Too Simple to Break

The Point

Some code is too simple to break. We think you should concentrate your effort on testing code that might break, rather than testing code that clearly cannot. The trick, of course, is to decide how simple is "too simple."

The Details

Let us begin with one of the classic testing questions: *Should I test getters and setters?* We discuss these in detail in [Reference:Chapter:Elementary Tests], so we will summarize here. This simple question divides the Programmer Testing community in two: many believe one ought to test these methods like any other and the rest believe that they are Too Simple to Break. The first camp cite this common problem as a reason to test these methods.

```
private String lastName;

public void setLastName(String lastName) {
    lastName = lastName;
}
```

Here is an example where an ambiguous statement – which `lastName` variable is being assigned to which? – creates a potential problem for the programmer. The above statement will likely be compiled and executed as if it were the following.

```
private String lastName;

public void setLastName(String lastName) {
    this.lastName = this.lastName;
}
```

This statement has no effect, and so is certainly a defect. The first of our two camps would say, *If we do not test our setters, then we will not catch these kinds of defects.* Well... yes and no. Yes, if we write no test for this method then no test will catch this defect; however, this is such an elementary error that there are other safety nets in place to catch it. Most notably, the compiler – or if that fails, the IDE. Our toolset has improved to the point where it is almost impossible for a defect such as this to survive long enough to be executed! Long before that time either the compiler or your IDE will warn you: *this statement has no effect.* We say that this method is Too Simple to Break, and therefore recommend against writing tests for it.

Now, we are not saying that testing this method is a *bad thing.* No-one will shun you if you decide to test it; however, the return on investment for this kind of test is quite low, and it pays to remember the goal of testing. The idea is to spend a certain amount of time and effort on uncovering ways in which the system behaves differently from what you would expect, with the goal of saving money by catching these problems as soon as possible after you created them. There is, therefore, a tradeoff between the effort you expend on testing and the confidence level that you attain in the process. At some point, spending more effort on testing is not worth the corresponding increase in confidence that the code works – or at least, does not fail in a painful way. It is for this reason alone that we have the notion of Too Simple to Break. If testing were free, we would test absolutely every line of code absolutely every way we could imagine.

Back to our getters and setters, there are some kinds of setters that warrant being tested. In particular, a setter that updates more than one variable based on some calculation is a good candidate to be tested. We would test this setter because it does something more complex than merely assign a value to a variable. It is possible for us to write code that the compiler does not criticize, that the IDE does not criticize, and have it do the wrong thing. This is a reasonable definition for Complex Enough to be Tested. Still, there are some such methods that we would generally not bother testing.

It is common in Object-Oriented Programming to see a large amount of *delegation*. That is, an object that wraps another and delegates much of its behavior to the object it wraps. The Decorator design pattern is built on this notion: the decorator performs some small service, but mostly invokes the object it decorates. In Java, the I/O libraries are great examples of Decorators. If you browse the source for, say, `ObjectOutputStream`, you will notice that some of its methods merely invoke the corresponding method on the `BlockDataOutputStream` it contains. Consider the method `ObjectOutputStream.write (int)`, for example.

```
public void write(int val) throws IOException {
    bout.write(val);
}
```

We believe that this method – indeed any method that merely delegates to another object – is Too Simple to Break. There is no way for this method to fail a test, assuming that the neighboring classes have been reasonably well-tested. Our reasoning goes like this: the only things that could go wrong are either that `bout` is `null` or that `BlockDataOutputStream.write()` is broken. This method cannot possibly be held responsible for either of these unfortunate circumstances; therefore, we ought to focus our attention on ensuring that those two things never happen, rather than worrying about this method. This method is Too Simple to Break *on its own*, and that is what we really mean by this catchphrase: a method that cannot break in any way on its own is a method not worth testing.

There may be no complete way to classify methods as Too Simple to Break. We have provided a couple of reasonable heuristics in this essay and if you participate in the JUnit mailing lists, you will occasionally see this topic discussed – often in gory detail. We recommend that you test until fear turns to boredom: if you think it might break, then test it until your thinking changes. If you are just starting out with JUnit, then you may find it comforting to test getters and setters, just to get your feet wet with testing. If you have just spent four hours debugging a problem related to a malfunctioning setter, then you might decide to be extra vigilant about testing setters – for a while. We recommend focusing your energy on code that can easily break, rather than code that may be Too Simple to Break, but never forget that they're just rules.