

Don't Mock Me: Design Considerations for Mock Objects

Jeff Langr
Langr Software Solutions
Jeff.Langr@LangrSoft.com

Abstract

Mock objects, an object-oriented testing construct used predominantly in test-driven development, are often abused, leading to inflexible and defective systems. Developers abuse mocks because they don't fully understand the driving forces for when to use and not use them; they also don't fully understand the implications of their improper use. Mocks inherently introduce design concessions in code. This experience report discusses when to use mocks, presents design categories that most mock implementations fall into, and discusses the considerations for each design category.

1. Introduction

Mock objects [1] allow you to write unit tests for classes that have heavy dependencies on other parts of a system. When writing tests for a class, you can replace an instance of one of its collaborators with a stub or “mock” version of the collaborator. The mock provides controlled behavior that meets only the needs of specific tests.

When building a system using test-driven development (TDD), mocks can help you build a good object-oriented system with low coupling. However, they have the potential to introduce more problems in a system than they solve.

Done properly, TDD should result in very flexible systems. Even major design changes should be easy to make. Improper mocking has the potential to destroy the benefits brought about by TDD.

This experience report is based on my encountering several systems where rampant mocking created significant design problems. These abuses were very costly to the health of the systems in question. Defects were higher and small changes became very expensive.

2. Mock Abuse

I consulted at a shop where the developers had built a client-server system. I'll refer to this client as

International Foods Corp, or IFC. The developers used a couple of third-party mock tools for their tests. They also hand-coded many mocks. The tools made it considerably easier to code a test using mocks. A staggering twenty percent of the source files in this system used mocks or provided mock support.

Part of the reason developers used mocks in this system was because they didn't want to be troubled with figuring out how to write unit tests without them. The system had large dependency chains. Sending a message to an object often had the potential of breaking dozens of postconditions.

The developers found it easier to mock just about everything. Instead of testing that object states were valid after calling a public interface method, developers tested, through mocking, that the public interface method had called other methods with appropriate parameters.

The result of this rampant mocking was an extremely inflexible system. The mock-based tests depended upon the specifics of the implementation. Even small changes to the implementation adversely impacted dozens of tests. A simple refactoring took hours instead of minutes.

3. Reasons to Mock

Developers should not introduce mock objects into a system unless there is a good reason for them to exist. For any valid mock object, you should be able to trace its origin to at least one of a few main reasons. These reasons are centered around testing effectiveness.

The first reason is that tests take too long to run. As a general guideline, a suite of tests should run in a matter of seconds. If tests take longer to execute, developers will not run them as often. As a result, developers will introduce more defects into the system as the amount of feedback diminishes. Also, in the event of test failures, the developer must remember what changed since the last time tests were run. All of these problems will cause the development pace to slow.

Tests can take too long to execute because of continued re-execution of code that accesses external resources. I've encountered this situation many times in business applications that make heavy use of databases. An individual access of a database resource may take a hundred milliseconds. In test suites I saw, typically containing a few thousand unit tests, database access times increased the execution time of a typical suite by several minutes.

A second reason to mock is that tests might not run consistently. I worked on a system that housed an external API on a test server located in a locked room upstairs. The server crashed often, resulting in a half-hour of testing downtime while we physically traveled to and rebooted the server.

Further, the code under test may return variable results. Suppose your system requires access to an external service to obtain a price for a stock. This service may return a different price each time it is accessed. Writing a test that deals with the variability of the external service can be difficult.

A third reason is that you may not be able to run tests at all. Often, developers are faced with the challenge of writing code that must interact with something (such as hardware) that doesn't exist yet. Also, developers in a team environment must frequently build classes to interact with software that other developers are currently writing.

Another reason is that a collaborator could generate events that you cannot recreate easily within a test. For example, the class you are testing might need to handle an IOException thrown by a collaborator. Writing a test to generate the appropriate exception might be difficult. In this situation, the test can use a mock of the collaborator that always generates the exception.

A final reason often cited for mocking is that the system contains a large dependency chain. In order to write a test for a single class, the test must create a context. Building the context involves constructing a series of dependent objects that work in conjunction with the target object (the object under test). Having to construct a few dependent objects is acceptable. But it demonstrates a design flaw if building the test context becomes onerous.

While having a large dependency chain seems like a legitimate reason to code mocks, avoid the temptation. Expend your efforts instead on redesigning the system.

As I'll demonstrate in this paper, mocking is always a design concession. By using bad design as an excuse to mock, you only make things worse by adding more questionable design to the system.

Mocks present a leap of faith. They claim that you, the developer, fully understood the behavior of the classes being mocked. Yet having a system of objects with a large dependency chain means that you likely

don't understand the full implication of actions on those objects. You run the risk of introducing mocks that do not properly emulate the behavior of their production counterparts.

As an example, the IFC system also used database mocking. Instead of writing tests against a live database, developers wrote tests to ensure that the SQL being used matched what they expected. The unit tests always passed. But the developers were unaware of the complete nature of data in the database. When unexpected data was read, the system failed.

Mocking database access in this system was acceptable. However, when mocking, a second level of testing (such as automated acceptance tests) is essential. Otherwise, you may not be aware that your mock is an incomplete or inaccurate representation of the product class it targets.

4. The Stock Portfolio Example

This paper will use a simple example, a stock portfolio application, to demonstrate the various mock implementations. The relevant classes are Portfolio, StockLookupService, and Holding.

```
public class Portfolio {
    private Holding holding;
    private StockLookupService service =
        new StockLookupService();

    public int currentValue() {
        if (holding == null) return 0;
        return
            service.lookupStockValue(
                holding.getSymbol()) *
                holding.getNumberOfShares();
    }

    public void add(Holding holding) {
        this.holding = holding;
    }
}
```

Figure 1. Portfolio

A Portfolio collects the stock holdings for an owner. It must be able to supply the total value of all holdings. Figure 1 shows how you might code Portfolio without consideration for testing. For the sake of brevity and incrementalism, Portfolio for the time being supports only a single holding. You would later expand the code to support multiple holdings by writing additional tests.

In order to calculate a portfolio's value, code in Portfolio must interact with a StockLookupService object. The StockLookupService contains code to determine a price for each stock holding by making calls to a remote service.

A Holding object contains a stock symbol and the number of shares held by the portfolio owner. Code for this class is not shown.

5. Building a parameterized mock

For testing the portfolio application, a mock StockLookupService can provide consistent and fast results. A test class for Portfolio, PortfolioTest, can prove Portfolio functionality without needing to use a “live” StockLookupService.

```
public void testSingleHoldingSingleShare() {
    final int numberOfShares = 1;
    final String vitriaSymbol = "VITR";
    final int vitriaValue = 5099;

    Holding holding =
        new Holding(vitriaSymbol,
            numberOfShares);
    StockLookupServiceIF service =
        new StockLookupServiceIF() {
            public int lookupStockValue(
                String symbol) {
                if (symbol.equals(vitriaSymbol))
                    return vitriaValue;
                return 0;
            }
        };

    Portfolio portfolio =
        new Portfolio(service);
    portfolio.add(holding);
    assertEquals(vitriaValue,
        portfolio.currentValue());
}
```

Figure 2. testSingleHoldingSingleShare

Figure 2 shows a mock-based test coded in the class PortfolioTest. A Java interface, StockLookupServiceIF (Figure 3), declares the method for looking up a value for a stock symbol.

```
public interface StockLookupServiceIF {
    int lookupStockValue(String symbol);
}
```

Figure 3. StockLookupServiceIF

Using an interface is the key to making mocks work. (You can also create a mock class as a subclass of its production target.) Instead of interacting directly with a StockLookupService that it creates, the production class Portfolio now takes a StockLookupServiceIF reference in its constructor. In a production environment, some other class passes an instance of the live StockLookupService to a Portfolio object.

In the test environment, the test defines and passes a mock instance of StockLookupServiceIF to a Portfolio. In Figure 2, testSingleHoldingSingleShare creates the

service instance using an anonymous inner class definition. The anonymous inner class overrides the method lookupStockValue to ensure that the expected symbol was passed, and, if so, return an appropriate fixed value.

In either environment, test or production, code in Portfolio is oblivious to the implementation type of the stock lookup service. See Figure 4. Test code passes a stock lookup service mock as a *parameter* to the Portfolio class, hence the reason I refer to this technique as a parameterized mock.

```
public class Portfolio {
    private Holding holding;
    private StockLookupServiceIF service;
    = new StockLookupService();

    public Portfolio(
        StockLookupServiceIF service) {
        this.service = service;
    }

    public int currentValue() {
        if (holding == null) return 0;
        return
            service.lookupStockValue(
                holding.getSymbol()) *
                holding.getNumberOfShares();
    }

    public void add(Holding holding) {
        this.holding = holding;
    }
}
```

Figure 4. Parameterized mock

5.1. Experiences with parameterized mocking

Most literature explaining mocks demonstrates the parameterized mock. It is the most easily understood, both conceptually and from a code standpoint. Parameterized mocking does introduce some problems.

In the production system, some other client class will be responsible for constructing instances of Portfolio. Since the Portfolio constructor takes a StockLookupServiceIF as a parameter, this client class must instantiate and pass a live instance.

In the absence of the need to mock, the preferred design would be to have Portfolio instantiate or obtain its own live StockLookupService instance. The client of Portfolio would not even know that a lookup service class existed.

The need to mock has thus forced a design decision. A detail of Portfolio that was previously encapsulated from any client—the existence of a stock lookup service—is now exposed to the client.

For example, suppose a class named PortfolioApplication is responsible for constructing

Portfolio. Without the design change caused by the need to mock, the PortfolioApplication class is only directly dependent upon Portfolio. With the design change, PortfolioApplication now directly depends upon two additional classes, StockLookupService and StockLookupServiceIF—see Figure 5.

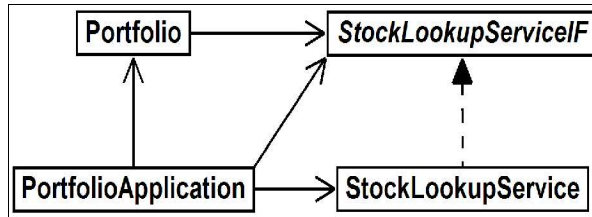


Figure 5. New dependencies

Using parameterized mocking also introduces “trickling dependencies.” Often, a service such as StockLookupService must be used in multiple places. Having to repeatedly pass an instance of StockLookupService through multiple layers of objects clutters the code and exacerbates the dependency issue.

After using parameterize mocks initially, ultimately I concluded that they are too “invasive” with respect to the production system. Once unnecessary dependencies begin to ooze out of objects, it becomes difficult to contain and manage them.

5.2. Parameterized mocking variant

```

// from class PortfolioTest:
public void testSingleHoldingSingleShare() {
    ...
    // overloaded constructor only used by test!
    Portfolio portfolio =
        new Portfolio(service);
    portfolio.add(holding);
    assertEquals(vitriaValue,
        portfolio.currentValue());
}

// from class Portfolio:
public class Portfolio {
    private Holding holding;
    private StockLookupServiceIF service;
    = new StockLookupService();

    public Portfolio() {
        this(new StockLookupService());
    }

    // constructor for testing use:
    Portfolio(StockLookupServiceIF service) {
        this.service = service;
    }
    ...
}
  
```

Figure 6. Parameterized replacement mock

A variant to parameterized mocking that I've since encountered requires an additional constructor or setter method that *only* the test would use. The Portfolio class encapsulates the creation of the production service instance. The test uses either an overloaded constructor or an extraneous setter to replace this instance. In Java, this constructor or setter can have package access, exposing it only for purposes of a test in the same package.

I'll refer to this variant as a parameterized replacement mock. I show the relevant portions of test and production code in Figure 6.

With the variant that provides an additional constructor, one constructor takes no parameters and constructs the production instance. The other constructor takes a service instance as parameter.

A parameterized replacement mock can eliminate many of the problems exhibited through use of parameterized mocking. Only the test is required to construct a mock instance, overriding the production instance that is otherwise constructed.

Some downsides exist. First, the test is misleading—it should represent appropriate production use, but instead demonstrates code only appropriate in a test. If you use such mocks, you should clearly annotate both the production code and test to indicate that the additional setter or constructor is designed primarily for use by tests.

Also, you must ensure that code in your production constructor does not operate on the service or pass it to collaborators. If you use a setter to replace the production service with a mock service, these operations would occur prior to the mock service being set. If you use an additional constructor to replace the production service with a mock, you should use constructor chaining. Otherwise operations in the production constructor will not be executed.

6. Factory-based mocking

Instead of receiving the StockLookupServiceIF reference through its constructor, Portfolio can obtain a StockLookupServiceIF reference for itself by interacting with a factory. The job of the factory is to return an instance of the appropriate type. By default, the factory returns a “live” StockLookupService object. Figure 7 shows the factory code.

```

public class ServiceFactory {
    private static
        StockLookupServiceIF instance = null;

    public static
        StockLookupServiceIF create() {
        if (instance != null)
            return instance;
        return new StockLookupService();
    }

    public static void setInstance(
        StockLookupServiceIF mock) {
        instance = mock;
    }

    public static void resetInstance() {
        instance = null;
    }
}

```

Figure 7. ServiceFactory

In contrast, the test tells the factory to return a mock StockLookupService object. See Figure 8.

```

public void testSingleHoldingSingleShare() {
    final int numberOfShares = 1;
    final String vitriaSymbol = "VITR";
    final int vitriaValue = 5099;

    Holding holding =
        new Holding(vitriaSymbol,
            numberOfShares);

    StockLookupServiceIF service =
        new StockLookupServiceIF() {
        public int lookupStockValue(
            String symbol) {
            if (symbol.equals(vitriaSymbol))
                return vitriaValue;
            return 0;
        }
    };

    ServiceFactory.setInstance(service);
    try {
        Portfolio portfolio = new Portfolio();
        portfolio.add(holding);
        assertEquals(vitriaValue,
            portfolio.currentValue());
    }
    finally {
        ServiceFactory.resetInstance();
    }
}

```

Figure 8. Factory-based mock test

The Portfolio class uses the factory to obtain an instance of StockLookupService. The design requires no Portfolio clients to create implementation instances

of StockLookupServiceIF. The UML is shown in Figure 9.

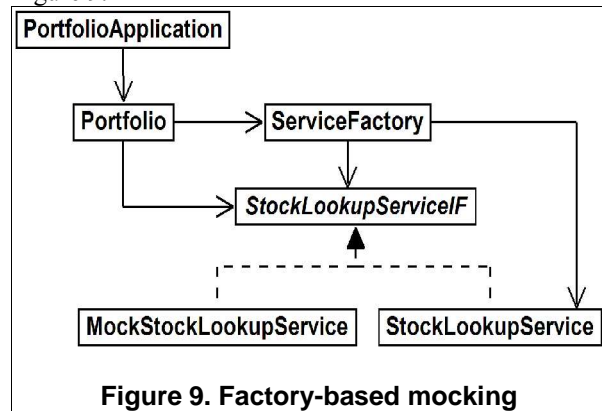


Figure 9. Factory-based mocking

6.1. Problems with factory-based mocking

At IFC, the use of factory-based mocking caused some coding and testing issues. If a test sets a mock instance into the factory, the test must ensure that the factory is reset. Occasionally, a developer would forget to reset the factory in one such test.

At this client, developers were using factory-based mocking in their acceptance test suite. Since these tests took eight hours to run, developers only ran tests they worked on. The next day, other acceptance tests expecting to use a live instance would break since the factory was returning a mock instance. With one of over 400 acceptance tests as the potential culprit, the resulting defect was costly to decipher.

The Java IFC code used try/finally blocks to initialize and reset the factories. Try/finally blocks tend to clutter the code. A significant goal of TDD is for tests to act as comprehensive documentation of system capabilities. At IFC, overuse of factory-based mocking detracted considerably from this goal. JUnit setup/teardown methods could have been used. However, this would have required breaking the tests down into smaller fixture-oriented classes, since not all tests in a given test case required mock instances.

An alternative is to use a system property, or other external switch, to designate when mocks should be used. This solution works as long as all tests require a mock. But as soon as one test requires use of the production class instead of the mock, developers must once again ensure that the switch is set and reset properly.

Introducing a completely new production class (the factory class) solely for purposes of testing is questionable. The factory is an additional layer of overhead and complexity in the production code.

7. Deferred factory-based mocking

```

public class Portfolio {
    private Holding holding;
    private StockLookupService service =
        new StockLookupService();

    public int currentValue() {
        if (holding == null) return 0;
        return
            getService().lookupStockValue(
                holding.getSymbol()) *
                holding.getNumberOfShares();
    }

    // exposed as package access for testing
    StockLookupServiceIF getService() {
        return new StockLookupService();
    }

    public void add(Holding holding) {
        this.holding = holding;
    }
}

```

Figure 10. Portfolio with factory method

Instead of accessing the stock lookup service directly via an instance variable, Portfolio defines a factory method that returns the service. In Figure 10, getService is a factory method that returns a production StockLookupService instance.

```

public void testSingleHoldingSingleShare() {
    final int numberOfShares = 1;
    final String vitriaSymbol = "VITR";
    final int vitriaValue = 5099;

    Holding holding =
        new Holding(vitriaSymbol,
            numberOfShares);

    final StockLookupServiceIF service =
        new StockLookupServiceIF() {
        public int
            lookupStockValue(String symbol) {
            if (symbol.equals(vitriaSymbol))
                return vitriaValue;
            return 0;
            }
        };

    Portfolio portfolio = new Portfolio() {
        StockLookupServiceIF getService() {
            return service;
        }
    };
    portfolio.add(holding);
    assertEquals(vitriaValue,
        portfolio.currentValue());
}

```

Figure 11. Deferred factory-based mock test

In general terms, the deferred factory method is a mock override of a production method. Implementation variants of deferred factory-based mocking include using decorators and inheritance. In either case, a mock definition for getService would replace the production definition.

The UML for this deferred factory mock example is shown in Figure 12.

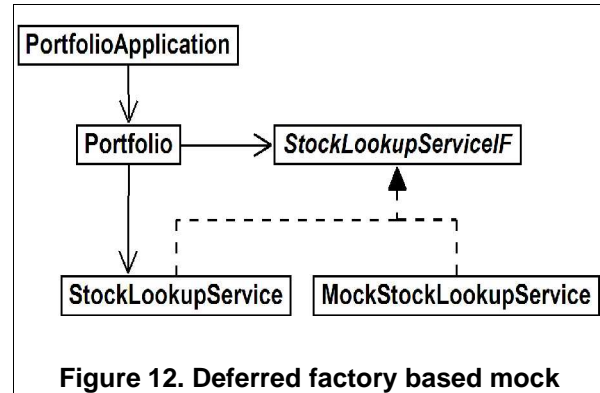


Figure 12. Deferred factory based mock

7.1. Problems with deferred factory-based mocking

A goal in building object-oriented systems is to maintain encapsulation. The more clients send messages to objects through their public interface, the more clients are decoupled from those objects. The more implementation details clients know about the objects they interact with, the tighter the coupling in your system, and ultimately the poorer the system design.

Should tests know implementation details of the classes they test? In the above example, getService is not and should not be publicly accessible. Is exposing it for use by only the test still too much?

In test-driven development, tests prove the functionality of classes [2]. PortfolioTest acts as just another client of Portfolio. Code in PortfolioTest sends messages to a Portfolio object through its public interface, then makes assertions to ensure all post-conditions are met. These assertions are usually made through the public interface of the class.

Occasionally developers will need to expose a method that was originally private in order to allow clients to inspect object state. In Java, many TDD developers place each test in the same package as the production class it tests. This allows the developer to expose the previously private method to package access (but not public access).

And also on occasion, developers may want to implement deferred factory-based mocking. To do so,

they must expose what would otherwise be a private method.

Once a test accesses these non-public methods, it becomes tightly bound to the production class implementation, and vice versa. This might not be a problem, but when taken to the extreme, refactoring becomes extremely difficult. Developers can't change the implementation of the production class—the things that should be private, per encapsulation goals—without breaking tests.

In section 2, I mentioned how IFC developers mocked virtually everything, such that a test would only have to prove that it called other methods in the proper sequence, with the proper parameters. I refer to this technique as “sequence-based mocking.”

At IFC, dramatic refactoring changes were required to server-side code for performance reasons. With hundreds of unit tests using sequence-based mocking, each and every refactoring broke several unit tests. Changing the name of a private method meant that sequence tests failed—they expected a method with a different name to execute. The tests were so tightly coupled to the production code that they were a barrier to system modification.

Based on the problems sequence-based mocking causes, I highly recommend never using it.

Done properly, deferred factory-based mocking avoids this potential disaster by isolating this tight coupling to a single factory method. Developers concede to a slightly less-than-ideal design in this case. But the likelihood that the `getService` factory method needs to change is minimal, negating the impact of the design concession.

Used in an isolated and minimal fashion, I have found deferred factory-based mocking to be very useful. Its abuse can result in an extremely rigid system—anything preventing the ability to refactor at will is bad.

8. Comments on Mocking Tools

IFC heavily used two tools to aid in generating mocks. A handful of additional tools are available; I've worked with most of them on various projects.

The use of mocking tools results in code that can be difficult to comprehend. A developer learns to read a mock-based test only after some experience with the tool. The mechanisms of the mock become implicit; the tests become idiomatically understood.

If mocking is sparingly used, as it should be, each time a developer encounters a mock-based test will require additional re-familiarization time. Hand-coded mocks done properly, on the other hand, can supply all information needed to understand the test in one place.

Mocking tools can make it too easy to write mock-based tests. The temptation to code mocks when none are needed increases, as was the case at IFC. It should take a bit of effort to code a mock.

The use of mocks should stand out as something exceptional about the system—why is a mock needed here? Can we design the system better, so mocks are perhaps not needed in as many places?

While mocking tools can eliminate some of the effort required to code mocks, I don't think that's a problem that needs to be solved. I recommend hand-coding mocks, which results in a clearer solution.

9. Final Notes

No perfect way to implement mocks exists. Each technique carries with it some design issues, however minor. This should suggest that while mocks can be useful in testing, they introduce other issues into a system that might be a cause for concern.

I prefer use of deferred factory-based mocking (section 7) over any other mock form. They appear to cause the fewest problems, and I believe they connote intent more clearly.

Parameterized replacement mocks are a good second choice. However, the concession of exposing overrideable getter methods to package-level access (when using deferred factory-based mocking) seems safer than adding constructors or setters.

Mocks can degrade confidence in the system. By definition, when developers test with mocks, they are not testing with “the real thing.” Developers can misunderstand the collaborator they are mocking. If so, a real possibility exists that they have overlooked something that will introduce a defect in the system. Developers must adequately capture all the test variants based on the collaborator's possible behaviors. For example, what happens if the collaborator throws an exception? Tests for `Portfolio` must handle this and other circumstances that the live `StockLookupService` creates.

To help restore lost confidence, developers should continually execute acceptance-level tests that exercise the system from a “live” user standpoint. The acceptance tests should work against production code, and not use mocks.

Mocking is still an extremely valuable tool. Slightly imperfect design is almost always the better choice over the inability to test. But the rule for mocking is still, *don't... unless you must*. Ensure you have a valid reason for mocking. The reason will normally be one of the reasons for mocking I discussed.

10. References

- [1] T. Mackinnon, S. Freeman, and P. Craig, “Endo-Testing: Unit Testing Using Mock Objects”,
<http://www.connextra.com/aboutUs/mockobjects.pdf>.
[2] “Code Unit Tests First,”
<http://c2.com/cgi/wiki?CodeUnitTestFirst>

11. Acknowledgements

Thanks to Michael Feathers for providing the term “deferred factory-based mocking.” Also, thanks to Debbie Utley for reviewing the article and providing many excellent insights and suggestions.