Evolving an Embedded Domain-Specific Language in Java

Steve Freeman
M3P
London, UK
steve.freeman@m3p.co.uk

Nat Pryce
B13 Services
London, UK
nat.pryce@gmail.com

Abstract

This paper describes the experience of evolving a domain-specific language embedded in Java over several generations of a test framework. We describe how the framework changed from a library of classes to an embedded language. We describe the lessons we have learned from this experience for framework developers and language designers.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Software libraries;* D.3.2 [**Programming Languages**]: Language Classifications – *Specialized application languages;* D.3.3 [**Programming Languages**]: Language Constructs and Features – *classes and objects, frameworks.*

General Terms Design, Human Factors, Languages

Keywords Embedded Domain-Specific Language, Java, Mock Objects.

1. A Good Programmer does Language Design.

> *"[...] a good programmer in these times does not just write programs. [...] a good programmer does language design, though not from scratch, but building on the frame of a base language."*
> *— Guy Steele Jr. [13]*

Every program is a new language. That language may be confused and implicit, but at a minimum there will be conventions and programming interfaces that color the structure of the code. The art of writing software well is to tease out the concepts in a domain and make them concrete and tractable, to make the language within the program explicit.

1.1 Programs as language

Consider how experts talk to each other. As part of becoming expert, they will have acquired a dialect that succinctly expresses the concepts of their discipline. This allows them to make progress without explaining everything from scratch each time and to skip what is not important. This efficient communication is a form of Domain-Specific Language (DSL). It is specific and focused, and only applies to the context of its domain[1]

We can think of a program as embodied expertise, a concrete implementation of the understanding that a group of people has developed about an activity. The team members are experts in the application they are working on, and successful teams develop a shared language to talk about it. Well-written programs reflect this and their code expresses its behavior at the level of this shared language. As Abelson and Sussman [12] put it,

> "Expert engineers stratify complex designs […] The parts constructed at each level are used as primitives at the next level. Each level of a stratified design can be thought of as a specialized language with a variety of primitives and means of combination appropriate to that level of detail."

DSLs promise advantages over general-purpose languages. Not least, programmers tend to produce statements at the same rate whatever the language, so high-level languages are more productive [2]. Even so, few teams go to the trouble of writing their own DSL: it's hard to get right, it's another syntax to learn, and it can be expensive to maintain. The usual result is to give up and just write the code in one of the standard programming languages. Unfortunately, these do not support abstraction well and we end up with code that is full of noise about programming, not about the domain. It's as if experts were forced to communicate using lay terms, explaining every detail each time. As one programmer put it, "If I had a nickel for every time I've written 'for (i = 0; i < N; i++)' in C I'd be a millionaire" [14].

1.2 Embedded Domain-Specific Languages

In practice, however, no experts invent a new spoken language, their dialect is based on the same language that they use to buy groceries and read novels. The same can be done within programming languages, we can embed domain specific features into an existing programming language to take advantage of its implementation and tools — an Embedded Domain Specific Language (EDSL). Sharing an underlying language might even allow us to combine EDSLs within a program. This is straightforward, even standard practice, in "small syntax languages" such as Lisp, Smalltalk, and Haskell [6][7]. It's harder in large-syntax languages, such as Java and C#. These usually require techniques such as preprocessing or extending the compiler [11], which breaks some of the advantages of working with a host language and makes combining EDSLs difficult.

We think this is unacceptable, and this paper describes the experience of and lessons from jMock, one attempt at making a language explicit and embodying it in a framework. Through all the changes to the framework, we were very concerned with the

Copyright is held by the author/owner(s).
OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

[1] For example, Computer Scientists use "constraints" to find the widest range of possibilities; prison warders do the opposite.

readability of the code and its output. We wanted the code to be concise and self-explanatory; in this we were influenced by our experience in using dynamic languages, particularly Smalltalk. We also wanted the error messages to be obvious.

The step-change for us was when we realized we were embedding a language in Java, rather than just writing a framework. jMock's use of chained interfaces to define syntax is its most disconcerting feature and, we believe, one of its most effective.

The rest of this paper covers five topics: a brief introduction to jMock, the framework we developed; a history of the ancestor versions, showing how each one arose and what we learned from it; a deeper discussion of jMock, its structure and qualities; some design lessons from our experiences; and, finally, some conclusions.

2. JMOCK: A Language for mock objects

In a previous paper [4], we described jMock, a library to support Test Driven Development by making Mock Objects easy to create. Mock Objects are used to guide the design and test the implementation of object-oriented code. When using Mock Objects, an object is tested in isolation from the real objects with which it will interact in the production system. In a test, the object is connected to "mock" implementations of the interfaces that it uses. Those mock objects verify that the object calls them as expected and fail the test as soon as they detect an unexpected invocation or an invocation with incorrect parameters.

For example, if we were implementing a cache object we would want to ensure that the cached value is loaded only once. A test using jMock would have a `Test` containing a `Cache`, the object under test, and a `MockLoader`, which simulates the behavior of a real value loader. `MockLoader` implements an `ObjectLoader` interface. The `Test` sets up the `Cache` object by passing it a `MockLoader`.

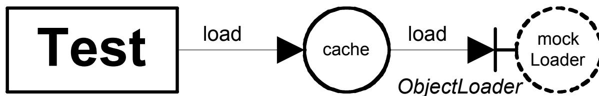

Figure 1. A Cache loading from an ObjectLoader

When the `Test` runs, it tells the `MockLoader` how it should expect to be called by the `Cache`, and then exercises the `Cache`. If the `MockLoader` is called incorrectly it will fail. Afterwards, the `Test` will verify the results and that the `MockLoader` has been called correctly. With jMock, the code might look like:

```
public void testDoesNotReloadCachedObjects() {
  mockLoader.expects(once())
    .method("load").with( eq(KEY) )
    .will( returnValue(VALUE) );

  assertSame( "loaded object",
              VALUE, cache.lookup(KEY) );
  assertSame( "cached object",
              VALUE, cache.lookup(KEY) );
}
```

The expectation line says that the Mock Loader is expecting the method `load` to be called exactly once with an argument equal to `KEY` and that it will return `VALUE`. The test calls `lookup` twice to show repeated calls.

jMock has evolved over several years from a primitive class library into a more complex framework. The driving force for the change was the need for clearer and more powerful specification of the expectations. As jMock evolved, its API changed from being an object-oriented library into what we now understand to be an embedded domain-specific language. The domain of that language is the specification of how objects should interact within a test scenario and the interpreter of the language is the testing framework itself.

The following sections describe the evolution of jMock, illustrating the forces that led to the development of an EDSL. To stretch a metaphor, we arrived at the current design through several generations of evolution. As we struggled with the limits of each implementation, the environment changed and we moved to new designs that were more effective. Curiously, some of our rejected designs survive in frameworks that were developed in isolation from the original team.

3. CAMBRIAN: Mockobjects.com

3.1 History

The first mockobjects.com [10] was an object-oriented framework to help with hand-coded mock object classes. The original concept arose out of several experiments, such as asking "what if we wrote code with no getters" and, after frustrations with testing against web servers, using IBM's VisualAge for Java to generate a stub implementation of `Servlet`. Practice and discussion within the London XP community clarified the ideas, and the mockobjects.com library was spun out from the code at Connextra Ltd., where most of the work was done. It defined core concepts such as `MockObject`, `Expectation` and `Verifiable` and provided a library of Expectation classes that allowed some flexibility in matching arguments beyond just equality. The design was essentially a refactoring of duplicated assertions from test cases into stub code.

3.2 Example

The mockobjects.com framework did not use dynamic proxies, so mock objects had to be created by hand, derived from a MockObject parent. We implemented a library of common expectations (value, list, and set) to compare expected and actual values during a test. The example below confirms that the `agent` buys a quantity of something via the `mainframe` and notifies `auditing` that this has happened. The test creates an agent, passing in mock implementations of the mainframe and auditing types.

```
public class MockMainframe extends MockObject
  implements Mainframe
{
  public ExpectationValue quantity =
    new ExpectationValue("quantity");
  public Ticket ticket;
  public void buy(long aQuantity) {
    quantity.setActual(aQuantity);
    return ticket;
  }
}
public void testBuysWhenPriceEqualsThreshold() {
  MockMainframe mainframe = new MockMainframe();
  MockAuditing auditing = new MockAuditing();
  Agent agent = new Agent(QUANTITY,
                          mainframe,
                          auditing);
  mainframe.quantity.setExpected(QUANTITY);
  mainframe.ticket = TICKET;
  auditing.boughtItems.addExpected(TICKET);
  agent.onPriceChange(NEW_PRICE);
  mainframe.verify();
  auditing.verify();
}
```

When the test calls the agent, the agent will call the `buy` method on the mock mainframe which, in turn, will tell the `quantity` expectation the value that has actually been passed in. The expectation will confirm that this value is expected and fail if not. Before finishing, the test must verify that each expectation has been met to catch missed calls (there is some reflective infrastructure in `MockObject` to simplify this).

The main benefit of this approach is that it is straightforward to code the simple cases. The expectation libraries worked well for testing equalities and generated good error messages, and it is easy to ignore parameters that don't matter. The main disadvantage is that the mock classes have to be written by hand, which breaks the momentum of the TDD process and clutters the code base. It could also be hard to specify multiple expectations on the same method.

The other disadvantage was that verification had to be called explicitly. Although this made the intention of a test explicit and we could implement stub behavior by simply leaving out the verification, it was prone to error by people forgetting to verify mock objects.

3.3 Experience

One of the striking features about this version was the influence of the development environment, in this case VisualAge which was the first Java IDE in which everything was "live". On the positive side, the Smalltalk-like environment with incremental compilation made both navigation and stub generation very easy, so there was no build cycle to slow the developer. VisualAge also worked well with the "endo-testing" approach. Failing an assertion from inside the target code meant that we could walk the stack in the debugger, find the problem, fix it, and continue.

On the negative side, VisualAge never implemented dynamic proxies, which meant that mockobjects.com could not exploit Java reflection and so stagnated. This limitation did, however, lead us in another useful direction. It made the cost of creating mock classes in a conventional style (with private fields and accessors) so high that we started to break the rules and concentrate on usability. We designed the Expectation classes to be readable when used in a chain of method calls, as in the line:

```
mock.boughtQuantity.setExpected(QUANTITY);
```

where `mock` is a mock object that has a public instance variable `boughtQuantity`. VisualAge's excellent code completion made this approach very easy under the fingers.

Ironically, we discovered that the best way to develop code that conformed to the Law of Demeter [9] was to have test code that violated it. We also discovered that it is worth putting a great deal of effort into making an API comfortable to work with and into generating good error reports.

Our biggest mistake was to attempt to provide common mocks for the entire Java API. First, this used up huge amounts of effort just maintaining compatibility with different versions of the JDK and J2EE. Second, the larger interfaces required enormous mock implementations to cover all the options. These were too large to understand, when all a test required was to verify a single method call. A worse issue, however, was it diverted attention from the most important use of mock objects which is as a *design* aid rather than a *testing* tool. Third-party APIs cannot be changed, so the tests do not drive their design. What we actually wanted people to do was wrap external types in objects that meant something in the domain of the program, using Mock Objects to help design those wrappers.

4. DEVONIAN: Early DynaMock

4.1 History

One of the authors wrote a Mock Object library for Ruby and found the advantages of dynamic types and complex argument matching to be overwhelming, so he ported his ideas back to Java. He wasn't using VisualAge so he could use the latest versions of Java with dynamic proxies, which meant that the DynaMock library could define a mock object and its expectations entirely within the code of a test. This streamlined the flow of test-driven development because it meant that programmers were no longer side-tracked by writing mock object classes. This development was in parallel with the mockobjects.com framework.

4.2 Example

Here, the `MockMainframe` class has been replaced by a dynamic proxy generated by the *Mock* class. Mock also includes methods for setting expectations and stubs (not shown here) on itself. Setting up the agent is now slightly more complex since the test has to specify the interface types and cast the result.

```
public void testBuysWhenPriceEqualsThreshold() {
  Mock mainframe = new Mock();
  Mock auditing = new Mock();
  Agent agent = new Agent(QUANTITY,
    (Mainframe)mainframe.createInterface(
        Mainframe.class),
    (Auditing)auditing.createInterface(
        Auditing.class));

  mainframe.expectReturn("buy”,
                         P.eq(QUANTITY), TICKET);
  auditing.expectVoid("bought",
                       P.same(TICKET));

  agent.onPriceChange(NEW_PRICE);

  mock.verify();
}
```

The major differences in writing tests were the use of strings to identify which method should be called ("buy" and "bought" in this example), and the introduction of Predicates to match parameters. `Predicate` defined a simple interface that received a value and reported whether it matched or not. In this example, the first expectation matches on equality and the second matches on identity. Predicates were later renamed to Constraints in response to user feedback; enterprise programmers were unfamiliar with the term "predicate".

Tests still had to verify each mock object explicitly.

4.3 Experience

This was our first attempt at "loosening up" our use of Java. DynaMock had a simple, imperative object-oriented API that let the user create expectations and stubs. Initially, the programmer could use the framework to define an expected sequence of calls that had to happen in the order specified. This proved too restrictive and the next release allowed expected calls in any order, but only let the programmer specify one set of constraints per method. The library only supported simple expectations and was not very extensible, so this basic API was sufficient.

DynaMock's main innovation compared to the mockobjects.com library and EasyMock [5] (another Mock Object library that uses dynamic proxies) was the use of arbitrary Predicate objects to match parameter values, not just Java equality. We had already

seen the need for extending matching with the mockobjects.com library and had sometimes implementing matcher objects that cheated by hijacking the `equals` method. Reifying the concept of Predicate meant that the programmer could define any type of match within a test, of which the most useful was substring.

In another early attempt at syntactic sugar, predicates were created by factory methods of the `P` class that had terse but readable names. For example, the `P.eq(QUANTITY)` clause above actually implements

```
public static Predicate eq(Object expected) {
  return new IsEqual(expected);
}
```

where `IsEqual` implements the interface `Predicate`. Programmers can create Predicate objects inline but this usually made the test too hard to read. We were not yet prepared to subclass JUnit's `TestCase`, so the best we could do was to reduce the clutter to "P".

The other syntactic trick was to use overloading to handle expectations with different numbers of arguments. For example, we had:

```
expectVoid(String name);
expectVoid(String name, Object arg1);
expectVoid(String name, Object arg1, Object arg2);
// and so on, until
expectVoid(String name, Object[] args);
```

This was just about manageable in this version but caused difficulties later when we introduced Java basic types, as will be discussed below.

The use of strings to describe methods was clumsy and is not handled by refactoring tools. In practice, however, we found that this did not cause major problems as method names are referenced by a limited number of tests; after changing a method name, the programmer runs the tests and fixes any broken ones. Sometimes it was even an advantage as it allows programmers to just type in a new method name without having to define it in the interface until they were ready. This approach was helped by the coding style encouraged by using Mock Objects, which makes dependencies as local as possible. In return, we found that the use of constraints to specify precise, flexible expectations more than compensated for the loss of flexibility caused by poor refactoring support.

5. JURASSIC: DynaMock rewrite

1.1 History

Over time, the inherent limitations of both mockobjects.com and DynaMock's became increasingly evident. By now, the community had moved off VisualAge to IDEs that supported Java 1.3 and hence dynamic proxies, so writing mock object classes by hand became less tolerable. Similarly, DynaMock was too inflexible with limitations such as only calling a method once. The community started an effort to rewrite DynaMock, keeping the same basic style of API but extending it to be more expressive and extensible, and to generate better failure messages.

1.2 Example

At this level, the test is very similar but there have been some improvements: instances of Mock are now bound to a mocked interface at construction; some of the Mock methods have been renamed to be more explicit, for example `expectReturn` is now `expectAndReturn`; there are methods for specifying stub behaviour (`matchAndThrow`) which can be mixed with expectations in the same test; and, `Predicate` has been renamed to `Constraint`.

```
Mock mainframe = new Mock(Mainframe.class);
Mock auditing = new Mock(Auditing.class);
Agent agent =
  new Agent( QUANTITY,
             (Mainframe)mainframe.proxy(),
             (Auditing)auditing.proxy() );

public void testBuysWhenPriceEqualsThreshold() {
  mainframe.expectAndReturn(
    "buy", C.eq(QUANTITY), TICKET);
  auditing.expect("bought", C.same(TICKET));

  agent.onPriceChange(THRESHOLD);

  mainframe.verify();
  auditing.verify();
}

public void testDoesNotBuyIfMainframeUnavailable()
{
  mainframe.matchAndThrow(
    "buy", C.ANY_ARGS,
    new NotAvailableException());

  agent.onPriceChange(THRESHOLD);
  mainframe.verify();
}
```

This test also shows a variety of Constraint, `C.ANY_ARGS` that will match on any argument value. This shows that we don't care what is passed in here because what matters about the test is the failure of the mainframe connection, the arguments are irrelevant.

As before, setting up mock objects was still clumsier than we liked and the test still had to verify them explicitly.

1.3 Experience

This version was intended to combine the precise control of mockobjects.com with the convenience of DynaMock. The most important change was that a mock object could expect the same method to be called more than once and with different arguments. Argument constraints and other rules were now used to *dispatch* invocations to expectations.

As the code evolved, it became more compositional. DynaMock turned into a high-level convenience API for specifying expected behaviour, layered above a framework. The framework implemented a test for that behaviour with pluggable objects that communicated through interfaces. Programmers could write their own implementation of these interfaces to extend the framework, adding new parameter constraints, matching rules, types of expectation, or stubs for the behaviour of mocked methods. These extension points were accessible through the high-level API.

Although we improved the API to be more consistent and to make test code read more like a specification, the simple, imperative style of the API became a problem. First, *Mock* had methods to define two basic types of expectation: `match` allowed but did not require the specified invocation, and `expect` required exactly one invocation (multiple calls required multiple expectations). There were variants of these methods for common stubbed behaviours, for example, `matchAndReturn` returned a given value after firing and `matchAndThrow` threw an given exception. To keep test code easy to read, the return variant had overloads for each of the primitive types so that the result of a mocked method could be specified as a literal in the test. We also kept the overloaded versions of the methods for up to four arguments to make it easy to specify expected argument values. This produced a

combinatorial explosion of variants and overloads; in essence, we were composing functionality in method signatures.

The result was a mess. First, additions to the DynaMock API required so much extra work that the implementation of new expectation types, such as `expectAtLeastOnce`, ground to a halt. Second, another set of overloads let the user leave out the argument constraints for an expectation. The intention was to make writing tests simpler, but the reality was confusing: did a missing argument specification mean that the expected method had no arguments, or that the mock ignored arguments? Third, code completion in the editor became unusable because the list of possibilities was so large. This was worse than it might sound because a significant requirement for the library was that it should *feel* comfortable to use; we wanted it to work well in IntelliJ.

Another weakness was that the extensibility hooks made tests difficult to read. Extensions to the API were not seamless: API calls that used framework extensions looked very different from those that used built-in DynaMock functionality. Finally, the generic, extensible dispatching algorithm ended up making failure messages harder to interpret not easier, despite the original goal of the rewrite.

In spite of its failings, DynaMock helped us to understand the structure of the domain. We had a reasonable implementation layer but a weak published interface. One symptom of this weakness was our choice of name for stubs: "match" describes the *implementation* of the framework not the intent of the test.

2. CENOZOIC: JMOCK

2.1 History

Clearly we needed a rewrite. We started work on jMock to clean up DynaMock. Our goals were to:

- make the API more self-consistent (some inconsistencies had slipped into DynaMock)
- improve the readability of test code
- reduce the size of the API to make completion in the IDE easier to use
- improve failure reporting
- allow the user to specify the partial ordering of expected method invocations
- add more expectation types: at least once, never, exactly, etc.

In the end, the underlying implementation did not change much, but we substantially reworked the public interface.

2.2 Example

We now specify expectations using a "call-chain" syntax which we will describe below. This gives a more declarative style of specification, built up from the component parts of an expectation. Creating and passing in mocks is still clumsy because of the type declarations, we have accepted this as a fundamental limitation.

```
Mock mainframe = mock(Mainframe.class);
Mock auditing = mock(Auditing.class);
Agent agent =
  new Agent( QUANTITY,
             (Mainframe)mainframe.proxy(),
             (Auditing)auditing.proxy() );
```

```
public void testBuysWhenPriceEqualsThreshold() {
  mainframe.expects(once())
    .method("buy").with(eq(QUANTITY))
    .will(returnValue(TICKET));
  auditing.expects(once())
    .method("bought").with(same(TICKET));
    agent.onPriceChange(THRESHOLD);
}
public void testDoesNotBuyIfMainframeUnavailable()
{
  mainframe.stubs().method("buy")
     .will(throwException(
        new NotAvailableException());
  auditing.expects(never()).method("bought");
  agent.onPriceChange(THRESHOLD);
}
```

We introduced a `MockObjectTestCase` class that extends JUnit's `TestCase` for two main reasons. First, mock objects are now verified automatically. Mock objects are created with a factory method *mock* which registers the new mock object with the test case. We have overridden the test case implementation to verify any available mock objects after the test method has run but before tear down. Second, to minimise syntax noise we moved the factory methods for constraints and other features into `MockObjectTestCase`. In the example, `eq` and `same` return constraint objects, and `returnValue` and `throwExpection` return method behaviour stubs.

2.3 jMock's call-chain syntax

As with the mockobjects.com library, we have found that the way to encourage good style in our production code is to break the rules in the test code—or at least follow a different set of rules.

The original intention was just to reduce unnecessary text by using something like Smalltalk cascades, a syntax for sending multiple messages to the same object:

```
anExpectation
  count: Once;
  method: 'buy';
  argument: (Quantity equalTo);
  result: Ticket;
  self.
```

Writing Java such as:

```
expectation.setCount(once());
expectation.setMethod("buy");
expectation.setArgument(eq(QUANTITY));
expectation.setResult(TICKET);
```

is just too noisy, so we had each method return the object itself to support cascade-like chaining. We quickly realised that most of the methods available at any given point in the set up of an expectation are not immediately relevant, which gave us the idea of limiting the options with interfaces.

Briefly, each method that defines part of an expectation returns an interface that can define the options for the next part; it's like a workflow defined in the Java type system. In our example, `mainframe` is of type `Mock`, `expect` takes a matcher (usually one that checks how often the target method is called) and returns a `NameMatchBuilder`. In `NameMatchBuilder`, `method` takes a constraint that identifies the target method (usually a `String`) and returns an `ArgumentMatchBuilder`, and so on. The chain looks like:

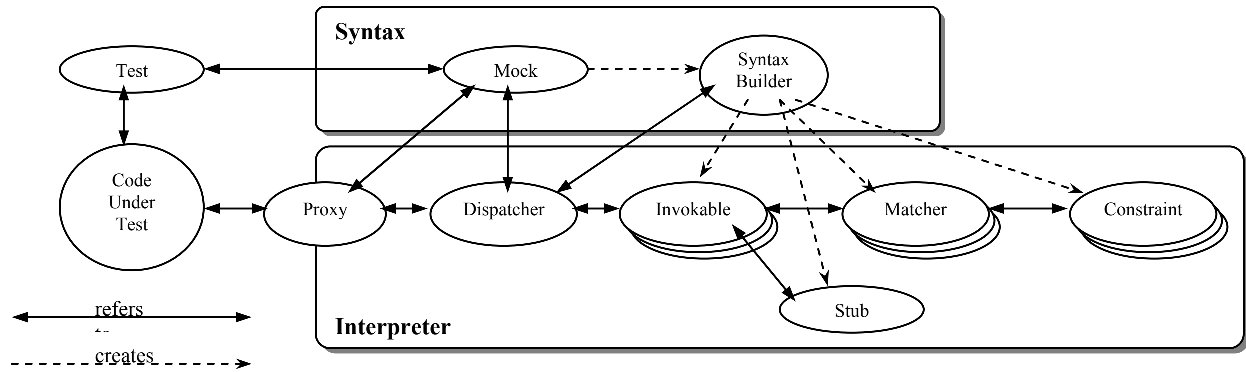

Figure 2. jMock Syntax and Interpreter layers

```
mainframe.Mock.expects(once())               →
NameMatchBuilder.method("buy")               →
ArgumentMatchBuilder.with(eq(QUANTITY))      →
StubBuilder.will(returnValue(TICKET));
```

The result is that the editor will prompt the developer with the available actions for each clause in the expectation.

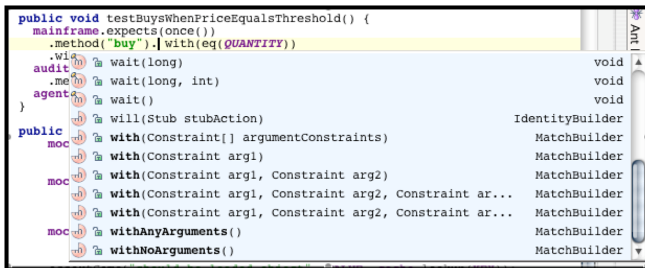

Figure 3. Code completion within an expectation

There are also interfaces for specifying additional constraints such as ordering between expectations. In addition, some of the interfaces extend each other to allow users to drop unnecessary clauses, for example:

```
mainframe.expects(once())
  .method("finish").after("start");
```

says that we expect the `finish` method to be called exactly once at some point after the `start` method. We don't care what arguments the `finish` methods takes, it's not part of this test. If it's important to specify a method that has no arguments, then we would write:

```
mainframe.expects(once())
  .method("finish").withNoArguments();
```

Under the covers, all these interfaces are implemented by an `InvocationMockerBuilder` class that gathers the arguments and constructs an expectation object.

2.4 Embedded language and core

JMock now consists of a "builder" layer, its public API, and an "interpreter" layer that runs the definitions built with the API.

The jMock interpreter accepts method invocations from a test and determines how to respond based on the expectations that have been set up by the builder layer. The appropriate response might be to fail an assertion, return a value, throw an exception, or invoke a user-supplied behavior. An expectation is stored as an `Invokable`, a combination of a set of `InvocationMatchers` (which decide if this `Invokable` will match the invocation), and a `Stub` (which implements any behavior we want the mock object to reproduce). Each `InvocationMatcher` matches a feature of an invocation: the method name, the values of the arguments, the number of times the method has been called, and so on. An `InvocationMatcher`, in turn, is usually implemented with `Constraint` objects which check incoming values, perhaps for equality or the presence of a substring.

When a test runs, it will trigger the target code to invoke a method on a mock object that is standing in for one of the target code's collaborators. The mock object dispatches to each of its `Invokables` in turn until one of them accepts the invocation as a match, at which point it will call the `Invokable`'s `Stub`. The test fails if there is no match. At the end of the test, all the `Invokables` are verified to make sure nothing has been missed. In practice, this verification is passed on to the `InvocationMatchers` as they will know whether they're missing an invocation.

In the syntax builder layer, calling `expects` or `stubs` creates a new `Invokable` within the mock object. The subsequent clauses in the expectation definition create `InvocationMatchers` in the new `Invokable` and populate them with `Constraint` objects and associated values to be matched. The `will` method assigns a `Stub` object to the `Invokable`.

5.1 jMock benefits

We have found some real advantages from jMock's peculiar builder syntax:

Orthogonality: each aspect of an expectation is handled separately which makes it easy to add new expectation styles and options. In retrospect, much of the design of the builder API came from simply removing duplication. This resulted in a huge improvement in maintainability since methods became focussed and simpler. We avoid the DynaMock explosion by having pieces we can compose together rather than trying to implement all the combinations as methods. This also helped with handling some of Java's quirks. All the method overloading to handle primitive types is now limited to a small number of clauses that can be used in several places.

Guidance: there are many software libraries where programmers are required to set up complex object state with no more guidance

than a group of setters. Given a statically typed language, we should exploit its features and tools. Combining type-chaining and IDE completion means that it's actually difficult to leave a complex object's state inconsistent. In addition, the orthogonality of the builder library means that the programmer is prompted with a manageable list of options when using completion in an expectation.

Seamless extensibility: we have defined intervention points throughout jMock to allow users seamlessly to extend the language. This means that users can make tests as expressive as possible, without breaking out of the EDSL. Programmers can inject their own implementations of components in the interpreter layer and use the same syntactic sugar techniques to extend the embedded language. This is discussed further in Section 6.4.

Consistency: each clause of the embedded language fits in exactly one place, so tests always look the same. This makes the tests easier to read, especially at a glance, even for more complex specifications.

Clear shorthand: dropped clauses default to weaker assertions. For example, if there is no `with` clause then the expectation ignores the parameters included in an invocation. This is simpler and clearer than the earlier use of overloaded methods.

5.2 Experience

Once we realised that the API had become a domain-specific language, development became more straightforward. Firstly, it encouraged a clean separation of the syntax and interpretation into layers. Secondly, it freed us from following the common Java coding style where appropriate. The interpreter framework is written as normal Java but we invented our own conventions for the builder level where the "call-chain" style made code read like a declarative specification instead of an imperative API. The result is striking if we remove the punctuation from the Java:

```
mainframe expects once method "buy"
  with eq QUANTITY will returnValue TICKET
auditing expects once method "bought"
  with same TICKET
```

Once again the development environment was a significant inspiration for some of the ideas. Using a modern IDE with good code completion, it seemed obvious to exploit the technology to guide the programmer to the next action.

We were surprised by some of the benefits of working with the builder syntax. For example, the ability to drop clauses to weaken an assertion meant that tests could focus just on the relevant criteria. Apart from making the intention clearer, this made refactoring easier as tests would not break when irrelevant features changed.

A further advantage we now have is that we believe we can superimpose different syntaxes over the same engine. We started writing an EasyMock layer for jMock and the results looked promising but we ran out of time.

One regret was the need to subclass `TestCase`, which makes it harder to integrate jMock with other frameworks that also extend `TestCase`. In our view, this shows that version 3 of JUnit, although wildly successful, is too closed. We couldn't do what we needed to without overriding part of the infrastructure.

6. How to write a language in Java (and C#)

> *"[…] like a dog's walking on his hinder legs. It is not done well; but you are surprised to find it done at all."*
> *— Dr. Johnson*

6.1 Introduction

We find that when we "refactor mercilessly" we end up reifying concepts that have previously been implicit within the methods of our classes. We discover logic duplicated among classes and move that logic into shared objects. In a well factored object-oriented program, the behavior of the *system* is an emergent property of the collaborations between simple objects. We now define our program's behavior by writing code to instantiate objects and plug them together, not by scripting it explicitly in imperative code. The collaborating objects have become an *interpreter* for a new, higher-level language and the setup code has become *statements* in that language.

This raises the level of abstraction at which we program, changing our programming language from being an imperative language, in which we script the exact behavior we want, to a declarative language, in which we state what we want to happen and let lower layers handle the actual interpretation of our wishes.

Throughout the history of programming there have always been languages that are more compatible with this approach, such as Lisp, Smalltalk, and Haskell. These languages make it easy to build up to higher level abstractions: few syntax rules; few keywords; little or no syntactic distinction between language, library and user program; lightweight syntax for anonymous functions that close over their lexical scope; syntactic macros; combinators; monads. In these environments the act of programming is to develop a language that describes the domain and then write the program in that language.

> "As well as top-down design, [Lisp programmers] follow a principle which could be called bottom-up design — changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. […] Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it." [6]

When programming in the usual commercial languages, such as Java and C#, just refactoring away duplication is not enough. The resulting code that creates the graph of collaborating objects does not clearly express the system behavior being defined. There are few features that let the programmer hide the implementation details and express only the higher-level concepts. The statements that instantiate objects and connect them with their collaborators have so much "administration syntax" that they obscure the intent of the programmer. What would be a single statement in a domain-specific language must be written as a list of `new` statements combined with calls to getters and setters.

In conventional languages we need an additional layer to help the programmer express intent — to provide a syntax for the interpreter our refactoring has produced. While writing jMock (Java) and NMock2 (C#) we have discovered some techniques

that can be used to write an EDSL in a language with heavyweight syntax.

6.2 Separate syntax and interpretation into layers

The code that defines the syntax of an EDSL will, by necessity, make quite unconventional use of the host language. If the implementation of the syntax is mixed with the implementation of its interpretation, the underlying framework will be hard to understand and maintain. Therefore, separate the two concerns, syntax and interpretation, into different layers. The interpretation layer should be an object-oriented framework implemented in the conventional style of the host language. The syntax layer can abuse the facilities of the host language as described below.

6.2.1 Use interfaces to define the syntax.

The grammar of the embedded language can be defined by interfaces. Each interface method defines a clause and returns a reference to the interface that defines the next clauses. For example, to define *method*, *with*, *withAnyArguments*, *withNoArguments*, *after,* and *will* as clauses in an EDSL, we declare these interfaces:

```
interface MethodNameSyntax {
  WithSyntax method(String name);
}
interface WithSyntax {
  OrderSyntax withAnyArguments();
  OrderSyntax withNoArguments();
  OrderSyntax with(Constraint c1);
  // etc.
}
interface OrderSyntax {
  StubSyntax after(String id);
}
interface StubSyntax {
  void will(Stub stub);
}
```

Chaining the interface types together ensures that the sequence of clauses must be:

- method
- with *or* withAnyArguments *or* withNoArguments
- after
- will

So the syntax will allow

```
mock.expects(once())
  .method("m")
  .withNoArguments()
  .after("n")
  .will(returnValue(20));
```

but not

```
mock.expects(once())
  .withNoArguments()
  .will(returnValue(20)) // out of sequence
  .method("m")
  .after("n");
```

6.2.2 Use interface inheritance to define optional clauses

To make a clause optional, define it in an interface derived from an interface that defines later clauses. For example, if we change `OrderSyntax` to extend `StubSyntax`, then we can call either `after` or `will` after any of the `with` clauses.

```
interface OrderSyntax extends StubSyntax {
  StubSyntax after(String id);
}
mock.expects(once())
  .method("m")
  .withNoArguments()      // no after() clause
  .will(returnValue(20));
```

Our experience is that allowing clauses to be optional makes jMock specifications easier to read. They only need to define the expectations that are relevant to a test; everything else can be left out.

6.2.3 Implement the syntax interfaces in Builder objects.

The syntax interfaces are naturally implemented according to the Builder pattern [3]. The Builder classes implement the syntax interfaces by having the syntax methods create and set up objects in the interpretation layer.

The simplest approach is to have a single Builder class implement all the syntax interfaces. Each syntax method returns the builder object itself as the next interface in the chain.

```
class ExpectationBuilder
  implements MethodNameSyntax, WithSyntax,
             OrderSyntax, StubSyntax
{
  private Expectation expectation;

  public WithSyntax method(String name) {
    expectation.setMethodNameConstraint(
      new IsEqual(name));
    return this;
  }
  public OrderSyntax withNoArguments() {
    expectation.setArgumentConstraints(
      new Constraint[0]);
    return this;
  }
  // etc.
}
```

6.3 Use, and abuse, the host language.

The conventions of the host language are unlikely to apply to an EDSL, given that the motivation for writing an EDSL is to overcome limitations in the host. To make the EDSL readable, it may need to break conventions such as capitalisation, formatting, and naming for classes and methods. One of the distinctions that encourages this practice is that EDSLs tend to be declarative while the host language is imperative.

In jMock, for example, one of the most startling practices is our extensive use of "train-wreck" statements[2]. We would normally regard this as very bad practice in object-oriented code because it violates the Law of Demeter, exposing the internal structure of objects and increasing coupling. Train wreck statements are, however, the only way we have found to emulate a new syntax in Java or C#. We limit their use to the syntax layer, which exists to abstract away the interpretation layer, so interface-chained code is not tightly coupled to any implementation details.

6.3.1 Implement a "container" to provide syntactic sugar for code in its scope.

Java code to create and set up objects generates a lot of syntax noise that is not relevant to the domain. To keep the EDSL readable and focussed, we use helper methods to clean up the

---

[2] An object-oriented "train-wreck" is a list of method calls chained together, often used to navigate a data structure, as in: `order.getParty().getAddress().getPhoneNumber()`

syntax for creating the initial builder object and other objects to be loaded into the interpreter. These helper methods must be defined in a scope that can be referenced by the code using the EDSL.

In jMock, for example, test fixtures extend `MockObjectTestCase` to inherit methods that specify expectations on a mock object. In this example, the `mock` method creates a `MockObject` and registers it to be verified at the end of the test. The other helpers create constraints and stubs to implement the expectation.

```
public class BuyerTest extends MockObjectTestCase
{
  void testAcceptsOfferIfLowPrice() {
    offer = mock(Offer.class);
    offer.expects( once() )
         .method("buy")
         .with( eq(QUANTITY) )
         .will( returnValue(receipt) );
    // etc.
  }
}
abstract class MockObjectTestCase
{
  Mock mock(Class mockedType, String roleName) {
    Mock mock = new MockObject(mockedType,
                               roleName);
    registerMockForValidation(mock);
    return mock;
  }
  Stub returnValue(Object value) {
    return new ReturnStub(value);
  }
  InvocationMatcher once() {
    return new CallCountMatch(1);
  }
  Constraint eq(Object value) {
    return new IsEqual(value);
  }
  // etc.
}
```

NMock-2, on the other hand, defines static methods of classes with names that work well when used in expectations. Whereas jMock uses a method `eq` to create an equality constraint, NMock uses the method `Is.Equal`, a static method named `Equal` of a class named `Is`.

We have whimsically termed this scope a "sugar bowl" because it contains the syntactic sugar of the EDSL.

6.3.2 Take advantage of host language features.

Not all features of heavyweight languages are a hindrance. C#, for example, supports operator overloading which has been used in Nmock-2 to define operators that combine constraints.

```
Expect.Once.On(mockLogger)
  .Method("LogError")
  .With( Has.Substring(USER_NAME)
         & Has.Substring("access denied") );
```

The equivalent statement in jMock is unwieldy and hard to read[3] because Java does not allow operator overloading:

```
mockLogger.expects(once())
  .method("LogError")
  .with( and( stringContaining(USER_NAME),
              stringContaining("access denied")));
```

[3] Except to lisp programmers

6.3.3 Appropriate intrusive syntax.

Sometimes the host language's intrusive syntax can be twisted into being useful for the EDSL. For example, this NMock-2 example specifies the order in which we expect an alarm clock object to call the Play method of a sound player: first the alarm clock will play the "on" sound, followed by "tick" and "tock" in either order, finally followed by the "alarm" sound. NMock-2 defines blocks of ordered or unordered expectations with the C# `using` statement.

```
Mockery mocks = new Mockery();
ISoundPlayer soundPlayer =
 (ISoundPlayer)mocks.NewMock(
   typeof(ISoundPlayer));
AlarmClock alarmClock =
  new AlarmClock(soundPlayer);
using (mocks.Ordered) {
  Expect.Once.On(soundPlayer)
        .Method("Play").With(ON_SOUND);
  using (mocks.Unordered) {
    Expect.Once.On(soundPlayer)
          .Method("Play").With(TICK_SOUND);
    Expect.Once.On(soundPlayer)
          .Method("Play").With(TOCK_SOUND);
  }
  Expect.Once.On(soundPlayer)
        .Method("Play").With(ALARM_SOUND);
}
```

We've learned to experiment and try different designs of our embedded language until we find the best fit between expressiveness and use of language features. Often simple changes to naming conventions can make all the difference. The name used for the `Mockery` above was chosen to make the following `using` statements clearly express their effect.

6.3.4 Tread a fine line.

As always, there is a balance to be struck when overloading operators or abusing keywords to create an embedded language. If in doubt, be conservative and define operators and keywords to have as close a meaning in the embedded language to that which they have in the host language or its standard library.

6.4 Don't Trap the User in the EDSL

DynaMock did not let programmers use the framework objects to extend the mock class; they were stuck with the constructs that we had implemented. We received a constant stream of feature requests which we could not fulfill because there were too many and, more importantly, because they were not generic enough to include in a shared library.

This taught us that even a simple DSL will not meet every user's needs; it must be extensible to be useful in practice. This is especially true of an *embedded* DSL where a key advantage is the ability to integrate code written in the host language. Given that we wrote an EDSL to provide the expressiveness that we could not get from the base language, we expect our users to need similar expressiveness in the EDSL.

There are two parts to making an EDSL extensible:

Extension points in the interpreter: most of the key objects in the framework are defined as interfaces and very few are instantiated directly. There is always a route for the programmer to substitute different implementations of components in the interpreter layer.

In the current version of jMock it is even possible to replace the way the dispatcher searches for a matching `Invokable`. Similarly, the arguments to builder methods are declared as interfaces,. The builder framework provides a veneer over these "low-level" features but still makes them accessible.

Seamless extensions to the language: the point of embedding a DSL is to let programmers clearly express their intent, which means extending the language into their domain. This will not work well if extensions look different from the rest of the DSL. The EDSL syntax should make no distinction between the built-in features of the language and those provided by the user to support their application, like the small-syntax languages we prefer.

Programmers can use the same syntactic sugar techniques as jMock by extending `MockObjectTestCase`. For example, if I want to ensure that the target code sends expired tickets to be cleaned up, I might write:

```
cleaner.expects(once())
  .method("remove").with(ticketExpiredOn(DATE))
```

where `ticketExpiredOn` is a sugar method:

```
public Constraint ticketExpiredOn(Date date) {
  return new ExpiredOn(date);
}
```

and `ExpiredOn` is an implementation of `Constraint` that returns true if it's passed an expired ticket. This expectation expresses exactly what I'm trying to achieve in the test in terms of the application domain, rather than in terms of dates.

The `ExpiredOn` class is an extension of the *interpreter* and `ticketExpiredOn` is an extension of the *language*. Both have been added with no change to the shared framework.

6.5 Map error reports to the syntax layer

Error reporting is critical to the usability of an EDSL. Errors are detected in the interpreter level but the user is programming to the syntax level. We cannot expect the user to translate errors back to the syntax level by hand; they may not even know how the internal features are implemented. Error reporting is hard but, in our experience, poor error reports will drive users away.

For an example of jMock error reporting, if we have not yet implemented the `Agent.onPriceChange` method, running `testBuysWhenPriceEqualsThreshold` would generate an error report (compressed for this paper format):

```
mock object mockMainframe:
  expected method was not invoked:
expected once: buy( eq(<1>) ), returns "Ticket"
```

This tells us that `mainframe` was expecting to be called in a certain way and that it did not happen. The last line describes the unfulfilled expectation and is close to the specification in the test:

```
mainframe.expects(once())
  .method("buy").with(eq(QUANTITY))
  .will(returnValue(TICKET));•
```

We can also report if a method is called incorrectly. Imagine that the `Agent.onPriceChange` method corrupts the ticket. Running the test will produce an error report like:

```
mockAuditing: unexpected invocation
Invoked:
 mockAuditing.bought("icket")
Allowed:
 expected once: bought( same("Ticket") ), is void
```

This tells us that someone has incorrectly called `bought` on the `Auditing` object with an argument of "icket". What we actually wanted was to call `bought` with a given `Ticket` object.

We cannot hard-code this kind of error reporting since we do not know how the framework will be extended by its users, so we require that all objects in the interpreter can describe themselves. We have an interface `SelfDescribing` that all the core objects must implement. For example, the `ExpiredOn` constraint might implement the interface with:

```
public StringBuffer describeTo(StringBuffer buf) {
  return
    buffer.append("is expired on ")
          .append(this.date);
}
```

When an assertion fails, the runtime visits the objects with a Java `StringBuffer`, collecting a description of the current state which it then shows in the error message.

This is essentially the same requirement as allowing users to extend the framework. Consistency and readability is critical, so users need the programming hooks to make any extensions they write indistinguishable from core features in error reporting.

7. Conclusions

> *MONSIEUR JOURDAIN "Well, what do you know about that! These forty years now, I've been speaking in prose without knowing it!"*
> *— Molière, Le Bourgeois Gentilhomme*

This paper describes our experience of developing an EDSL in Java. We worked with the concept of Mock Objects over several years, absorbing ideas from many different sources, such as Ruby and even IntelliJ. The various generations of the library were picked over at length, mainly by members of the London XP community.

Throughout this process we kept an absolute commitment to maintaining readability and consistency; we wanted the library to work well "under the fingers" in a modern IDE. The lesson we learned is never to skimp on this aspect of a framework, ever. We also try to apply the same rigour to the unit tests we write using jMock. We also learned that the heuristics that we've learned for writing good Object Oriented code do not always apply to EDSLs.

Another lesson is that there is no higher art in writing software than finding and reifying implicit concepts. Once we had a way of describing the communications between objects, which is at the core of our approach to Test-Driven Development, new ideas about the design clicked into place.

7.1 On the limits of EDSLs

We don't know how practical it is to scale up to a complex EDSL. The visible part of the jMock syntax has six builder interfaces, and one test class. Immediately beneath that are twenty six constraints, ten matchers, and eight stubs; these can be thought of

as the built-in features of the jMock EDSL. This is not a large language — which is a good attribute of a DSL.

One of the authors worked on a project which wrote an EDSL for a much larger domain, with mixed success. The straightforward cases worked well, new developers said that they found the EDSL code easy to work with and even the Business Analysts could understand it. Complex cases were more difficult to express and we started to create special extension objects to handle them. These were, in effect, very powerful verbs in the language, which kept the domain code compact but made it less obvious to read.

What is not clear from this experience is whether the difficulties were in the approach or its implementation. jMock is the result of several years experience whereas the project in question suffered from the usual commercial time pressures. We only implemented sugar methods and an interpreter layer, we did not clean up and implement the chained builder interfaces. The result was rather like working in a procedural rather than an object-oriented language. In retrospect, it might have been better to spend more time earlier working on the language syntax, but there is no evidence either way.

7.2 On host programming languages

Extending Java has at times been a frustrating experience. Java has several features that very much helped us: garbage collection, interfaces, a base object type, reflection, and dynamic proxies. Java has some other features that made our task needlessly difficult. The worst is basic types (such as `int`) which required us to add lots of overloading. We would also like to find an easy way to refer to methods, which should be possible since they're statically compiled. In comparison, C# has more features which gives us more options when defining the EDSL (such as the exploitation of the `using` clause) but requires more effort to implement the extra edge cases in the interpreter. Also, the IDEs available for C# have not yet caught up to those available for Java.

Ironically, there have been antecedents for this approach since the first structured languages. Alan Kay [8] cites

> "a little-known syntactic variant in the Algol 60 official syntax that encouraged a more readable form for made-up procedures. This allowed a comment in a procedure call to be replaced by the following construct:
>
> ```
> ): <some comment> (
> ```
>
> and this would allow [for (i, 1, 10, print(a[i]))] to be written as follows [...]:
>
> ```
> for (i): from (1): to (10): do ( print( a[i] ))
> ```
>
> which looks a lot like the Algol base language but done as a meta-extension by the programmer for the benefit of other programmers."

In jMock, we appear to have invented keyword-based messaging once again.

On the whole, it's too hard to extend conventional host languages, the syntax and the low-level operations get in the way. We look forward to a new generation of language, such as Fortress [1], that explicitly address our needs.

8. Acknowledgments

Our thanks to Tim Mackinnon, Keith Braithwaite and the eXtreme Tuesday Club.

We gratefully acknowledge the support of Imperial College London through EPSRC grant GR/R95715/01.

9. References

[1] Allen, E., Chase, D., Luchangco, V., Maessen, J., Ryu, S., Steele, G., Tobin-Hochstadt, S. *The Fortress Language Specification*, Sun Microsystems.

[2] Brooks, F. P. 1995 *The Mythical Man-Month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc.

[3] Gamma et. al. 95 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, Boston, MA, 1995

[4] Freeman, S., Mackinnon, T., Pryce, N., and Walnes, J. 2004. Mock roles, not objects. In *Companion To the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, BC, CANADA, October 24 - 28, 2004). OOPSLA '04. ACM Press, New York, NY, 236-246. DOI= http://doi.acm.org/10.1145/1028664.1028765

[5] Freese, T. EasyMock. At: http://www.easymock.org

[6] Graham, P. 1993 *On Lisp: Advanced Techniques for Common LISP*. Prentice-Hall, Inc. p. 3

[7] Hudak, P. 1996. Building domain-specific embedded languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996), 196. DOI= http://doi.acm.org/10.1145/242224.242477

[8] Kay, A. *The Future of Programming As Seen from the 1960s*. Foreword to Ducasse, S. 2005 *Squeak: Learn Programming with Robots (Technology in Action)*. Apress.

[9] Lieberherr, K., Holland, I., and Riel, A. 1988. Object-oriented programming: an objective sense of style. In *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications* (San Diego, California, United States, September 25 - 30, 1988). N. Meyrowitz, Ed. OOPSLA '88. ACM Press, New York, NY, 323-334. DOI= http://doi.acm.org/10.1145/62083.62113

[10] Mackinnon, T., Freeman, S., and Craig, P. 2001. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, G. Succi and M. Marchesi, Eds. The XP Series. Addison-Wesley Longman Publishing Co., Boston, MA, 287-301.

[11] Mernik, M., Heering, J., and Sloane, A. M. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37, 4 (Dec. 2005), 316-344. DOI= http://doi.acm.org/10.1145/1118890.1118892. Also CWI Technical Report, SEN-E0309, 2003.

[12] Norvig, P. Finding and Reusing Programmer's Work, *Proceedings of Object World Conference*, January 1994. Also at http://www.norvig.com/ow.ps

[13] Steele, G. L. 1998. Growing a language. In Addendum To the 1998 *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Addendum) (Vancouver, British Columbia, Canada). J. Haungs, Ed. OOPSLA '98 Addendum. ACM Press, New York, NY DOI= http://doi.acm.org/10.1145/346852.346922

[14] Vanier, M. On http://www.paulgraham.com/quotes.html