Приложение II
Фибоначчи

В ответ на просьбу одного из моих рецензентов я включил в книгу описание разработки функции вычисления последовательности Фибоначчи в стиле TDD. Некоторые утверждают, что именно этот пример раскрыл им глаза на механику работы TDD. Однако этот пример очень короток, к тому же в нем не используются многие важные приемы, применяемые в рамках TDD. По этой причине его невозможно использовать в качестве замены примеров, рассмотренных ранее в данной книге. Если, ознакомившись с рассмотренными ранее примерами, вы до сих пор не можете понять, как осуществляется разработка в стиле TDD, ознакомьтесь с данным материалом, возможно, он поможет вам прояснить ситуацию.

Первый тест показывает, что fib(0) = 0. Реализация возвращает константу.

```
public void testFibonacci() {
assertEquals(0, fib(0));
}
int fib(int n) {
return 0;
}
```

(Я использую класс TestCase как вместилище кода, так как мы разрабатываем всего одну функцию.)

Второй тест показывает, что fib(1) = 1.

```
public void testFibonacci() {
assertEquals(0, fib(0));
assertEquals(1, fib(1));
}
```

Я просто добавил еще один оператор assert() в тот же самый тестовый метод, так как не вижу особого смысла создавать новый

метод с именем testFibonacciOfOneIsOne.

Чтобы заставить тест работать, можно воспользоваться одним из нескольких методов. Я решаю использовать значение 0 как специальный случай:

```
int fib(int n) {
if (n == 0) return 0;
return 1;
}
```

Дублирование в тестирующем методе начинает действовать мне на нервы. По мере добавления новых тестов, дублирование будет только усугубляться. Давайте попробуем выделить общую структуру операторов assert(), для этого добавим в тест таблицу входных и ожидаемых значений функции fib():

```
public void testFibonacci() {
int cases[][] = {{0,0},{1,1}};
for (int i = 0; i < cases.length; i++)
assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Теперь добавление нового теста требует всего шесть нажатий на клавиши и никаких дополнительных строк:

```
public void testFibonacci() {

int cases[][] = {{0,0},{1,1},{2,1}};
for (int i = 0; i < cases.length; i++)
assertEquals(cases[i][1], fib(cases[i][0]));
}
```

Как это ни удивительно, но новый тест работает. Это происходит потому, что константа 1 также подходит и для входного значения 2. Переходим к следующему тесту:

```
public void testFibonacci() {
int cases[][] = {{0,0},{1,1},{2,1},{3,2}};
for (int i = 0; i < cases.length; i++)
assertEquals(cases[i][1], fib(cases[i][0]));
```

```
}
```

Ура! Наконец-то тест не сработал. Воспользуемся прежней стратегией (рассматриваем меньшие входные значения как специальные случаи):

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return 2;
}
```

Теперь мы готовы обобщить код. Мы написали 2, однако на самом деле мы имели в виду 1 + 1.

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return 1 + 1;
}
```

Первая единица в сумме – на самом деле fib(n-1):

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return fib(n-1) + 1;
}
```

Вторая единица в сумме – на самом деле fib(n-2):

```
int fib(int n) {
if (n == 0) return 0;
if (n <= 2) return 1;
return fib(n-1) + fib(n-2);
}
```

Теперь окончательно чистим код. Та же самая структура должна работать для fib(2), поэтому мы можем преобразовать второй условный оператор:

```
int fib(int n) {
if (n == 0) return 0;
if (n == 1) return 1;
return fib(n-1) + fib(n-2);
}
```

Это и есть функция вычисления последовательности Фибоначчи, целиком и полностью разработанная в рамках методики TDD.