1 Мультивалютные деньги

Вначале мы рассмотрим объект, созданный Уордом для системы WyCash, — мультивалютные деньги (см. «Введение»). Допустим, у нас есть отчет вроде этого.

Компания	Количество акций	Цена	Всего
IBM	1000	25	25 000
GE	400	100	40 000
		Итого:	65 000

Добавив различные валюты, получим мультивалютный отчет.

Компания	Количество акций	Цена	Всего
IBM	1000	25 USD	25 000 USD
Novartis	400	150 CHF	60 000 CHF
		Итого	65 000 USD

Также необходимо указать курсы обмена.

Из	В	Курс
CHF	USD	1,5

```
$5 + 10 CHF = $10, если курс обмена 2:1
$5 * 2 = $10
```

Что нам понадобится, чтобы сгенерировать такой отчет? Или, другими словами, какой набор успешно выполняющихся тестов сможет гарантировать, что созданный код правильно генерирует отчет? Нам понадобится:

- выполнять сложение величин в двух различных валютах и конвертировать результат с учетом указанного курса обмена;
- выполнять умножение величин в валюте (стоимость одной акции) на количество акций, результатом этой операции должна быть величина в валюте.

Составим список задач, который будет напоминать нам о планах, не даст запутаться и покажет, когда все будет готово. В начале работы над задачей выделим ее жирным шрифтом, **вот так**. Закончив работу над ней — вычеркнем, ~~вот так~~. Когда придет мысль написать новый тест, добавим новую задачу в наш список.

Как видно из нашего списка задач, сначала мы займемся умножением. Итак, какой объект понадобится нам в первую очередь? Вопрос с подвохом. Мы начнем не с объектов, а с тестов. (Мне приходится постоянно напоминать себе об этом, поэтому я просто притворюсь, что вы так же забывчивы, как и я.)

Попробуем снова. Итак, какой тест нужен нам в первую очередь? Если исходить из списка задач, первый тест представляется довольно сложным. Попробуем начать с малого — умножение, — сложно ли его реализовать? Займемся им для начала.

Когда мы пишем тест, мы воображаем, что у нашей операции идеальный интерфейс. Попробуем представить, как будет выглядеть операция снаружи. Конечно, наши представления не всегда будут находить воплощение, но в любом случае стоит начать с наилучшего возможного программного интерфейса (API) и при необходимости вернуться назад, чем сразу делать вещи сложными, уродливыми и «реалистичными».

Простой пример умножения[1]:

```
public void testMultiplication() {
   Dollar five = new Dollar(5);
   five.times(2);
   assertEquals(10, five.amount);
}
```

(Знаю, знаю: публичные поля, побочные эффекты, целые числа для денежных величин и все такое. Маленькие шаги — помните? Мы отметим, что где-то есть душок[2], и продолжим дальше. У нас есть тест, который не выполняется, и мы хотим как можно скорее увидеть зеленую полоску[3].)

```
$5 + 10 CHF = $10, если курс обмена 2:1
$5 * 2 = $10
Сделать переменную amount закрытым членом класса
Побочные эффекты в классе Dollar?
Округление денежных величин?
```

[1] Название метода `times()` можно перевести на русский как «умножить на». — *Примеч. пер.*

[2] Код с душком (code that smells) — распространенная в XP метафора, означающая плохой код (содержащий дублирование). — *Примеч. пер.*

[3] Имеется в виду индикатор успешного выполнения тестов в среде JUnit, имеющий форму полосы. Если все тесты выполнились успешно, полоса становится зеленой. Если хотя бы один тест потерпел неудачу, полоса становится красной. — *Примеч. пер.*

Тест, который мы только что создали, даже не компилируется, но это легко исправить. (О том, когда и как создаются тесты, я расскажу позже — когда мы будем подробнее говорить о среде тестирования, JUnit.) Как проще всего заставить тест компилироваться (пусть он пока и будет терпеть неудачу)? У нас четыре ошибки компиляции:

- нет класса `Dollar`;
- нет конструктора;
- нет метода `times(int)`;
- нет поля (переменной) `amount`.

Устраним их одну за другой. (Я всегда ищу некоторую численную меру прогресса.) От одной ошибки мы избавимся, определив класс `Dollar`:

`Dollar`

```
class Dollar
```

Одной ошибкой меньше, осталось еще три. Теперь нам понадобится конструктор, причем совершенно необязательно, чтобы он что-то делал — лишь бы компилировался.

`Dollar`

```
Dollar(int amount) {
}
```

Осталось две ошибки. Необходимо создать заготовку метода `times()`. Снова мы выполним минимум работы, только чтобы заставить тест компилироваться:

`Dollar`

```
void times(int multiplier) {
}
```

Теперь осталась только одна ошибка. Чтобы от нее избавиться, нужно создать поле (переменную) `amount`:

`Dollar`

```
int amount;
```

Отлично! Теперь можно запустить тест и убедиться, что он не выполняется: ситуация продемонстрирована на рис. 1.1.

Загорается зловещий красный индикатор. Фреймворк тестирования (JUnit в нашем случае) выполнил небольшой фрагмент кода, с которого мы начали, и выяснил, что вместо ожидаемого результата «10» получился «0». Ужасно...

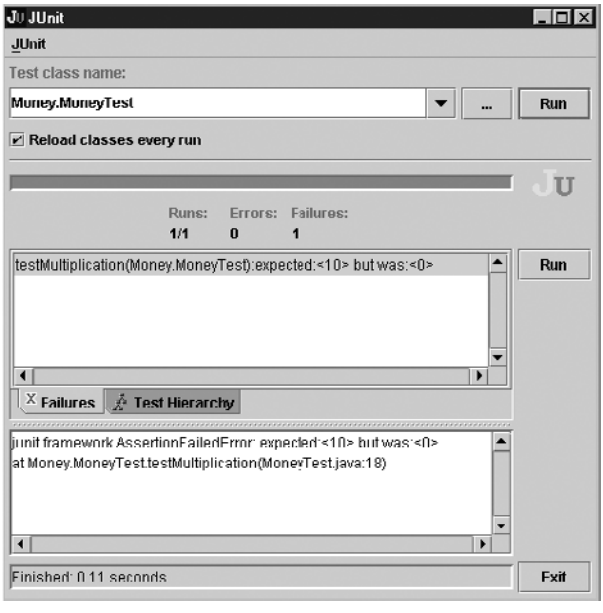

Рис. 1.1. Прогресс! Тест терпит неудачу

Вовсе нет! Неудача — это тоже прогресс. Теперь у нас есть конкретная мера неудачи. Это лучше, чем просто догадываться, что у нас что-то не так. Наша задача «реализовать мультивалютность» превратилась в «заставить работать этот тест, а потом заставить работать все остальные тесты». Так намного проще и намного меньше поводов для страха. Мы заставим этот тест работать.

Возможно, вам это не понравится, но сейчас наша цель не получить идеальное решение, а заставить тест выполняться. Мы принесем свою жертву на алтарь истины и совершенства чуть позже.

Наименьшее изменение, которое заставит тест успешно выполняться, представляется мне таким:

Dollar

```
int amount = 10;
```

Рисунок 1.2 показывает результат повторного запуска теста. Теперь мы видим ту самую зеленую полоску, воспетую в поэмах и прославленную в веках.

Вот оно, счастье! Но радоваться рано, ведь цикл еще не завершен. Уж слишком мал набор входных данных, которые заставят такую странно попахивающую и наивную реализацию работать правильно. Перед тем как двигаться дальше, немного поразмышляем.

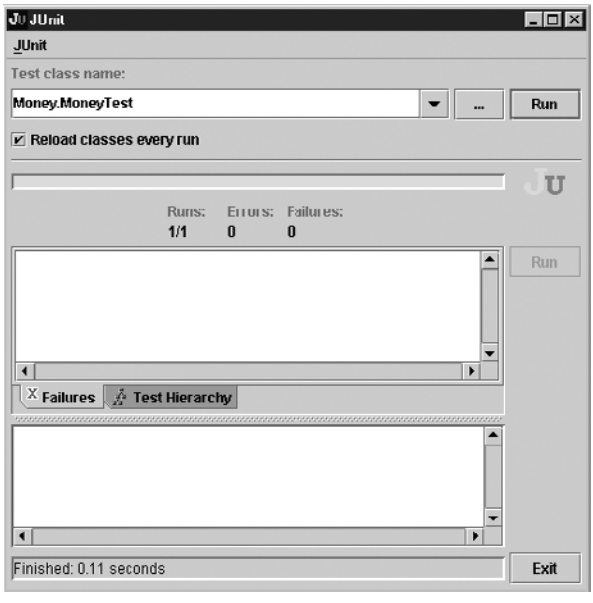

Рис. 1.2. Тест успешно выполняется

Вспомним, полный цикл TDD состоит из следующих этапов:

1. Добавить небольшой тест.
2. Запустить все тесты и убедиться, что новый тест терпит неудачу.
3. Внести небольшое изменение.
4. Снова запустить тесты и убедиться, что все они успешно выполняются.
5. Устранить дублирование с помощью рефакторинга.

ЗАВИСИМОСТЬ И ДУБЛИРОВАНИЕ

Стив Фримен (Steve Freeman) указал, что проблема с тестами и кодом заключается не в дублировании (на которое я еще не указал вам, но сделаю это, как только закончится отступление). Проблема заключается в зависимости между кодом и тестами — вы не можете изменить одно, не изменив другого. Наша цель — иметь возможность писать новые осмысленные тесты, не меняя при этом код, что невозможно при нашей текущей реализации.

Зависимость является ключевой проблемой разработки программного обеспечения. Если фрагменты SQL, зависящие от производителя используемой базы данных, разбросаны по всему коду и вы хотите поменять производителя, то непременно окажется, что код зависит от этого производителя. Вы не сможете поменять производителя базы данных и при этом не изменить код.

Зависимость является проблемой, а дублирование — ее симптомом. Чаще всего дублирование проявляется в виде дублирования логики — одно и то же выражение появляется в различных частях кода. Объекты — отличный способ абстрагирования, позволяющий избежать данного вида дублирования.

> В отличие от большинства проблем в реальной жизни, где устранение симптомов приводит только к тому, что проблема проявляется в худшей форме где-то еще, устранение дублирования в программах устраняет и зависимость. Именно поэтому существует второе правило TDD. Устраняя дублирование перед тем, как заняться следующим тестом, мы максимизируем наши шансы сделать его успешным, внеся всего одно изменение.

Мы выполнили первые четыре пункта цикла, и все готово к устранению дублирования. Но где же оно? Обычно мы замечаем дублирование в нескольких разных фрагментах кода, однако в нашем случае — друг друга дублируют тест и тестируемый код. Еще не видите? Как насчет того, чтобы написать так:

Dollar

```
int amount = 5 * 2;
```

Теперь ясно, откуда мы взяли число 10. Видимо, мы в уме произвели умножение, причем так быстро, что даже не заметили. Произведение «5 умножить на 2» присутствует как в тесте, так и в тестируемом коде. Только изначально в коде оно было представлено в виде константы 10. Сейчас же 5 и 2 отделены друг от друга, и мы должны безжалостно устранить дублирование, перед тем как двинуться дальше. Такие вот правила.

Действия, с помощью которого мы устранили бы 5 и 2 за один шаг, не существует. Но что, если переместить установку поля (переменной) `amount` в метод `times()`?

Dollar

```
int amount;

void times(int multiplier) {
   amount= 5 * 2;
}
```

Тест все еще успешно выполняется, и индикатор остался зеленым. Успех нам пока сопутствует.

Такие шаги кажутся вам слишком мелкими? Помните, TDD не обязывает двигаться только микроскопическими шагами, речь идет о способности совершать эти микроскопические шаги. Буду ли я программировать день за днем такими маленькими шагами? Нет. Но когда дела совсем плохи, я рад возможности выполнять хоть такие шаги. Примените микроскопические шаги к любому собственному примеру. Если вы сможете продвигаться маленькими шагами, вы сумеете делать шаги более крупного и подходящего размера. Если же вы способны делать только огромные шаги, вы никогда не распознаете ситуацию, в которой более уместны меньшие шаги.

Оставим рассуждения. На чем мы остановились? Ну да, мы избавлялись от дублирования между кодом теста и рабочим кодом. Где мы можем взять 5? Это значение передавалось конструктору, поэтому его можно сохранить в переменной `amount`:

Dollar

```
Dollar(int amount) {
   this.amount = amount;
}
```

и использовать в методе `times()`:

Dollar

```
void times(int multiplier) {
   amount = amount * 2;
}
```

Число 2 передается в параметре `multiplier`, поэтому подставим параметр вместо константы:

Dollar

```
void times(int multiplier) {
   amount = amount * multiplier;
}
```

Чтобы продемонстрировать, как хорошо мы знаем синтаксис языка Java, используем оператор `*=` (который, кстати, уменьшает дублирование):

Dollar

```
void times(int multiplier) {
   amount *= multiplier;
}
```

```
$5 + 10 CHF = $10, если курс обмена 2:1
~~$5 * 2 = $10~~
Сделать переменную amount закрытым членом класса
Побочные эффекты в классе Dollar?
Округление денежных величин?
```

Теперь можно пометить первый тест как завершенный. Далее мы позаботимся о тех странных побочных эффектах; но сначала давайте подведем итоги. Мы сделали следующее:

- создали список тестов, которые — мы знаем — нам понадобятся;
- с помощью фрагмента кода описали, какой мы хотим видеть нашу операцию;
- временно проигнорировали особенности среды тестирования JUnit;
- заставили тесты компилироваться, написав соответствующие заготовки;
- заставили тесты работать, использовав сомнительные приемы;
- слегка улучшили работающий код, заменив константы переменными;
- добавили пункты в список задач, вместо того чтобы заняться всеми этими задачами сразу.