



## Electronic Course Reserves

### Notice: Warning Concerning Copyright Restriction

The copyright Law of the United States (Titles 17, United State Code) governs the making of photocopies or other reproductions of copyright material.

Any electronic copy or copies, photocopies or any other type of Reproduction of this article or other distribution of this copyrighted Material may be an infringement of the Copyright Law. This copy is not to be “used for any other purposes other than private study, scholarship, or research” [section 107]. If a user makes or later uses any form of reproduction of this copyrighted work for purposes in excess of section 107, Fair Use, that user may be liable for copyright infringement.

This material is provided by the Ball State University Libraries. If you have questions concerning this material or are unable to access an electronic document, contact Course Reserve Services via email at [libreserves@bsu.edu](mailto:libreserves@bsu.edu) or by telephone at 765-285-5146.

### This material is on Electronic Reserve for:

|             |   |
|-------------|---|
| Instructor: | <u>GESTWICKI</u>                          |
| Course:     | <u>CS 222</u>                             |
| CREF:       | <u>Beck2003-Ch1MultiCurrencyMoney.pdf</u> |

### The work from which this copy is made:

|                           |  |
|---------------------------|--|
| Author Name:              | <u>Beck, K.</u>                            |
| Book or Periodical Title: | <u>Test-driven development: by example</u> |
| Vol.(iss), Date:          |  |
| Year Published:           | <u>2003</u>                                |
| Publisher & Location:     | <u>Addison-Wesley : Boston, MA</u>         |
| Pages in excerpt:         | <u>3-10</u>                                |
| Total pages in book or    |  |
| Publication:              | <u>220</u>                                 |

# PART I

## The Money Example

In Part I, we will develop typical model code driven completely by tests (except when we slip, purely for educational purposes). My goal is for you to see the rhythm of Test-Driven Development (TDD), which can be summed up as follows.

1. Quickly add a test.
2. Run all tests and see the new one fail.
3. Make a little change.
4. Run all tests and see them all succeed.
5. Refactor to remove duplication.

The surprises are likely to include

- How each test can cover a small increment of functionality
- How small and ugly the changes can be to make the new tests run
- How often the tests are run
- How many teensy-weensy steps make up the refactorings

# Chapter 1

---

## Multi-Currency Money

We'll start with the object that Ward created at WyCash, multi-currency money (refer to the Introduction). Suppose we have a report like this:

| Instrument | Shares | Price | Total |
|------------|--------|-------|-------|
| IBM        | 1000   | 25    | 25000 |
| GE         | 400    | 100   | 40000 |
| Total      |        |       | 65000 |

To make a multi-currency report, we need to add currencies:

| Instrument | Shares | Price   | Total     |
|------------|--------|---------|-----------|
| IBM        | 1000   | 25 USD  | 25000 USD |
| Novartis   | 400    | 150 CHF | 60000 CHF |
| Total      |        |         | 65000 USD |

We also need to specify exchange rates:

| From | To  | Rate |
|------|-----|------|
| CHF  | USD | 1.5  |

$\$5 + 10 \text{ CHF} = \$10$  if rate is 2:1  
 $\$5 * 2 = \$10$

What behavior will we need to produce the revised report? Put another way, what set of tests, when passed, will demonstrate the presence of code we are confident will compute the report correctly?

- We need to be able to add amounts in two different currencies and convert the result given a set of exchange rates.
- We need to be able to multiply an amount (price per share) by a number (number of shares) and receive an amount.

We'll make a to-do list to remind us what we need to do, to keep us focused, and to tell us when we are finished. When we start working on an item, we'll make it bold, **like this**. When we finish an item, we'll cross it off, ~~like this~~. When we think of another test to write, we'll add it to the list.

As you can see from the to-do list on the previous page, we'll work on multiplication first. So, what object do we need first? Trick question. We don't start with objects, we start with tests. (I keep having to remind myself of this, so I will pretend you are as dense as I am.)

Try again. What test do we need first? Looking at the list, that first test looks complicated. Start small or not at all. Multiplication, how hard could that be? We'll work on that first.

When we write a test, we imagine the perfect interface for our operation. We are telling ourselves a story about how the operation will look from the outside. Our story won't always come true, but it's better to start from the best-possible application program interface (API) and work backward than to make things complicated, ugly, and "realistic" from the get-go.

Here's a simple example of multiplication:

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

(I know, I know, public fields, side-effects, integers for monetary amounts, and all that. Small steps. We'll make a note of the stinkiness and move on. We have a failing test, and we want the bar to go to green as quickly as possible.)

\$5 + 10 CHF = \$10 if rate is 2:1

\$5 \* 2 = \$10

Make "amount" private

Dollar side-effects?

Money rounding?

The test we just typed in doesn't even compile. (I'll explain where and how we type it in later, when we talk more about the testing framework, JUnit.) That's easy enough to fix. What's the least we can do to get it to compile, even if it doesn't run? We have four compile errors:

- No class Dollar
- No constructor
- No method times(int)
- No field amount

Let's take them one at a time. (I always search for some numerical measure of progress.) We can get rid of one error by defining the class Dollar:

**Dollar**

```
class Dollar
```

One error down, three errors to go. Now we need the constructor, but it doesn't have to do anything just to get the test to compile:

**Dollar**

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

Two errors to go. We need a stub implementation of times(). Again we'll do the least work possible just to get the test to compile:

**Dollar**

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

Down to one error. Finally, we need an amount field:

**Dollar**

```
int amount;
```

Bingo! Now we can run the test and watch it fail, as shown in Figure 1.1.

You are seeing the dreaded red bar. Our testing framework (JUnit, in this case) has run the little snippet of code we started with, and noticed that although we expected "10" as a result, instead we saw "0". Sadness.

No, no. Failure is progress. Now we have a concrete measure of failure. That's better than just vaguely knowing we are failing. Our programming problem has been transformed from "give me multi-currency" to "make this test work, and then make the rest of the tests work." Much simpler. Much smaller scope for fear. We can make this test work.

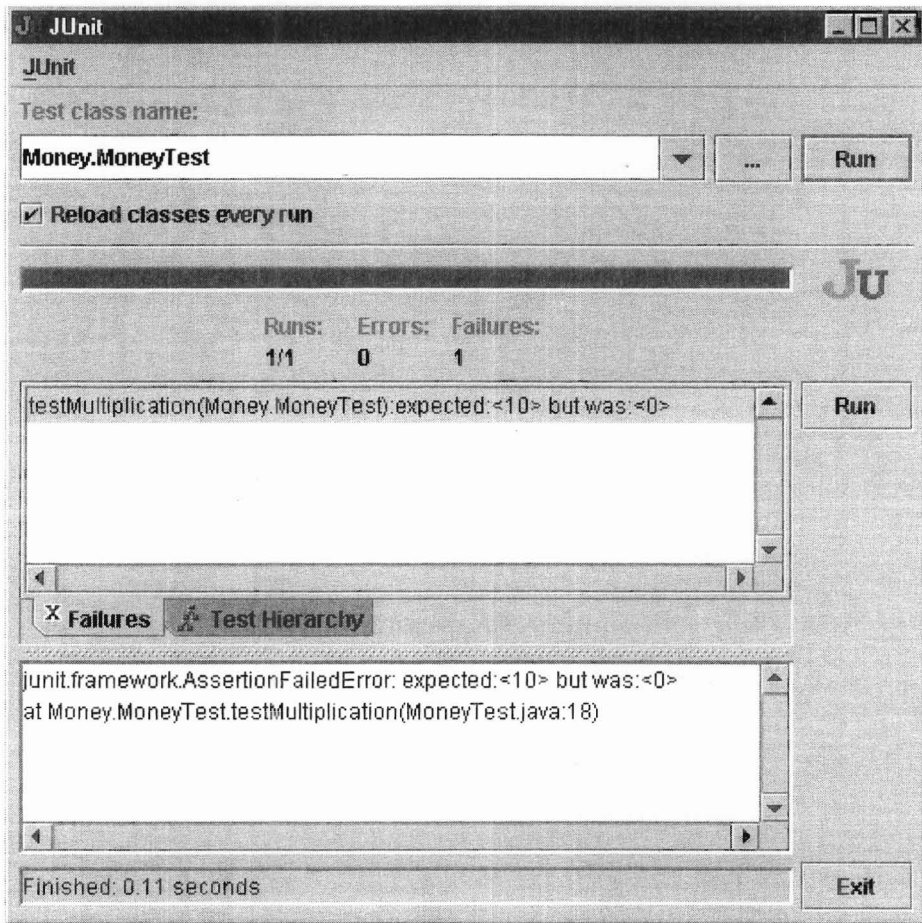


Figure 1.1 *Progress! The test fails*

You probably aren't going to like the solution, but the goal right now is not to get the perfect answer but to pass the test. We'll make our sacrifice at the altar of truth and beauty later.

Here's the smallest change I could imagine that would cause our test to pass:

Dollar

```
int amount= 10;
```

Figure 1.2 shows the result when the test is run again. Now we get the green bar, fabled in song and story

Oh joy, oh rapture! Not so fast, hacker boy (or girl). The cycle isn't complete. There are very few inputs in the world that will cause such a limited, such

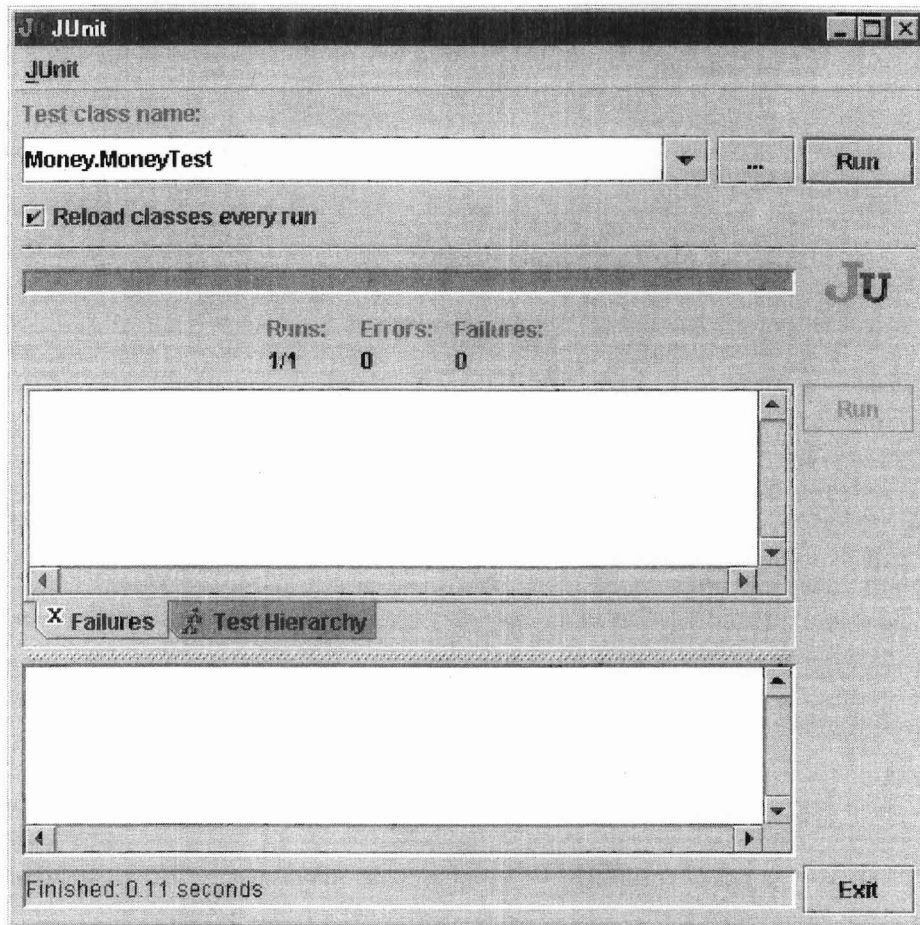


Figure 1.2 *The test runs*

a smelly, such a naïve implementation to pass. We need to generalize before we move on. Remember, the cycle is as follows.

1. Add a little test.
2. Run all tests and fail.
3. Make a little change.
4. Run the tests and succeed.
5. Refactor to remove duplication.

---

### Dependency and Duplication

Steve Freeman pointed out that the problem with the test and code as it sits is not duplication (which I have not yet pointed out to you, but I promise to as soon as this digression is over). The problem is the dependency between the code and the test—you can't change one without changing the other. Our goal is to be able to write another test that "makes sense" to us, without having to change the code, something that is not possible with the current implementation.

Dependency is the key problem in software development at all scales. If you have details of one vendor's implementation of SQL scattered throughout the code and you decide to change to another vendor, then you will discover that your code is dependent on the database vendor. You can't change the database without changing the code.

If dependency is the problem, duplication is the symptom. Duplication most often takes the form of duplicate logic—the same expression appearing in multiple places in the code. Objects are excellent for abstracting away the duplication of logic.

Unlike most problems in life, where eliminating the symptoms only makes the problem pop up elsewhere in worse form, eliminating duplication in programs eliminates dependency. That's why the second rule appears in TDD. By eliminating duplication before we go on to the next test, we maximize our chance of being able to get the next test running with one and only one change.

---

We have run items 1 through 4. Now we are ready to remove duplication. But where is the duplication? Usually you see duplication between two pieces of code, but here the duplication is between the data in the test and the data in the code. Don't see it? How about if we write the following:

**Dollar**

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

That 10 had to come from somewhere. We did the multiplication in our heads so fast we didn't even notice. The 5 and 2 are now in two places, and we must ruthlessly eliminate duplication before moving on. The rules say so.

There isn't a single step that will eliminate the 5 and the 2. But what if we move the setting of the amount from object initialization to the times() method?

**Dollar**

```
int amount;
```

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



The test still passes, the bar stays green. Happiness is still ours.

Do these steps seem too small to you? Remember, TDD is not about taking teeny-tiny steps, it's about *being able to* take teeny-tiny steps. Would I code day-to-day with steps this small? No. But when things get the least bit weird, I'm glad I can. Try teeny-tiny steps with an example of your own choosing. If you can make steps too small, you can certainly make steps the right size. If you only take larger steps, you'll never know if smaller steps are appropriate.

Defensiveness aside, where were we? Ah, yes, we were getting rid of duplication between the test code and the working code. Where can we get a 5? That was the value passed to the constructor, so if we save it in the amount variable,

**Dollar**

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

then we can use it in times():

**Dollar**

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

The value of the parameter “multiplier” is 2, so we can substitute the parameter for the constant:

**Dollar**

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

To demonstrate our thorough knowledge of Java syntax, we will want to use the \*= operator (which does, it must be said, reduce duplication):

**Dollar**

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

\$5 + 10 CHF = \$10 if rate is 2:1

~~\$5 \* 2 = \$10~~

Make “amount” private

Dollar side effects?

Money rounding?

We can now mark off the first test as done. Next we'll take care of those strange side effects. But first let's review. We've done the following:

- Made a list of the tests we knew we needed to have working
- Told a story with a snippet of code about how we wanted to view one operation
- Ignored the details of JUnit for the moment
- Made the test compile with stubs
- Made the test run by committing horrible sins
- Gradually generalized the working code, replacing constants with variables
- Added items to our to-do list rather than addressing them all at once