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 hash-Code() 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()

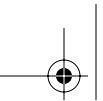We could have used Triangulation to drive the generalization of `times()` also. If we had $5 x 2 = $10 and $5 x 3 = $15, then we would no longer have been able to return a constant.

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