# Mock Object Patterns

**Matthew A. Brown**
**&**
**Eli Tapolcsanyi**

**Version 1.3.2**

**5/19/2003**

***"You take the red pill,***

***you stay in Wonderland, and I show you how deep the***

***rabbit hole goes."***

***-Morpheuas, The Matrix***

*[Author's note: The Mock Object pattern paper is going to be a 2 year PLOP paper- We found a good deal of material in the area, and did not have time to write about all of the patterns that we found.]*

## Introduction

### Background: JUnit

This paper is concerned with solving some of the issues encountered when unit testing with the JUnit( for java) or NUnit (for .Net) testing frameworks. JUnit's primary goals are as follows:

- Unit test code in isolation.
- Make these tests easy to use and easier to write.
- Leverage these tests wherever or whenever you can. Run these tests frequently. This means having the ability to run them in an automated fashion.

With JUnit, a test class is written for each class that requires unit testing. These test classes implement the TestCase interface of the Junit framework. Each of these test classes has various assertions that are tested against, to see if the code passed unit test. These assertions allow for the code to be tested automatically. As part of the TestCase interface, a "setup"method is called as well, to allow for initilialization of resources before the test.

For more information on Junit, please go to www.junit.org. For the remainder of this paper, we are assuming that the audience has a grasp on the concepts and implementation of Junit.

## Background: Mock Objects

My son Mark likes to build with Legos. He can spend hours constructing castles, cars, alien spacecraft, or a police station. He goes about his work, and without even knowing it, is testing constantly. He'll test to see how the parts fit together, or see how easily they fall apart (usually via some sort of alien attack). He is not aware that he is testing, but through his play, he is doing it. He knows how to test, but sometimes he doesn't know how to test things the "best way." For example, if he's building a battery powered Lego car, it might be difficult for him to test whether the power unit works without actually hooking up the wheels/drive train and seeing it work.

For a 9 year old, his testing methods are perfectly fine, and especially when the systems he is building are Lego based. An improvement he could make is to test the power unit in isolation, usually via some sort of power tester. That way, he can defer his choice of options on how to put the car together. Furthermore, he would be able to isolate what he's trying to test. Testing the power unit by running it with the wheels and drive train could result incorrect conclusions. If the wheels and drive train are not attached properly, he might come to the incorrect conclusion that the power unit doesn't work. The difficult part of testing, whether it be Legos or a Nuclear Reactor, is that it can be difficult to test only one feature at a time. . The power tester is serving as a "stand in" for the actual drive train mechanism. The tester is built to check, or assert that the power unit is exerting a charge, and nothing more.

In the programming world, its difficult to test database driven applications without actual an actual database and related schema setup. It's the same issue as the Legos, and the solution is the same. You can test through the database through validating the data returned from the database (assuming you've already set the database and schema up). However, just like in the Legos, your testing is not isolated. To solve this issue, we can use the equivalent of the power tester, or a "stand in," that is built for testing the requirements you want, in isolation. Furthermore, it is difficult to write unit tests that validate on a changing system, where the data is constantly fluctuating. Moreover, the new distributed systems should be able to run on any kind of database, as databases become more commoditized in capabilities. Accordingly, an attempt needs to be made to test the application isolated from the database.

*"An essential aspect of unit testing is to test one feature at a time; you need to know exactly what you are testing and where any problems are."(Mock Object Paper).*

The solution is a Mock Object. The main objective of Mock Objects is to ensure unit tests are done in true isolation. Today's distributed systems make it difficult to be isolated, as objects are constantly interacting with servers or other components distributed far and wide. Whether messaging channels or database connections, these objects do not allow the unit test to occur in isolation. Alastaire Cockburn wrote of unit tests:

*"In Software, the trick is to fake communication against the outside world, then run the tests locally. Then your testing is partitioned"*

Mock Objects are a way of "faking" this communication with the outside world. Since Mock Objects are at the core, Objects(with a capital "O"), there are many strategies that can be employed to fake the ObjectToTest is communicating with the outside world. To *Morpheuas*'s point, the rabbit hole is pretty deep. This paper will present some of the strategies that we've found, for utilizing Mock Objects. Mock Objects truly are like a chapter out of "Alice In Wonderland" or "The Matrix", as you are faking reality for the ObjectToTest.

## Patterns Catalog

The following table lists the patterns provided, with a brief description of each pattern:

| ***Pattern Name*** | ***Synopsis*** |
|---|---|
| MockObject | Basic mock object pattern that allows for testing a unit in isolation by "faking" communication with collaborating objects. |
| Test Stubs | |
| MockObject via Factory | A way of generating mock objects, utilizing existing factory methods. |
| Self Shunt | Unit Test code serves as the mock object by passing an instance of itself. |
| Pass in Mock Collaborator | Pass in a mock object in place of the actual collaborating object. |
| Mock Object via Delegator | Creates a mock implementation of a collaborating interface in the Test class or mock object. |

*The following patterns will be added next year for 2004 PLOP:*
*-Mock Objects via CrossPoints*
*-Write Testable Code*
*-Mock Object with Guard*

# Title

Mock Object [See Mock Objects web site www.mockobjects.com]
Original Research Paper/Pattern is available at the mockobjects web site.

# Problem

We are assuming that the readers of this paper have an understanding of Junit, the java unit testing framework. For more information, see www.Junit.org. Furthermore, it would be to the readers benefit to have an understanding of MockObjects as well. For more information on Mock Objects, go to www.mockobjects.org.

Junit is a remarkably powerful unit testing framework, in that it helps to build highly leveraged, easily written unit tests. Some forces that we can consider with Junit testing:

- Independent testing of classes is desirable because of better test coverage
- Dependencies between classes prevent them from being tested independently
- It is desirable to test that classes correctly call other code
- It is desirable to test that classes handle failures and exceptions correctly that occur further down the call chain
- Test code must be simple and maintainable
- It is desirable to have the ability to test many types of error conditions without going through the trouble of creating errors in your actual domain code.
- The whole test suite must not take much time to run
- Interaction with external resources (remote systems etc.) can take much time.
- Having a structured approach to confirming and verifying test results is desirable.
- It is desirable to avoid duplicating validation logic in the tests.

During our experience with distributed systems, the following observations can be made, that directly impacted our ability to use Junit:

- Systems use databases to persist data.
- Systems use "middleware" or an application server to handle transactions and requests.
- Buying and incorporating components is common place in many systems. "Plug and Play" component architectures make this very easy.

Given JUnit's objectives, in light of the realities of distributed systems, and forces to consider, the following problems occur:

- How can code that accesses an outside database or application server be tested in isolation? The database or application server exposes the unit test to all kinds of outside factors that could affect the results of the test. Factors would include "bugs" in application/database server or network communication issues (bottlenecks). How can we limit variability in our testing environment?
- How can unit tests be run frequently or whenever its necessary, without having to worry about the application/database server being up and running?
- How can we unit test code written by outside vendors, when source code is not available or changeable?

- How can we ensure that our Junit test has a structured method for confirming and verifying test results?
- How can we test for error conditions without having to implement errors in the testing environment?

## Solution

Call other classes only through interfaces. For testing purposes, provide specific 'mock' implementations that you call instead of the live implementations. These mock objects should perform only cheap operations and contain test code that allows the test to inspect the calls or inject faults and exceptions. Lastly, these mock objects should have a self-validation mechanism built in.

Benefits of Mock Objects:

- Unit Tests are more localized.
- Unit Tests are independent of each other.
- Infrastructure choice can be deferred.
- A structured method for setting and confirming testing expectations is provided. Expectations are set when initializing Mock Objects, and are confirmed during test execution. Expectations are applied to both the return values from method calls, as well as the number of times a specific method is called. Mock Object's have the ability to validate themselves.
- All test validation logic is within the Mock Object, which prevents duplication of this logic throughout the Junit test code.

Some liabilities that come with Mock Objects

- More code. Mock Objects require that the real object be emulated, usually through the implementation of an interface. Implementing these interfaces could result in lots of "dummied up" methods, as not all of the interface's methods may even be used. Furthermore, more code also means there is a greater chance that errors/bugs may be introduced, producing side effects in our tests.
- Added complexity. The whole point of Junit is to make testing fast and easy. Mock Objects can add complexity by having to implement a "fake" structure.
- Mock Objects need to be synchronized with domain code. This synchronization has a cost.
- At times, making calls through an interface is not possible. Depending on the structure of the called code, interfaces might not be a viable option. Purchased components may provide no capability to alter how object's are called.
- Changing existing code to accommodate mock objects adds additional risk that errors or bugs will be introduced into code that was stable.
- Testing is too narrow. Mock Objects cause your tests to be very specific as to what is being tested. If the implementation changes in the domain code, the Mock Object's narrow scope can cause the test to become obsolete very quickly. [http://saloon.javaranch.com/cgi-bin/ubb/ultimatebb.cgi?ubb=get_topic&f=68&t=000020- Nicholas Lesiecki].

# Implementation

As we worked with Mock Objects, we found that our unit tests developed a common format :

- Create instances of Mock Objects
- Set state in the Mock Objects
  - Set any parameters or attributes that could be used by the object to test.
- Set expectations in the Mock Objects
  - Setting expectations in Mock Objects is where the desired or expected outcome is set. This includes the number of method calls and the return values of Mock Object invocations.
- Invoke domain code with Mock Objects as parameters
  - Now that all of the expectations have been set, use the Mock Objects within the domain code.
- Verify consistency in the Mock Objects
  - A common practice within Mock Object testing is to implement a "verify" method, which is called as a final step in the test to verify that the expected outcomes match the actual.

[www.mockobjects.com]

The following issues need to be considered during implementation:

- Accessing Collaborators can cause issues. Many people have complained that to accommodate mock objects, they needed to open access to methods to allow for testability. Some of the strategies listed in this paper can help prevent the need to open access.
- Use of interfaces. Mock Objects are easier to implement through utilizing interfaces. Being aware of the "testability" of the objects can result in implementation code that is refactored into interfaces. Interfaces
- Mock Implementations can be difficult to initially setup. Depending on the setup, the implementation can have a steep curve. However, the payoff can be great, as reuse of Mock Implementations can be high [http://www.c2.com/cgi/wiki?MockingLegacyCode].
- Assertions need to be mapped to possible objects, method calls, or end state(s) in the real collaborator. A full and robust set of assertions is required for the mock objects to be effective at testing. This includes:
  - Number of method calls(method specific)
  - Expected Return Value

## Title

Test Stubs[placeholder for Test stubs]

## Problem

## Solution

## Implementation

# Title

Self Shunt

# Problem

See Mock Objects Pattern for more information on what Mock Object's are. This pattern uses the Mock Objects pattern as a foundation.

Mock Objects can be used to build repeatable, automated, and highly leveraged Unit Tests. In many cases, setting up Mock Object frameworks that "emulate" the "real world" objects is necessary. However, if the value derived from creating Mock Objects for Test Classes does not outweigh the effort required in coding and maintenance on those Mock Objects, a different approach should be explored.

Forces at hand when implementing Mock Objects:

- Mock Objects should be simple.
- Mock Objects should not add significant time for testing.
- Mock Objects must be simple and maintainable
- Mock Objects should be implemented quickly.
- In a given situation, agility or speed in software development can be highly desirable.

A Mock Object like test is desired, without creating lots of extra Mock Objects and "dummied up" code. How can we accomplish this objective?

# Solution

Call other classes only through interfaces. For testing purposes, implement the interfaces within the test class and provide specific 'mock' implementation. When the 'mock' object call is made, pass a reference of the test class in, instead of an actual mock object.

Self Shunt is a way of creating a pseudo Mock Object for testing, without requiring that Mock Objects be created. Furthermore, it requires little extra coding, and can provide the needed object to allow for testing. A Self Shunt is akin to a LoopBack adapter in the networking world, where you run decoupled, and it defaults to a fixed IP address and routes that to some internal code.

The resulting class from using the Self Shunt is a mock object implementation without creating an "explicit" mock object (as the test class is the mock object). The object under test has a way to talk to itself properly when disconnected from the actual server. This is provided without any additional classes. Ideal for non-complex interface implementations, Self Shunt can be quicker to implement than other patterns.
Benefits for using Self-Shunt

- Code is tested in isolation. As the Self Shunt is a "stand in", much like a Mock Object, it allows the test to be performed in isolation.
- Less code. A formal Mock Object structure does not need to be created.

- Allows for speed through being able to implement all Test Code in one place, the Test Case.
- Real Object is guarded. By passing in a reference to itself, the Self-Shunt class ensures that the real object from the domain is not actually being called.

Liabilities for using Self-Shunt:

- Expectations are not used. Since Self Shunt is a pseudo mock object, it does not actually have the expectations setup within it. Consequently, verifying the testing results is more difficult.
- Behavior and responsibilities of Test Class are too broad. Leads to a "bloated" code base.
- Self Shunt avoids implementation of Mock Object interfaces, which are constructed explicitly for testing. Avoiding this implementation permits less structure and discipline in the test code.
- Increases complexity of test code. Accordingly, increases probability that errors will be introduced into the test code.
- Constrained to using interfaces. Building object structures with a layer of abstraction, such as with interfaces, is generally a good idea. However, there are many cases where using interfaces is not possible. This constraint limits the use of Self Shunt.

# Implementation

Consider the following issues when implementing Self Shunt:

1. Self Shunt is ideal for creating mock collaborators during Test Driven Development(TDD), where speed and agility in development is paramount.
2. Test Class must conform to implemented interface. Since the test class, and not a separate mock object, implements the collaborators interface, the Test class will be a mixture of test and mock object code.
3. Test Class should be monitored to ensure that the number of methods implemented does not become too large.

Sources

**The 'Self'-Shunt Unit Testing Pattern**
**Michael Feathers**
**Object Mentor, Inc.**
mfeathers@objectmentor.com

# Title

Pass in Mock Collaborator

# Problem

See Mock Objects Pattern for more information on what Mock Object's are. This pattern uses the Mock Objects pattern as a foundation.

Mock Objects are effectively a "stand in" for the actual collaborating resource/object, whether its a database, a file server, or a message queue. Once the Mock Object "stand in" is established, Unit Testing will benefit, as there will be less variability in test results and more certainty that the tests are truly testing as expected. The challenge of using Mock Objects is in using them without requiring changes in implementation code.

The following forces impact our unit tests:

- Loose coupling of business logic from native resources, such as database and file systems, is desirable.
- Encapsulating the creation and management of collaborating resource within an object structure separate from the domain code is desirable.
- "Polluting" domain code with test code is not desirable.
- If the logic for calling the mock object code is buried within domain code, there is a risk that the "real" objects could be called accidentally. For example, in a banking application, a test could accidentally call a real ATM object versus a MockATM object.
- Having code that is easily tested is desirable.
- Modular code is typically desirable. Modular code allows for greater reuse.

Some issues that we saw in our use of Mock Objects:

- How can a Mock Object be used without resorting to including a reference to it in the actual implementation code?
- What if we want to test 3rd party components are being tested, where the source code is not provided?
- How can we write our implementation code to be more "testable"?

# Solution

Avoid a strong coupling of collaborating objects with domain objects. Pass in a reference to the collaborating object parameterized method calls. Extend or implement the collaborator object's interface within a Mock Object(See Mock Object Pattern). During testing, pass in the reference to a mock object instead of the actual collaborator object.

Much like in Visitor, the impersonator object passed in contains various operations that the domain class has no knowledge of, thus preserving encapsulation of the objects. Furthermore, as in the Visitor, the impersonator can easily have operations added to it

without having any effect on the class using the Visitor. Even without the source code, you can impersonate an object through extending the class or implementing an interface. The following benefits are associated with this pattern:

- ObjectToTest is significantly more testable, as a Mock Object can be passed in.
- ObjectToTest more closely follows the Law Of Demeter, which emphasizes reduced coupling and increased cohesion, by removing the creation and management of collaborative objects from within the actual class.
- Actual Collaborator objects are guarded from being called during the test, as there is type safety introduced using this pattern.

The following liabilities are associated with Mock Object Collaborators:

- Existing code may be not allow Collaborator objects to be passed in.
- Change of existing interfaces to existing callers of the ObjectToTest would introduce changes to stable code, possibly de-stablizing the code.
- Implementation code must be written with testing in mind.
- For existing code, re-writes would need to be done. Many times this isn't possible.
- Collaborator might not allow itself to be extended by a Mock Object.

# Implementation

The following steps can be taken to implement visitor:

1. Construct a UnitTest class that will be used to create the actual implementation object.
2. Construct an ObjectToTest class. Using the Law of Demeter, design your method calls to permit easy testing.
3. Create a Collaboration Object. The Collaboration object could be a database connection, a Http Servlet Request Object, or any other type of object that might be self managed.
4. Create a Mock Collaboration Object. The Mock Collaboration Object could either extend from the actual Collaboration Object or implement a Collaboration Interface.
5. Modify the Unit Test to allow the Mock Collaboration Object to be passed into the ObjectToTest.

# Title

Mock Object via Delegator

# Problem

See Mock Objects Pattern for more information on what Mock Object's are. This pattern uses the Mock Objects pattern as a foundation.

Mock Objects can be used to build repeatable, automated, and highly leveraged Unit Tests. In many cases, setting up Mock Object frameworks that emulate "real world" objects is necessary. However, if the value derived from creating Mock Objects for Test Classes does not outweigh the effort required in coding and maintenance on those Mock Objects, a different approach should be explored.

Let's say that we want to create a Mock Object for testing Java Messaging Messages. With the javax.jms.TextMessage interface we could create a Mock Object that looks identical to a TextMessage. The downside is that we would need to implement over 50 methods, many of which would go unused. So how can you get the best of both worlds, where you only implement the methods you want to test with, and still have the ability to do "Mock Object" type activities?

The following forces need to be considered in light of this problem:

- Unit Tests should be kept simple
- Unit Tests should be quick to implement
- Unit Tests should not rapidly expand the code-base.
- Modular code is desirable.
- Mock Objects should include methods that require testing, but should not include methods that are never called.
- Abstracting common object structure into interfaces is desirable.
- It is desirable to code to an interface over coding to an implementation.

# Solution

For most collaborating objects, code to an interface not an implementation. Implement a Delegator class( one is available at `http://www.ejgroeneveld.com`) which effectively allows the calling class to implement an interface at runtime instead of having the test class implement the collaborator interface directly. In the Junit test class, create collaborator objects through using a Delegator class. Within the Junit Class, implement the required methods from the collaborating object's interface. With a Delegator, there is no need to "dummy up" methods that you do not need in your Mock Object. Also, Delegator ensures that you do Mock methods called in the ObjectToTest on the Mock Object. Much like a Self Shunt, but without all of the hassle of having to implement all of the methods.
The following benefits are accrued by using this pattern:

- Less Code. Instead of needing to implement a given interface through a Mock Object directly, the Delegator can create an instance of the interface being tested.
- Only the methods actually called by the code being tested will need to be implemented.
- Delegator enforces that called methods be implemented. The Delegator will throw an exception if the methods in the interface that are called are not implemented in the test class.
- Mock Object Delegator helps with creating mock objects with only the methods needed, and still ensuring that the methods called by the ObjectToTest are included.

Liabilities are as follows:

- Expected method call and return value logic is not "built in" to the mock object class. Weakness is that you must implement expected method call logic, expected value logic, and other types of logic for testing the object.
- Increased complexity of code. Implementing a Delegate is complex. The Delegate class depends upon the Java reflection API, which adds complexity.
- Reflection API performance is poor.
- Security Access for some classes may need to be changed to accommodate the use of Delegator's, as the methods will be called through the reflection API.
- Limitations exist on the reflection API.(see implementation section on specific list of limitations).
- Solution is limited to objects that implement interfaces.

## Implementation

Consider the following issues when applying the Mock Object Delegator pattern:

1. Mock Object Encapsulation vs. Test Class Serving as Mock. In the structure diagram provided, the assumption is that Test Class is "doubling" as the mock object. Much like a Self Shunt pattern, the methods are co-located with the unit testing methods. However, the case could be made that the mock object implementation should be encapsulated within an actual mock object, rather than a Self Shunt.
2. All standard proxy limitations apply[JavaWorldNov2000]
    a. The array of interfaces passed to the proxy constructor must not contain duplicates of the same interface. Sun specifies that, and it makes sense that you wouldn't be trying to implement the same interface twice at the same time. For example, an array { IPerson.class, IPerson.class } would be illegal, but the code { IPerson.class, IEmployee.class } would not. The code calling the constructor should check for that case and filter out duplicates.
    b. All the interfaces must be visible to the ClassLoader specified during the construction call. Again, that makes sense. The ClassLoader must be able to load the interfaces for the proxy.

c. All the nonpublic interfaces must be from the same package. You cannot have a private interface from package com.xyz and the proxy class in package com.abc. If you think about it, it is the same way when programming a regular Java class. You couldn't implement a nonpublic interface from another package with a regular class either.
d. The proxy interfaces cannot have a conflict of methods. You can't have two methods that take the same parameters but return different types. For example, the methods public void foo() and public String foo() cannot be defined in the same class because they have the same signature, but return different types (see *The Java Language Specification*). Again, that is the same for a regular class.

The resulting proxy class cannot exceed the limits of the VM, such as the limitation on the number of interfaces that can be implemented.

# Title

Mock Object via Factory

# Problem

See Mock Objects Pattern for more information on what Mock Object's are. This pattern uses the Mock Objects pattern as a foundation.

We need to code and maintain test objects that depend on complex system, business, or third-party objects. This can become difficult if not impossible based on complex object availability. In some cases the "real-world" objects cannot be available or we cannot control their availability i.e. third-party object, object not implemented yet, or infrastructure not available.

Mock Objects provide a manner of localizing unit tests and removing this complex object dependance. However, implementing complex logic via a Mock Object can be laborious. It may require the developer to write a sizeable amount of code. Furthermore, if the complex logic being called changes, the developer must maintain the mock objects accordingly.

Forces impacting unit testing:

- Isolating or refactoring code to be tested from complex logic is desirable.
- Encapsulating the "real-world" complex object and Mock Object creation via factory is desirable.
- Keeping all Mock Object references out of "real-world" object implementation is desirable.
- Implementing a Mock Object is necessary when the complex logic is not available during testing.
- Calling "real-world" complex logic may have undesirable impacts such as web service which decrements/increments bank account or replenishes inventory with new order in such cases a Mock Object is necessary.
- Eliminating infrastructure dependencies via Mock Objects is desirable.
- The Mock Objects implementing complex logic must be kept as simple for maintenance purposes.
- Quick Mock Object creation is desirable.
- Mock Objects should be used if the benefits gained outweigh the cost of time in creation and maintenance.

# Solution

Implementing a Mock Object via factory allows us to remove the "real-world" complex object availability requirements when performing unit testing. We isolate the ObjectToTest from complex object via refactoring. Refactoring the complex object logic allows us to provide a mock object for the complex object. Extracting the creation of the

complex object into a factory allows us to encapsulate object creation and removes requirements of a secondary creation method. This can accommodate the "real-world" versus Mock Object instantiation by checking a server parameter startup parameter or system property, reading from a configuration file, or passing the creation method a parameter.

From the perspective of the ObjectToTest, the actual complex object implementation is completely unknown. We have abstracted the details of creation and implementation. The factory creates and manipulates the correct complex object, whether it is a mock implementation of the "real-world".

Benefits associated with factories creating Mock Objects:

- ObjectToTest becomes easier to maintain and test.
- "Real-world" object becomes easier to maintain and test.
- Complex logic object becomes reusable.
- Encapsulation and isolation of test code from "real-world" code is accomplished.

Liabilities associated with factories creating Mock Objects:

- If the ObjectToTest is calling a particular static method, retrieving a particular singleton, or calling a particular factory method, we cannot substitute mock object.
- If the "real-world" object changes often or the complexity increases significantly, it makes maintenance of the Mock Object difficult.
- To insure type safety we need to implement interfaces or shoulder the burden of the possibility of type errors. Strong typing can also be added to the Mock Object, but this also adds complexity to the Mock Object.

# Implementation

Steps to implement Mock Objects via Factory:

1. Extract the complex logic into an object.
2. Extract the creation code for this complex object into a factory method.
3. Create ObjectToTest class.
4. Add ability in factory to return Mock Object or "real-world" object
5. Add unit test requiring the extracted creation code to return an object of the correct type.

Concerns of Implementation:

- Logical errors in test code or incorrect assumptions about object not yet implemented.
- Unavailable logic or incorrect assumptions about third-party object implementation.