Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

Before Clarity

Michael Feathers

If you're at all like me, you've spent a lot of time thinking about what makes "good design" good. Most software developers become preoccupied with this question at some point in their careers, usually after witnessing the effects of bad design first hand. At that point, we start to reflect. We go through a stage where we feel we know what good design is but can't really define it. Then we learn various design principles and rules of thumb that make it easier to judge what constitutes good design. But when these principles and rules conflict, we have to make trade-offs and decide what's most important in each situation.

To help me out, a few years ago I created my own blanket rule of thumb: Keep design as clear as possible. I was pretty sure that, regardless of the trade-offs, the most important thing was clarity. If a system uses a straightforward coding style—the classes and methods are well named and small enough to be clearly understood, and the system isn't littered with snarls of obscure code—then you can do just about anything. You can change the system with impunity, write tests for it, make adjustments, and add features, all with relative ease.

So "clear design is good design" seemed like a reasonable rule of thumb because so much of what makes code impossible to maintain comes down to a lack of clarity. If you can understand your system, you can change it effectively. If you can't, it's much harder. Sounds simple and straightforward, right? Well, it might be a little too simple. Recently, I've discovered that I'm periodically sacrificing clarity and subordinating it to another standard.

Changing clear code

Let's take a look at an example. In Figure 1, we have some reasonably clear code for a method on a C++ class in a trading application. The `adjustedShares` method computes the number of shares that should be sold under certain circumstances. To do this, it gets information from a static method named `getRateAdjustment` on a class named `TradeUtils`. `TradeUtils` is just a utility class. All of its methods are static.

So, what would we do if we needed to change the way `getRateAdjustment` calculates `baseAdjustment`? The answer seems relatively straightforward—we'd go into the code and change the calculation. But how would we know if we got it right?

The most direct way to ensure that our changes are correct is to build some tests around the code we're changing to sense how it currently works. Then, once we've made our changes, we can write more tests to see if the changed code works as expected.

So, the first step is to create a `ShareBias` instance in a test so we can call the `adjustShares` method and to see what values we're currently producing. Figure 2 shows a test written using the CppUnit testing framework (http://cppunit.sourceforge.net).

This seems simple enough, but we discover a problem when running the test. The static method `getRateAdjustment` on the `TradeUtils` class throws an exception. It turns out that it talks to a remote system that provides real-time rate adjustments over a socket. It also turns out that other methods on `Share-`

0740-7459/04/$20.00 © 2004 IEEE

Bias use many other static methods on the TradeUtils class, presenting similar problems. How in the world can we test this little piece of code without bringing up that other system?

Refactoring to make the code testable

This sounds like a job for refactoring. Refactoring is the process of changing code's structure without changing its behavior to make it more maintainable (for more information, see Martin Fowler's *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999). In this case, we'd like to refactor ShareBias to make it more testable, which would also make it more maintainable. The problem is, when we refactor, we should have tests around the code we're changing to ensure we don't introduce errors.

To address this problem, we could use a series of dependency-breaking techniques to make the class testable. Dependency-breaking techniques are refactorings, but they're very conservative—we can perform them safely without running tests. (I discuss this further in *Working Effectively with Legacy Code*, Prentice Hall, 2004.)

So, how can we refactor this code to make it testable? Here's one technique. We can find the getRateAdjustment method on the TradeUtils class and make it a nonstatic method (see Figure 3). Then we could subclass TradeUtils and override the nonstatic method so it returns values we supply when we're testing. If we change the ShareBias class's constructor to accept a TradeUtils instance, we could use that instance in the adjustedShares method (see Figure 3) and eliminate dependencies on the remote system. Doing this would also make it possible to write tests for all other ShareBias methods that use TradeUtils.

The technique just used is a variation of a technique called *Introduce Instance Delegator*. Although it gets us past the immediate problem—we can now write tests for the adjustedShares method—the result is a rather ugly TradeUtils class (it has a bunch of static methods and only one nonstatic method). Furthermore, we'll have to find all the places in the code base where we access getRateAdjustment statically and declare a TradeUtils object so that we can replace calls such as TradeUtils::getRateAdjustment (id, 3) with someUtilsObject.getRateAdjustment(id, 3).

So this isn't the cleanest solution. TradeUtils is now a class in limbo—sometimes we need to create an instance of it and other times we can just use its methods statically. However, over time, we can make it more like a normal class by making its static methods nonstatic. Should we feel bad that our code is less than clear now? We could, but we're making progress. The fact is, we had to do this or something else equally ungainly to test the method. The original code was clear but not testable. To me, that just makes it poorly designed code.

```
double ShareBias::adjustedShares(ClientAccount& account,
                                 int sharesToSell,
                                 double balanceRate) {
    double baseAdjustment = 0;
    double adjustedShares = 0;

    if (balanceRate > m_orgBase) {
         baseAdjustment = TradeUtils::getRateAdjustment(
               account.getID(), sharesToSell);
    } else {
       baseAdjustment = m_state_min + m_bucket[POST_DIST]
         + m_bucket[PRE_STATE_DIST];
    }

    adjustedShares = (3 * sharesToSell)
         - Math.sqrt(baseAdjustment * 2.0) * m_defaultBias;
    …
    return adjustedShares;
}
```

Figure 1. Reasonably clear code for a C++ method in a trading application.

```
void ShareBiasTest::testAdjustedShares() {
    ShareBias bias;
    ClientAccount  account("Joe", 1);

    assertEquals(0.0, bias.adjustedShares(account, 10, 0.9));
}
```

Figure 2. Creating a ShareBias instance in a test.

I like clean, clear code. I consider clarity to be one of the most important things that we can achieve in our designs. But as I try to get tests in place in ostensibly clear code bases, I've discovered that I don't mind strategically dropping clarity if it'll get me testability. When code is testable, we get a different kind of clarity. We can write tests that make the code's functionality very clear, and we can use those tests to support us as we move the code back to conventional clarity.

Is there a good clear yet testable de-

```
class TradeUtils
{
    static double getForeignRate(int id, date transactionDate);
    static double getMarginalRate(int id, date transactionDate);
    static double setAccountAllowanceFactor(int id, double factor);
    …

    // this method was static
    virtual double getRateAdjustment(int id, int sharesToSell);
};

class FakeTradeUtils : public TradeUtils
{
public:
    double nextAdjustment;
    …
    virtual double getRateAdjustment(int id, int sharesToSell) {
      return nextAdjustment;
    }
};

double ShareBias::adjustedShares(ClientAccount& account,
                                 int sharesToSell,
                                 double balanceRate) {
    double baseAdjustment = 0;
    double adjustedShares = 0;

    if (balanceRate > m_orgBase) {
        baseAdjustment = m_tradeUtils.getRateAdjustment(
            account.getID(), sharesToSell);
    } else {
        baseAdjustment = m_state_min + m_bucket[POST_DIST]
            + m_bucket[PRE_STATE_DIST];
    }

    adjustedShares = (3 * sharesToSell) - Math.sqrt(baseAdjustment * 2.0)
        * m_defaultBias;
    …
    return adjustedShares;
}

// Test using TradeUtils

void ShareBiasTest::testAdjustedShares() {
    FakeTradeUtils tradeUtils;
    ShareBias bias (tradeUtils);
    ClientAccount account("Joe", 1);

    tradeUtils.nextAdjustment = 0.1;
    assertEquals(0.7, bias.adjustedShares(account, 10, 0.9));
}
```

Figure 3. Code modifications to support testing of `adjustedShares`.

sign for the `ShareBias` class? I think so. When we rename `TradeUtils` and start to make its static methods nonstatic, we'll get closer to it, and the tests we write in the process will help us make sure that we get there without breaking the code. In much the same way, we can take most designs and make them easier to maintain if we decide to make them testable first. Once we do, we can make the code clearer as we move forward.

Michael Feathers is a consultant at Object Mentor. Contact him at mfeathers@objectmentor.com.