Introduction

Early one Friday, the boss came to Ward Cunningham to introduce him to Peter, a prospective customer for WyCash, the bond portfolio management system the company was selling. Peter said, "I'm very impressed with the functionality I see. However, I notice you only handle U.S. dollar denominated bonds. I'm starting a new bond fund, and my strategy requires that I handle bonds in different currencies." The boss turned to Ward, "Well, can we do it?"

Here is the nightmarish scenario for any software designer. You were cruising along happily and successfully with a set of assumptions. Suddenly, everything changed. And the nightmare wasn't just for Ward. The boss, an experienced hand at directing software development, wasn't sure what the answer was going to be.

A small team had developed WyCash over the course of a couple of years. The system was able to handle most of the varieties of fixed income securities commonly found on the U.S. market, and a few exotic new instruments, like Guaranteed Investment Contracts, that the competition couldn't handle.

WyCash had been developed all along using objects and an object database. The fundamental abstraction of computation, `Dollar`, had been outsourced at the beginning to a clever group of software engineers. The resulting object combined formatting and calculation responsibilities.

For the past six months, Ward and the rest of the team had been slowly divesting `Dollar` of its responsibilities. The Smalltalk numerical classes turned out to be just fine at calculation. All of the tricky code for rounding to three decimal digits got in the way of producing precise answers. As the answers became more precise, the complicated mechanisms in the testing framework for comparison within a certain tolerance were replaced by precise matching of expected and actual results.

Responsibility for formatting actually belonged in the user interface classes. As the tests were written at the level of the user interface classes, in particular the report framework,[1] these tests didn't have to change to accommodate this refinement. After six months of careful paring, the resulting `Dollar` didn't have much responsibility left.

[1] For more about the report framework, refer to c2.com/doc/oopsla91.html.

One of the most complicated algorithms in the system, weighted average, likewise had been undergoing a slow transformation. At one time, there had been many different variations of weighted average code scattered throughout the system. As the report framework coalesced from the primordial object soup, it was obvious that there could be one home for the algorithm, in `AveragedColumn`.

It was to `AveragedColumn` that Ward now turned. If weighted averages could be made multi-currency, then the rest of the system should be possible. At the heart of the algorithm was keeping a count of the money in the column. In fact, the algorithm had been abstracted enough to calculate the weighted average of any object that could act arithmetically. One could have weighted averages of dates, for example.

The weekend passed with the usual weekend activities. Monday morning the boss was back. "Can we do it?"

"Give me another day, and I'll tell you for sure."

`Dollar` acted like a counter in weighted average; therefore, in order to calculate in multiple currencies, they needed an object with a counter per currency, kind of like a polynomial. Instead of $3x^2$ and $4y^3$, however, the terms would be 15 USD and 200 CHF.

A quick experiment showed that it was possible to compute with a generic `Currency` object instead of a`Dollar`, and return a `PolyCurrency` when two unlike currencies were added together. The trick now was to make space for the new functionality without breaking anything that already worked.

What would happen if Ward just ran the tests?

After the addition of a few unimplemented operations to `Currency`, the bulk of the tests passed. By the end of the day, all of the tests were passing. Ward checked the code into the build and went to the boss. "We can do it," he said confidently.

Let's think a bit about this story. In two days, the potential market was multiplied several fold, multiplying the value of WyCash several fold. The ability to create so much business value so quickly was no accident, however. Several factors came into play.

- Method— Ward and the WyCash team needed to have constant experience growing the design of the system, little by little, so the mechanics of the transformation were well practiced.
- Motive— Ward and his team needed to understand clearly the business importance of making WyCash multi-currency, and to have the courage to start such a seemingly impossible task.
- Opportunity— The combination of comprehensive, confidence-generating tests; a well-factored program; and a programming language that made it possible to isolate design decisions meant that there were few sources of error, and those errors were easy to identify.

You can't control whether you ever get the motive to multiply the value of your project by spinning technical magic. Method and opportunity, on the other hand, are entirely under your control. Ward and his team created method and opportunity through a combination of superior talent, experience, and discipline. Does this mean that if you are not one of the ten best software engineers on the planet and don't have a wad of cash in the bank so you can tell your boss to take a hike, then you're going to take the time to do this right, that such moments are forever beyond your reach?

No. You absolutely can place your projects in a position for you to work magic, even if you are a software engineer with ordinary skills and you sometimes buckle under and take shortcuts when the pressure builds. Test-driven development is a set of techniques that any software engineer can follow, which encourages simple designs and test suites that inspire confidence. If you are a genius, you don't need these rules. If you are a dolt, the rules won't help. For the vast majority of us in between, following these two simple rules can lead us to work much more closely to our potential.

- Write a failing automated test before you write any code.
- Remove duplication.

How exactly to do this, the subtle gradations in applying these rules, and the lengths to which you can push these two simple rules are the topic of this book. We'll start with the object that Ward created in his moment of inspiration—multi-currency money.

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 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) {
}
```

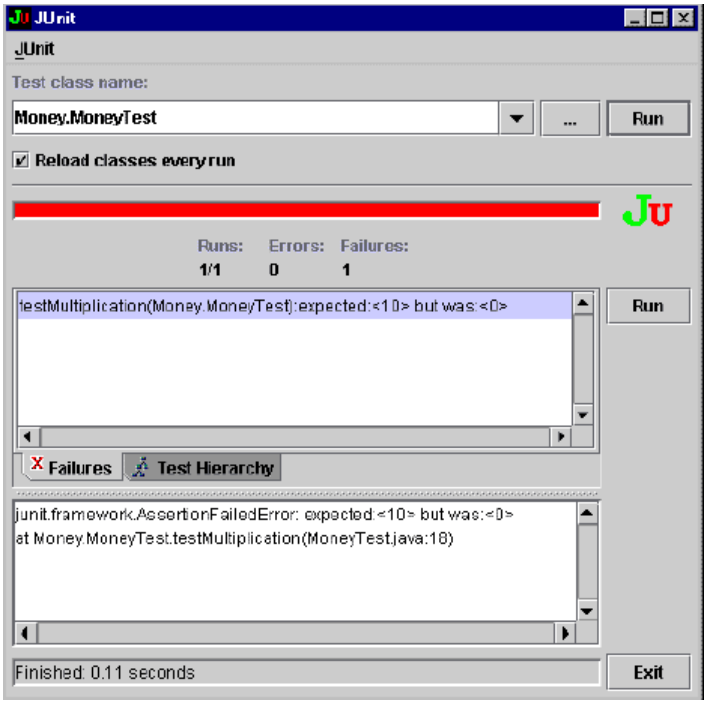
Down to one error. Finally, we need an `amount` field:

Dollar

```
int amount;
```

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

Figure 1.1. Progress! The test fails



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.

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.

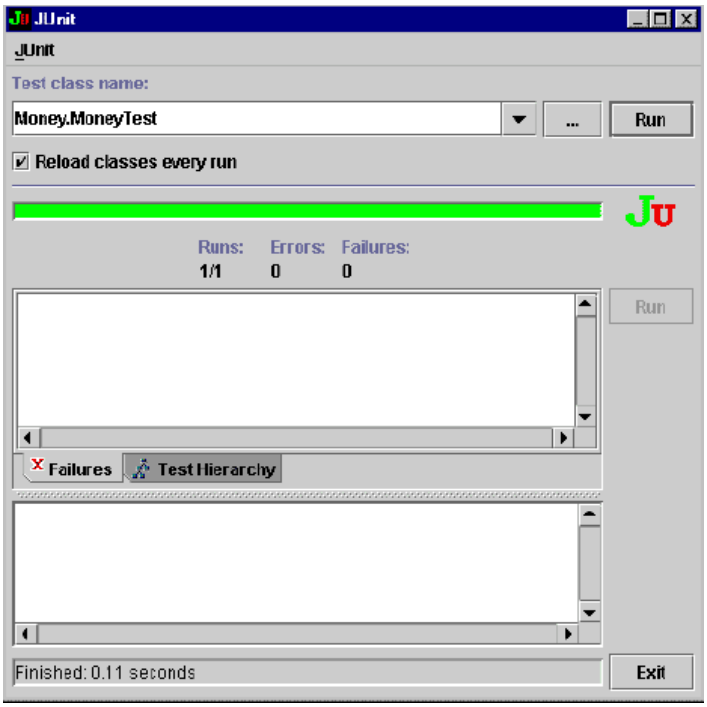
Here's the smallest change I could imagine that would cause our test to pass:

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

Figure 1.2. The test runs



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 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

Chapter 2. Degenerate Objects

The general TDD cycle goes as follows.

1. Write a test. Think about how you would like the operation in your mind to appear in your code. You are writing a story. Invent the interface you wish you had. Include all of the elements in the story that you imagine will be necessary to calculate the right answers.
2. Make it run. Quickly getting that bar to go to green dominates everything else. If a clean, simple solution is obvious, then type it in. If the clean, simple solution is obvious but it will take you a minute, then make a note of it and get back to the main problem, which is getting the bar green in seconds. This shift in aesthetics is hard for some experienced software engineers. They only know how to follow the rules of good engineering. Quick green excuses all sins. But only for a moment.
3. Make it right. Now that the system is behaving, put the sinful ways of the recent past behind you. Step back onto the straight and narrow path of software righteousness. Remove the duplication that you have introduced, and get to green quickly.

The goal is clean code that works (thanks to Ron Jeffries for this pithy summary). Clean code that works is out of the reach of even the best programmers some of the time, and out of the reach of most programmers (like me) most of the time. Divide and conquer, baby. First we'll solve the "that works" part of the problem. Then we'll solve the "clean code" part. This is the opposite of architecture-driven development, where you solve "clean code" first, then scramble around trying to integrate into the design the things you learn as you solve the "that works" problem.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
Dollar side effects?
Money rounding?

We got one test to work but in the process noticed something strange: when we perform an operation on a `Dollar`, the `Dollar` changes. I want to be able to write:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    five.times(2);
    assertEquals(10, five.amount);
    five.times(3);
    assertEquals(15, five.amount);
}
```

I can't imagine a clean way to get this test to work. After the first call to `times()`, five isn't five any more, it's really ten. If, however, we return a new object from `times()`, then we can multiply our original five bucks all day and never have it change. We are changing the interface of `Dollar` when we make this change, so we have to change the test. That's okay. Our guesses about the right interface are no more likely to be perfect than our guesses about the right implementation.

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

```
    product = five.times(3);
    assertEquals(15, product.amount);
}
```

The new test won't compile until we change the declaration of `Dollar.times()`:

Dollar

```
Dollar times(int multiplier) {
    amount *= multiplier;
    return null;
}
```

Now the test compiles, but it doesn't run. Progress! Making it run requires that we return a new `Dollar` with the correct amount:

Dollar

```
Dollar times(int multiplier) {
    return new Dollar(amount * multiplier);
}
```

$5 + 10 CHF = $10 if rate is 2:1 ~~$5 * 2 = $10~~ Make "amount" private ~~Dollar side effects?~~ Money rounding?

InChapter 1, when we made a test work we started with a bogus implementation and gradually made it real. Here, we typed in what we thought was the right implementation and prayed while the tests ran (short prayers, to be sure, because running the test takes a few milliseconds). Because we got lucky and the test ran, we can cross off another item.

Following are two of the three strategies I know for quickly getting to green:

- Fake It— Return a constant and gradually replace constants with variables until you have the real code.
- Use Obvious Implementation— Type in the real implementation.

When I use TDD in practice, I commonly shift between these two modes of implementation. When everything is going smoothly and I know what to type, I put in Obvious Implementation after Obvious Implementation (running the tests each time to ensure that what's obvious to me is still obvious to the computer). As soon as I get an unexpected red bar, I back up, shift to faking implementations, and refactor to the right code. When my confidence returns, I go back to Obvious Implementations.

There is a third style of TDD, Triangulation, which we will demonstrate in Chapter 3. However, to review, we

- Translated a design objection (side effects) into a test case that failed because of the objection
- Got the code to compile quickly with a stub implementation

- Made the test work by typing in what seemed to be the right code

The translation of a feeling (for example, disgust at side effects) into a test (for example, multiply the same`Dollar` twice) is a common theme of TDD. The longer I do this, the better able I am to translate my aesthetic judgments into tests. When I can do this, my design discussions become much more interesting. First we can talk about whether the system should work like *this* or like *that.* Once we decide on the correct behavior, we can talk about the best way of achieving that behavior. We can speculate about truth and beauty all we want over beers, but while we are programming we can leave airy-fairy discussions behind and talk cases.

Chapter 3. Equality for All

If I have an integer and I add 1 to it, I don't expect the original integer to change, I expect to use the new value. Objects usually don't behave that way. If I have a contract and I add one to its coverage, then the contract's coverage should change (yes, yes, subject to all sorts of interesting business rules which do *not* concern us here).

We can use objects as values, as we are using our `Dollar` now. The pattern for this is Value Object. One of the constraints on Value Objects is that the values of the instance variables of the object never change once they have been set in the constructor.

There is one huge advantage to using Value Objects: you don't have to worry about aliasing problems. Say I have one check and I set its amount to $5, and then I set another check's amount to the same $5. Some of the nastiest bugs in my career have come when changing the first check's value inadvertently changed the second check's value. This is aliasing.

When you have Value Objects, you needn't worry about aliasing. If I have $5, then I am guaranteed that it will always and forever be $5. If someone wants $7, then they will have to make an entirely new object.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
equals()

One implication of Value Objects is that all operations must return a new object, as we saw in Chapter 2. Another implication is that Value Objects should implement `equals()`, because one $5 is pretty much as good as another.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
equals()
hashCode()

If you use `Dollars` as the key to a hash table, then you have to implement `hashCode()` if you implement`equals()`. We'll put that on the to-do list, too, and get to it when it's a problem.

You aren't thinking about the implementation of `equals()`, are you? Good. Me neither. After snapping the back of my hand with a ruler, I'm thinking about how to test equality. First, $5 should equal $5:

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
}
```

The bar turns obligingly red. The fake implementation is just to return `true`:

Dollar

```
public boolean equals(Object object) {
    return true;
}
```

You and I both know that `true` is really "5 == 5", which is really "amount == 5", which is really "amount == dollar.amount". If I went through these steps, though, I wouldn't be able to demonstrate the third and most conservative implementation strategy: Triangulation.

If two receiving stations at a known distance from each other can both measure the direction of a radio signal, then there is enough information to calculate the range and bearing of the signal (if you remember more trigonometry than I do, anyway). This calculation is called Triangulation.

By analogy, when we triangulate, we only generalize code when we have two examples or more. We briefly ignore the duplication between test and model code. When the second example demands a more general solution, then and only then do we generalize.

So, to triangulate we need a second example. How about $5 != $6?

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
    assertFalse(new Dollar(5).equals(new Dollar(6)));
}
```

Now we need to generalize equality:

Dollar

```
public boolean equals(Object object) {
    Dollar dollar= (Dollar) object;
    return amount == dollar.amount;
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()

We could have used Triangulation to drive the generalization of `times()` also. If we had $5 x 2 = $10 and $5 x 3 = $15, then we would no longer have been able to return a constant.

Triangulation feels funny to me. I use it only when I am completely unsure of how to refactor. If I can see how to eliminate duplication between code and tests and create the general solution, then I just do it. Why would I need to write another test to give me permission to write what I probably could have written in the first place?

However, when the design thoughts just aren't coming, Triangulation provides a chance to think about the problem from a slightly different direction. What axes of variability are you trying to support in

your design? Make some of them vary, and the answer may become clearer.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object

So, equality is done for the moment. But what about comparing with null and comparing with other objects? These are commonly used operations but not necessary at the moment, so we'll add them to the to-do list.

Now that we have equality, we can directly compare `Dollars` to `Dollars`. That will let us make "amount" private, as all good instance variables should be. To review the above, we

- Noticed that our design pattern (Value Object) implied an operation
- Tested for that operation
- Implemented it simply
- Didn't refactor immediately, but instead tested further
- Refactored to capture the two cases at once

Chapter 4. Privacy

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
Make "amount" private
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object

Now that we have defined equality, we can use it to make our tests more "speaking." Conceptually, the operation `Dollar.times()` should return a `Dollar` whose value is the value of the receiver times the multiplier. Our test doesn't exactly say that:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(10, product.amount);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

We can rewrite the first assertion to compare `Dollars` to `Dollars`:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(new Dollar(10), product);
    product= five.times(3);
    assertEquals(15, product.amount);
}
```

That looks better, so we rewrite the second assertion, too:

```
public void testMultiplication() {
    Dollar five= new Dollar(5);
    Dollar product= five.times(2);
    assertEquals(new Dollar(10), product);
    product= five.times(3);
    assertEquals(new Dollar(15), product);
}
```

Now the temporary variable `product` isn't helping much, so we can inline it:

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

```
    assertEquals(new Dollar(15), five.times(3));
}
```

This test speaks to us more clearly, as if it were an assertion of truth, not a sequence of operations.

With these changes to the test, `Dollar` is now the only class using its `amount` instance variable, so we can make it private:

Dollar

```
private int amount;
```

```
$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
```

And we can cross another item off the to-do list. Notice that we have opened ourselves up to a risk. If the test for equality fails to accurately check that equality is working, then the test for multiplication could also fail to accurately check that multiplication is working. This is a risk that we actively manage in TDD. We aren't striving for perfection. By saying everything two ways—both as code and as tests—we hope to reduce our defects enough to move forward with confidence. From time to time our reasoning will fail us and a defect will slip through. When that happens, we learn our lesson about the test we should have written and move on. The rest of the time we go forward boldly under our bravely flapping green bar (my bar doesn't actually flap, but one can dream.)

To review, we

- Used functionality just developed to improve a test
- Noticed that if two tests fail at once we're sunk
- Proceeded in spite of the risk
- Used new functionality in the object under test to reduce coupling between the tests and the code

Chapter 5. Franc-ly Speaking

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
5 CHF * 2 = 10 CHF

How are we going to approach the first test, the most interesting test, on that list? It still seems to be a big leap. I'm not sure I can write a test that I can implement in one little step. A prerequisite seems to be having an object like `Dollar`, but to represent francs. If we can get the object `Franc` to work the way that the object `Dollar` works now, we'll be closer to being able to write and run the mixed addition test.

We can copy and edit the `Dollar` test:

```
public void testFrancMultiplication() {
    Franc five= new Franc(5);
    assertEquals(newFranc(10), five.times(2));
    assertEquals(newFranc(15), five.times(3));
}
```

(Aren't you glad we simplified the test in Chapter 4? That has made our job here easier. Isn't it amazing how often things work out like this in books? I didn't actually plan it that way this time, but I won't make promises for the future.)

What short step will get us to a green bar? Copying the `Dollar` code and replacing `Dollar` with `Franc`.

Stop. Hold on. I can hear the aesthetically inclined among you sneering and spitting. Copy-and-paste reuse? The death of abstraction? The killer of clean design?

If you're upset, take a cleansing breath. In through the nose … hold it 1, 2, 3 … out through the mouth. There. Remember, our cycle has different phases (they go by quickly, often in seconds, but they are phases.):

1. Write a test.
2. Make it compile.
3. Run it to see that it fails.
4. Make it run.
5. Remove duplication.

The different phases have different purposes. They call for different styles of solution, different aesthetic viewpoints. The first three phases need to go by quickly, so we get to a known state with the new functionality. We can commit any number of sins to get there, because speed trumps design, just for that brief moment.

Now I'm worried. I've given you a license to abandon all the principles of good design. Off you go to your teams—"Kent says all that design stuff doesn't matter." Halt. The cycle is not complete. A four-legged Aeron chair falls over. The first four steps of the cycle won't work without the fifth. Good design at good times. Make it run, make it right.

There, I feel better. Now I'm sure you won't show anyone except your partner your code until you've removed the duplication. Where were we? Ah, yes. Violating all the tenets of good design in the interest of speed (penance for our sin will occupy the next several chapters).

Franc

```
class Franc {
    private int amount;

    Franc(int amount) {
        this.amount = amount;
    }

    Franc times(int multiplier) {
        return new Franc(amount * multiplier);
    }

    public boolean equals(Object object) {
        Franc franc = (Franc) object;
        return amount == franc.amount;
    }
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
Common equals
Common times

Because the step to running code was so short, we were even able to skip the "make it compile" step.

Now we have duplication galore, and we have to eliminate it before writing our next test. We'll start by generalizing `equals()`. However, we can cross an item off our to-do list, even though we have to add two more. Reviewing, we

- Couldn't tackle a big test, so we invented a small test that represented progress
- Wrote the test by shamelessly duplicating and editing
- Even worse, made the test work by copying and editing model code wholesale
- Promised ourselves we wouldn't go home until the duplication was gone

Chapter 6. Equality for All, Redux

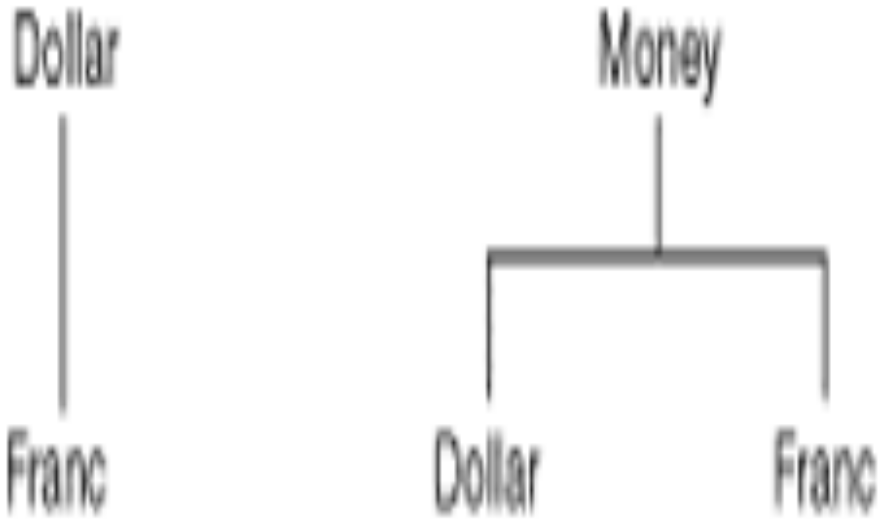
$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
Common equals
Common times

There is a fabulous sequence in *Crossing to Safety* in which author Wallace Stegner describes a character's workshop. Every item is perfectly in place, the floor is spotless, all is order and cleanliness. The character, however, has never made anything. "Preparing has been his life's work. He prepares, then he cleans up." (This is also the book whose ending sent me audibly blubbering in business class on a trans-Atlantic 747. Read with caution.)

We avoided this trap in Chapter 5. We actually got a new test case working. But we sinned mightily in copying and pasting tons of code in order to do it quickly. Now it is time to clean up.

One possibility is to make one of our classes extend the other. I tried it, and it hardly saves any code at all. Instead, we are going to find a common superclass for the two classes, as shown in Figure 6.1. (I tried this already, too, and it works out great, although it will take a while.)

Figure 6.1. A common superclass for two classes



What if we had a `Money` class to capture the common equals code? We can start small:

Money

```
class Money
```

All of the tests still run—not that we could possibly have broken anything, but it's a good time to run the tests anyway. If `Dollar` extends `Money`, that can't possibly break anything.

Dollar

```
class Dollar extends Money {
   private int amount;
}
```

Can it? No, the tests still all run. Now we can move the `amount` instance variable up to `Money`:

Money

```
class Money {
   protected int amount;
}
```

Dollar

```
class Dollar extends Money {
}
```

The visibility has to change from private to protected so that the subclass can still see it. (Had we wanted to go even slower, we could have declared the field in `Money` in one step and then removed it from `Dollar` in a second step. I'm feeling bold.)

Now we can work on getting the `equals()` code ready to move up. First we change the declaration of the temporary variable:

Dollar

```
public boolean equals(Object object) {
   Money dollar = (Dollar) object;
   return amount == dollar.amount;
}
```

All the tests still run. Now we change the cast:

Dollar

```
public boolean equals(Object object) {
   Money dollar = (Money) object;
   return amount == dollar.amount;
}
```

To be communicative, we should also change the name of the temporary variable:

Dollar

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

Now we can move it from `Dollar` to `Money`:

Money

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

Now we need to eliminate `Franc.equals()`. First we notice that the tests for equality don't cover comparing `Francs` to `Francs`. Our sins in copying code are catching up with us. Before we change the code, we'll write the tests that should have been there in the first place.

You will often be implementing TDD in code that doesn't have adequate tests (at least for the next decade or so). When you don't have enough tests, you are bound to come across refactorings that aren't supported by tests. You could make a refactoring mistake and the tests would all still run. What do you do?

Write the tests you wish you had. If you don't, you will eventually break something while refactoring. Then you'll get bad feelings about refactoring and stop doing it so much. Then your design will deteriorate. You'll be fired. Your dog will leave you. You will stop paying attention to your nutrition. Your teeth will go bad. So, to keep your teeth healthy, retroactively test before refactoring.

Fortunately, here the tests are easy to write. We just copy the tests for `Dollar`:

```
public void testEquality() {

    assertTrue(newDollar(5).equals(newDollar(5)));
    assertFalse(newDollar(5).equals(newDollar(6)));
    assertTrue(newFranc(5).equals(newFranc(5)));
    assertFalse(newFranc(5).equals(newFranc(6)));
}
```

More duplication, two lines more! We'll atone for these sins, too.

Tests in place, we can have `Franc` extend `Money`:

Franc

```
classFranc extends Money {
    private int amount;
}
```

We can delete `Franc`'s field `amount` in favor of the one in `Money`:

Franc

```
class Franc extends Money {
}
```

`Franc.equals()` is almost the same as `Money.equals()`. If we make them precisely the same, then we can delete the implementation in `Franc` without changing the meaning of the program. First we change the declaration of the temporary variable:

Franc

```
public boolean equals(Object object) {
    Money franc= (Franc) object;
    return amount == franc.amount;
}
```

Then we change the cast:

Franc

```
public boolean equals(Object object) {
    Money franc= (Money) object;
    return amount == franc.amount;
}
```

Do we really have to change the name of the temporary variable to match the superclass? I'll leave it up to your conscience.... Okay, we'll do it:

Franc

```
public boolean equals(Object object) {
    Money money= (Money) object;
    return amount == money.amount;
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~

hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
Compare Francs with Dollars

Now there is no difference between `Franc.equals()` and `Money.equals()`, so we delete the redundant implementation in `Franc`. And run the tests. They run.

What happens when we compare `Francs` with `Dollars`? We'll get to that in Chapter 7. Reviewing what we did here, we

- Stepwise moved common code from one class (`Dollar`) to a superclass (`Money`)
- Made a second class (`Franc`) a subclass also
- Reconciled two implementations—`equals()`—before eliminating the redundant one

PREVIOUS NEXT

Chapter 7. Apples and Oranges

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
Compare Francs with Dollars

The thought struck us at the end of Chapter 6: what happens when we compare `Francs` with `Dollars`? We dutifully turned our dreadful thought into an item on our to-do list. But we just can't get it out of our heads. What does happen?

```
public void testEquality() {
    assertTrue(new Dollar(5).equals(new Dollar(5)));
    assertFalse(new Dollar(5).equals(new Dollar(6)));
    assertTrue(new Franc(5).equals(new Franc(5)));
    assertFalse(new Franc(5).equals(new Franc(6)));
    assertFalse(new Franc(5).equals(new Dollar(5)));
}
```

It fails. `Dollars` are `Francs`. Before you Swiss shoppers get all excited, let's try to fix the code. The equality code needs to check that it isn't comparing `Dollars` with `Francs`. We can do this right now by comparing the class of the two objects—two `Moneys` are equal only if their amounts and classes are equal.

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

Using classes this way in model code is a bit smelly. We would like to use a criterion that makes sense in the domain of finance, not in the domain of Java objects. But we don't currently have anything like a currency, and this doesn't seem to be sufficient reason to introduce one, so this will have to do for the moment.

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Compare Francs to Dollars~~
Currency?

Now we really need to get rid of the common `times()` code, so we can get to mixed currency arithmetic. Before we do, however, we can review our grand accomplishments for this chapter. We

- Took an objection that was bothering us and turned it into a test
- Made the test run a reasonable, but not perfect way—`getClass()`
- Decided not to introduce more design until we had a better motivation

Chapter 8. Makin' Objects

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Compare Francs to Dollars~~
Currency?

The two implementations of `times()` are remarkably similar:

Franc

```
Franc times(int multiplier) {
   return new Franc(amount * multiplier);
}
```

Dollar

```
Dollar times(int multiplier) {
   return new Dollar(amount * multiplier);
}
```

We can take a step toward reconciling them by making them both return a `Money`:

Franc

```
Money times(int multiplier) {
   return new Franc(amount * multiplier);
}
```

Dollar

```
Money times(int multiplier) {
   return new Dollar(amount * multiplier);
}
```

The next step forward is not obvious. The two subclasses of `Money` aren't doing enough work to justify their existence, so we would like to eliminate them. But we can't do it with one big step, because that wouldn't make a very effective demonstration of TDD.

Okay, we would be one step closer to eliminating the subclasses if there were fewer references to the subclasses directly. We can introduce a factory method in `Money` that returns a `Dollar`. We would use it like this:

```
public void testMultiplication() {
    Dollar five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

The implementation creates and returns a `Dollar`:

Money

```
staticDollar dollar(intamount) {
    return new Dollar(amount);
}
```

But we want references to `Dollars` to disappear, so we need to change the declaration in the test:

```
public void testMultiplication() {
    Money five = Money.dollar(5);
    assertEquals(new Dollar(10), five.times(2));
    assertEquals(new Dollar(15), five.times(3));
}
```

Our compiler politely informs us that `times()` is not defined for `Money`. We aren't ready to implement it just yet, so we make `Money` abstract (I suppose we should have done that to begin with?) and declare`Money.times()`:

Money

```
abstractclass Money
abstractMoney times(intmultiplier);
```

Now we can change the declaration of the factory method:

Money

```
static Money dollar(intamount) {
    return new Dollar(amount);
}
```

The tests all run, so at least we haven't broken anything. We can now use our factory method everywhere in the tests:

```
public void testMultiplication() {
    Money five = Money.dollar(5);
```

```
    assertEquals(Money.dollar(10), five.times(2));
    assertEquals(Money.dollar(15), five.times(3));
}
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(newFranc(5).equals(newFranc(5)));
    assertFalse(newFranc(5).equals(newFranc(6)));
    assertFalse(newFranc(5).equals(Money.dollar(5)));
}
```

We are now in a slightly better position than before. No client code knows that there is a subclass called`Dollar`. By decoupling the tests from the existence of the subclasses, we have given ourselves freedom to change inheritance without affecting any model code.

Before we go blindly changing the `testFrancMultiplication`, we notice that it isn't testing any logic that isn't tested by the test for `Dollar` multiplication. If we delete the test, will we lose any confidence in the code? Still a little, so we leave it there. But it's suspicious.

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}

public void testFrancMultiplication() {
    Money five = Money.franc(5);
    assertEquals(Money.franc(10), five.times(2));
    assertEquals(Money.franc(15), five.times(3));
}
```

The implementation is just like `Money.dollar()`:

Money

```
staticMoney franc(intamount) {
    return new Franc(amount);
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Compare Francs to Dollars~~
Currency?
Delete testFrancMultiplication?

Next we'll get rid of the duplication of `times()`. For now, to review, we

- Took a step toward eliminating duplication by reconciling the signatures of two variants of the same method—`times()`
- Moved at least a declaration of the method to the common superclass
- Decoupled test code from the existence of concrete subclasses by introducing factory methods
- Noticed that when the subclasses disappear some tests will be redundant, but took no action

Chapter 9. Times We're Livin' In

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Compare Francs to Dollars~~
Currency?
Delete testFrancMultiplication?

What is there on our to-do list that might help us to eliminate those pesky useless subclasses? What would happen if we introduced the notion of currency?

How do we want to implement currencies at the moment? I blew it, again. Before the ruler comes out, I'll rephrase: How do we want to test for currencies at the moment? There. Knuckles saved, for the moment.

We may want to have complicated objects representing currencies, with flyweight factories to ensure we create no more objects than we really need. But for the moment, strings will do:

```
public void testCurrency() {
    assertEquals("USD", Money.dollar(1).currency());
    assertEquals("CHF", Money.franc(1).currency());
}
```

First we declare `currency()` in `Money`:

Money

```
abstract String currency();
```

Then we implement it in both subclasses:

Franc

```
String currency() {
    return "CHF";
}
```

Dollar

```
String currency() {
   return "USD";
}
```

We want the same implementation to suffice for both classes. We could store the currency in an instance variable and just return the variable. (I'll start going a little faster with the refactorings in the interest of time. If I go too fast, please tell me to slow down. Oh wait, this is a book—perhaps I just won't speed up much.)

Franc

```
privateString currency;
Franc(intamount) {
   this.amount = amount;
   currency = "CHF";
}
String currency() {
   return currency;
}
```

We can do the same with Dollar:

Dollar

```
privateString currency;
Dollar(intamount) {
   this.amount = amount;
   currency = "USD";
}
String currency() {
   return currency;
}
```

Now we can push up the declaration of the variable and the implementation of `currency()`, because they are identical:

Money

```
protectedString currency;
String currency() {
   return currency;
}
```

If we move the constant strings `"USD"` and `"CHF"` to the static factory methods, then the two constructors will be identical and we can create a common implementation.

First we'll add a parameter to the constructor:

Franc

```
Franc(intamount, String currency) {
```

```
    this.amount = amount;
    this.currency = "CHF";
}
```

This breaks the two callers of the constructor:

Money

```
staticMoney franc(intamount) {
    return new Franc(amount, null);
}
```

Franc

```
Money times(intmultiplier) {
    return new Franc(amount * multiplier, null);
}
```

Wait a minute! Why is `Franc.times()` calling the constructor instead of the factory method? Do we want to make this change now, or will we wait? The dogmatic answer is that we'll wait, not interrupting what we're doing. The answer in my practice is that I will entertain a brief interruption, but only a brief one, and I will never interrupt an interruption (Jim Coplien taught me this rule). To be realistic, we'll clean up `times()` before proceeding:

Franc

```
Money times(intmultiplier) {
    return Money.franc(amount * multiplier);
}
```

Now the factory method can pass `"CHF"`:

Money

```
staticMoney franc(intamount) {
    return new Franc(amount, "CHF");
}
```

And finally we can assign the parameter to the instance variable:

Franc

```
Franc(intamount, String currency) {
    this.amount = amount;
    this.currency = currency;
}
```

I'm feeling defensive again about taking such teeny-tiny steps. Am I recommending that you actually work this way? No. I'm recommending that you be able to work this way. What I did just now was to work in larger steps and make a stupid mistake halfway through. I unwound a minute's-worth of

changes, shifted to a lower gear, and did it over with little steps. I'm feeling better now, so we'll see if we can make the analogous change to `Dollar` in one swell foop:

Money

```
staticMoney dollar(intamount) {
    return new Dollar(amount, "USD");
}
```

Dollar

```
Dollar(intamount, String currency) {
    this.amount = amount;
    this.currency = currency;
}
Money times(intmultiplier) {
    return Money.dollar(amount * multiplier);
}
```

And it worked first time. Whew!

This is the kind of tuning you will be doing constantly with TDD. Are the teeny-tiny steps feeling restrictive? Take bigger steps. Are you feeling a little unsure? Take smaller steps. TDD is a steering process—a little this way, a little that way. There is no right step size, now and forever.

The two constructors are now identical, so we can push up the implementation:

Money

```
Money(intamount, String currency) {
    this.amount = amount;
    this.currency = currency;
}
```

Franc

```
Franc(intamount, String currency) {
    super(amount, currency);
}
```

Dollar

```
Dollar(intamount, String currency) {
    super(amount, currency);
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Compare Francs to Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

We're almost ready to push up the implementation of `times()` and eliminate the subclasses, but first, to review, we

- Were a little stuck on big design ideas, so we worked on something small we noticed earlier
- Reconciled the two constructors by moving the variation to the caller (the factory method)
- Interrupted a refactoring for a little twist, using the factory method in `times()`
- Repeated an analogous refactoring (doing to `Dollar` what we just did to `Franc`) in one big step
- Pushed up the identical constructors

Chapter 10. Interesting Times

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
Common times
~~Compare Francs to Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

When we are done with this chapter, we will have a single class to represent `Money`. The two implementations of `times()` are close, but not identical:

Franc

```
Money times(int multiplier) {
   return Money.franc(amount * multiplier);
}
```

Dollar

```
Money times(int multiplier) {
   return Money.dollar(amount * multiplier);
}
```

There's no obvious way to make them identical. Sometimes you have to go backward to go forward, a little like solving a Rubik's Cube. What happens if we inline the factory methods? (I know, I know, we just called the factory method for the first time just one chapter ago. Frustrating, isn't it?)

Franc

```
Money times(int multiplier) {
   return new Franc(amount * multiplier, "CHF");
}
```

Dollar

```
Money times(int multiplier) {
```

```
    return new Dollar(amount * multiplier, "USD");
}
```

In `Franc`, however, we know that the currency instance variable is always `"CHF"`, so we can write:

Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier, currency);
}
```

That works. The same trick works in `Dollar`:

Dollar

```
Money times(int multiplier) {
    return new Dollar(amount * multiplier, currency);
}
```

We're almost there. Does it really matter whether we have a `Franc` or a `Money`? We could carefully reason about this given our knowledge of the system, but we have clean code and we have tests that give us confidence that the clean code works. Rather than apply minutes of suspect reasoning, we can just ask the computer by making the change and running the tests. In teaching TDD, I see this situation all the time—excellent software engineers spending 5 to 10 minutes reasoning about a question that the computer could answer in 15 seconds. Without the tests you have no choice, you have to reason. With the tests you can decide whether an experiment would answer the question faster. Sometimes you should just ask the computer.

To run our experiment, we change `Franc.times()` to return a `Money`:

Franc

```
Money times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

The compiler tells us that `Money` must be a concrete class:

Money

```
class Money
Money times(int amount) {
    return null;
}
```

And we get a red bar. The error message says, "expected:<Money.Franc@31aebf> but was: <Money.Money@478a43>". Not as helpful as we perhaps would like. We can define `toString()` to give us a better error message:

Money

```
public String toString() {
    return amount + " " + currency;
}
```

Whoa! Code without a test? Can you do that? We could certainly have written a test for `toString()` before we coded it. However,

- We are about to see the results on the screen.
- Because `toString()` is used only for debug output, the risk of it failing is low.
- We already have a red bar, and we'd prefer not to write a test when we have a red bar.

Exception noted.

Now the error message says: `"expected:<10 CHF> but was:<10 CHF>"`. That's a little better, but still confusing. We have the right data in the answer, but the class is wrong—`Money` instead of `Franc`. The problem is in our implementation of `equals()`:

Money

```
public boolean equals(Object object) {
    Money money = (Money) object;
    return amount == money.amount
        && getClass().equals(money.getClass());
}
```

We really should be checking to see that the currencies are the same, not that the classes are the same.

We'd prefer not to write a test when we have a red bar. But we are about to change real model code, and we can't change model code without a test. The conservative course is to back out the change that caused the red bar so we're back to green. Then we can change the test for `equals()`, fix the implementation, and retry the original change.

This time, we'll be conservative. (Sometimes I plough ahead and write a test on a red, but not while the children are awake.)

Franc

```
Money times(int multiplier) {
    return new Franc(amount * multiplier, currency);
}
```

That gets us back to green. The situation that we had was a `Franc(10,"CHF")` and a `Money(10, "CHF")` that were reported to be not equal, even though we would like them to be equal. We can use exactly this for our test:

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

It fails, as expected. The `equals()` code should compare currencies, not classes:

Money

```
public boolean equals(Object object) {
   Money money = (Money) object;
   return amount == money.amount
      && currency().equals(money.currency());
}
```

Now we can return a `Money` from `Franc.times()` and still pass the tests:

Franc

```
Money times(int multiplier) {
   return new Money(amount * multiplier, currency);
}
```

Will the same work for `Dollar.times()`?

Dollar

```
Money times(int multiplier) {
   return new Money(amount * multiplier, currency);
}
```

Yes! Now the two implementations are identical, so we can push them up.

Money

```
Money times(int multiplier) {
   return new Money(amount * multiplier, currency);
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
~~Common times~~
~~Compare Francs to Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

Multiplication in place, we are ready to eliminate the stupid subclasses. To review, we:

- Reconciled two methods—`times()`—by first inlining the methods they called and then replacing constants with variables
- Wrote a `toString()` without a test just to help us debug
- Tried a change (returning `Money` instead of `Franc`) and let the tests tell us whether it worked
- Backed out an experiment and wrote another test. Making the test work made the experiment work

Chapter 11. The Root of All Evil

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
Dollar/Franc duplication
~~Common equals~~
~~Common times~~
~~Compare Francs to Dollars~~
~~Currency?~~
Delete testFrancMultiplication?

The two subclasses, `Dollar` and `Franc`, have only their constructors. But because a constructor is not reason enough to have a subclass, we want to delete the subclasses.

We can replace references to the subclasses with references to the superclass without changing the meaning of the code. First `Franc`:

Franc

```
static Money franc(int amount){
    return new Money(amount, "CHF");
}
```

Then `Dollar`:

Dollar

```
static Money dollar(int amount) {
    return new Money(amount, "USD");
}
```

Now there are no references to `Dollar`, so we can delete it. `Franc`, on the other hand, still has one reference, in the test we just wrote.

```
public void testDifferentClassEquality() {
    assertTrue(new Money(10, "CHF").equals(new Franc(10, "CHF")));
}
```

Is equality covered well enough elsewhere that we can delete this test? Looking at the other equality test,

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertTrue(Money.franc(5).equals(Money.franc(5)));
    assertFalse(Money.franc(5).equals(Money.franc(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

it looks as though we have the cases for equality well covered—too well covered, actually. We can delete the third and fourth assertions because they duplicate the exercise of the first and second assertions:

```
public void testEquality() {
    assertTrue(Money.dollar(5).equals(Money.dollar(5)));
    assertFalse(Money.dollar(5).equals(Money.dollar(6)));
    assertFalse(Money.franc(5).equals(Money.dollar(5)));
}
```

$5 + 10 CHF = $10 if rate is 2:1
~~$5 * 2 = $10~~
~~Make "amount" private~~
~~Dollar side effects?~~
Money rounding?
~~equals()~~
hashCode()
Equal null
Equal object
~~5 CHF * 2 = 10 CHF~~
~~Dollar/Franc duplication~~
~~Common equals~~
~~Common times~~
~~Compare Francs to Dollars~~
~~Currency?~~
~~Delete testFrancMultiplication?~~

The test we wrote forcing us to compare currencies instead of classes makes sense only if there are multiple classes. Because we are trying to eliminate the `Franc` class, a test to ensure that the system works if there is a `Franc` class is a burden, not a help. Away `testDifferentClassEquality()` goes, and `Franc` goes with it.

Similarly, there are separate tests for dollar and franc multiplication. Looking at the code, we can see that there is no difference in the logic at the moment based on the currency (there was a difference when there were two classes). We can delete `testFrancMultiplication()` without losing any confidence in the behavior of the system.

Single class in place, we are ready to tackle addition. First, to review, we

- Finished gutting subclasses and deleted them
- Eliminated tests that made sense with the old code structure but were redundant with the new code structure

Chapter 12. Addition, Finally

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

It's a new day, and our to-do list has become a bit cluttered, so we'll copy the pending items to a fresh list. (I like physically copying to-do items to a new list. If there are lots of little items, I tend just to take care of them rather than copy them. Little stuff that otherwise might build up gets taken care of just because I'm lazy. Play to your strengths.)

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

I'm not sure how to write the story of the whole addition, so we'll start with a simpler example: $5 + $5 = $10.

```
public void testSimpleAddition() {
    Money sum= Money.dollar(5).plus(Money.dollar(5));
    assertEquals(Money.dollar(10), sum);
}
```

We could fake the implementation by just returning `"Money.dollar(10)"`, but the implementation seems obvious. We'll try:

Money

```
Money plus(Money addend) {
    return new Money(amount + addend.amount, currency);
}
```

(In general, I will begin speeding up the implementations to save trees and keep your interest. Where the design isn't obvious, I will still fake the implementation and refactor. I hope you will see through this how TDD gives you control over the size of steps.)

Having said that I was going to go much faster, I will immediately go much slower—not in getting the tests working, but in writing the test itself. There are times and tests that call for careful thought. How are we going to represent multi-currency arithmetic? This is one of those times for careful thought.

The most difficult design constraint is that we want most of the code in the system to be unaware that it potentially is dealing with multiple currencies. One possible strategy is to immediately convert all money values into a reference currency. (I'll let you guess which reference currency American imperialist programmers generally choose.) However, this doesn't allow exchange rates to vary easily.

Instead we would like a solution that lets us conveniently represent multiple exchange rates, and still allows most arithmetic-like expressions to look like, well, arithmetic.

Objects to the rescue. When the object we have doesn't behave the way we want it to, we make another object with the same external protocol (an imposter) but a different implementation.

This probably sounds a bit like magic. How do we know to think of creating an imposter here? I won't kid you—there is no formula for flashes of design insight. Ward Cunningham came up with the "trick" a decade ago, and I haven't seen it independently duplicated yet, so it must be a pretty tricky trick. TDD can't guarantee that we will have flashes of insight at the right moment. However, confidence-giving tests and carefully factored code give us preparation for insight, and preparation for applying that insight when it comes.

The solution is to create an object that acts like a `Money` but represents the sum of two `Money`s. I've tried several different metaphors to explain this idea. One is to treat the sum like a *wallet:* you can have several different notes of different denominations and currencies in the same wallet.

Another metaphor is *expression,* as in "(2 + 3) * 5", or in our case "($2 + 3 CHF) * 5". A `Money` is the atomic form of an expression. Operations result in `Expression`s, one of which will be a `Sum`. Once the operation (such as adding up the value of a portfolio) is complete, the resulting `Expression` can be reduced back to a single currency given a set of exchange rates.

Applying this metaphor to our test, we know what we end up with:

```
public void testSimpleAddition() {
    ...
    assertEquals(Money.dollar(10), reduced);
}
```

The `reduced` `Expression` is created by applying exchange rates to an `Expression`. What in the real world applies exchange rates? A *bank.* We would like to be able to write:

```
public void testSimpleAddition() {
    ...
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

(It's a little weird to be mixing the *bank* and *expression* metaphors. We'll get the whole story told first, and then we'll see what we can do about literary value.)

We have made an important design decision here. We could just as easily have written `…reduce= sum.reduce("USD", bank)`. Why make the bank responsible? One answer is, "That's the first thing that popped into my head," but that's not very informative. Why did it pop into my head that reduction should be the responsibility of the bank rather than of the expression? Here's what I'm aware of at the moment:

- `Expression`s seem to be at the heart of what we are doing. I try to keep the objects at the heart as ignorant of the rest of the world as possible, so they stay flexible as long as possible (and remain easy to test, and reuse, and understand).
- I can imagine there will be many operations involving `Expression`s. If we add every operation to `Expression`, then `Expression` will grow without limit.

That doesn't seem like enough reasons to tip the scales permanently, but it is enough for me to start in this direction. I'm also perfectly willing to move responsibility for reduction to `Expression` if it turns out that `Bank`s don't need to be involved.

The `Bank` in our simple example doesn't really need to do anything. As long as we have an object, we're okay:

```
public void testSimpleAddition() {
    ...
```

```
    Bank bank= new Bank();
    Mbney reduced= bank.reduce(sum, "USD");
    assertEquals(Mbney.dollar(10), reduced);
}
```

The sum of two Moneys should be an Expression:

```
public void testSimpleAddition() {
    ...
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Mbney reduced= bank.reduce(sum, "USD");
    assertEquals(Mbney.dollar(10), reduced);
}
```

At least we know for sure how to get five dollars:

```
public void testSimpleAddition() {
    Mbney five= Mbney.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Mbney reduced= bank.reduce(sum, "USD");
    assertEquals(Mbney.dollar(10), reduced);
}
```

How do we get this to compile? We need an interface Expression (we could have a class, but an interface is even lighter weight):

Expression

```
interfaceExpression
```

Money.plus() needs to return an Expression,

Money

```
Expression plus(Mbney addend) {
    return new Mbney(amount + addend.amount, currency);
}
```

which means that Money has to implement Expression (which is easy, as there are no operations yet):

Money

```
classMbney implements Expression
```

We need an empty Bank class,

Bank

`class` **Bank**

which stubs out `reduce()`:

Bank

```
Money reduce(Expression source, String to) {
   return null;
}
```

Now it compiles, and fails miserably. Hooray! Progress! We can easily fake the implementation, though:

Bank

```
Money reduce(Expression source, String to) {
   return Money.dollar(10);
}
```

We're back to a green bar, and ready to refactor. First, to review, we

- Reduced a big test to a smaller test that represented progress ($5 + 10 CHF to $5 + $5)
- Thought carefully about the possible metaphors for our computation
- Rewrote our previous test based on our new metaphor
- Got the test to compile quickly
- Made it run
- Looked forward with a bit of trepidation to the refactoring necessary to make the implementation real

Chapter 13. Make It

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

We can't mark our test for $5 + $5 done until we've removed all of the duplication. We don't have code duplication, but we do have data duplication—the $10 in the fake implementation:

Bank

```
Money reduce(Expression source, String to) {
    return Money.dollar(10);
}
```

is really the same as the $5 + $5 in the test:

```
public void testSimpleAddition() {
    Money five= Money.dollar(5);
    Expression sum= five.plus(five);
    Bank bank= new Bank();
    Money reduced= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(10), reduced);
}
```

Before when we've had a fake implementation, it has been obvious how to work backward to the real implementation. It simply has been a matter of replacing constants with variables. This time, however, it's not obvious to me how to work backward. So, even though it feels a little speculative, we'll work forward.

```
$5 + 10 CHF = $10 if rate is 2:1
$5 + $5 = $10
Return Money from $5 + $5
```

First, `Money.plus()` needs to return a real `Expression`—a `Sum`, not just a `Money`. (Perhaps later we'll optimize the special case of adding two identical currencies, but that's later.)

The sum of two `Money`s should be a `Sum`:

```
public void testPlusReturnsSum() {
    Money five= Money.dollar(5);
    Expression result= five.plus(five);
    Sum sum= (Sum) result;
    assertEquals(five, sum.augend);
    assertEquals(five, sum.addend);
}
```

(Did you know that the first argument to addition is called the *augend?* I didn't until I was writing this. Geek joy.)

The test above is not one I would expect to live a long time. It is deeply concerned with the implementation of our operation, rather than its externally visible behavior. However, if we make it work, we expect we've moved one step closer to our goal. To get it to compile, all we need is a `Sum` class with two fields, `augend` and `addend`:

Sum

```
class Sum {
    Money augend;
    Money addend;
}
```

This gives us a `ClassCastException`, because `Money.plus()` is returning a `Money`, not a `Sum`:

Money

```
Expression plus(Money addend) {
    return new Sum(this, addend);
}
```

`Sum` needs a constructor:

Sum

```
Sum(Money augend, Money addend) {
}
```

And`Sum` needs to be a kind of `Expression`:

Sum

```
class Sum implements Expression
```

Now the system compiles again, but the test is still failing, this time because the `Sum` constructor is not setting the fields. (We could fake the implementation by initializing the fields, but I said I'd start going faster.)

Sum

```
Sum(Money augend, Money addend) {
    this.augend= augend;
    this.addend= addend;
}
```

Now`Bank.reduce()` is being passed a `Sum`. If the currencies in the `Sum` are all the same, and the target currency is also the same, then the result should be a `Money` whose amount is the sum of the amounts:

```
public void testReduceSum() {
    Expression sum= new Sum(Money.dollar(3), Money.dollar(4));
    Bank bank= new Bank();
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(7), result);
}
```

I carefully chose parameters that would break the existing test. When we reduce a `Sum`, the result (under these simplified circumstances) should be a `Money` whose amount is the sum of the amounts of the two `Money`s and whose currency is the currency to which we are reducing.

Bank

```
Money reduce(Expression source, String to) {
    Sum sum= (Sum) source;
    int amount= sum.augend.amount + sum.addend.amount;
    return new Money(amount, to);
}
```

This is immediately ugly on two counts:

- The cast. This code should work with any `Expression`.
- The public fields, and two levels of references at that.

Easy enough to fix. First, we can move the body of the method to `Sum` and get rid of some of the visible fields. We are "sure" we will need the `Bank` as a parameter in the future, but this is pure, simple refactoring, so we leave it out. (Actually, just now I put it in because I "knew" I would need it—shame, shame on me.)

Bank

```
Money reduce(Expression source, String to) {
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Sum

```
publicMoney reduce(String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

$5 + 10 CHF = $10 if rate is 2:1 **$5 + $5 = $10** Return Money from $5 + $5 Bank.reduce(Money)

(Which brings up the point of how we are going to implement, er … test `Bank.reduce()` when the argument is a `Money`.)

Let's write that test, since the bar is green and there is nothing else obvious to do with the code above:

```
public void testReduceMoney() {
    Bank bank= new Bank();
    Money result= bank.reduce(Money.dollar(1), "USD");
    assertEquals(Money.dollar(1), result);
}
```

Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money) return (Money) source;
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Ugly, ugly, ugly. However, we now have a green bar, and refactoring is possible. Any time we are checking classes explicitly, we should be using polymorphism instead. Because `Sum` implements `reduce(String)`, if `Money` implemented it, too, we could then add it to the `Expression` interface.

Bank

```
Money reduce(Expression source, String to) {
    if (source instanceof Money)
        return (Money) source.reduce(to);
    Sum sum= (Sum) source;
    return sum.reduce(to);
}
```

Money

```
publicMoney reduce(String to) {
    return this;
}
```

If we add `reduce(String)` to the `Expression` interface,

Expression

```
Money reduce(String to);
```

then we can eliminate all those ugly casts and class checks:

Bank

```
Money reduce(Expression source, String to) {
    return source.reduce(to);
}
```

I'm not entirely happy with the name of the method being the same in `Expression` and in `Bank`, but having different parameter types in each. I've never found a satisfactory general solution to this problem in Java. In languages with keyword parameters, communicating the difference between `Bank.reduce(Expression, String)` and `Expression.reduce(String)` is well supported by the language syntax. With positional parameters, it's not so easy to make the code speak for us about how the two are different.

$5 + 10 CHF = $10 if rate is 2:1
$5 + $5 = $10
Return Money from $5 + $5
~~Bank.reduce(Money)~~
Reduce Money with conversion
Reduce(Bank, String)

Next we'll actually exchange one currency for another. First, to review, we

- Didn't mark a test as done because the duplication had not been eliminated
- Worked forward instead of backward to realize the implementation
- Wrote a test to force the creation of an object we expected to need later (`Sum`)
- Started implementing faster (the `Sum` constructor)
- Implemented code with casts in one place, then moved the code where it belonged once the tests were running
- Introduced polymorphism to eliminate explicit class checking

Chapter 14. Change

$5 + 10 CHF = $10 if rate is 2:1
$5 + $5 = $10
Return Money from $5 + $5
~~Bank.reduce(Money)~~
Reduce Money with conversion
Reduce(Bank, String)

Change is worth embracing (especially if you have a book out with "embrace change" in the title). Here, however, we are thinking about a much simpler form of change—we have two francs and we want a dollar. That sounds like a test case already:

```
public void testReduceMoneyDifferentCurrency() {
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(Money.franc(2), "USD");
    assertEquals(Money.dollar(1), result);
}
```

When I convert francs to dollars, I divide by two. (We're still studiously ignoring all of those nasty numerical problems.) We can make the bar green in one piece of ugliness:

Money

```
publicMoney reduce(String to) {
    int rate = (currency.equals("CHF") && to.equals("USD"))
        ? 2
        : 1;
    return new Money(amount / rate, to);
}
```

Now, suddenly, `Money` knows about exchange rates. Yuck. The `Bank` should be the only place we care about exchange rates. We'll have to pass the `Bank` as a parameter to `Expression.reduce()`. (See? We*knew* we would need it, and we were right. In the words of the grandfather in *The Princess Bride*, "You're very clever… .") First the caller:

Bank

```
Money reduce(Expression source, String to) {
    return source.reduce(this,to);
}
```

Then the implementors:

Expression

```
Money reduce(Bank bank, String to);
```

Sum

```
public Money reduce(Bank bank, String to) {
    int amount = augend.amount + addend.amount;
    return new Money(amount, to);
}
```

Money

```
public Money reduce(Bank bank, String to) {
    int rate = (currency.equals("CHF") && to.equals("USD"))
    ? 2
    : 1;
    return new Money(amount / rate, to);
}
```

The methods have to be public because methods in interfaces have to be public (for some excellent reason, I'm sure).

Now we can calculate the rate in the `Bank`:

Bank

```
int rate(String from, String to) {
    return (from.equals("CHF") && to.equals("USD"))
    ? 2
    : 1;
}
```

And ask the `bank` for the right rate:

Money

```
public Money reduce(Bank bank, String to) {
    int rate = bank.rate(currency, to);
    return new Money(amount / rate, to);
}
```

That pesky 2 still appears in both the test and the code. To get rid of it, we need to keep a table of rates in the `Bank` and look up a rate when we need it. We could use a hashtable that maps pairs of currencies to rates. Can we use a two-element array containing the two currencies as the key? Does `Array.equals()` check to see if the elements are equal?

```
public void testArrayEquals() {
    assertEquals(new Object[] {"abc"}, new Object[] {"abc"});
}
```

Nope. The test fails, so we have to create a real object for the key:

Pair

```
private class Pair {
   private String from;
   private String to;

   Pair(String from, String to) {
   this.from= from;
   this.to= to;
   }
}
```

Because we are using `Pair`s as keys, we have to implement `equals()` and `hashCode()`. I'm not going to write tests for these, because we are writing this code in the context of a refactoring. If we get to the payoff of the refactoring and all of the tests run, then we expect the code to have been exercised. If I were programming with someone who didn't see exactly where we were going with this, or if the logic became the least bit complex, I would begin writing separate tests.

Pair

```
public boolean equals(Object object) {
   Pair pair= (Pair) object;
   return from.equals(pair.from) && to.equals(pair.to);
}

public int hashCode() {
   return 0;
}
```

0 is a terrible hash value, but it has the advantage of being easy to implement, and it will get us running quickly. Currency lookup will look like linear search. Later, when we get lots of currencies, we can do a more thorough job with real usage data.

We need somewhere to store the rates:

Bank

```
private Hashtable rates= new Hashtable();
```

We need to set the rate when told:

Bank

```
void addRate(String from, String to, int rate) {
   rates.put(new Pair(from, to), new Integer(rate));
}
```

And then we can look up the rate when asked:

Bank

```
int rate(String from, String to) {
   Integer rate= (Integer) rates.get(new Pair(from, to));
   return rate.intValue();
```

```
}
```

Wait a minute! We got a red bar. What happened? A little snooping around tells us that if we ask for the rate from USD to USD, we expect the value to be 1. Because this was a surprise, let's write a test to communicate what we discovered:

```
public void testIdentityRate() {
    assertEquals(1, new Bank().rate("USD", "USD"));
}
```

Now we have three errors, but we expect them all to be fixed with one change:

Bank

```
int rate(String from, String to) {
    if (from.equals(to)) return 1;
    Integer rate= (Integer) rates.get(new Pair(from, to));
    return rate.intValue();
}
```

Green bar!

$5 + 10 CHF = $10 if rate is 2:1
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~

Next we'll implement our last big test, $5 + 10 CHF. Several significant techniques have slipped into this chapter. For now, to review, we

- Added a parameter, in seconds, that we expected we would need
- Factored out the data duplication between code and tests
- Wrote a test (`testArrayEquals`) to check an assumption about the operation of Java
- Introduced a private helper class without distinct tests of its own
- Made a mistake in a refactoring and chose to forge ahead, writing another test to isolate the problem

Chapter 15. Mixed Currencies

$5 + 10 CHF = $10 if rate is 2:1
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~

Now we are finally ready to add the test that started it all, $5 + 10 CHF:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

This is what we'd like to write. Unfortunately, there are a host of compile errors. When we were generalizing from `Money` to `Expression`, we left a lot of loose ends laying around. I was worried about them, but I didn't want to disturb you. It's time to disturb you now.

We won't be able to get the preceding test to compile quickly. We will make the first change that will ripple to the next and the next. We have two paths forward. We can make it work quickly by writing a more specific test and then generalizing, or we can trust our compiler not to let us make mistakes. I'm with you—let's go slow (in practice, I would probably just fix the rippling changes one at a time).

```
public void testMixedAddition() {
    Money fiveBucks= Money.dollar(5);
    Money tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

The test doesn't work. We get 15 USD instead of 10 USD. It's as if `Sum.reduce()` weren't reducing the arguments. It isn't:

Sum

```
publicMoney reduce(Bank bank, String to) {
    int amount= augend.amount + addend.amount;
    return new Money(amount, to);
}
```

If we reduce both of the arguments, the test should pass:

Sum

```
public Money reduce(Bank bank, String to) {
    int amount = augend.reduce(bank, to).amount
        + addend.reduce(bank, to).amount;
    return new Money(amount, to);
}
```

And it does. Now we can begin pecking away at Moneys that should be Expressions. To avoid the ripple effect, we'll start at the edges and work our way back to the test case. For example, the augend and addend can now be Expressions:

Sum

```
Expression augend;
Expression addend;
```

The arguments to the Sum constructor can also be Expressions:

Sum

```
Sum(Expression augend, Expression addend) {
    this.augend = augend;
    this.addend = addend;
}
```

(Sum is starting to remind me of Composite, but not so much that I want to generalize. The moment we want a Sum with other than two parameters, though, I'm ready to transform it.) So much for Sum—how about Money?

The argument to plus() can be an Expression:

Money

```
Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

Times() can return an Expression:

Money

```
Expression times(int multiplier) {
    return new Money(amount * multiplier, currency);
}
```

This suggests that Expression should include the operations plus() and times(). That's all for Money. We can now change the argument to plus() in our test case:

```
public void testMixedAddition() {
    Money fiveBucks = Money.dollar(5);
```

```
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

When we change `fiveBucks` to an `Expression`, we have to make several changes. Fortunately we have the compiler's to-do list to keep us focused. First we make the change:

```
public void testMixedAddition() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Money result= bank.reduce(fiveBucks.plus(tenFrancs), "USD");
    assertEquals(Money.dollar(10), result);
}
```

We are politely told that `plus()` is not defined for `Expression`s. We define it:

Expression

```
Expression plus(Expression addend);
```

And then we have to add it to `Money` and `Sum`.`Money`? Yes, it has to be public in `Money`:

Money

```
publicExpression plus(Expression addend) {
    return new Sum(this,addend);
}
```

We'll just stub out the implementation in `Sum`, and add it to our to-do list:

Sum

```
publicExpression plus(Expression addend) {
    return null;
}
```

~~$5 + 10 CHF = $10 if rate is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~
Sum.plus
Expression.times

Now that the program compiles, the tests all run.

We are ready to finish generalizing `Money` to `Expression`. But first, to review, we

- Wrote the test we wanted, then backed off to make it achievable in one step
- Generalized (used a more abstract declaration) from the leaves back to the root (the test case)
- Followed the compiler when we made a change (`ExpressionfiveBucks`), which caused changes to ripple (added `plus()` to `Expression`, and so on)

Chapter 16. Abstraction, Finally

~~$5 + 10 CHF = $10 if rate is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~
Sum.plus
Expression.times

We need to implement `Sum.plus()` to finish `Expression.plus`, and then we need `Expression.times()`, and then we're finished with the whole example. Here's the test for `Sum.plus()`:

```
public void testSumPlusMoney() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).plus(fiveBucks);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(15), result);
}
```

We could have created a `Sum` by adding `fiveBucks` and `tenFrancs`, but the form above, in which we explicitly create the `Sum`, communicates more directly. We are writing these tests not just to make our experience of programming more fun and rewarding, but also as a Rosetta stone for future generations to appreciate our genius. Think, oh think, of our readers.

The test, in this case, is longer than the code. The code is the same as the code in `Money`. (Do I hear an abstract class in the distance?)

Sum

```
public Expression plus(Expression addend) {
    return new Sum(this, addend);
}
```

~~$5 + 10 CHF = $10 if rate is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~
~~Sum.plus~~
Expression.times

You will likely end up with about the same number of lines of test code as model code when implementing TDD. For TDD to make economic sense, you'll need to be able to either write twice as many lines per day as before, or write half as many lines for the same functionality. You'll have to measure and see what effect TDD has on your own practice. Be sure to factor debugging, integrating, and explaining time into your metrics, though.

~~$5 + 10 CHF = $10 if rate is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~
~~Sum.plus~~
Expression.times

If we can make `Sum.times()` work, then declaring `Expression.times()` will be one simple step. The test is:

```
public void testSumTimes() {
    Expression fiveBucks= Money.dollar(5);
    Expression tenFrancs= Money.franc(10);
    Bank bank= new Bank();
    bank.addRate("CHF", "USD", 2);
    Expression sum= new Sum(fiveBucks, tenFrancs).times(2);
    Money result= bank.reduce(sum, "USD");
    assertEquals(Money.dollar(20), result);
}
```

Again, the test is longer than the code. (You JUnit geeks will know how to fix that—the rest of you will have to read Fixture.)

Sum

```
Expression times(int multiplier) {
    return new Sum(augend.times(multiplier), addend.times(multiplier));
}
```

Because we abstracted augend and addend to `Expressions` in the last chapter, we now have to declare `times()` in `Expression` for the code to compile:

Expression

```
Expression times(int multiplier);
```

This forces us to raise the visibility of `Money.times()` and `Sum.times()`:

Sum

```
public Expression times(int multiplier) {
    return new Sum(augend.times(multiplier), addend.times(multiplier));
}
```

Money

```
public Expression times(int multiplier) {
        return new Money(amount * multiplier, currency);
}
```

~~$5 + 10 CHF = $10 if rate is 2:1~~
~~$5 + $5 = $10~~
Return Money from $5 + $5
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~
~~Sum.plus~~
~~Expression.times~~

And it works.

The only loose end to tie up is to experiment with returning a `Money` when we add $5 + $5. The test would be:

```
public void testPlusSameCurrencyReturnsMoney() {
   Expression sum= Money.dollar(1).plus(Money.dollar(1));
   assertTrue(sum instanceof Money);
}
```

This test is a little ugly, because it is testing the guts of the implementation, not the externally visible behavior of the objects. However, it will drive us to make the changes we need to make, and this is only an experiment, after all. Here is the code we would have to modify to make it work:

Money

```
public Expression plus(Expression addend) {
   return new Sum(this, addend);
}
```

~~$5 + 10 CHF = $10 if rate is 2:1~~
~~$5 + $5 = $10~~
~~Return Money from $5 + $5~~
~~Bank.reduce(Money)~~
~~Reduce Money with conversion~~
~~Reduce(Bank, String)~~
~~Sum.plus~~
~~Expression.times~~

There is no obvious, clean way (not to me, anyway; I'm sure you could think of something) to check the currency of the argument if and only if it is a `Money`. The experiment fails, we delete the test (which we didn't like much anyway), and away we go.

To review, we

- Wrote a test with future readers in mind
- Suggested an experiment comparing TDD with your current programming style
- Once again had changes of declarations ripple through the system, and once again followed the compiler's advice to fix them
- Tried a brief experiment, then discarded it when it didn't work out

Chapter 17. Money Retrospective

Let's take a look back at the money example, both the process we used and the results. We will look at:

- What's Next?
- Metaphor— The dramatic effect metaphor has on the structure of the design.
- JUnit Usage— When we ran tests and how we used JUnit.
- Code Metrics— A numerical abstract of the resulting code.
- Process— We say red/green/refactor, but how much work goes into each step?
- Test Quality— How do TDD tests stack up against conventional test metrics?