L07-Object-Design.pptx

<https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two>

[“Testing Private Methods with JUnit and SuiteRunner”](https://www.artima.com/suiterunner/private.html)

by Bill Venners

May 24, 2004

**Summary**

This article compares four different approaches to testing private methods in Java classes.

**MJW:**

Except where otherwise noted, the contents of these notes on unit testing come from the above article. Some parts are paraphrased, reworded or reformatted for clarity.

**Intro:**

* “When I asked Daniel about testing private methods, he gently suggested that I test the private methods indirectly by testing the package-access and public methods that call the private ones.
* This answer did not quite satisfy me, because on occasion I really did feel the urge to directly test a private method. My initial solution was to just make such private methods package access, which allowed me to test them directly with JUnit from the test classes in the same package in the parallel source tree.
* This worked fine, but made me feel a bit dirty somehow. Although in general I discovered that thinking about how to design interfaces so they could be easily unit tested helped me design better interfaces, in this case I felt I was making the design slightly worse to make it testable.”

**Four Basic Approaches:**

1. Don't test private methods. (C++)
   1. Testing private methods may be an indication that those methods should be moved into another class to promote reusability.
   2. if you have a private method so complex that it needs to be tested out of the context of its public callers, I would consider that a code-smell.
   3. Testing private helpers may lead us to create unnecessarily rigorous tests for those units. This will have us writing more code than is needed.
2. Give the methods package access.
   1. This is “breaking encapsulation for the sake of testing”
   2. When you designate methods as “private” you are sending a message about its relationship to the class. Breaking that changes the way we examine that class when considering how to use it.
3. Use a nested test class. (C++)
   1. Your production code would contain test code - OR-
   2. You would have to modify your development code for production
   3. From my experience in commercial software, both of these options are unacceptable because:
      1. I don’t want anyone having access to anything that is not exactly what I intended them to see.
      2. Any modification from the testing environment means the code going into the release is not the exact same as the code in the test environment
4. Use reflection - this exploits an aspect of Java that is not present in every language. C++ does not support reflection.

[**Languages Supporting Reflection**](https://en.wikipedia.org/wiki/Reflection_(computer_programming))

**MJW:**

Some private helpers that we write should become classes of their own so their methods would be public and available for testing at the class level. Parsing user input might fall into this category. It is of general utility. You could use this in other places for other applications.

* Yes, this results in more coding for you.
* Yes, it also results in better design.

The presence of methods like this in a class is considered a “code smell”.

**Avoid Utility Classes:**

<https://www.vojtechruzicka.com/avoid-utility-classes/>

## Conclusion

I wouldn't go as far as saying that utility classes are evil and should never ever be used. However, when creating one, think twice. Be aware of all the implications of such approach. Maybe you there is already a class where that functionality would fit. Maybe you can use a regular instantiable class instead. Should it really be a util class or are you just too lazy to find a proper place to put the new logic in?

**BUT!!!** Expedience is a thing. This is not real life but you should know that there are many things to consider.