Hammock 2.1 Internals

Test Doubles for Java ME

Carl Meijer

June 2009

1 Hammock Internals

This document gives an overview of the code structure of Hammock. Hopefully this will make it easier to understand the code if you want to modify it, extend it or use the classes effectively. The JavaDocs are the definitive guide to the API; this document provides more of a high-level view to see what classes exist and how they collaborate.

1.1 Core Classes

The core classes of Hammock fall into five categories as shown in figure 1:

- Test double classes (classes that implement `IMockObject`).

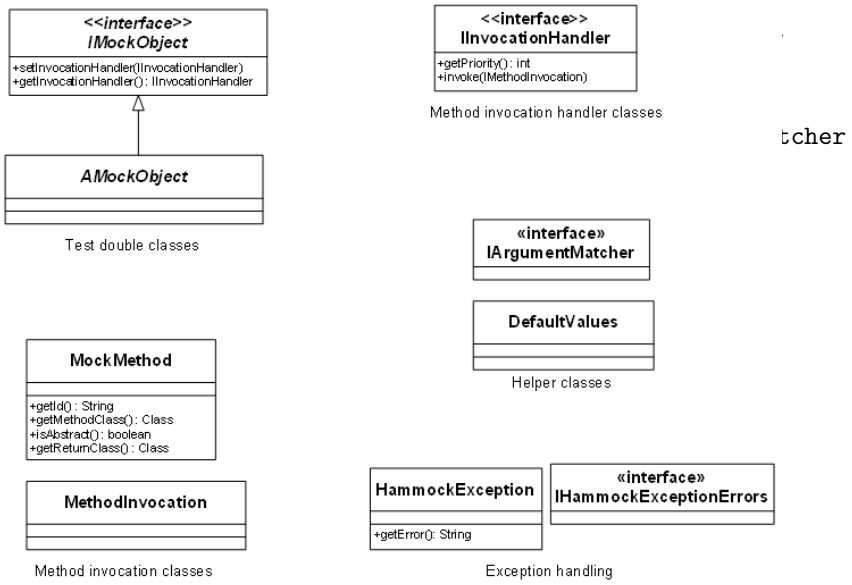

Figure 1: Core Hammock classes

1.2 Test Double Classes

Test double classes are classes that implement the `IMockObject` interface. The `AMockObject` abstract class (see figure 1) implements the methods of `IMockObject`. If a test double mocks an interface, it can extend `AMockObject` rather than implementing `IMockObject`'s methods explicitly. For example, `MockHttpConnection` implements the `HttpConnection` interface and extends `AMockObject`.

Test doubles can be configured to behave like mocks or spies by injecting an appropriate *method invocation handler* via the `setInvocationHandler()` method or the test double's constructor.

1.3 Method Invocation Handlers

A test double must respond to method invocations in some configurable manner. It is possible to implement `IMockObject`s that respond *directly* to method invocations, but that is not the Hammock idiom. Instead test doubles are associated with method invocation handler; the handler is configured to respond to method invocations or to provide some default response for method invocations that haven't been explicitly configured. When a method is invoked, the test double queries its invocation handler how to respond. The advantage of using invocation handlers is that the code for all test double classes remains the same irrespective of whether our test doubles behave like spy objects or mock objects; instead we only need to switch a single instance of our method invocation handler to change the behavior of our test doubles.

Method invocation handlers implement the `IInvocationHandler` interface. As shown in the UML class diagram of figure 2, Hammock provides three (concrete) implementations of the `IInvocationHandler` interface:

- `Hamspy`,
- `Hammock`, and
- `MethodHandler`.

The `Hamspy` invocation handler allows test doubles to behave like spy objects similar in spirit to Mockito's test doubles. The `Hammock` class provides mock object behavior similar to that provided by EasyMock and jMock.

Google have released a program, TestabilityExplorer, that determines how difficult a piece of software is to test using a metric known as "cyclomatic complexity". According to this measure the `MethodHandler` class is the most complex of the three invocation handlers with a score of 31 (the scores for `Hammock` and `Hamspy` are 19 and 26 respectively). While the `MethodHandler` class is comparatively complex, it is also very limited; a `MethodHandler` can only be configured to respond to invocations associated with one method. The `Hamspy` and `Hammock` classes can be configured to respond to invocations of more than one method using the `setExpectation()` and `setStubExpectation()` methods. The `setExpectation()` methods, though, return instances of `MethodHandler`. So, while it would be unusual to explicitly set a `MethodHandler` as the invocation handler for a test double, the `MethodHandler` class is an essential class.

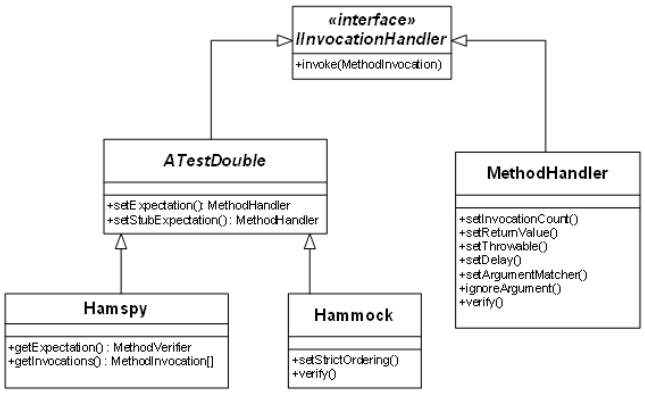

Figure 2: Method Invocation Handlers

1.4 The MockMethod and MethodInvocation Classes

Java SE has a `java.lang.reflect.Method` class to identify a method of a class. Java ME does not support reflection and there is no inherent language support for identifying methods. The `MockMethod` class encapsulates the attributes of a method of a test double. An instance of `MockMethod` has a number of attributes that may be retrieved via various getters:

- An identifier which is a `String` representation of the method's name and arguments.
- An associated class; the class that the method belongs to.
- Whether the method is abstract or concrete.
- The classes of the arguments passed to the methods.
- The class of the return value (which is `null` for a method of type `void`).
- The classes of the exceptions that the method can throw.

The `isAbstract()` method allows the `Hamspy` method invocation handler to determine how to respond to a method invocation if no expectation has been set. For example, if an expectation has not been set but a method is not abstract, then the processing of the method invocation can be delegated to the superclass rather than responding with some canned response. The `getNumberOfArguments()` and `validateArguments()` methods are used for verifying that expected arguments are consistent with a method's signature. Similarly the `validateThrowable()` method checks that the method signature allows the throwing of a particular exception.

The `MethodInvocation` class (see figure 3) depends on the the `MockMethod` class. Since test doubles do not respond directly to method invocations, a test double needs some way of telling its invocation handler about the method invocation. The details of the method invocation are passed using a `MethodInvocation` instance. The important attributes of an invocation are: the method (represented as an instance of `MockMethod`), the arguments passed to the method (which is an array of `Objects`) and the test double that was invoked (an instance of `IMockObject`).

If an invocation handler knows how to respond to a method invocation it will set a return value or an exception for the method invocation. If a return value

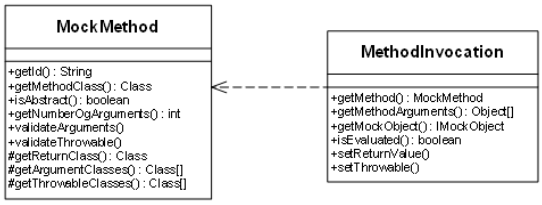

Figure 3: MockMethod and MethodInvocation Classes

or exception has been set for a `MethodInvocation`, the `isEvaluated()` method will return true. The `ATestDouble` invocation handlers use the `isEvaluated()` method to determine whether any of the `MethodHandler`s they created (via `setExpectation()`) were able to evaluate a method invocation. If not, they respond appropriately (for example, the `Hammock` class will throw an exception indicating that a method was unexpectedly invoked).

1.5 Helper Classes

1.5.1 Argument Matching Classes

When a method is invoked on a test double it is essential to be able to verify that the arguments passed to the method are as expected. It is also useful to be able to modify mutable objects (especially arrays) that are passed in a method invocation. For verifying arguments and modifying mutable objects, instances of the `IArgumentMatcher` interface can be used. Hammock comes with five classes that implement the interface (see figure 4):

- `DefaultArgumentMatcher` that is used if no other argument matching is specified.
- `PromiscuousArgumentMatcher` whose `areArgumentsEqual()` always returns `true` and is used if `ignoreArgument()` is invoked on a `MethodHandler` or `InvocationVerifier`.
- `ClassArgumentMatcher` that returns `true` if the actual argument is the same class as (or a subclass of) the expected argument class.
- `NotNullArgumentMatcher` checks that the actual argument is not `null`; any expected argument specified is ignored.
- `PopulateArrayMatcher` that can be used to populate an array when passed to a method invocation.

The `DefaultArgumentMatcher` class is the most complex of the argument matching class. In fact, according to the TestabilityExplorer, it is the most complex Hammock class with a cyclomatic complexity of 41. This complexity arises from the fact that the class doesn't simply test for equality using `equals` but, in the case of arrays, will check whether an expected and actual array have the same dimensions and, if so, whether their individual elements are `equals`.

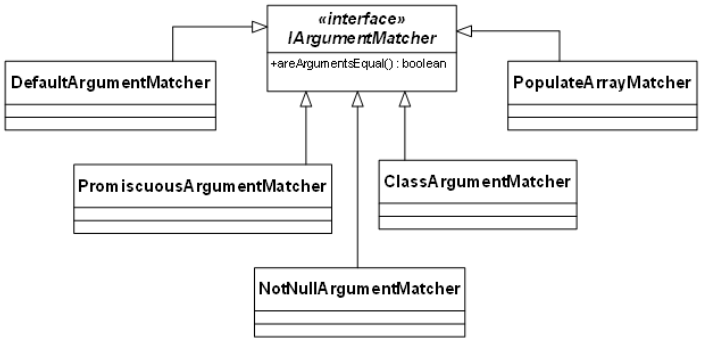

Figure 4: Argument matching classes

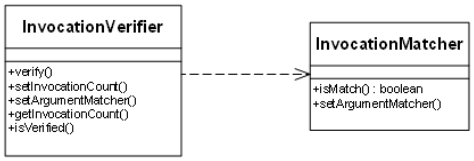

Figure 5: InvocationVerifier and InvocationMatcher Classes

1.5.2 The InvocationVerifier and InvocationMatcher Classes

When using spy objects with the `Hamspy` handler, one will normally, after exercising a class under test, verify that certain methods were invoked with particular arguments. This can be done by (tediously) retrieving the method invocations from the `Hamspy` instance. It's easier to use an instance of `InvocationVerifier` that compares actual and expected arguments. The `InvocationVerifier` class is shown in figure 5. This helper class provides methods for specifying how many times a method should have been invoked and a setter method for supplying argument matchers.

The `InvocationVerifier` class uses the `InvocationMatcher` class to verify whether an actual and expected method invocation are equal (via the `isMatch()` method). The `InvocationMatcher` class is also used by instances of the `MethodHandler` class to determine whether they should process a method invocation.

1.5.3 The Default Values Class

The `DefaultValues` class is used by the `Hamspy` class to return a default value when a method is invoked on a spy object but no expectation has been set. For example, `false` is returned for a `boolean` method, zero for a numeric type and `null` if the return type is an `Object`. The CLDC 1.1 distribution will return 0.0f or 0.0 if the return type is `float` or `double` respectively.

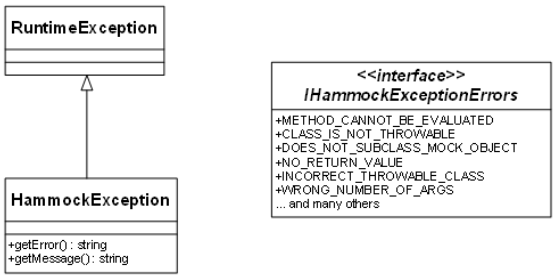

Figure 6: Hammock Exception Classes

1.6 Exception Classes

The `HammockException` class is an unchecked exception (see figure 6) that is thrown by the Hammock framework when things go wrong. Examples of problems are

- A call to `verify()` on a `Hammock` instance failed because a mock object wasn't invoked as expected.
- An expectation was set that a method would be invoked with an invalid number of arguments (the number of arguments passed in the expectation is inconsistent with the method's signature).
- A method handler was invoked and needed to return a value but no return value was specified in the expectation (via the `setReturnValue()` method).
- A method handler was instructed to throw a checked exception which is inconsistent with the signature of the method.

Examining the `IHammockExceptionErrors` interface is instructive in learning what can go wrong.

The `HammockException` class exposes two methods to get a text string describing the particular exception. The `getError()` message returns a succinct error message as defined in `IHammockExceptionErrors`. The `getMessage()` method provides a more detailed message that includes the name of the class that threw the exception.