The Addison-Wesley Signature Series

A KENT BECK SIGNATURE BOOK

TEST-DRIVEN DEVELOPMENT

BY EXAMPLE

KENT BECK

Test-Driven Development

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

This page intentionally left blank

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 `Moneys`. 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 `Expressions`, 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:

- `Expressions` 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 `Expressions`. 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 `Banks` 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();
   Money reduced= bank.reduce(sum, "USD");
   assertEquals(Money.dollar(10), reduced);
}
```

The sum of two `Moneys` should be an `Expression`:

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