

Testable Java

Michael Feathers

mfeathers@objectmentor.com

Object Mentor, Inc.

Have you ever tried to write unit tests for code that doesn't have them? If you have, chances are you've run into a number of problems. Code that wasn't designed to be testable is not testable. Testability is not a quality that appears accidentally. Or does it?

If you practice Test-Driven Development diligently, your code is testable by default. You simply don't get to add functionality without a triggering test case. Seen this way, TDD is a generative process. It answers the question of testability by being a procedure that gives us testability, but it isn't perfect. Or rather, it is and we aren't. TDD is a high-discipline approach and for a variety of reasons, there are still people who write code directly without tests.

In this paper, I'm going to outline a simple rule that you can use to write testable Java code, code that is easy to test once you've decided to test it. Like all simple rules, however, it requires background knowledge that you must draw upon to apply it well.

So, here's the rule:

Never hide a TUF within a TUC

Right, so what is a TUF and what is a TUC?

TUF is an acronym for *Test Unfriendly Feature*. It is some quality of code, or piece of functionality embodied in code, which makes unit testing difficult.

Here are some example TUFs:

- Database access
- File system access
- Network access
- Access to side effecting APIs (GUI, etc)
- Lengthy computations

- Inscrutable computations (computations which are hard to test because they are difficult to understand)
- Static variable usage

TUFs can be unfriendly for a variety of reasons. Sometimes the unfriendliness consists of the amount of time that it takes to setup or maintain a test case. At other times the unfriendliness comes from the fact that the TUF produces side effects that we would not like to have occur during test execution. TUFs are, essentially, features in production code that are impediments during unit testing.

Now, that we know what TUFs are, let's talk about TUCs. A TUC is a *Test Unfriendly Construct*. It is some construct in a programming language that makes it hard to swap one behavior with another. That swapping is really the essence of mocking and stubbing.

Here's a list of TUCs in Java

- Final Methods
- Final Classes
- Static Methods
- Private Methods
- Static Initialization Expressions
- Static Initialization Blocks
- Constructors
- Object Initialization Blocks
- New-Expressions

In the section that follows, I'll discuss each of them.

Test Unfriendly Constructs in Java

One classic move when trying to get code under test is to Subclass and Override anything that gets in the way during testing. Let's look at an example:

```
public class EventResponder
{
    void showNotification(String notificationMessage) {
        JOptionPane.showMessageDialog(null, notificationMessage);
    }

    public void respond() {
        ...
        showNotification(message);
    }
}
```

If we are testing code in the *respond* method, we probably don't want to have a message dialog pop up when we run our tests. To get past this, we create a testing subclass and we override the offending method:

```
public class TestingEventResponder extends EventResponder
{
    void showNotification(String notificationMessage) {
    }
}
```

In this case, it's enough to just override *showNotification* with an empty method. We were just lucky that it was not a private method.

The Subclass and Override move is simple, easy, and almost always doable – except in a few oddball cases. One of them is the use of final methods. If a method is *final*, you simply can't override it in a testing subclass. And, in many situations, that may not be a problem at all. We definitely don't need to override all of the methods in a class to be able to test methods that use them; however, final methods are TUCs because they have the potential to hide a TUF. I'm not saying that *final* methods are intrinsically bad, but I do think that it is worth being extremely careful about their use. There can be quite a bit of distance between a TUC and the TUF it hides: the TUF can be rather far the down the call-stack. When someone walks in later and attempts to write tests for that code, every stray TUC that is lying around is an irritation: it might even be a showstopper.

Final Classes are TUCs also, and for fundamentally the same reason that Final Methods are. They get in the way of Subclass and Override. A *final* class can't be sub-classed, and while that might be a constraint that you'd like to document in your design, documenting it with a compiler-enforced mechanism like *final* is a rather severe move. Strictly speaking, *final* classes and *final* methods are only a problem in testing when they hide a TUF, but it is nice to have a carefully considered reason for using them rather than just using them by default.

Static Methods, as well, get in the way of Subclass and Override. You simply can not override a *static* method in a testing subclass; however, there are some dependency breaking tricks that you can use. The most powerful one is Introduce Instance Delegator [1], but that technique does leave your code in an odd state. It is better, yet again, to avoid writing *static* methods that hide a TUF. Other uses of *static* methods are perfectly fine. In Java, *Math.cos(double)* is a *static* method, and I don't think anyone has ever wanted to mock it out in a testing situation.

Private Methods suffer from essentially the same problem as *final* and *static* methods. They can be very useful, but if they hide a TUF, trouble ensues.

Static Initialization Expressions and Static Initialization Blocks are TUCs also. They are executed on class load, in order, within a class. And, you really don't have any

choice in the matter. They simply are going to run and you can't override them. Imagine this scenario:

```
public class Configuration
{
    private static final List<ConfigurationItem> items;

    static {
        items = Loader.load("../ourpainintheassconfigurationfile.cfg");
    }
    ...
}
```

In this code, our configuration is set for all time. There is no way to choose to do in-memory initialization for testing, short of changing the code that you want to test, and, let's face it, that's what you are going to have to do here. But, that whole issue could be avoided if people avoided Static Initialization Blocks in the first place, especially when they hide TUFs. In some of my more glib moments, I say that 90% of all testability problems in Java would go away if teams just stopped using the *static* and *final* keywords. For many Java developers it would be a tough pill to swallow, but it's worth noting that there are many languages which have nothing like *static* or *final* methods. Ruby and Smalltalk have class methods, but they are override-able. The recently designed Newspeak language [2] avoids *static* entirely and it is a very elegant language.

If you've ever been in a situation where you've had to consider testability in Java, the issues with *static* and *final* are bound to have occurred to you. But, it often takes people a while to discover the fact that constructors are TUCs. Why are they?

Well, the reason is kind of obvious when you think about it. Constructors have to run, and in Java there is no way to override them. The best you can do if a constructor is hiding a TUF is to use an alternative constructor (if there's one available) or add an alternative constructor for testing; but that is a little hacky, and the hacks get even worse when the TUF is hiding in a superclass constructor. It's better still to avoid the whole problem and keep TUFs out of constructors.

Object Initializer Blocks suffer from the same problem. There is no way to override them.

People have struggled with the use of constructors throughout the history of OO. One common idiom, in the early days, was to perform all meaningful resource acquisition on construction. Imagine having a *SpecialFile* class with a constructor like this:

```
// opens the file
SpecialFile file = new SpecialFile(fileName, File.MODE_READ);
```

This seems perfectly reasonable, until you want to test aspects of a *SpecialFile*'s construction without actually opening a file, or want to test a method that constructs a *SpecialFile* object within it. The best you can do in the latter situation is to extract a factory method:

```
void someMethod() {
    ...
    SpecialFile file = new SpecialFile(fileName, File.READ);
    ...
    ... code we'd like to test
    ...
}

public void someMethod() {
    SpecialFile file = openFile(fileName, File.READ);
    ...
    ... code we'd like to test
    ...
}

protected SpecialFile openFile(String filePath, FileMode mode) {
    return new SpecialFile(filePath, mode);
}
```

Now, we can Subclass and Override *openFile*, but we will have to find some way to make a valid file object. If there is no other constructor for *SpecialFile*, we'll have to introduce one or we can just return null if, for some reason, the *SpecialFile* object isn't actually going to be used in the branches we cover with our test.

Notice that the whole problem could be avoided if we simply kept low-level file operations out of the constructor. Here is an alternative interface:

```
// Constructor does not open the file
SpecialFile file = new SpecialFile(fileName, File.MODE_READ);
file.open();
```

Sometimes people see this as *two-stage initialization*, but to me, it is really a re-conceptualization of the class. The class now represents a file rather than an open file. There are many design problems that we can avoid with a little finesse.

New-Expressions are another TUC. This should really come as no surprise. Dependency Injection Frameworks were pretty much invented to pull *new* out of enterprise applications. The problem with new-expressions is that they hardcode particular concrete classes within a block of code. If those classes don't contain TUFs, it might be fine. But, if they do, you might have to modify production code to use a subclass when you apply Subclass and Override.

Conclusion

Testability is a rarely discussed quality of code. Fortunately, however, it is rather easy to achieve. You can use Test-Driven Development and avoid testability traps effortlessly, or you can pay attention to TUCs and TUFs during development and, if your reasoning is correct, you'll end up with code that is easily testable after the fact.

References

- [1] Feathers, Michael – *Working Effectively with Legacy Code* - Prentice Hall 2004
- [2] Bracha, Gilad – *Newspeak Language* - <http://bracha.org/Site/Newspeak.html>