Test-Oriented Languages: Is it Time for a New Era?

Benjamin Stopford
The Royal Bank of Scotland
London, UK
e-mail: Benjamin.stopford@rbs.com

Abstract

More than a decade has passed since the advent of Test Driven Development and the introduction of the tools that facilitate its practice. However, it is our belief that we are nearing the limits through which functional decoration can aid the testing of current imperative languages.

This paper presents a *thought experiment* to explore improvements to the testability of current imperative languages. We use the guise of a hypothetical language, Quilt[1], to present one path that such a language might take. For brevity we retain the language constructs of current imperative languages like Java and C# and explore alterations in the compiler operation that make the language more test-oriented.

Quilt extends the Mockist Test Driven Development approach [2,9] by integrating the role of unit test isolation into the compiler. The application is split into a patchwork of independently testable units. However, unlike current Mocking frameworks [5,11,12], Quilt isolates through the provision of stub *Methods*, not Objects. Methods that do not return state (or mutate passed references) are automatically stubbed. Methods that do return state cause compilation failures if a stub has not been provided.

Through static analysis the compiler minimises the number of interactions that require isolation, reducing coupling between test and class (when compared to current testing practices). The effect is to significantly reduce the barriers to testing: Less test setup is needed, there is no need to inject dependencies for the purpose of testing and even preexisting code is easy to test[2].

We conclude that testing in current object-oriented programming languages is already largely incumbent and ultimately inevitable. However, the penetration of the Mockist approach has been limited somewhat by a high barrier to entry and adverse side effects experienced under certain conditions. We make a case for the value of unit test isolation and describe a mechanism for lowering this barrier for entry, reducing coupling issues, and generally making TDD easier.

Keyword - testing; programming language; test driven development

[1] The name Quilt alludes to it being a 'patchwork' of independently testable units.

[2] We are not advocating the practice of 'test last', we simply acknowledge that the practice of TDD is not for everyone (however much we would like it to be). We believe the language should facilitate testing regardless for your preference in this matter.

I. DEFINTITION OF TERMS

Some of the terminology in this field is overloaded so we define a few terms used throughout this paper:

- Stub: A test implementation, usually passed by interface, which provides pre-canned answers. It is used simply to isolate the code under test.
- Mock: Similar to a stub but allowing the tester to assert on whether the mock was called and in what way.
- Mockist TDD / Interaction Testing: This is the process of testing both the state an object changes to and the interactions that it makes. The approach is described in [9].

II. INTRODUCTION

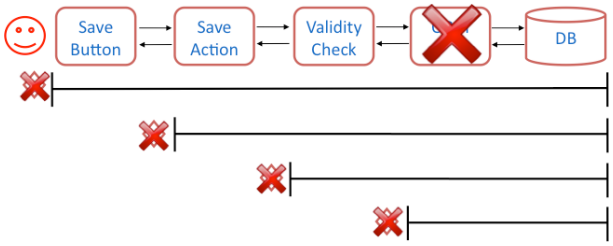
There have been a number of epochs in software development: Object Orientation (OO) took over from the Procedural paradigm as practices moved away from the writing of code towards the modelling of software as a cognitive artefact. The foundations for this shift were laid down by the development community as they attempted to promote reuse, increase encapsulation and model their software, pushing the boundaries of the procedural constructs they had. These ideas were extended and solidified through the creation of languages such as Simula and Smalltalk that actively engaged such tenets.

This paper investigates what may be a comparable shift in current OO programming languages as the community strives to embrace Programmer Testing. During the last decade or so the testing of software has gained prominence and has, for some, become a prime focus in the process of computer programming. Test Driven Development (TDD) is now a mature practice. It remains our contention that it is inhibited by the constructs of current mainstream Object Oriented languages. In particular Mockist TDD requires the application of strict practices [9] to avoid test code becoming highly coupled to the implementation. Whilst we acknowledge that such practices are beneficial to the program [9], the fact that the code must be written in a certain way to make it easy to mock makes the practice unsuitable for mainstream programming. This paper proposes a method for lowering the barrier for entry, making it easier for both new and experienced programmers to embrace testing. We posit that such a progression requires changes in the programming language, but believe it is a price worth paying

A. *The Motivations for Isolating the Class Under Test*

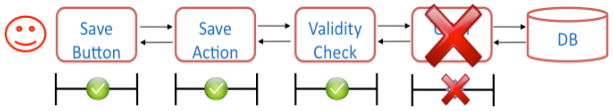
If an application is built without the use of stubs to demarcate the units under test, it will inevitably suffer from test case overlap: multiple tests will exercise the same sections of source code. This overlap of code sections from different tests degrades the feedback provided when failures occur: A single bug will manifest itself as a multitude of failures in the overlapping tests (Figure 1).

Figure 1. A single bug causing multiple failures in overlapping tests.



Using stub objects to isolate the code under test breaks the call stack into separate sections that are tested independently. This provides more accurate feedback on the location of the test failure (see Figure 2).

Figure 2. Each test isolates the code being tested using stubs. The same failure seen in Figure 1 causes only one test to break.



In addition, by segregating the code under test the scope of the problem is reduced to a more manageable size. This process of compartmentalization makes the software easier to develop and maintain.

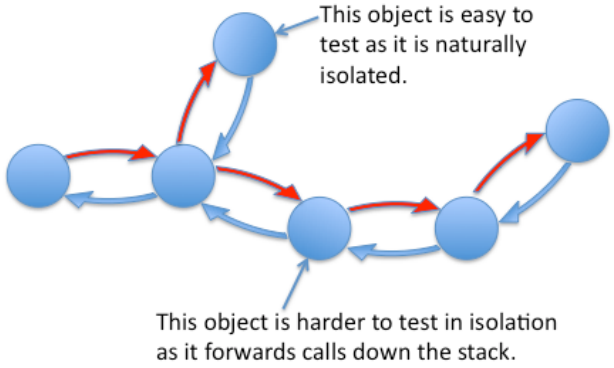
B. *The Motivations for Testing Interactions*

In traditional TDD, stubs are used to isolate the code under test, but stubs alone leave the developer with a problem: Some classes simply do not expose a change in state, making them hard to test using state-based assertions: there is simply nothing to assert on. An example of such a class is a Proxy [13].

One way to test such objects is to use an "active" stub, one that tracks calls made to it and exposes them to- be-asserted upon. The development of such 'active' stubs is the first step on the road to Interaction Testing.

Interaction testing (or Mockist TDD), as a testing methodology, goes far beyond the use of 'active' stubs; it is a change in the testing paradigm. Tests become about the interactions we expect an object to make rather than the changes we expect in its exposed state: When the Proxy is called we *expect* it to call the object it is proxying. This is demonstrated in Figure 3. The class at the top of the figure is suitable for state-based testing; it has no collaborators down the stack. The lower, mid-chain example interacts with classes further down the stack so interaction testing is likely to be more appropriate.

Figure 3. Representation of a call graph in a typical program showing the difference between a naturally isolated class that can be tested through its exposed state and one that requires interaction testing.



III. Problems Associated with the Testing of Current Imperitive Languages

A. *Problems with Coupling in Test Driven Code*

One of the key criticisms of Mock objects and Mockist TDD is the increased coupling between class and test [2,8]. Martin Fowler, for example states:

"I don't see any compelling benefits for mockist TDD, and am concerned about the consequences of coupling tests to implementation." [2]

The problems he refers to are real. Stub objects add an extra layer of coupling between test and implementation as they depend on more than the classes external interface, they couple to calls the object makes internally. Thus, should the inner workings of a class be changed the stub may need changing, even if the interface to the class and its behaviour remain the same.

Mock objects increase the coupling further by asserting on the specifics of the class's implementation: More specifically the interactions it makes with collaborating classes.

There have been successful attempts to reduce this coupling. The Mockito framework [5,6] represented a significant evolution over its predecessors by encouraging programmers to stub first, get the test running, and then layer expectations on top. This change does not physically reduce the coupling between test and implementation but it encourages programmers not to form unnecessary ones.

However a further problem exists: Current languages require the stubbing of objects that may not affect the output of the test, for example because they provide a reference that plays a role in the object's function but not one pertinent to the section under test. To address these a compiler change is needed. The Quilt compiler determines the minimum set of methods that require stub substitution at compile time. In this

way the number of stubbed methods can be reduced and hence so is the coupling between test and implementation.

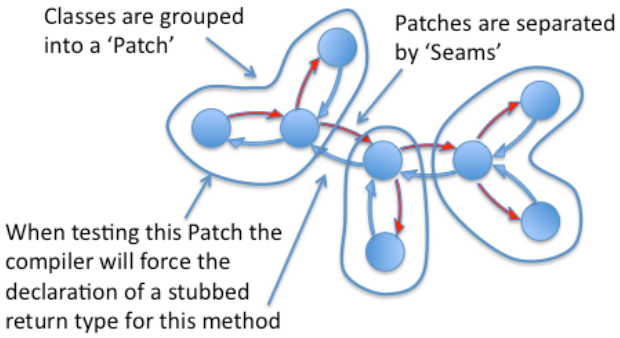
B. *The Testing Barrier: It's hard to test if you don't test first.*

There are a variety of things that a developer can do when writing "program first" that will make that code very hard to test in an isolated way. There are two common symptoms of this:

(a) The programmer must refactor the code to inject dependencies into the class so that test implementations (mocks, stubs etc) can be used instead of the real ones.
(b) Tests require a lot of setup code in which multiple stub objects, often unrelated to the test condition, need to be injected. The number of stubs that need to be created becomes disproportionate to the amount of code under test. The testing process then takes a disproportionately large amount of time. Also the larger number of stub objects ties the class and test together more tightly. As described in the previous section, this coupling makes the class very hard to refactor: The test code is brittle.

These problems can be addressed by writing well-formed Object Oriented code. One technique for doing this is "listening to the tests" [9] and rigorously applying the law of Demeter [10]. However significant expertise is needed to practice this technique. Quilt however makes it very easy to isolate the functionality under test, even in conditions such as these, helping to ensure that the barrier for testing is as low as possible.

Figure 4. A Quilt application, split into a patchwork of independently testable units composed of groups of classes.

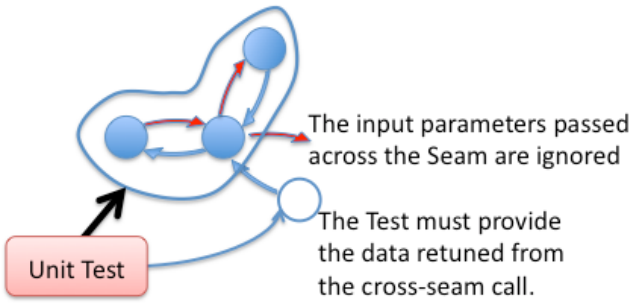


IV. What Makes Quilt Different?

(a) The compiler forces the isolation of testable units.
(b) Methods are isolated (stubbed), not Classes.
(c) Methods that do not return state are automatically stubbed. Methods that do return state, or mutate passed parameters, cause compilation failure if a stub has not been provided.
(d) The unit of isolation can be one or many Classes.
(e) There is no need to inject dependencies for the pupose of testing.
(f) The barrier for testing is very low: Even existing code is easier to isolate and test[3].

The Quilt application has two modes of execution: Unit tests, which test Patches of code in isolation, and Quilt tests that exercise the entire application. The arrangement of classes, grouped as Patches, and the Seams that separate them are shown in Figure 4.

Figure 5. Calls across Seams are stubbed by the compiler. If a return object is required the programmer must provide one in the test.



During compilation the Quilt compiler looks for methods that traverse Seams (the barriers between groups of independently tested classes). If such calls into different Patches return (or mutate) state then the Quilt compiler necessitates the programmer stubbing the method from inside the test. This is shown in Figure 5.

A. *Compilation Ensures that Stubs are Required only if they Affect the Output of the Test.*

Quilt ensures that a test and its corresponding class (or groups of classes, known as a Patch) are independently executable. That is to say that a class, or Patch, can be tested as an isolated unit, without depending on any other part of the code base. Quilt does this by isolating all calls that cross a Patch boundary (known as a Seam) to determine the minimum set of stubs required for the test to execute. If the method call *is deemed by the compiler to have an effect on the execution of the class under test* then the programmer is prompted with a compilation failure. This failure requires that a stub be created so that the test can run in isolation. The Quilt compiler is careful to only force the programmer to provide interactions that are absolutely necessary for test execution: Cross-Seam calls that affect state in a way that can influence the test output.

This process will likely seem familiar to those accustomed to Mocking frameworks and the practice of Interaction Testing [9]. However because the compiler is test-aware Quilt is able to reduce the amount of code (number of stubs) needed to make the test run in isolation. Reducing the number of stubs reduces the coupling between test and source making the language easier to refactor.

To demonstrate this, consider the Java code in Example 1(a). This code is tested using Quilt: Example 1(b), and Java/Mockito: Example 1(c).

[3] It should be noted that we are not advocating the writing of tests last.

Example 1(a): A class we wish to test in Pseudo Java (and Mockito)

```
class ConstructionSite{
  Digger digger = new Digger();
  Mixer mixer = new CementMixer();
  Foreman foreman = new Foreman();

  ConstructionSite(){}

  ConstructionSite(Digger d, Mixer m, Foreman f){
    digger = d;
    mixer = m;
    foreman = f;
  }

    boolean buildFoundation(Bricks bricks){

      Cement cement = mixer.mix();
      Foundation foundation = digger.dig();
      BrickLayer layer = foreman.getLayer();

      if(!cement.isSolid()  && bricks.size()> 100){
        Posts posts = layer.lay(bricks, cement);
        foundation.fill(posts);
        return true;
      }
      return false;
  }}
```

Example 1(b): A Quilt test for this class

```
shouldBuildFoundationsWithLotsOfBricksAndSlowDrying
Cement(){

   Seam:
    cement.isSolid() returns false;
    bricks.size returns 100;

   AssertTrue:
   new ConstructionSite().buildFoundation(..);

}}
```

This Quilt test does not require the setup of all dependent classes used by the class under test: Only ones methods that return state that contributes to the test output are required. In this case there are only the two of significance: the two inside the if-condition. However, if either of these methods are omitted from the test (i.e. the programmer does not provide either real or stub implementations), the quilt compiler will fail with a error such as "`Compliation Failure: Return value required for Seam transition cement.isSolid()`"

Implementing a similar test in a language like Java, as in Example 1(c), requires more code, and importantly more stub objects. Each stub object and method increases the coupling between the test and the implementation. Thus it should be apparent that the Quilt test has far less coupling to the code under test than it's Java counterpart.

The Quilt compiler works by evaluating whether each method call, which crosses a seam boundary, changes the internal state in a way that can affect the output of the test. This necessitates that the compiler execute bottom up.

In the above Example 1(a) the `foundation.fill(..)` method is analysed first (as it is at the bottom of the stack). The compiler deduces that it cannot affect the output of the test. This implies that the `posts` variable is also irrelevant and hence there is no need to consider the `layer.lay(..)` method either.

Example 1(c): A Java/Mockito version of the same test

```
@Test
shouldBuildFoundationsWithLotsOfBricksAndSlowDrying
Cement(){
  Digger digger = mock(Digger.class);
  CementMixer mixer = mock(CementMixer.class);
  Foreman foreman = mock(Foreman.class);
  Cement cement = mock(Cement.class);
  BrickLayer layer = mock(BrickLayer.class);
  Foundation foundation = mock(Foundation.class);

  when(mixer.mix()).thenReturn(cement);
  when(digger.dig()).thenReturn(foundation);
  when(cement.isSolid()).thenReturn(Boolean.FALSE);
  when(foreman.getLayer()).thenReturn(layer);

  ConstructionSite site = new
  ConstructionSite(digger, mixer, foreman);
  assertTrue(site.buildFoundation(new Bricks(101)))
}
```

Moving further up the stack, the compiler recognises that the methods `cement.isSolid(..)` and `bricks.size(..)` will affect the output of the test and hence the complier ensures that these are provided by either real objects (which must be in the same Patch) or through stub methods provided by the programmer in the test. If the programmer does not provide an implementation of these methods a compiler failure occurs. The compilation process is covered in more detail in Section VI(A)

B. Quilt Stubs Methods not Objects

Quilt stubs *methods* not objects. This avoids the need for stub objects to be created as part of the test. Only the methods need to be stubbed, and only if they cause a state change that affects the test output.

Example 1(b) includes a Quilt stub that must be declared as it returns state that affects the output of the test:

```
cement.isSolid() returns false;
```

No stub object is declared. The variable name, *cement*, provides a convenient way for the programmer to communicate the location of the method to be stubbed. The variable, *cement*, is defined in the program code only. The Quilt compiler makes reference to the scope of the program when compiling the test (if the variable name is ambiguous compilation failure occurs).

In this example only the *method* `isSolid` is stubbed. There is no need for the programmer to create the `Cement` object itself. There is no need for the programmer to create the `CementMixer` object either (which the cement came from). This is contrasted by the Java/Mockito version in Example 1(c), which needs these *objects* to be created in the test.

C. Quilt Avoids Mock Object Chains

As Quilt stubs method calls, object chains can be stubbed in a single line. For example consider the code:

```
A a = input.do();
B b = a.do();
String x = b.do();
```

In current programming languages isolating this code require the creation of three separate stub objects and the stubbing of each method individually. However in Quilt it can be stubabed in a single line:

```
input.do().do().do() returns “foo”;
```

It should be noted that such object chains are considered bad programming practice [10] and should be avoided. The feature is included in the language only as a result of the founding tenet: The language should make testing any style of code as easy as possible.

D. *The Unit Under Test Should Be More Than One Class*

One of the arguments levelled against the Mockist form of TDD is that it introduces too much coupling between test and implementation [2,8]. This has been noted elsewhere in the developer community [4] where developers report that the single “Test, Pass, Refactor” cycle inhibits the design process.[4] We have argued previously [3] that this coupling can be mitigated by isolating *groups* of classes to be tested as an autonomous unit.

Both class and test definitions in Quilt must be assigned to a Patch. This is loosely comparable to a package in Java or Namespaces in C# except that a Patch is a group of classes that will be tested as a single autonomous unit and the compiler forces isolation (through stubs) along Seams.

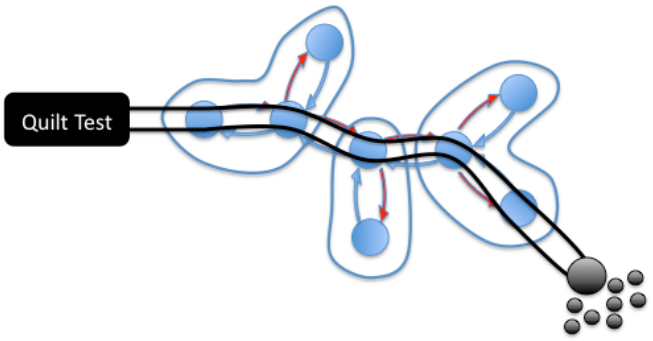
By increasing the size of the testable unit the ratio between classes and Seams is reduced, which in turn reduces coupling, making tests less brittle. Put another way, mocks and stubs always increase coupling (mocks more so than stubs) as they tie themselves to a facet of the classes’ implementation. By increasing the size of the testable unit (the number of classes being tested together), fewer stubs are needed to isolate the functionality and hence there is less coupling. This is the driver behind the concept of Patches being multiple classes in Quilt.

E. *Quilt Tests: Putting all the bits together in combined execution.*

Executing the various Patches in isolation is an important part of the development process and provides fast, accurate feedback on failures that may occur. However a set of tests that verify the behaviour of individual Patches will not ensure the correct running of the program as a whole. For this reason Quilt has Quilt Tests.

Quilt tests allow a set of Patches to be run in a single test with or without the use of external libraries as shown in Figure 6. By default execution of such tests will include external calls to external libraries. Quilt’s Aspect-like stub declaration model allows these to be overridden should they need to be. This model is similar to the provisioning of stubs for unit tests described in a previous section.

Figure 6. A Quilt test running a set of patches end to end. The black circles represent a library that is not part of the compilation unit but is exercised as part of this test



V. IMPLEMENTATION CONSIDERATIONS

Quilt can be implemented either as a stand-alone language or as an alteration to the compilation process of an existing imperative language such as Java. In the later case the compilation of program and tests would be segregated to facilitate changes to the test semantics.

A. *Quilt Compiler*

A key feature of Quilt is the compiler’s ability to determine the minimum user input required to isolate the code under test. To do this the compiler must perform Static Analysis to determine which Seam transitions can affect the output and hence require substitution (aka stubbing). This process is similar to Abstract Interpretation [1]. Full details of the implementation are beyond the scope of this paper but a short overview is given.

Data can flow across a seam boundary both through parameters (going outwards) and through return variables (coming inwards). The quilt compiler attempts to determine whether either outward or inward state is relevant to the test’s execution. The process includes the following steps:

- The compiler is executed against a single Patch. This being a group of related classes and tests that must be tested in isolation.
- The compiler lists all points at which program code is called from the test cases. It iterates through each one in turn.
- For each call it traverses the possible paths execution can take. If static state is used to drive the program from the test it can be used to reduce the possible execution paths.
- Every time execution leaves the scope of the Patch, through a method call that returns state, the call is flagged for substitution and the returned variable is flagged as being a ‘substituted variable’. The parameters passed are also flagged. This is a breadth first search.

[4] This is not the opinion of the authors. It is simply an example of TDD only facilitating a single style of development.

- The usage of each substituted variable in the Patch is analysed to see if it affects the test result. This is done bottom up.
- If a method that returns a substituted variable is found to not affect the test output it is ignored (no stub needs to be provided).
- The next level of the search identifies parameters passed to cross-seam methods to see if they are mutated. If they are, and the stub has not been provided in the test, a compilation failure occurs. This process continues to a maximum depth set in the compiler.
- The compiler has two modes of execution: Optimistic and Pessimistic. The optimistic model searches for the mutation of parameters passed to cross-seam methods and, on completion, assumes that no stubbing is necessary. Pessimistic mode assumes stubbing is necessary.

This type of breadth-first, bottom up evaluation of execution paths allows paths to be eliminated quickly if they do not affect the test output. A maximum depth is used to avoid a combinatorial explosion in paths. When the compiler reaches this depth it ceases further analysis.

It should be noted that a significant simplification could be made to the compiler method by forcing parameters to be immutable in the Quilt language, as is done in some functional languages. However we believe this would limit the applicability of the language.

VI. Conclusions

This paper explores how a relatively small amount of change could facilitate better program testing. Quilt represents an accessible, test-driven language designed to make testing as easy and pain-free as possible. The compiler significantly reduces coupling between test and implementation by not requiring the declaration of stubs that are not relevant to the test. The stub definition language is both terse and crosscutting. There is no need to inject dependencies for testing purposes as Methods are mocked, not Objects. Finally, Quilt encourages the testing of groups of classes as units rather than the traditional one-class-per-test paradigm, reducing coupling further. The result is a language that opens TDD to a far broader audience than traditional mock-driven methods can. It would be possible to implement Quilt as either a stand-alone language or as an addition to an existing compiler.

VII. Further Work

A. *An Empirical Study of the Benefits of Applying Execution Path Dependency Analysis to a Test Driven Code Base.*

Static code analysis of test and program code for a Test Driven code base should be used to determine the quantity of stub objects, and ensuing coupling, that would be removed through the use of Execution Path Dependency Analysis.

B. *A Fuller Description of the Asserting of Program State and Interactions.*

The description of Quilt has focussed on test isolation with only minimal consideration for how state and interactions are asserted upon.

C. *Consideration of the Concepts of BDD and in particular the Implementatinos of Rspec and Cucumber*

There are a number of current testing approaches that have not been explored. RSpec and Cucumber are testing tools derived from the Behavioural Driven Development (BDD) movement [7] and focus on tests as documents describing the behaviours of the system.

[1] On the Design of Generic Static Analyzers for Imperative Languages (TR), with Patricia M. Hill, Andrea Pescetti, and Enea Zaffanella. Technical Report. Quaderno 485 (2008), Department of Mathematics, University of Parma, Italy.

[2] Mocks Aren't Stubs, Martin Fowler, http://martinfowler.com/articles/MocksArentStubs.html

[3] Refactoring Tested Code: Has Mocking Gone Wrong? Benjamin Stopford, 1st REFTEST Network Workshop, Brunel University, London, 28th January 2010

[4] http://searchsoftwarequality.techtarget.com/news/1285151/Barriers-remain-for-test-driven-development

[5] Mockito: http://en.kioskea.net/

[6] http://monkeyisland.pl/2008/02/01/deathwish/

[7] http://en.wikipedia.org/wiki/Behavior_Driven_Development

[8] http://tech.groups.yahoo.com/group/easymock/message/1261

[9] Growing Object-Oriented Software Guided By Tests Steve Freeman, Nat Pryce, Addison Wesley 2009

[10] The Law of Demeter: http://www.ccs.neu.edu/home/lieber/LoD.html

[11] JMock http://www.jmock.org

[12] EasyMock http://www.easymock.org

[13] http://en.wikipedia.org/wiki/Proxy_pattern