приведен контрольный пример непрохождения теста для такого изменения в программе (когда тест выполняется, переменной `expectedMessage` (ожидаемое сообщение) присваивается сообщение, пересылаемое объектом класса `MessageFowarder`).

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new MessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
```

```
    assertEquals ("anon-members@" + forwarder.getDomain(),
    expectedMessage.getFrom ()[0].toString());
}
```

Далее мы выполняем подклассификацию согласно рис. 8.1.

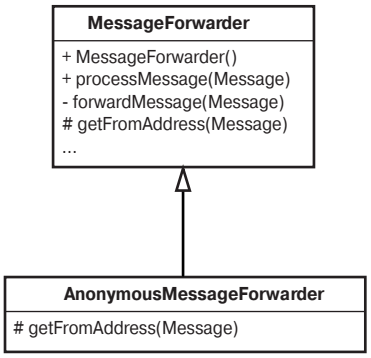

Рис. 8.1. Подклассификация класса `MessageFowarder`

В данном случае метод `getFromAddress` сделан защищенным, а не частным в классе `MessageFowarder`. Затем он был переопределен в классе `AnonymousMessageForwarder` (пересылка анонимных сообщений). В этом классе он выглядит следующим образом:

```
protected InternetAddress getFromAddress(Message message)
        throws MessagingException {
    String anonymousAddress = "anon-" + listAddress;
    return new InternetAddress(anonymousAddress);
}
```

Что же это нам дает? Мы решили поставленную перед нами задачу, но ввели новый класс в систему для очень простого поведения. А стоило ли выполнять подклассификацию целого класса пересылки сообщений ради одного лишь изменения в нем адреса отправителя? В долгосрочной перспективе, пожалуй, и не стоило, но в то же время это позволило нам быстро пройти тест. А если этот тест проходит, то мы можем быть уверены в том, что новое поведение сохранится, если мы решим внести изменения в структуру кода.

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

Что-то уж слишком просто. В чем же загвоздка? А в том, что если пользоваться данным способом слишком часто и не уделять внимания главным элементам структуры кода, то он начнет быстро деградировать. Для того чтобы увидеть, что может при этом произойти, рассмотрим еще одно изменение. Нам нужно не только пересылать сообщения по списку рассылки получателей, но и отправлять их слепую копию (bcc) ряду других получателей, которых нельзя ввести в официальный список рассылки. Таких получателей назовем внесписковыми.

Такое изменение выглядит достаточно простым. Ведь мы можем вновь выполнить подклассификацию класса `MessageForwarder` и переопределить его метод обработки, чтобы отправлять сообщения по данному месту назначения, как показано на рис. 8.2.

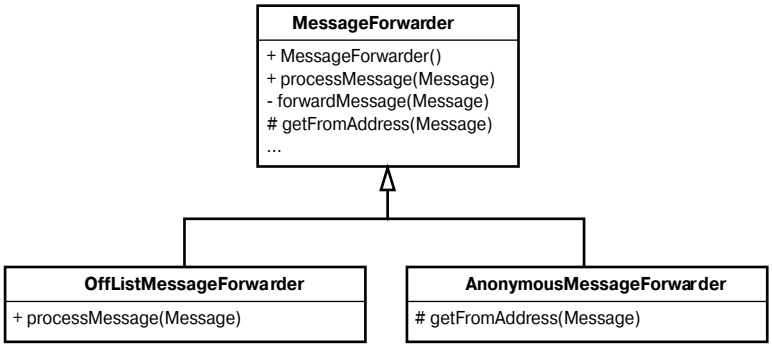

Рис. 8.2. Подклассификация по двум разностям

Такой вариант может оказаться вполне работоспособным, за одним исключением. Что, если в классе `MessageForwarder` нам требуется реализовать две функции — отправку всех сообщений внесписковым получателям и анонимную пересылку сообщений?

Это одна из самых больших проблем, возникающих в связи со слишком широким применением наследования. Если мы поместим свойства в отдельные классы, то они станут доступными нам лишь по очереди.

Как же развязать этот узел? Для этого можно, в частности, отложить на время ввод свойства внесписковых получателей и реорганизовать код, чтобы сделать его более ясным. К счастью, мы написали тест и поместили его в нужном месте, а теперь можем воспользоваться им, чтобы проверить, сохраняется ли поведение при переходе к другому алгоритму.

Свойство анонимной пересылки сообщений мы могли бы реализовать и без подклассификации, сделав его одним из вариантов конфигурации программы. Для этого можно, например, изменить конструктор класса, чтобы он воспринимал совокупность свойств.

```
Properties configuration = new Properties();
configuration.setProperty("anonymous", "true");
MessageForwarder forwarder = new MessageForwarder(configuration);
```

Можем ли мы подготовить наш тест к прохождению? Рассмотрим тест еще раз.

```
public void testAnonymous () throws Exception {
    MessageForwarder forwarder = new AnonymousMessageForwarder();
    forwarder.forwardMessage (makeFakeMessage());
    assertEquals ("anon-members@" + forwarder.getDomain(),
        expectedMessage.getFrom ()[0].toString());
}
```

В настоящий момент этот тест проходит. В классе `AnonymousMessageForwarder` переопределяется метод `getFrom` из класса `MessageForwarder`. А что, если изменить метод `getFrom` в классе `MessageForwarder` следующим образом?

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
```

```
    if (configuration.getProperty("anonymous").equals("true")) {
        fromAddress = "anon-members@" + domain;
    }
    else {
        Address [] from = message.getFrom ();
        if (from != null && from.length > 0) {
            fromAddress = from [0].toString ();
        }
    }
    return new InternetAddress (fromAddress);
}
```

Теперь в классе `MessageForwarder` имеется метод `getFrom`, способный обрабатывать как анонимные, так и обычные сообщения. Для проверки этого факта мы можем закомментировать переопределение метода `getFrom` в классе `AnonymousMessageForwarder` и посмотреть, проходят ли тесты в этом случае.

```
public class AnonymousMessageForwarder extends MessageForwarder
{
/*
    protected InternetAddress getFromAddress(Message message)
            throws MessagingException {
        String anonymousAddress = "anon-" + listAddress;
        return new InternetAddress(anonymousAddress);
    }
*/
}
```

Конечно, они проходят.

Класс `AnonymousMessageForwarder` нам уже не понадобится, и поэтому мы можем его удалить. Для этого нам нужно отыскать в коде все места, где создается класс `AnonymousMessageForwarder`, и заменить вызов его конструктора вызовом конструктора, воспринимающего упомянутую выше совокупность свойств.

Эту совокупность свойств можно использовать и для ввода нового свойства в программу. Для этого достаточно выбрать свойство, активизирующее обработку сообщений для внесписковых получателей.

Можно ли считать дело сделанным? Не совсем. Мы внесли в код метода `getFrom` в классе `MessageForwarder` некоторый беспорядок, но, поскольку у нас имеются соответствующие тесты, мы можем очень быстро извлечь этот метод и немного поправить его. В настоящий момент он выглядит следующим образом.

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        fromAddress = "anon-members@" + domain;
    }
    else {
        Address [] from = message.getFrom ();
        if (from != null && from.length > 0)
            fromAddress = from [0].toString ();
```

```
    }
    return new InternetAddress (fromAddress);
}
```

А после реорганизации кода он выглядит следующим образом.

```
private InternetAddress getFromAddress(Message message)
        throws MessagingException {

    String fromAddress = getDefaultFrom();
    if (configuration.getProperty("anonymous").equals("true")) {
        from = getAnonymousFrom();
    }
    else {
        from = getFrom(Message);
    }
    return new InternetAddress (from);
}
```

Теперь код выглядит чуть более ясным, но в то же время свойства рассылки анонимных сообщений и обработки сообщений для внесписковых получателей сведены в класс `MessageForwarder`. Плохо ли это с точки зрения *принципа единственной ответственности*? Возможно, и плохо — все зависит от того, насколько крупным получается код, связанный с такой ответственностью, и насколько запутанным он оказывается по сравнению с остальным кодом. В данном случае определить анонимность списка не так уж и сложно. Этому способствует выбранный нами подход к совокупности свойств. А что мы будем делать, если таких свойств окажется много и код в классе `MessageForwarder` начнет засоряться условными операторами? В таком случае можно воспользоваться отдельным классом вместо совокупности свойств. В частности, мы можем создать класс под названием `MailingConfiguration` (конфигурация рассылки), чтобы хранить в нем совокупность свойств (рис. 8.3).

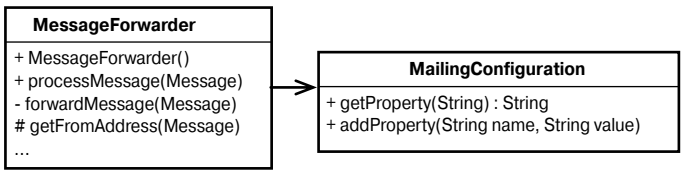

Рис. 8.3. Делегирование полномочий классу `MailingConfiguration`

Такая мера кажется действенной, но не является ли она чрезмерной? Ведь класс `MailingConfiguration`, по-видимому, выполняет те же функции, что и совокупность свойств.

Что, если переместить метод `getFromAddress` в класс `MailingConfiguration`? Класс `MailingConfiguration` мог бы тогда воспринимать сообщение и определять возвращаемый адрес отправителя. Если же конфигурация программы настроена на анонимность сообщений, то он мог бы возвращать адрес отправителя для анонимной рассылки сообщений. В противном случае он мог бы извлечь первый адрес из сообщения и возвратить этот адрес. Тогда структура кода выглядела бы так, как показано на рис. 8.4. Обратите внимание на то, что нам уже не требуется метод для получения и задания свойств. Теперь класс `MailingConfiguration` поддерживает функции более высокого уровня.

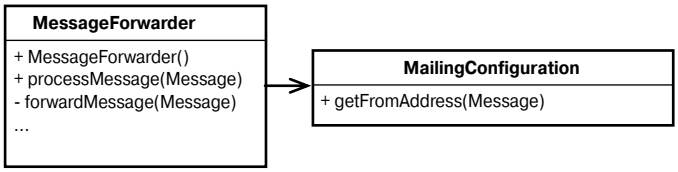

Рис. 8.4. Перенос поведения в класс `MailingConfiguration`

После этого мы могли бы приступить к вводу других методов в класс `Mailing Configuration`. Так, если нам потребовалось бы реализовать свойство обработки сообщений для внесписковых получателей, мы могли бы ввести метод под названием `build RecipientList` (составить список получателей) в класс `MailingConfiguration` и предоставить классу `MessageForwarder` возможность использовать его так, как показано на рис. 8.5.

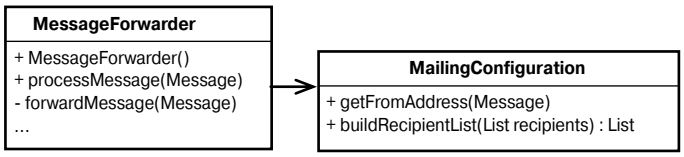

Рис. 8.5. Перенос дополнительного поведения в класс `MailingConfiguration`

При таких изменениях имя рассматриваемого здесь класса не вполне соответствует его функциям. Конфигурация обычно носит довольно пассивный характер, а этот класс активно создает и видоизменяет данные для объектов класса `MessageForwarder` по их запросу. Для такого класса более подходящим будет имя `MailingList` (список рассылки), если в системе отсутствует класс с аналогичным именем. Объекты класса `MessageForwarder` обращаются к спискам рассылки для определения адресов отправителей и составления списков получателей. Можно сказать, что ответственность за определение характера изменений в сообщениях лежит на списке рассылки. На рис. 8.6 показана структура кода после переименования упомянутого класса.

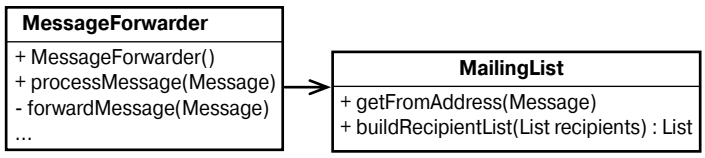

Рис. 8.6. Переименование класса `MailingConfiguration` *в класс* `MailingList`

Среди многих видов эффективной реорганизации кода самым эффективным считается переименование класса (Rename Class). Такая реорганизация кода изменяет точку зрения программистов на код и позволяет им обнаружить в нем те возможности, о которых они прежде и не подозревали.

Программирование по разности — полезный способ, позволяющий быстро вносить изменения в код и использовать тесты для перехода к более ясной конструкции. Но для того чтобы делать это правильно, нам придется принять во внимание два скрытых препятствия. Первым из них является нарушение *принципа подстановки Лискова* (LSP).

Принцип подстановки Лискова

Использование наследования может привести к появлению некоторых едва заметных ошибок. Рассмотрим следующий код.

```
public class Rectangle
{
    ...
    public Rectangle(int x, int y, int width, int height) { … }
    public void setWidth(int width) { ... }
    public void setHeight(int height) { ... }
    public int getArea() { ... }
}
```

В этом коде имеется класс `Rectangle` (прямоугольник). Можем ли мы создать подкласс с именем `Square` (квадрат)?

```
public class Square extends Rectangle
{
    ...
    public Square(int x, int y, int width) { ... }
   ...
}
```

Подкласс `Square` наследует методы `setWidth` (задать ширину) и `setHeight` (задать высоту) от класса `Rectangle`. Какой же должна получиться площадь квадрата в результате выполнения следующего кода?

```
Rectangle r = new Square();
r.setWidth(3);
r.setHeight(4);
```

Если площадь равна 12, то можно ли считать такую фигуру квадратом? Мы могли бы переопределить методы `setWidth` и `setHeight`, чтобы сохранить в подклассе `Square` форму квадрата, или же видоизменить переменные ширины и высоты в этих методах, но это привело бы к не совсем логичным результатам. Так, если задать высоту 4 и ширину 3, то вместо ожидаемой площади 12 будет получена площадь 16 прямоугольника.

Это классический пример нарушения принципа подстановки Лискова. Объекты подклассов должны быть подставлены в коде вместо объектов их суперклассов. В противном случае в коде возникают скрытые ошибки.

Принцип подстановки Лискова подразумевает, что клиенты отдельного класса должны иметь возможность использовать объекты его подкласса, даже не зная о том, что они на самом деле являются объектами подкласса. Каких-то действенных способов полностью исключить нарушение принципа подстановки Лискова не существует. Соответствие класса данному принципу зависит от клиентов этого класса и их ожиданий. Но для соблюдения этого принципа можно руководствоваться следующими эмпирическими правилами.

1. Избегайте, по возможности, переопределения конкретных методов.
2. Если вы все же переопределяете конкретные методы, то проверьте, сможете ли вы вызвать переопределяемый метод в переопределяющем методе.

Но в приведенном выше примере с классом `MessageForwarder` мы не делали ничего подобного. В действительности, мы поступили совсем наоборот: переопределили конкретный метод в подклассе (`AnonymousMessageForwarder`). И что же в этом особенного?

Дело в том, что когда мы переопределяем конкретные методы, как, например, метод `getFromAddress` класса `MessageForwarder` в классе `AnonymousMessageForwarder`, то можем изменить поведение некоторого кода, использующего объекты класса `MessageForwarder`. Если же ссылки на класс `MessageForwarder` размещены по всему приложению и одна из них настроена на обращение к классу `AnonymousMessageForwarder`, то пользователи такой ссылки могут посчитать, что они обращаются к классу `MessageForwarder`, который извлекает адрес получателя из обрабатываемого им сообщения и использует его для обработки других сообщений. Имеет ли для пользователей данного класса значение, какой именно адрес отправителя он использует: упомянутый выше или же другой специальный адрес? Это зависит от конкретного применения. Но в целом код становится более запутанным, когда мы слишком часто переопределяем конкретные методы. Кто-нибудь может заметить в коде ссылку на класс `MessageForwarder`, проанализировать этот класс и посчитать, что код в этом классе служит для выполнения метода `getFromAddress`, не имея даже представления о том, что ссылка на самом деле указывает на класс `AnonymousMessageForwarder` и что в данном случае используется метод `getFromAddress` именно этого класса. Если бы мы действительно хотели сохранить наследование, то могли бы сделать абстрактными класс `MessageForwarder` и его метод `getFromAddress`, предоставив подклассам возможность подставлять конкретные тела своих методов. На рис. 8.7 показано, как должна выглядеть структура кода после таких изменений.

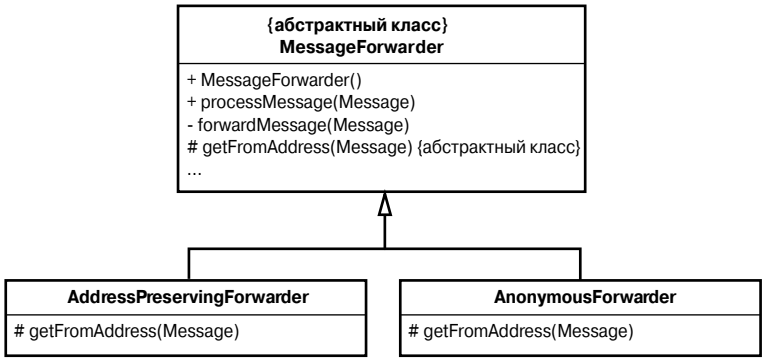

Рис. 8.7. Нормализованная иерархия

Такую иерархию классов можно назвать *нормализованной*. В нормализованной иерархии ни у одного из классов нет более одной реализации метода. Иными словами, ни у одного из классов нет метода, переопределяющего конкретный метод, наследуемый из суперкласса. В этом случае, анализируя функции абстрактного класса, можно обнаружить в нем абстрактные методы, каждый из которых реализуется в одном из подклассов данного класса. В нормализованной иерархии вам не нужно беспокоиться о переопределении в подклассах поведения, наследуемого ими от их суперклассов.

Следует ли делать это постоянно? Несколько переопределений конкретных методов никогда не помешают, если они не нарушают принцип подстановки Лискова. Но всякий

раз рекомендуется подумать о том, насколько классы отдаляются от нормализованной формы, и иногда обращаться к ней перед тем, как разделять ответственность.

Программирование по разности позволяет нам быстро вносить изменения в систему. При этом мы можем точно определить новое поведение с помощью написанных нами тестов и перейти, если потребуется, к более подходящим структурам. Тесты позволяют сделать такой переход довольно быстро.

Резюме

Способы, описанные в этой главе, можно использовать для ввода новых свойств в любой код, подлежащий тестированию. На тему разработки посредством тестирования за последние годы написано немало литературы. В частности, рекомендуются следующие книги: *Экстремальное программирование: разработка через тестирование* Кента Бека, издательство "Питер", 2003, а также *Test-Driven Development: A Practical Guide* (Разработка посредством тестирования. Практическое руководство) Дэйва Эстела (Dave Astel), Prentice Hall Professional Technical Reference, 2003.