PNEHM! #3 - 2005 CITERUS

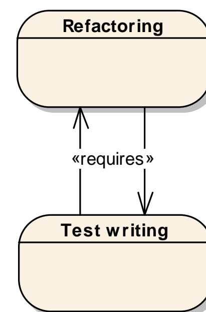
CITERUS NYHETSBREV FÖR DIG SOM VILL LYCKAS MED MJUKVARUUTVECKLING

JUNIT ASSIST - ONE WAY TO UNFOLD THE CATCH 22

Tobias Hill

Have you ever been faced with legacy code that is in such a poor state that a major refactoring is absolutely needed before any new functionality can be added? And then, have you to your very horror realized that there is no test suite, let alone a single test, to facilitate your refactoring endeavours? Citerus consultant Tobias Hill has, and it made him realize how frustrating that situation is. From frustration came inspiration. From inspiration came a suggestion.

The problem when trying to refactor legacy code lacking a test suite somewhat boils down to this catch 22:



- To be able to do refactoring we need a test suite to ensure that we do not break the external contract of components and the system as a whole.

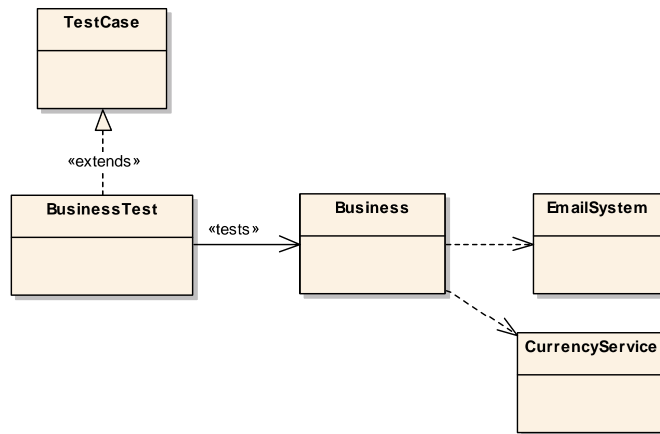
- To be able to write the tests we need to do, or would benefit much from some refactorings. The reason for this is that we might be forced to mock or stub some of the dependencies of the class we want to write tests for. Quite often this mocking/stubbing requires refactoring when working with badly written and tightly coupled applications.

A common way to solve this is to work it out little by little, taking small steps of refactorings followed by some unit tests. To some degree dependencies can be extracted to methods within the class we want to test. Those can in turn be overridden and thereby mocked. This approach still requires changes to code for which you have no test suite and is as a consequence both risky and tedious.

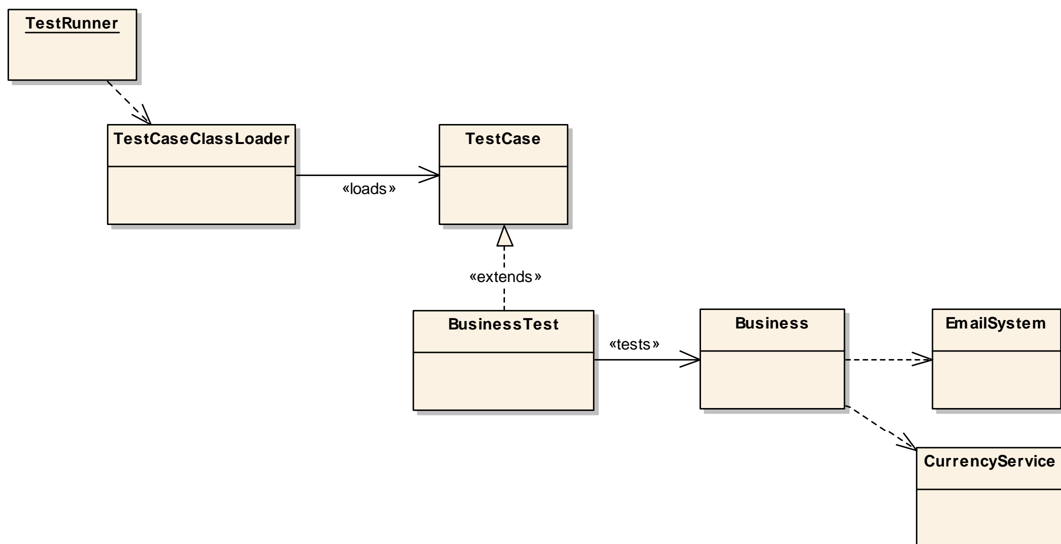
© 2005, Citerus | www.citerus.se

One day on my way home from a typical very-small-step-refactorings-due-to-missing-testsuite-workday I realized that the solution to this quite unsatisfactory situation might be to add better functionality to our present test frameworks, in my case JUnit. I envisioned the following.

A typical test scenario with hard to cut out dependencies might look like this:

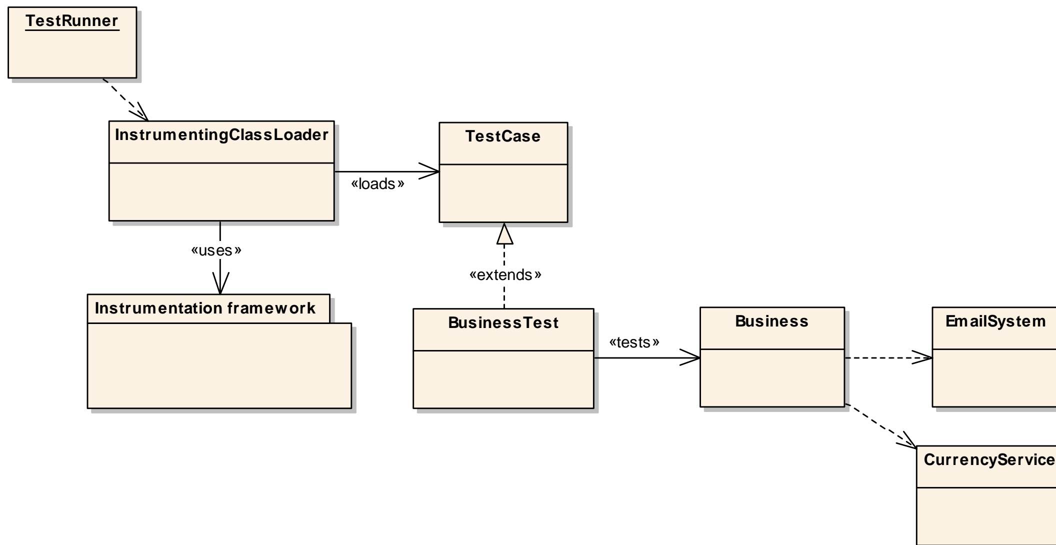


A schematic and slightly simplified view of how this BusinessTest is handled by the JUnit framework can be seen below.

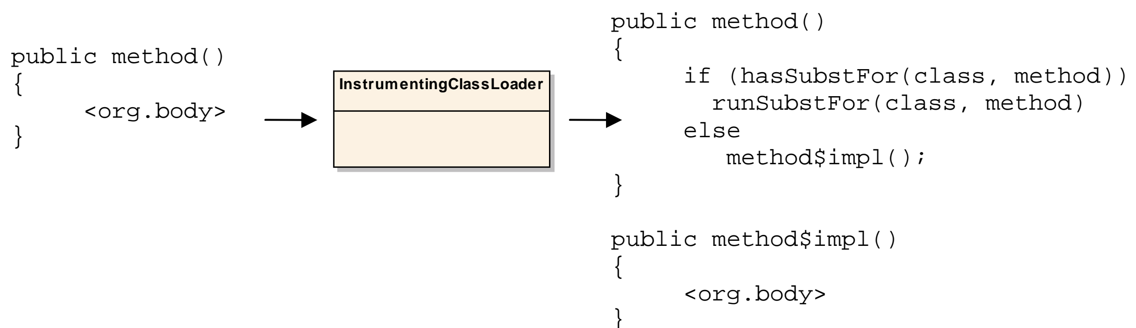


When launching the TestRunner of JUnit it will load the test class with its internal TestCaseClassLoader and then run method by method on it while collecting results.

When running the test, dependent classes such as Business, EmailSystem and CurrencyServer, will dynamically be loaded by the JVM. The rules of class loading dictates that the JVM will delegate class loading to the same class loader as loaded the parent of the class current being loaded. Consequently, if we modify JUnit so that it instead of its internal classloader uses an *instrumenting* classloader provided by us we will be in the very powerful position where we can modify every class and every method in the entire class tree. That is exactly what we want.



The instrumentation I had in mind was to change every single method in the loaded class tree so that running it becomes **conditional** and **dependent on a whether any substitutions exists for it** in a static structure carrying substitutions. Schematically something like this:



A *substitution* is simply an *alternative body* provided by the user anytime before the method is reached. This could be from a setUp-method in the test for instance.

So what will this give us? Actually it gives us the power of dynamically and declaratively replace – i.e. mock or stub – any method-body in the entire class tree without doing any refactorings. In other words it opens up our catch 22. Of course this might be a bad thing if misused by some lazy moron during TDD (morons don't do TDD anyway, do they?), but on the other hand a very powerful feature when working with untested legacy code.

Well, enough said about the envisioned inner workings – the technicalities above are nothing an end user will have to care about anyway. What's more important here is: *Did these visions ever concretise?*

Yes they did in a quick but promising spike. It was pretty straightforward to add a custom classloader to JUnit.

For instrumenting the classes I used Javassist. This very powerful class library makes high-level instrumentation, structural changes and bytecode engineering at classload-time a breeze.

For easy additions and removal of method substitutions I added two methods to the TestCase base class:

- `addSubstition(Class c, Object o)` - for telling JUnit that any method present in o should be called instead of the corresponding one in c.
- `removeSubstitutions(Class c)` - for removing all substitutions for class c

My result, which currently goes under the name JUnitAssist, is very straightforward to use. See this example:

```
public class StubbingTest extends TestCase{

   /**
    * Tell the class loader to pick methods from
    * MockEmailSystem that overrides methods
    * (signature wise) in EmailSystem.class
    */
   protected void setUp() throws Exception {
      addSubstitution(EmailSystem.class, new
         MockEmailSystem());
   }
   ...
}
```

There is nothing more too it than that (except of course implementing the MockEmailSystem class). Any method in MockEmailSystem that has the same method name and signature will be called instead of the corresponding one in EmailSystem. This dynamic override will also be possible to do for private, package or protected methods. You can of course also use anonymous substitution classes.

Try it out yourself by downloading JUnitAssistExample.zip. The zip contains two jars: junitassist.jar and javassist.jar and a few self explanatory examples showing how it works. Add the jars to the classpath to be able to compile and run the examples.

Of course there are still at least two issues to be resolved.

First of all some IDE:s have a "bootstrapping"-loader that might load the classes before they are loaded in JUnitAssist. This unfortunately means JUnitAssist never gets a chance to instrument the class and an already loaded class cannot be instrumented.

Secondly and probably a quite hard issue to resolve is how to add this quite powerful feature to a well established standard tool such as JUnit as non-invasive as possible. Simplicity and elegance go hand in hand which is why I would prefer to make a feature like this optional and pluggable. The authors of JUnit, Kent Beck and Erich Gamma, are currently working on the 4.0 release. How open it will be for plugging in this kind of behaviour I do not yet know. How open the authors are for a discussion about it I plan to find out. While I do that, you go ahead and find out what you think about the whole idea. ■ ■ ■

CHECK OUT JUNITASSIST

- http://www.citerus.se/pnehm/junitassist

Tobias Hill is a partner at Citerus who is committed to improve upon the ways software is built. Fuel his interest further by sharing your experiences with him at tobias dot hill at citerus dot se.