Глава 4

Создание тестов

Рефакторинг — ценный инструмент, но он не может существовать в отрыве от тестов. Чтобы правильно выполнить рефакторинг, мне нужен солидный набор тестов, чтобы найти неизбежные ошибки, допущенные мной. Даже при использовании автоматизированных инструментов рефакторинга многие из моих рефакторингов все равно будут нуждаться в проверке с использованием набора тестов.

Я не считаю это недостатком. Даже без рефакторинга написание хороших тестов повышает эффективность моей работы как программиста. Это было неожиданностью для меня и контринтуитивно для большинства программистов, поэтому на этом вопросе стоит остановиться подробнее.

Важность самотестируемого кода

Если вы посмотрите на то, как большинство программистов тратят свое время, то обнаружите, что написание кода на самом деле является довольно малой его частью. Некоторое время тратится на выяснение того, что должно быть сделано, некоторое время тратится на проектирование, но большая часть времени тратится на отладку. Я уверен, что каждый читатель может вспомнить долгие часы отладки, часто до поздней ночи или раннего утра. Каждый программист может рассказать историю ошибки, на поиск которой ушел целый день (а то и больше). Исправление ошибки, как правило, делается довольно быстро, но найти ее — сущий кошмар. А когда вы исправляете ошибку, всегда есть вероятность, что появится другая, которую вы заметите только намного позже, и потратите целую вечность, чтобы найти эту ошибку.

Событием, подтолкнувшим меня на путь создания самотестируемого кода, стал разговор на OOPSLA в 1992 году. Некто (припоминается, это был Дэйв Томас (Dave Thomas)) небрежно заметил: "Классы должны содержать тесты для самих себя". Мне пришло в голову, что это неплохой способ организации тестирования. Я истолковал эти слова так, что в каждом классе должен быть свой метод (например, с именем `test`), с помощью которого класс может протестировать сам себя. Тогда я занимался инкрементной разработкой, а потому попытался по завершении каждого шага добавлять в классы тестирующие методы. Проект был совсем небольшим, и каждый шаг занимал у нас примерно неделю. Выполнять тесты

стало достаточно просто, но хотя их было легко запускать, само тестирование оставалось весьма утомительным, потому что каждый тест выводил на консоль результаты, которые приходилось проверять. Я человек достаточно ленивый, так что готов как следует потрудиться, лишь бы избавиться от лишней работы. Очевидно, что можно не смотреть на экран в поисках некоторой информации, которая должна быть выведена, а заставить заниматься этим компьютер. Все, что требовалось, — это поместить ожидаемые данные в код теста и провести в нем сравнение. После этого можно было просто вызывать тестирующий метод каждого класса, и если все было в порядке, то он просто выводил на экран сообщение `OK`. Так классы стали самотестируемыми.

Тесты должны быть полностью автоматизированы и проверять свои результаты.

После этого выполнять тесты стало ничуть не труднее, чем компиляцию, так что я начал проводить тестирование при каждой компиляции. Вскоре обнаружилось, что производительность моего труда резко возросла. Я понял, что перестал тратить много времени на отладку! Если я допускал ошибку, перехватываемую предыдущим тестом, это обнаруживалось, как только я запускал этот тест. Поскольку раньше тест работал, я понимал, что ошибка находится там, где я внес изменения после предыдущего тестирования. Тесты я запускал часто, буквально каждые несколько минут, так что всегда знал, где именно находится ошибка — в том коде, который я только что написал. Этот код был еще свеж в памяти, мал по размеру, так что найти ошибку было легко. Часы, которые ранее приходилось тратить на поиск ошибок, превратились в считанные минуты. Такое мощное средство обнаружения ошибок появилось у меня не только благодаря тому, что я писал классы с самотестированием, но и потому, что часто выполнял тестирование.

Придя к таким выводам, я стал тестировать свой код еще более агрессивно. Я стал добавлять тесты сразу же после написания небольших фрагментов функций, не дожидаясь завершения этапа разработки. За день, как правило, добавлялась пара новых функций и тесты для них. На отладку я стал тратить не более нескольких минут в день.

Набор тестов является мощным детектором ошибок, резко сокращающим время их поиска.

Инструменты для написания и организации этих тестов получили большое развитие со времени моих экспериментов. Во время полета из Швейцарии в Атланту на OOPSLA 1997 года Кент Бек в паре с Эрихом Гаммой (Erich Gamma) перенес свою среду модульного тестирования с Smalltalk на Java. Получившийся

каркас под названием JUnit оказал огромное влияние на тестирование программ, вдохновив на создание огромного количества похожих инструментов [41] для множества разных языков.

Убедить других последовать тем же путем, конечно же, было нелегко. Тестам необходим большой объем кода. Самотестирование кажется бессмысленным, пока не убедишься сам, насколько оно ускоряет работу. К тому же многие не просто совершенно не умеют писать тесты, но даже никогда о них и не задумываются. Проводить тестирование вручную — занятие не из веселых, но если его автоматизировать, то написание тестов может даже доставлять удовольствие.

Фактически очень полезно писать тесты еще до начала программирования. Когда требуется добавить новую функциональность, начинайте с создания теста. Это не так глупо, как может показаться. Когда вы пишете тест, то спрашиваете себя, что нужно сделать для добавления этой функциональности. При написании теста вы сосредоточиваетесь на интерфейсе, а не на реализации, что всегда хорошо. Кроме того, это означает наличие точного критерия завершения кодирования — в конечном итоге тест должен успешно проходиться.

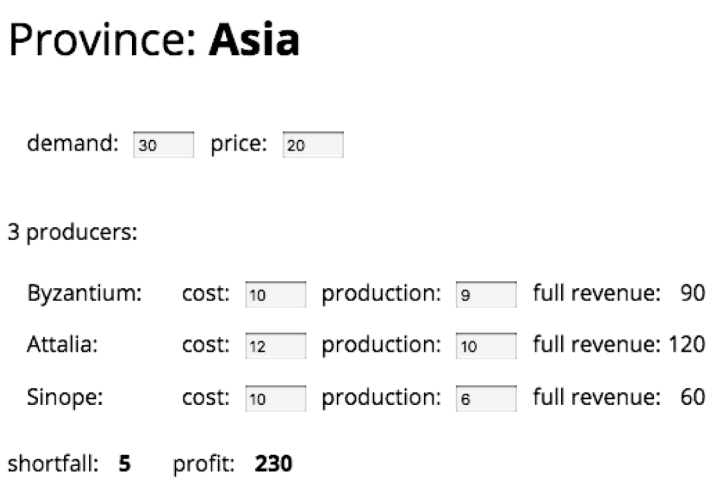
Кент Бек превратил привычку сначала писать тест в методику под названием *Разработка на основе тестов* (Test-Driven Development — TDD) [36]. Такой подход к разработке основывается на коротких циклах написания теста, создания кода, успешно проходящего этот тест, и рефакторинга для обеспечения максимально чистого результата. Этот цикл проверки-кода-рефакторинга должен выполняться много раз в час и может быть очень продуктивным и успокаивающим способом написания кода. Я не собираюсь обсуждать его здесь, но я его использую сам и горячо рекомендую всем.

Но прекратим эту бесполезную дискуссию. Хотя я уверен, что написание самотестируемого кода крайне выгодно для всех, эта книга все же посвящена другой теме — рефакторингу. Рефакторинг требует тестов. Тому, кто собирается заниматься рефакторингом, просто необходимо писать тесты. В этой главе дается беглое представление о том, как это делается при работе на языке программирования JavaScript. Книга посвящена не тестированию, так что я не стану погружаться в детали. Но что касается тестирования, я твердо убедился, что даже малое его количество может принести очень большую пользу.

Как и все остальное в этой книге, подход к тестированию описан с применением большого количества примеров. При разработке кода лично я пишу тесты прямо по ходу работы; но зачастую, когда над рефакторингом работает группа людей, приходится иметь дело с большим объемом кода при отсутствии самотестирования. Поэтому прежде чем применить рефакторинг, нужно сделать код самотестируемым.

Пример кода для тестирования

Вот небольшой код, который нужно просмотреть и проверить. Код поддерживает простое приложение, которое позволяет пользователю изучать производственный план и манипулировать им. (Необработанный) графический интерфейс пользователя выглядит следующим образом.



Производственный план имеет спрос (demand) и цену (price) для каждой географической области. В каждой области есть производители, каждый из которых может производить определенное количество единиц продукции по определенной цене. Интерфейс также показывает, сколько дохода получит каждый производитель, если продаст всю свою продукцию. Внизу экрана показан дефицит производства (спрос минус общий объем производства) и прибыль по этому плану. Пользовательский интерфейс позволяет пользователю манипулировать спросом, ценой, производством и затратами отдельного производителя, чтобы увидеть их влияние на дефицит производства и прибыль. Всякий раз, когда пользователь меняет любое число на дисплее, все остальные значения немедленно обновляются.

Здесь я показал пользовательский интерфейс, чтобы вы могли почувствовать, как используется программное обеспечение, но я планирую сосредоточиться только на бизнес-логике программного обеспечения, т.е. на классах, которые рассчитывают прибыль и дефицит, а не на коде, который генерирует HTML и передает изменения полей к базовой бизнес-логике. Эта глава — просто введение в мир самотестируемого кода, поэтому для меня имеет смысл начать с самого простого случая — кода, который не включает пользовательский интерфейс, сохранение или взаимодействие с внешними службами. Такое разделение является хорошей

идеей в любом случае: как только бизнес-логика усложнится, я отделю ее от механики пользовательского интерфейса, чтобы было легче ее обдумывать и тестировать.

Этот код бизнес-логики включает два класса: класс, представляющий одного производителя, и класс, представляющий целую область. Конструктор области принимает объект JavaScript — тот, который мы могли бы предоставлять в виде документа JSON.

Вот код, который загружает информацию об области из данных JSON:

```
class Province...
constructor(doc) {
  this._name = doc.name;
  this._producers = [];
  this._totalProduction = 0;
  this._demand = doc.demand;
  this._price = doc.price;
  doc.producers.forEach(d=>this.addProducer(new Producer(this,d)));
}

addProducer(arg) {
  this._producers.push(arg);
  this._totalProduction += arg.production;
}
```

Эта функция создает соответствующие данные JSON. Я могу создать образец области для тестирования, создавая объект области из результата следующей функции.

```
Верхний уровень..
function sampleProvinceData() {
  return {
    name: "Asia",
    producers: [
      {name: "Byzantium", cost: 10, production: 9},
      {name: "Attalia",   cost: 12, production: 10},
      {name: "Sinope",    cost: 10, production: 6},
    ],
    demand: 30,
    price: 20
  };
}
```

У класса области есть методы доступа к различным значениям данных.

```
class Province...
get name()               {return this._name;}
get producers()          {return this._producers.slice();}
get totalProduction()    {return this._totalProduction;}
set totalProduction(arg) {this._totalProduction = arg;}
get demand()             {return this._demand;}
```

```
set demand(arg)          {this._demand = parseInt(arg);}
get price()              {return this._price;}
set price(arg)           {this._price = parseInt(arg);}
```

Методы установки вызываются со строками из пользовательского интерфейса, которые содержат числа, поэтому мне нужно выполнить преобразования строк в числа, чтобы использовать их в расчетах.

Класс производителя в основном представляет собой простое хранилище данных:

```
class Producer...
constructor(aProvince, data) {
  this._province = aProvince;
  this._cost = data.cost;
  this._name = data.name;
  this._production = data.production || 0;
}

get name()    {return this._name;}
get cost()    {return this._cost;}
set cost(arg) {this._cost = parseInt(arg);}

get production() {return this._production;}
set production(amountStr) {
  const amount = parseInt(amountStr);
  const newProduction = Number.isNaN(amount) ? 0 : amount;
  this._province.totalProduction+=newProduction - this._production;
  this._production = newProduction;
}
```

Способ, которым `set production` обновляет данные в области, уродлив, и всякий раз, когда я вижу такое, я хочу выполнить рефакторинг, чтобы удалить этот ужас. Но прежде чем я смогу выполнить рефакторинг, следует написать тесты.

Расчет дефицита прост.

```
class Province...
get shortfall() {
  return this._demand - this.totalProduction;
}
```

Расчет прибыли немного более сложен.

```
class Province...
get profit() {
  return this.demandValue - this.demandCost;
}
get demandCost() {
  let remainingDemand = this.demand;
  let result = 0;
  this.producers
    .sort((a,b) => a.cost - b.cost)
```

```
    .forEach(p => {
      const contribution=Math.min(remainingDemand,p.production);
      remainingDemand -= contribution;
      result += contribution * p.cost;
    });
  return result;
}
get demandValue() {
  return this.satisfiedDemand * this.price;
}
get satisfiedDemand() {
  return Math.min(this._demand, this.totalProduction);
}
```

Первый тест

Чтобы протестировать этот код, мне понадобится какой-то каркас тестирования. Их имеется немало, даже только для JavaScript. Я использую довольно распространенный каркас Mocha [43]. Не буду вдаваться в подробное объяснение того, как использовать этот каркас, просто покажу несколько примеров тестов с ним. Вы легко адаптируете используемый вами каркас для создания похожих тестов.

Вот простой тест для расчета дефицита.

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
  });
});
```

Каркас Mocha разделяет код теста на блоки, каждый из которых объединяет набор тестов. Каждый тест отображается в блоке `it`. Для данного простого случая тест состоит из двух этапов. На первом этапе настраиваются некоторые "приборы" тестирования — данные и объекты, необходимые для теста; в данном случае в роли такого прибора выступает загруженный объект области. Вторая строка проверяет некоторые характеристики кода; в данном случае, что дефицит соответствует сумме, которую следует ожидать для используемых исходных данных.

Разные разработчики используют описательные строки в блоках `describe` и `it` по-разному. Некоторые пишут пояснение, что именно проверяет тест, другие предпочитают оставлять их пустыми, утверждая, что описательное предложение просто дублирует код так же, как и комментарий. Я предпочитаю небольшое описание, чтобы понимать при сбое, какой тест оказался не пройден.

Когда я запускаю этот тест в консоли NodeJS, вывод выглядит следующим образом:

```
''''''''''''''

1 passing (61ms)
```

Обратите внимание на простоту вывода — просто краткое указание, сколько тестов было выполнено и сколько прошло успешно.

Необходимо убедиться, что тест не даст ложное положительное срабатывание (не покажет, что все в порядке при наличии ошибки).

Когда я пишу тест для существующего кода, как в данном случае, увидеть, что все хорошо, приятно, но я отношусь к этому скептически. В частности, я всегда нервничаю, что тест на самом деле не выполняет код так, как я предполагаю, а потому не обнаружит имеющуюся ошибку. Поэтому я предпочитаю, чтобы каждый тест оказался провален хотя бы один раз. Мой любимый способ сделать это — временно ввести в код ошибку, например:

```
class Province...
get shortfall() {
  return this._demand - this.totalProduction * 2;
}
```

Вот как теперь выглядит вывод на консоль:

```
!

  0 passing (72ms)
  1 failing

  1) province shortfall:
    AssertionError: expected -20 to equal 5
      at Context.<anonymous> (src/tester.js:10:12)
```

Каркас указывает, какой тест не пройден, и дает некоторую информацию о природе сбоя (в данном случае — какое значение ожидалось и какое было получено на самом деле). Поэтому я сразу замечаю, что что-то не так, вижу, какие тесты не пройдены, и это дает мне подсказку о том, что пошло не так (и в данном случае подтверждение того, что проблема именно там, где я ее внес).

Запускайте тесты почаще. Запускайте тесты для кода, над которым вы работаете, хотя бы раз в несколько минут; запускайте все тесты хотя бы раз в день.

В реальной системе могут быть тысячи тестов. Хороший тестовый каркас позволяет легко их запускать и сразу же определять, успешно ли они пройдены. Такая простая обратная связь необходима для самотестирования кода. Работая, я очень часто запускаю тесты для проверки нового кода или наличия ошибок при рефакторинге.

Каркас Mocha может использовать различные библиотеки, которые он называет *библиотеками утверждений* (assertion libraries) для проверки настроек теста. Для JavaScript их насчитывается неимоверное количество. В настоящее время я использую Chai [5], что позволяет мне писать свои проверки в стиле "утверждений":

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
  });
});
```

или в стиле "ожиданий":

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });
});
```

Я обычно предпочитаю стиль утверждений, но в настоящее время при работе в JavaScript я в основном использую стиль ожиданий.

Различные среды предоставляют разные способы запуска тестов. Программируя на Java, я использую интегрированную среду разработки, которая предоставляет графический тестер. Индикатор выполнения остается зеленым, когда проходят все тесты, и становится красным в случае неудачи любого из них. Мои коллеги для описания состояния тестов часто используют фразы "зеленая полоса" и "красная полоса". Я мог бы сказать: "Никогда не выполняй рефакторинг на красной полосе", т.е. что не следует приступать к рефакторингу, если в вашем тестовом наборе есть сбойный тест. Или можно сказать "Вернитесь к зеленому", т.е. что вы должны отменить последние изменения и вернуться к последнему состоянию, когда у вас успешно проходил весь набор тестов (обычно это означает вернуться к последней контрольной точке системы управления версиями).

Графические тестеры хороши, но не обязательны. Обычно мои тесты "привязаны" к комбинации клавиш в Emacs, и я наблюдаю текстовую обратную связь в окне компиляции. Ключевым моментом является то, что я могу быстро увидеть, все ли тесты в порядке.

Добавление другого теста

Теперь я продолжу добавлять другие тесты. Стиль, которому я следую, заключается в том, чтобы посмотреть все действия, которые должен выполнять класс, и проверить каждое из них на наличие условий, которые могут привести к сбою работы класса. Это не то же самое, что тестирование каждого открытого метода (подход, который защищают некоторые программисты). Тестирование должно быть ориентировано на риск; я пытаюсь найти ошибки — сейчас или в будущем. Поэтому я не тестирую методы доступа к полям, которые просто читают или записывают поле: они настолько просты, что вряд ли я найду там ошибку.

Это важный момент, так как попытка написать слишком много тестов обычно приводит к тому, что их оказывается недостаточно. Я прочел не одну книгу о тестировании, вызывавшую одно лишь желание любой ценой уклониться от той необъятной работы, которую предлагалось проделать. Всеохватывающее тестирование контрпродуктивно, поскольку заставляет считать, что тестирование всегда связано с чрезмерным трудом. Однако тестирование приносит ощутимую пользу, даже если осуществляется в небольшом объеме. Главное для решения проблемы тестирования — тестировать в основном код, возможность ошибок в котором вызывает наибольшее беспокойство. При этом подходе усилия, затраченные на тестирование, дают максимальную выгоду.

Лучше написать и выполнить неполные тесты, чем не выполнить полные тесты.

Так что я начну с другого основного вывода этого кода — расчета прибыли. И вновь я просто выполняю базовый тест прибыли для исходных данных.

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });

  it('profit', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.profit).equal(230);
  });
});
```

Здесь показан конечный результат, но способ, которым я его получил, заключался в том, чтобы сначала установить ожидаемое значение в качестве заполнителя, а затем заменить его значением, полученным от программы (`230`). Я мог бы рассчитать его вручную, но так как код должен работать правильно, сейчас я

просто доверяю ему. После того как этот новый тест заработал для правильного случая, я проверяю сам тест, изменяя расчет прибыли путем добавления ложного умножения на 2. Я убеждаюсь, что тест (как и следует) проваливается, а затем убираю введенную мною ошибку. Этот шаблон — запись с использованием заполнителя для ожидаемого значения, замена заполнителя фактическим значением, возвращаемым кодом, введение ошибки, восстановление корректного кода — является наиболее распространенной моделью, которую я использую при добавлении тестов в существующий код.

Между этими тестами есть определенное дублирование — они оба работают с одной и той же первой строкой. Так же, как я с подозрением отношусь к дублированию в обычном коде, я с подозрением отношусь к нему и в тестовом коде, и поэтому постараюсь удалить его. Один из вариантов заключается в поднятии константы во внешнюю область видимости.

```
describe('province', function() {
  const asia = new Province(sampleProvinceData()); // Не делай так
  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });

  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

Но, как указывает комментарий, я никогда так не делаю. В данный момент это сработает, но на самом деле это рассадник самых неприятных ошибок в тестировании — совместно используемый объект, который заставляет тесты взаимодействовать. Ключевое слово `const` в JavaScript означает только константность ссылки на `asia`, а не константность содержимого этого объекта. Если будущий тест изменит этот общий объект, то в конечном итоге могут появляться периодические сбои тестов из-за их взаимодействия через общие объекты, что может приводить к разным результатам в зависимости от порядка выполнения тестов. Этот недетерминизм в тестах может привести к долгой и трудной отладке в лучшем случае и полному недоверию к тестам — в худшем. Вместо этого я предпочитаю поступать следующим образом.

```
describe('province', function() {
  let asia;
  beforeEach(function() {
    asia = new Province(sampleProvinceData());
  });

  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });
```

```
  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

Конструкция `beforeEach` запускается перед каждым тестом, сбрасывая объект `asia` и устанавливая каждый раз новое значение переменной. Таким образом, перед каждым тестом создается новый объект, что делает тесты изолированными один от другого и предотвращает недетерминированность, вызывающую столько проблем.

Когда я даю этот совет, некоторые программисты беспокоятся, не будет ли создание нового объекта каждый раз тормозить тесты. В большинстве случаев это будет незаметно. Если же это создает проблему, я бы рассмотрел возможность совместного использования объекта, но с большими предосторожностями, чтобы ни один тест никогда его не изменял. Прибегнуть к совместно используемому объекту можно при гарантии, что он действительно неизменный. Но мой рефлекс состоит в том, чтобы использовать новый объект, потому что в прошлом у меня были слишком большие неприятности, связанные с ошибкой из-за совместно используемого объекта.

Учитывая, что я запускаю код настройки в `beforeEach` для каждого теста, почему бы не оставить его внутри отдельных блоков `it`? Но мне нравится, когда все мои тесты работают с одним и тем же объектом. Наличие блока `beforeEach` сигнализирует читателю кода, что я использую стандартный объект для тестирования. Затем вы можете просмотреть все тесты в рамках этого блока `describe`, зная, что они принимают одни и те же базовые данные.

Изменение прибора тестирования

До сих пор написанные мною тесты показывали, как я проверяю свойства прибора тестирования — объекта `asia` — после его загрузки. Но при использовании этот объект будет регулярно обновляться пользователями по мере изменения значений.

Большинство обновлений являются простыми методами установки значений, и я обычно не пытаюсь их тестировать в силу крайне малой вероятности, что они станут источником ошибки. Но у метода установки `production` достаточно сложное поведение, так что, я думаю, его стоит проверить.

```
describe('province'...
  it('change production', function() {
    asia.producers[0].production = 20;
    expect(asia.shortfall).equal(-6);
    expect(asia.profit).equal(292);
  });
```

Это распространенный шаблон. Я беру исходный стандартный прибор, *настроенный* блоком `beforeEach`, выполняю *тестирование*, затем *проверяю*, что сделано именно то, что должно быть сделано. Если вы прочтете о тестировании побольше, то увидите эту последовательность, возможно, описанную другими словами. Иногда вы увидите все шаги в самом тесте, в других случаях некоторые ранние фазы могут быть перенесены в стандартные процедуры настройки, такие как `beforeEach`.

(Есть еще один неявный четвертый этап, который обычно не упоминается: *удаление*, которое удаляет прибор между тестами, чтобы разные тесты не взаимодействовали друг с другом. Выполняя все настройки в `beforeEach`, я позволяю тестовому каркасу неявно удалять мой тестирующий прибор между тестами, так что я применяю эту фазу неявно. Большинство авторов тестов также задействуют ее неявно, но иногда может оказаться важным иметь явную операцию удаления, особенно если наш прибор совместно используется разными тестами в силу сложности/длительности его создания.)

В этом тесте я проверяю две разные характеристики в одной конструкции `it`. В общем случае целесообразно иметь только одну инструкцию проверки в каждой конструкции `it`. Это связано с тем, что сбойный тест может скрывать полезную информацию, когда вы выясняете, почему именно тест не пройден. В этом случае я чувствую, что проверки достаточно тесно взаимосвязаны, и я проверяю их в одном тесте. Если я захочу разделить их на отдельные конструкции `it`, то смогу сделать это позже.

Проверка границ

До сих пор мои тесты были сосредоточены на регулярном использовании, часто именуемом “счастливым путем”, когда все идет хорошо и все работает, как и ожидалось. Но стоит также проводить тесты и на границах этих условий — чтобы увидеть, что происходит, когда что-то может пойти не так.

Всякий раз, когда у меня есть коллекция чего-либо (в данном примере — производителей), я предпочитаю проверить, что происходит, когда такая коллекция пуста.

```
describe('no producers', function() {
  let noProducers;
  beforeEach(function() {
    const data = {
      name: "No producers",
      producers: [],
      demand: 30,
      price: 20
    };
```

```
    noProducers = new Province(data);
  });
  it('shortfall', function() {
    expect(noProducers.shortfall).equal(30);
  });
  it('profit', function() {
    expect(noProducers.profit).equal(0);
  });
```

В случае обычных чисел неплохо бы проверить как нулевые значения:

```
describe('province'...
  it('zero demand', function() {
    asia.demand = 0;
    expect(asia.shortfall).equal(-25);
    expect(asia.profit).equal(0);
  });
```

так и отрицательные:

```
describe('province'...
  it('negative demand', function() {
    asia.demand = -1;
    expect(asia.shortfall).equal(-26);
    expect(asia.profit).equal(-10);
  });
```

В этот момент я могу задаться вопросом, действительно ли отрицательный спрос, приводящий к отрицательной прибыли, имеет какой-то смысл для данной предметной области? Не должен ли минимальный спрос быть нулевым? В этом случае, возможно, метод установки значения должен реагировать на отрицательный аргумент иначе — выдавая ошибку или устанавливая значение равным нулю. Это хорошие вопросы, и написание подобных тестов помогает мне подумать о том, как код должен реагировать на граничные случаи.

Подумайте о граничных условиях, которые могут быть неправильно обработаны, и сосредоточьте на них свои усилия.

Методы установки значений берут из полей в графическом интерфейсе пользователя строки, которые, хотя и ограничены только числами, все еще могут быть пустыми, а потому у меня должны быть тесты, которые гарантируют, что код реагирует на пустые строки так, как мне нужно.

```
describe('province'...
  it('empty string demand', function() {
    asia.demand = "";
    expect(asia.shortfall).NaN;
    expect(asia.profit).NaN;
  });
```

Обратите внимание, как я играю роль врага моего кода. Я активно размышляю о том, как мне его поломать. Я нахожу это не только полезным, но и веселым (это соответствует подлым наклонностям моей души).

Это интересно:

```
describe('string for producers', function() {
  it('', function() {
    const data = {
      name: "String producers",
      producers: "",
      demand: 30,
      price: 20
    };
    const prov = new Province(data);
    expect(prov.shortfall).equal(0);
  });
```

Это не приводит к простому сообщению об отказе, что дефицит не равен 0. Вот вывод на консоль:

```
'''''''''!

  9 passing (74ms)
  1 failing

  1) string for producers :
    TypeError: doc.producers.forEach is not a function
      at new Province (src/main.js:22:19)
      at Context.<anonymous> (src/tester.js:86:18)
```

Каркас Mocha рассматривает это как сбой, но многие тестовые среды различают данную ситуацию, которую они называют ошибкой, и регулярный сбой. Сбой (failure) означает шаг проверки, когда фактическое значение выходит за границы, ожидаемые инструкцией проверки. Но **ошибка** (error) — дело другое, это исключение, сгенерированное на более ранней стадии (в данном случае на этапе настройки). Она выглядит как исключение, которого авторы кода не ожидали, поэтому мы получаем хорошо знакомую программистам на JavaScript ошибку (“… не является функцией”).

Как код должен реагировать на такой случай? Один из подходов состоит в том, чтобы добавить некоторую обработку, которая дала бы лучшую реакцию на ошибку — либо выводя более информативное сообщение об ошибке, либо просто делая `producers` пустым массивом (возможно, с записью соответствующего сообщения в журнальный файл). Но могут быть и веские причины оставить все как есть. Возможно, входной объект создается доверенным источником, таким как другая часть той же кодовой базы. Включение большого количества проверок между модулями в одной и той же кодовой базе может привести к дублированию проверок, что вызовет больше проблем, чем пользы, особенно если они

дублируют проверки, выполняемые в другом месте. Но если этот входной объект поступает из внешнего источника, такого как запрос в кодировке JSON, проверки корректности необходимы, и они должны быть выполнены. В любом случае, написание тестов приводит к такого рода вопросам.

Если бы я писал подобные тесты перед рефакторингом, вероятно, я бы отказался от этого теста. Рефакторинг должен сохранять наблюдаемое поведение; данная же ошибка выходит за границы наблюдаемого поведения, поэтому мне не нужно беспокоиться, если выполняемый мною рефакторинг изменит реакцию кода на данное условие.

Если эта ошибка может привести к некорректным данным и сбою программы, который будет трудно отладить, можно воспользоваться рефакторингом *Введение утверждения* (с. 346). Я не добавляю тесты для выявления таких сбоев утверждений, поскольку они сами по себе являются разновидностью тестирования.

Опасения по поводу того, что тестирование не выявит все ошибки, не должно мешать писать тесты, которые выявят большинство ошибок.

Когда нужно остановиться? Наверняка вы не раз слышали, что тестирование не доказывает отсутствие ошибок в программе. Это верно, но это не мешает тестированию повысить скорость работы программиста. Было предложено немало правил, призванных гарантировать тестируемость всех мыслимых комбинаций данных. Ознакомиться с ними полезно, но не стоит принимать их слишком всерьез. Они могут уменьшить выгоду, приносимую тестированием, а кроме того, имеется опасность, что попытка написать слишком много тестов оставит в душе программиста такой след, что в итоге он вообще перестанет писать тесты. Необходимо сконцентрироваться на наиболее подозрительных местах. Изучите код и определите, где он становится сложным. Изучите функцию и установите области, где могут возникнуть ошибки. Тесты не выявят все ошибки, но в процессе рефакторинга помогут лучше понять программу и благодаря этому найти больше ошибок. Я уже говорил, что всегда начинаю рефакторинг с создания комплекта тестов; по мере продвижения вперед я обязательно пополняю этот комплект.

И многое другое…

Это все, что я собирался сказать в данной главе — в конце концов, эта книга о рефакторинге, а не о тестировании. Но тестирование — важная тема, как потому, что это необходимая основа для рефакторинга, так и потому, что это ценный инструмент сам по себе. Хотя я был рад видеть, что со времени написания первого издания этой книги рефакторинг стал повседневной практикой

программирования, я был еще более счастлив увидеть изменение отношения к тестированию. Ранее считавшееся обязанностью отдельной (зачастую подчиненной) группы, ныне тестирование становится все более первостепенной задачей любого достойного разработчика программного обеспечения. Архитектуры часто справедливо оцениваются по возможности их тестирования.

Типы тестов, которые я показал в этой главе, — это модульные тесты, разработанные для работы с небольшими фрагментами кода и быстрого выполнения. Они являются основой самотестируемого кода; большинство тестов в такой системе являются юнит-тестами. Существуют и другие разновидности тестов, сосредоточенные на интеграции между компонентами, использующие одновременно несколько уровней программного обеспечения, находящие проблемы с производительностью и т.д.

Как и большинство аспектов программирования, тестирование — действие итеративное. Если только вы не очень опытны или не очень удачливы, вы не сможете получить правильные тесты с первого раза. Я обнаружил, что постоянно работаю над набором тестов так же, как и над основным кодом. Естественно, это означает добавление новых тестов наряду с добавлением новых функциональных возможностей, но сюда входит и пересмотр существующих тестов. Достаточно ли они ясны? Не стоит ли их реорганизовать, чтобы было легче понять, что они делают? Есть ли у меня необходимые правильные тесты? Очень хорошая привычка — реагировать на ошибку путем написания теста, который четко ее выявляет. Только после создания такого теста я могу исправить ошибку. Имея тест, я знаю, остается ошибка в коде или она уже исправлена. Я также пытаюсь понять: не может ли эта ошибка и ее тест дать подсказку о других возможных проблемах?

Получив сообщение об ошибке, начните с написания модульного теста, который ее выявляет.

Распространенным вопросом является следующий — “Сколько тестов достаточно?” Некоторые разработчики рекомендуют использовать в качестве меры покрытие тестами [35], но анализ такого покрытия хорош только для определения непроверенных областей кода, но не для оценки качества набора тестов.

Лучшая мера для достаточно хорошего набора тестов — субъективная: насколько вы уверены, что если кто-то внесет дефект в код, то тесты это выявят? Это не тот показатель, который можно объективно проанализировать, и он не учитывает ложную уверенность, но цель самотестируемого кода — в получении такой уверенности. Если я смогу реорганизовать свой код и быть уверенным, что я не внес никаких ошибок, потому что мои тесты остаются “зелеными” — я могу быть доволен своим набором тестов.

Можно написать и слишком много тестов. Одним из признаков этого является то, что я трачу больше времени на изменение тестов, чем на тестируемый код, и чувствую, что возня с тестами не ускоряет, а замедляет мою работу. Но хотя избыточное тестирование и случается, оно является исчезающе редким по сравнению с тестированием недостаточным.