StaticMock: A Mock Object Framework for Compiled Languages

Dustin Bingham and Kristen R. Walcott

University of Colorado, Colorado Springs
dbingham@uccs.edu, kwalcott@uccs.edu

Abstract

Mock object frameworks are very useful for creating unit tests. However, purely compiled languages lack robust frameworks for mock objects. The frameworks that do exist rely on inheritance, compiler directives, or linker manipulation. Such techniques limit the applicability of the existing frameworks, especially when dealing with legacy code.

We present a tool, StaticMock, for creating mock objects in compiled languages. This tool uses source-to-source compilation together with Aspect Oriented Programming to deliver a unique solution that does not rely on the previous, commonly used techniques. We evaluate the compile-time and run-time overhead incurred by this tool, and we demonstrate the effectiveness of the tool by showing that it can be applied to new and existing code.

Keywords

Test-Driven Development, Mock Objects,Seams, Stubs, Expectations, AspectC++, Aspect Oriented Programming

1. Introduction

Unit testing is fundamental to the process of writing quality software. Test Driven Development (TDD for short) places such value on the importance of unit testing that we create unit tests first, before even one line of code is written [8]. In the past, unit tests were written ad hoc, but a collection of techniques have arisen in the eXtreme Programming and TDD communities that ease the process of creating worthwhile tests quickly. These techniques focus on isolating a section of code, such as a class or method, and strictly controlling the behavior of its surrounding environment. Mock objects - objects that have the same interface as the original, but provide no behavior [13] - are one such technique, and are the focus of this paper.

While interpreted languages such as Ruby or Java have a wealth of frameworks available [7, 3] that provide a toolkit for mock objects, purely compiled languages such as C++ lag behind. This is due to the dynamic faculties that interpreted languages can offer. Metaprogramming and reflection allow us to inspect and alter both the state and the interface of an object at runtime, and are common features of interpreted languages. In compiled languages, such features are absent or, if they exist at all, highly limited. This leaves us with a whole class of languages that are deficient in the powerful isolating features that mock objects offer.

As TDD continues to blossom, the need to bring a robust mock object framework to compiled languages also grows. Some attempts have been made, following the approaches laid out in Michael Feathers' seminal work [13]. Google Mocks [16], for example, applies the technique of inheritance: mock classes are subclasses of their full featured parents. The original methods are overriden and provide the appropriate mock behavior instead.

Other less common approaches have also been taken to solve this issue. Mockator [5, 26] cleverly manipulates the order of the link step in compilation, effectively shadowing the original code with mocked code. On the other side of the compilation process, a tool such as `fff` [21] makes use of conditional compiler directives. These compiler directives replace original code with the mocked code, depending on the state of some value specified at compile time. The replacement is done just before the formal compilation process begins.

Unfortunately, the existing solutions introduce challenges that limit their effectiveness. For example, the inheritance approach requires code modification. Every method that belongs to the interface of that class now must be changed to allow the methods to be be overridden in the child mock class. Like most things in engineering, making a method overridable comes with a trade-off: some overhead must be paid to facilitate this polymorphism [12]. In a performance critical system, that price may not be permissible, especially if it is a price paid to aid in unit testing, but carries its cost in the production code. Unfortunately, compiled languages tend be used most frequently in applications where the demands of performance outweigh other considerations.

We must also consider the impact introducing child classes with overriden, mocked methods requires in a large, legacy code base. For example, consider an application with more than 10 million lines of code. Modifying each class in such a large system is a non-trivial expenditure of time, even if the task is a simple one. An organization may be loathe to open these classes for modification, even though the only change is simply marking methods as overridable.

The conditional compiler directive solution has a similar limitation, but worse. Every class that needs to be mocked will require some sort of change to incorporate the directives. While no price is seen at runtime because the replacement is done at compile time, careful application of these directives will require a careful and thorough modification of the code base. Furthermore, the compiler directive approach can remove type safety as well as make the code more obfuscated [26].

Finally, the approach that cleverly manipulates the compiler tool itself avoids both the performance hit as well as the code maintenance problem. However, it is necessarily limited to the compiler that it is manipulating. If compiler the tool is created for is the one and only one that is in use in a particular organization for a particular program, then this approach can be successful. However, in an environment where different compilers may be used to target different platforms (g++ for the various flavors of Linux and Microsoft Visual Studio for Windows, for example), then this approach also falls short. The mock framework will only be available in one of the multiple targeted environments of the program. Different unit test code must be written for environments that have access to the tool versus the environments that do not.

In this work, we use source-to-source compilation avoids these pitfalls. We can take existing code, transform it, and output source that is ready for mocking. The original source code remains untouched in the baseline, avoiding the risk, time, and tedium of opening them for modification. Importantly, the performance penalty of inheritance is avoided in the actual production code as well.

While there are many tools that can perform source-to-source compilation [4, 14], it would be beneficial to extend from a bare source-to-source compiler and utilize an approach that closely matches the problem of introducing a mock object framework into compiled languages.

The Aspect Oriented Programming (AOP) paradigm can help meet this need. AOP gathers together similar logic spread through multiple parts of a codebase into an abstraction called an *aspect* [19]. We use an aspect to intercept all method calls and provide the appropriate mock behavior as necessary. The aspect is then woven through the original code through source-to-source compilation using an `aspect weaver`, a feature of AOP that merges aspect logic with the original source code.

By combining source-to-source compilation with AOP through the aspect weaver, the capability to create mock objects can then be introduced at compile time. Source code is modified for the unit test driver and left untouched in the main program. We avoid the issues surrounding inheritance, compiler directives, and linker manipulation. In our tool, StaticMock, we instead provide a strategy for transforming code through AOP to provide a framework for mock objects in compiled languages.

In summary, the main contributions of this paper are:

1. Background discussion on mock objects (Section), source-to-source compilation(Section , and Aspect Oriented Programming (Section)
2. Overview of the StaticMock tool (Section)
3. Evaluation of StaticMock (Section)
4. Discussion of future work and ways to improve StaticMock (Section)

2. Mock Objects, Source-to-Source Compilation, and AOP

Before we discuss our tool, we more fully articulate the idea of a mock object. In the TDD community, there is some disagreement on the definition of the terms that surround this concept, so we clarify our usage of these terms.

We also give an overview of source-to-source compilation, and AOP. We describe how these techniques are used with our StaticMock strategy.

2.1. Mock Objects, Stubs, Seams, and Expectations

Providing a way to use mock objects for compiled languages is the heart of the StaticMock tool. In this section we describe mock objects, as well as the related techniques of stubs, seams, and expectations.

The term *mock object* is used to denote an object that is used to 'stand-in' for a real object [22, 27]. Mock objects exactly mirror the interface of the real object but provide no implementation. In a unit test, the surrounding classes interacting with a Class Under Test (CUT) are typically created as mock objects. This isolates the behavior of the CUT while still allowing us to provide all the necessary interacting pieces required to test. Since "no object is an island" [9], the surrounding objects around the CUT form a system that can greatly influence how it behaves. By making mock objects that stand in for the surrounding objects, we attain the isolation needed to evaluate only the CUT.

Mock objects usually forbid any of their methods from being called. If the mock object receives an unexpected method call, then the unit test can be failed. But merely having a mock object reject all method calls would not allow us to adequately exercise the CUT. A certain subset of method calls should be *stubbed.* A stubbed method is a method on a mock object that allows a call to be made to it without failing the unit test. Furthermore, a stubbed method can be configured at run time to return a specific value. This is highly useful: the output of the stubbed method may be the input to a method in the CUT. Through configuring the return value, we can generate a range of values to that method to ensure the behavior is correct across them.

Besides allowing calls to proceed through a mock object, we also allow a stub to be introduced to a real object. Instead of performing the logic within the stubbed method, the method returns instantly with the specified return value. In a unit test, this is useful to avoid invoking behavior that could be destructive, irreversible, or bothersome. One-way, permanent changes to a database is one example. Calling an external API owned by a different organization that charges per invocation is another. In such cases, returning immediately is desirable, and allows us to write unit tests that would otherwise be impossible.

Stubs fall into two basic categories: Seams and Expectations. While a seam behaves exactly in the manner described above, an expectation extends the power of the stub. An expectation causes a unit test to fail if the method was never called by the time the unit test terminates. In other words, an expectation is a seam that expects to be called.

A mock object should retain any arguments passed to its stubbed methods for later verification in the unit test. All stubs should count the number of calls as well, and fail the test if the stub method was called the incorrect number of times.

2.2. Source to Source Compilation

Source to source compilation translates from one type of source to another. For example, consider a class `Foo` with method `Bar()` as it exists before source-to-source compilation. This class is shown in Figure 1.

```
class Foo{
  void Bar(int p1, int p2){
    // Bar logic...
  }
}
```

Figure 1: Class Foo Before Source-to-source Compilation

Source-to-source compilation gives us the ability to transform the input source code and output modified source code as shown in Figure 2. Since we are creating a mock object framework, the example compilation injects logic to provide for mock object behavior. The code surrounding the `bar()` method determines at runtime if a particular object instance of `Foo` is a mock object or a full featured object. If the `Foo` object is a mock object, then the method performs the null behavior necessary. If the object is a full featured object, however, it continues on its normal path.

```
class Foo {
  void Bar(int p1, int p2) {
    if (isMocked) { return; }
    else {
      // Bar logic...
    }
  }
}
```

Figure 2: Class Foo After Source-to-source Compilation

2.3. Aspect Oriented Programming

StaticMock combines Aspect Oriented Programming with source-to-source compilation to create mockable objects. The following section describes the key concepts taken from AOP and used in the StaticMock tool.

Figure 1 introduced a class `Foo` with a method `Bar()`. In source-to-source compilation, logic is injected directly around the implementation of the `Bar()` method (Figure 2). However, with AOP, that logic instead becomes an *aspect* as shown in figure 3. The aspect is merged with those methods by the aspect *weaver*. The weaver performs the role of the source-to-source compiler, introducing behavior for creating a mock object around the `Bar()` method. The original code of `Foo` is preserved and the mocking logic is clearly separated away into its own abstraction.

```
class Foo {
  void Bar(int p1, int p2) {
    // Bar logic...
  }
}

aspect Mocker {
  // weave this logic with methods of foo
  advice execution("Foo") : around() {
    if (isMocked()) {
      return getMockReturnValue();
    }
    else {
      // continue to do original method logic
      proceed();
    }
  }
}
```

Figure 3: Class Foo and Aspect Mocker

Specifically, AOP deals with the idea that there are challenges in computer programming that are best thought of as *cross-cutting concerns* [19]. A cross cutting concern is one in which the code to implement the solution is scattered across many sub-systems or classes in the program. For example, logging the execution of every function called (both entering and exiting) is a cross cutting concern.

Another example of a cross-cutting concern, as shown in the appendix , is synchronization for a collection of data structure classes. Two typical data structure classes, `Stack` and `List` are described in the figure in pseudocode. The code is simple and clear, and the methods deal only with their direct main concern: operations for pushing, popping, and inserting. However, if these classes are extended such that only one thread at a time can modify the internal state of the structure, the code becomes obfuscated, as seen in the appendix . The clean data structure operations are now blended with logic necessary to provide thread safety. That logic is similar between all three operations: lock a mutex, do the data structure operation, then unlock that mutex. If that code could be abstracted away, then the clean data structure operations would reappear.

This abstraction is called an *aspect*, and gives AOP its name. In the appendix , the low level synchronization details are bundled together, forming the aspect. Many of the advantages realized with the inheritance model of class layout also apply to aspects. Aspects coalesce code into one common place, preserving the DRY principle (Don't Repeat Yourself) [17]. This greatly aids in reusability. Aspects also further the goals of information hiding: the lower-level details and data members are hidden away within the aspect and exposed only via methods.

While an aspect abstracts cross cutting concerns into one place, mechanisms are then necessary to introduce that code into the necessary classes. Following the synchronized data structures example in Figure 8, the synchronization logic is abstracted into an aspect. This aspect then must be applied in some way to each of the data structure classes, `Stack` and `List`. To do so, three concepts are used: point cuts, advice, and class slicing.

Point cuts are the insertion points in the originating code from which the aspect is called. Point cuts define which classes the aspect are attached to and where. In the example, Point cuts are defined on the entrance and exit of the public methods in each of the data structure classes. This is shown with the `before` and `after` keywords in the example.

The second concept, *advice*, is then introduced to the originating class through the point cut. Advice can be thought of as the methods that perform the work of the aspect. In Figure 8 one piece of advice locks the synchronizing primitive. This is joined through the `begin` point cut. A second advice method, joined through the `after` point cut, unlocks the synchronizing primitive. Together, these pieces of advice provide the syncronization behavior to allow the data structure to be safe in a multi-threaded environment.

The final mechanism, *class slicing*, introduces new data members and methods directly to the originating class. Class slices are attached to the originating class similarly to advice: point cuts define what classes gain the new members and methods, and the slice defines the members and methods gained. In the synchronization example, the synchronization mutex is sliced into each of the data structure classes. However, it is encapsulated purely in the aspect and the logic that operates against it is hidden away in the advice.

3. StaticMock Overview

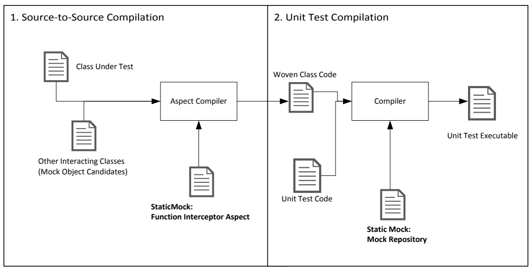
The StaticMock tool binds the previous discussed ideas together to form a solution to the problem of creating a mock object framework in compiled languages. It uses source-to-source compilation through AOP to allow us to create mock objects without introducing unnecessary inheritance, code modification, or linker manipulation. To accomplish this task, the tool is broken into two phases as shown in figure 4.

The first step of the tool is source-to-source compilation, using an AOP compiler. The CUT, any supporting classes, and the AOP portion of StaticMock, the Function Interceptor Aspect, are input to that compiler. The compiler produces woven code as output. This woven code, together with the code for the unit test driver and one other piece of our StaticMock implementation, the Mock Repository, are fed to the actual language compiler to produce the unit test executable.

Our tool, StaticMock, is comprised of two halves that work together across these compilation steps. In figure 4, the bolded parts show these two halves. The first half, the Function Interceptor Aspect, is weaved in to all classes at compile time. It provides the necessary advice to all the methods of a class, and through that advice communicates with the second half of StaticMock, the Mock Repository.

The Mock Repository is a stand-alone class that maintains a record of all objects mocked, seams introduced, methods called and arguments seen. Classes woven together with the Function Interceptor Aspect inform the Mock Repository of these records. The Mock

Figure 4: Steps of Compilation in Static Mock



Repository is also the part of StaticMock that we use when we write an actual unit test. When we want to validate arguments, transform objects into mocks, or to stub out methods in the unit test, the Mock Repository is relied upon to performed these tasks.

These two pieces are now explained in detail.

3.1. The Function Interceptor Aspect

The Function Interceptor Aspect is the part of StaticMock that surrounds all the method calls for the classes in the unit test. By intercepting these calls, the Function Interceptor Aspect is able to determine the status of the object, and provide either mock object behavior, or the behavior that the normal, full featured object behavior. The Function Interceptor works in tandem with the Mock Repository, which is described in section .

The Function Interceptor Aspect catches all method calls made in the unit test. There are 3 categories of calls caught: Constructors, Destructors, and Function Calls. The calls are caught by introducing advice to these function categories. The advice uses an "around" Point Cut expression to completely replace the function call instructions with the instructions in the advice description. A special method in the advice, `proceed()`, allows the original function to be invoked if necessary: if the current object is not a mock object and the function is not stubbed, then the original behavior should be allowed to continue.

Constructors must be intercepted in a mock object, as the constructor may attempt to initialize or otherwise make use of other objects or resources that should not be touched. For example, the constructor may try to initialize a connection to a database, or connect to a live website's API. It is a little tricky to prevent the constructor from being invoked, however. By the time an object is brought to life, the constructor has already been called and its work performed. Because of this, we cannot uniquely identify the object that should have its constructor disabled. Instead, one must register with the Mock Repository beforehand the classes that should not have their constructors invoked upon object instantiation. Unfortunately, such a registration is necessarily at the class level, not object level, because no object yet exists, and so such registration would impact all future object instantiations of that class. Therefore, StaticMock provides a way to also unregister forbidden class constructors. The preferred idiom for creating mock objects then is to register the classes that should have their constructors intercepted with the Mock Repository, create the objects one plans to use, then unregister those classes. This allows one to create a real object of that class, complete with full initialization, later down the line in the unit test if needed. Pseudo-code for the constructor advice follows:

Listing 1: Constructor Advice

```
className = determineClassName()
if (MockRepository.canConstruct(className))
{
  MockRepository.setCtorCalled(object);
  proceed();
}
```

Destructors, unfortunately, have a similar concern. Resources normally acquired through construction or method use could be freed at the end of the object's lifetime. It would be disastrous if one were to attempt to free these resources that were never actually acquired because the object was mocked. Thus, destructors must also be intercepted. There are several considerations that need to be kept in mind in the implementation of the destructor advice. It is possible that the normal construction of a mock object was allowed by the user of the tool. In this case, the destructor must also be permitted to continue and free any resources acquired during normal initialization. Furthermore, the process of function interception for a given mock object acquires dynamic memory that must be freed. Finally, any expectations that were unmet by this object should be reported to the end user. Sadly, an exception typically should not be thrown from a destructor, and so the best that can be done is to send the failure out to some error stream. However, a user can ask for the status of the expectations on a mocked object at any time, and so perform the assertions necessary for the unit test in that way. The tasks the Destructor must follow are shown below:

Listing 2: Destructor Advice

```
MockRepository.assertExpectationsMet(object);

MockRepository.unseam(object);
MockRepository.unexpect(object);

if (MockRepository.wasCtorCalled(object))
{
  MockRepository.eraseCtorCalled(object);
  proceed();
}
```

Finally, all other Function Calls are also intercepted. When a call is intercepted, the Interceptor must check with the Mock Repository to determine whether the function has been seamed, if the target object is a mock object or not, and (in the case that the function is a method of a mock object) whether or not that object is expecting this call. If the function has been seamed, or if the object is a mock object and it is expecting the call, the Interceptor informs the Mock Repository of the call. The function call arguments are cataloged with the Mock Repository for later verification in the unit test. The interceptor then retrieves the registered return value from the Repository and returns that value to the caller. If the object is a mock object, however, and a call is not expected, an exception is thrown so that the unit test can fail. If neither of the above cases are met, then the function was neither seamed, nor was the object a mock object with an expectation set up, and so the original function is allowed to be called through the `proceed()` method. Note that the more general term /emphfunction is used here, instead of /emphmethod, as the Function Interceptor Aspect intercepts not just object methods, but also free functions that do not belong to an object at all. The interaction of this logic is captured below:

Listing 3: Execution Advice

```

if (isSeamed || hasExpectation)
{
  // save the arguments for later
  // verification
  Arguments arguments;
  for (int ii=0; ii< FunctionArgs; ++ii)
  {
    Arg arg = getArg(ii);
    arguments.push_back(arg);
  }

  MockRepository.markFunctionCalled(object, signature, arguments);
  return MockRepository.getSeamReturnValue(object, signature);
}
else if (isMocked)
{
  throw exception("Unexpected call to" + signature);
}
else
{
  proceed();
}
```

3.2. The Mock Repository Class

The Mock Repository is the part of StaticMock that one interacts with when actually writing the unit test. The Function Interceptor Aspect described above in section feeds the Mock Repository as methods are called throughout the execution of the program. By querying this class, the necessary assertions can be performed in the unit test to verify the behavior of the CUT. We now detail the workings of the Mock Repository.

The Mock Repository is a standalone class, so there is ever only one in existence as the program executes. This class is responsible for handling the registration of seams, mocks, and expectations made by the unit test. It also keeps a log of the function calls made at runtime as the Function Interceptor Aspect encounters them.

Central to most of the Mock Repository methods is the concept of the Function Signature. It is a string representation of a function's declaration, and takes the form:

Listing 4: Function Signature

```
ReturnType [Namespace::]*[ClassName::]?FunctionName( ArgumentType1,
    ArgumentType2, ... ArgumentTypeN)
```

There are a few subtleties to point out in the Function Signature. Namespaces can be nested, or nonexistent, as is the case for the global namespace. Free functions, global functions that do not belong to a class, have no class name specified. The argument list only contains the types of the arguments and the names of the variables are not listed. This signature uniquely identifies a method.

The Mock Repository also frequently requires the address of an object. It uses this address to discriminate among the various objects that may exist at run time, allowing us to setup seams and expectations on a per object basis. Furthermore, when an object is destroyed, it automatically unregisters itself from the repository by using its address. If the seam or expectation is a free function, or a class level method, then the value of null is used for the address to indicate that the seam or expectation is not tied to any object.

3.2.1. Registration

Mock objects are registered through the `mock()` method. This method turns the object into a mock object, forbidding all methods on that object from being called. An exception is thrown if any method is called on the object henceforth. To allow a method on a mock object to receive a call a seam can be introduced or an expectation set up.

A seam is created through the `seam()` method. The method is now allowed on the mock object, but is stubbed out. A return value can be specified, which will be returned to the caller when the method is invoked in the future. Expectations are similarly registered, but they use the `expect()` method. Expectations are used to ensure that the mock object receives a call to that method some time before the object is destroyed. Expectations can be checked at any time through the `metExpectation()` method. All expectations on an object are checked automatically on object destruction. Any missing expectations are written out to the error stream.

Seams can also be introduced over methods of normal, full featured, objects. This is useful for writing unit tests against the CUT, but preventing that class from performing behavior that is undesirable in a unit test, such as deleting files, making database changes, or calling external APIs.

3.2.2. Logging Function Calls

The Mock Repository contains a map of all calls made to stubbed functions during the execution of the test. When the Function Interceptor Aspect catches a method call, it informs the Mock Repository of that call so that it can be logged. The arguments that were passed to the method are bound together with the Function Signature and the originating object address and are stored in the map. These arguments can then be retrieved through the `getArgument()` method for appropriate validation within the unit test to ensure that values received by the method were correct. The map also contains a count of how many times a method was called so that can also be verified by the test.

4. StaticMock for C++

In this section, we discuss how we implemented the previously described pieces of the StaticMock tool in the C++ language.

C++ was chosen as the target language for evaluation as it is a language in wide use, especially when performance is a concern. It is a language that does have mocking frameworks available [16, 5], but these frameworks rely upon the aforementioned techniques of inheritance or compiler manipulation.

We chose AspectC++ [1] to perfors the AOP source-to-source compilation described in figure 4. It weaves in aspect header files and outputs valid C++ code. This code can then be included and built with unit test code to verify the operation of classes and methods.

The implementation of the Function Interceptor Aspect described in section resides in the file `smock.ah`. AspectC++ transforms all classes fed to it to generate mock-ready classes and functions.

The other part of StaticMock, the Mock Repository class (as detailed in section), lives in the files `MockRepository.h` and `MockRepository.cpp`. Including the header in a unit test code file and linking MockRepository.cpp is all that is required to gain access to the all the functionality that the Mock Repository provides.

4.1. Metrics

To evaluate this implementation of StaticMock the following criteria were considered:

1. New capabilities enabled by the tool
2. Compile build time overhead
3. File size overhead
4. Run-time execution overhead

Two experiments were performed to evaluate the new capabilities criteria: a Test Driver that demonstrates the functionality of StaticMock with new code, and a TinyXml Driver that uses the tool against an already existing library.

The overhead induced criteria was measured through two separate builds of a small application (again using the TinyXml Library) that either has StaticMock framework compiled in, or it does not. This experiment considers the cost the StaticMock framework charges by comparing the differences in compile and execution times between these two configurations.

All experiments were performed on a machine with these characteristics:

- AOP Compiler: AspectC++ version 0.9
- C++ Compiler: g++ version 4.8.1, optimization level set to -O2
- Processor: Intel i7@2.4GHz
- Memory: 8GB

4.2. Experimental Evaluation

The two experiments below investigate the ability to use StaticMock as a mock object framework. A simple test driver experiment was performed first, ensuring that all the necessary pieces of our mock framework can be utilized. The second of these two experiments use the tool with an already existing library.

In our third and final experiment we investigate the overhead introduced to the executables that may use this tool.

4.2.1. Test Driver

The Test Driver experiment was an initial proof of the StaticMock concept. A simple hierarchy of two classes, `Base` and `Derived`, was created. Inside these classes, permutations of virtual and non-virtual functions were added to ensure that both overridden and non-overriden types of methods were able to stubbed - a key requirement for object oriented programming.

A class method, `gcd()`, was also added to the `Derived` class. It calculates the greatest common divisor between two integers, using the recursive euclidean algorithm. A separate method on the `Derived` class, `produce()`, makes use of that `gcd()` function. It loops five times, on each pass generating two integers randomly and calling `gcd()` to find the common divisor between them.

A third class, `Consumer`, makes use of the `Derived` class through its `consume()` method. This is used (as shown below) to test use between objects. `consume()` takes a reference to an object of the `Derived` class as an argument, and, inside the method implementation, invokes the `produce()` method of the `Derived` class.

A unit test was then written to exercise this simple code. This unit test first creates an object of `Derived`, `d`. Before the demonstration of the mocking capabilities, the `produce()` method on `d` is called. `produce()`, as explained above, iterates five times, each time calculating the gcd of two random numbers through the `gcd()` class method. These are sent to Standard Out for inspection and validation.

Next, the `gcd()` function is seamed, such that it always returns `-1`. `produce()` is again invoked, but this time an exception is thrown, as `produce()` does not expect to receive a negative value from `gcd()`. While a contrived example, it does show how StaticMock can stub over a function and control its return value. Through this feature, a critical error that exists in `Derived`'s `produce()` method is uncovered.

We now turn `d` into a mock object, forbidding all calls to it (except ones that are explicitly stubbed through seams or expectations). A `Consumer` object, `c` is also constructed. The `consume()` method on `c` is called, passing in `d` as the argument. `c`, in turn, calls back to `d`, through the `produce()` method. Since `d` is now a mock object and no expectation is set up for the `produce()` method, an exception is thrown. This demonstrates another use of the capabilities of mock objects in general and of StaticMock in particular. Interactions between objects are easily shown. It is clear now that `Consumed` is tightly coupled to `Derived` through the behavior of `Derived`'s `produce()` method. While `produce()` does nothing of consequence in this little function, in real code it could potentially write changes to a database or invoke outside APIs. By refusing to invoke this code and instead throwing an exception, this sort of behavior is avoided. Furthermore, to test `consume()` fully, one would want to modulate all the return values of `produce()` across the entire range of outputs that could returned. These are all capabilities delivered by using the seam and expectation features of StaticMock.

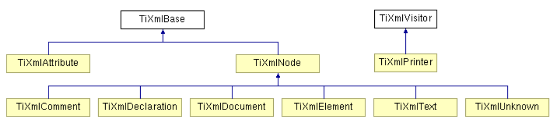
Continuing with this experiment, another instance of `Derived` is created, called `e`. A seam is set up on one of the overridable methods in the hierarchy, `abstractfn1()`. An expectation is created as well, on the other overridable method, `abstractfn2()`. `abstractfn1()` is invoked, which is allowed through the seam. However, `abstractfn2()` has an expectation that it should be called. The state of the expectations of an object can be checked through the `assertExpectationsMet()` method on the Mock Repository. Accordingly, that method is called, and an exception thrown - `abstractfn2()` never received a call.

4.2.2. TinyXML Driver

While the first experiment was created to evaluate and demonstrate the features of StaticMock, the intention of this second experiment is to apply the StaticMock tool to an existing code base. To that end, StaticMock was applied to write a small unit test against the TinyXML library.

The TinyXML [28] program is a fast and small XML parser. It is very mature and stable, and is used in many real world applications. While small, it has enough complexity to demonstrate the usefulness of the StaticMock tool's capabilities. The object model for Tiny XML is shown in figure 5.

Figure 5: Tiny XML Object Model



TiXmlDocument is the central class in the design of this library, and represents an XML Document, as one would expect. It was chosen as the class under test for the experiment. The unit test for the experiment focused on printing functionality surrounding this class.

Two classes interact with TiXmlDocument during printing. TiXmlElement represents an arbitrary XML element that is attached to the TiXmlDocument. TiXmlPrinter implements the visitor design pattern [15] and is used to visit each item in the XML document. As it visits the item, the Print() method of that object is invoked.

Two methods in the unit test driver test the behavior of TiXmlDocument. test_print() evaluates the DOM walking capabilities of TiXmlDocument. That is to say, it ensures that the children in its document receive a call. To accomplish this task, it attaches three TiXmlElement mock objects to the TiXmlDocument object being tested through the linkEndChild() method. An expectation is set up on each of the three mock objects, hoping to see their own individual Print() methods are called. The test then invokes the Print() of the document itself. If all expectations are met, then test succeeds.

Similarly, test_visitor() ensures that the VisitEnter() and VisitExit() methods are called on a TiXmlPrinter object from the TiXmlDocument. A mock TiXmlPrinter is instantiated, and the TiXmlDocument accepts the printer visitor through the accept() method. If the expectations are met, the test succeeds. Otherwise, it fails.

4.2.3. Overhead Introduced

To evaluate the various overhead costs introduced by the tool, the TinyXML library was again used. Instead of creating a unit test, however, a small application was created. This application reads the contents of a local XML file from disk (about 142KB in size), parses it into an internal DOM representation, then writes that representation back out to disk in a separate file.

Two different build configurations for this application were tested: one with the aspect code weaved in, and one without. For each configuration, 10 separate builds were performed and timed. The average time was then recorded for each, and slowdown calculated by dividing the average time taken with StaticMock code by the average time take without that code. The file size of both executables were also recorded. The results are shown in Table 1.

Next, the application was executed 10 times in both configurations, recording how long it took to perform its task. The results are presented in Table 2.

4.3. Discussion

In both the Test Driver experiment and the TinyXML Driver experiment the mocking capabilities of StaticMock are shown. Seams and Expectations can be stubbed over any arbitrary method. Return values can be specified for those methods, allowing us to control the interactions between mocked objects and the CUT. Any class that is provided to the aspect compiler can be then be turned into a mock object in the unit test. Argument data to methods are saved in the MockRepository. That data can then retrieved later in the unit test, and be evaluated to assert that their values are correct. Taken in toto, this demonstrates that unit tests can be successfully written take advantage of this new functionality that the StaticMock tool provides, for both new and existing code.

The Overhead Introduced experiment shows that there is around 6.5 factor increase in time to compile the executable when StaticMock is added. In the experiment, the output executable took an additional 14.5 additional seconds to compile. More dramatically, the

Table 1: Compilation Overhead

Trial	Compile Time with StaticMock (ms)	Compile Time without StaticMock (ms)
1	17138	2654
2	17340	2700
3	17263	2733
4	17347	2691
5	17192	2669
6	17209	2686
7	17184	2687
8	17326	2706
9	17196	2688
10	17268	2690
Avg Time	17246	2690
Avgerage Slowdown	6.457	
File Size (Bytes)	1448652	161187
File Size Increase	8.987	

Table 2: Execution Overhead

heightTrial	Execution Time with StaticMock (ms)	Execution Time without StaticMock (ms)
1	1603	18
2	1603	18
3	1604	19
4	1605	19
5	1604	17
6	1599	18
7	1602	17
8	1606	18
9	1599	19
10	1603	18
Avg. Exec. Time	1602.8	18.1
Execution Slowdown	89.1	

execution time suffered an 89.1 factor slowdown. While the total compile and run times with StaticMock still appear to be overall within reasonable parameters (seconds, not minutes) unit tests should be first of all be fast, both to execute as well as to build. In an environment where thousands of unit tests must be run, this increase in build and run time may not be acceptable, which could potentially limit the usefulness of the tool. However in [20] it was discussed that the runtime overhead introduced by aspect code may be disproportionately large in small code, making this overhead appear larger in the experiment than it actually would be in practice.

Furthermore, the point cuts defined for introducing advice in the implementation of StaticMock are very aggressive: they catch all method calls, whether necessary to the unit test or not. The tool could be optimized to only catch method calls for the objects surrounding the CUT. This would limit the interference of the mocking framework, reducing overhead. In a way, the results here provide a 'worst-case' analysis of the tool's use, where every class provided to StaticMock will need to be capable of becoming a mock object in the unit test.

5. Threats to Validity

While StaticMock was implemented with only one type of compiled language (C++), it should be generally applicable to other purely compiled languages. Similarly, while only one type of aspect oriented compiler was chosen for experimentation, the feature set across aspect oriented languages is similar enough that the core ideas will be transferable.

It may also be noted that the experiments chosen to demonstrate our tool were small; however each experiment shows the complete spectrum of possibilities that the tool can achieve. Expectations and seams with return values configured were demonstrated to be achievable. Mock objects with their attendant expectation validation and method forbiddence abilities are shown. And finally, argument values can be stored, retrieved, and validated. With these capabilities, StaticMock is established as a viable toolkit for creating unit tests.

6. Related Work

There are many mock object frameworks already in existence. For interpreted languages, there are a wealth of choices available [7, 3, 10, 6, 11]. These frameworks rely on dynamic features of interpreted languages, which are not available in compiled languages.

For compiled languages, the selection is slimmer [16, 25, 5]. These frameworks either depend on inheritance or compiler manipulation to achieve their goals. Our solution does neither: instead it intercepts method calls at runtime through an aspect advice woven in at compile time through an AOP compiler.

The concrete implementation of the StaticMock tool is closely related to the Virtual Mock Object technique proposed in [18], which uses a central class to function as repository and method call logger. It similarly uses AOP to intercept method calls. No concrete mock objects are created with this tool, however. Instead, individual methods on particular objects are registered and intercepted in an ad-hoc basis. Our implementation differs in that an actual mock object can be instantiated, with the entire interface of that object acting as a mock object. This leads to a natural, intuitive, and readable unit test. A second, critical, difference is that StaticMock is focused on bringing a mock object framework to compiled languages, while the Virtual Mock Object technique was implemented for Java using AspectJ [2].

GenuTest [24] makes use of the Virtual Mock Object technique described above in the automatic generation of its unit tests.

Discussion of the applicability of mock objects and frameworks can be found in [23]. The genesis of using mock objects in unit tests was discussed in [22].

Object, link, and and preprocessor seams was explored in [13]. A fourth technique for introducing seams into an object, compile seams, was introduced in [26].

7. Conclusion and Future Work

StaticMock is a unique approach to the problem of creating a mock object framework for compiled languages. Other approaches rely on code modification, inheritance, compiler directives, or linker manipulation to achieve their ends. When dealing with legacy code, these approaches have significant drawbacks that limit their applicability. StaticMock instead uses Aspect Oriented Programming and source-to-source compilation to deliver that framework. Unit tests can now be created with StaticMock without making changes to legacy code or by introducing unnecessary runtime overhead. Our tool shows slow both the compilation and runtime of unit tests where it is used. It would be helpful to explore ways to reduce the time overhead of the tool. StaticMock could be extend toward thread safety and protecting access to a mock object's member data.

8. References

[1] Aspect c++. `http://www.aspectc.org/`. Accessed: 2015-02-08.

[2] The aspectj project. `https://eclipse.org/aspectj/`. Accessed: 2015-02-08.

[3] jmock: an expressive mock object library for java. http://www.jmock.org/. Accessed: 2015-02-08.

[4] The llvm compiler infrastructure. http://llvm.org/. Accessed: 2015-02-08.

[5] Mockator. http://www.mockator.com/. Accessed: 2015-02-08.

[6] Mockito. http://mockito.org/. Accessed: 2015-02-08.

[7] Rspec: Behavior driven development for ruby. http://rspec.info/. Accessed: 2015-02-08.

[8] D. Astels. *Test driven development: A practical guide.* Prentice Hall Professional Technical Reference, 2003.

[9] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *ACM Sigplan Notices*, volume 24, pages 1–6. ACM, 1989.

[10] I. Clarius, Manas. Moq. https://github.com/Moq/moq4. Accessed: 2015-02-08.

[11] E. contributors. Easymock. http://easymock.org/. Accessed: 2015-02-08.

[12] K. Driesen and U. Hölzle. The direct cost of virtual function calls in c++. In *ACM Sigplan Notices*, volume 31, pages 306–323. ACM, 1996.

[13] M. Feathers. *Working effectively with legacy code.* Prentice Hall Professional, 2004.

[14] F. S. Foundation. Gnu bison. https://www.gnu.org/software/bison/. Accessed: 2015-02-08.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader).* Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[16] Google. googlemock: Google c++ mocking framework. https://code.google.com/p/googlemock/. Accessed: 2015-02-08.

[17] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master.* Addison-Wesley Professional, 2000.

[18] R. Jeffries. Virtual mock objects using aspectj with junit. http://ronjeffries.com/xprog/articles/virtualmockobjects/. Accessed: 2015-02-08.

[19] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97Object-oriented programming*, pages 220–242. Springer, 1997.

[20] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of aop with generative programming in aspectc++. In *Generative Programming and Component Engineering*, pages 55–74. Springer, 2004.

[21] M. Long. fff: Fake function framework. https://github.com/meekrosoft/fff#readme. Accessed: 2015-02-08.

[22] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2000.

[23] S. Mostafa and X. Wang. An empirical study on the usage of mocking frameworks in software testing. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 127–132. IEEE, 2014.

[24] B. Pasternak, S. Tyszberowicz, and A. Yehudai. Genutest: a unit test and mock aspect generation tool. *International journal on software tools for technology transfer*,

11(4):273–290, 2009.

[25] Pe'er. Fakeit. `https://github.com/eranpeer/FakeIt`. Accessed: 2015-02-08.

[26] M. Rüegg and P. Sommerlad. Refactoring towards seams in c++. In *Proceedings of the 7th International Workshop on Automation of Software Test*, pages 117–123. IEEE Press, 2012.

[27] D. Thomas and A. Hunt. Mock objects. *Software, IEEE*, 19(3):22–24, 2002.

[28] L. Thomason. Tinyxml. `http://www.grinninglizard.com/tinyxml/`. Accessed: 2015-02-08.

9. Appendix

```
class Stack {
public:
  void push(Object o) {
    if (top != max) {
      internal[top++] = o;
    }
  }

  Object pop()  {
    if (top > 0)
      return internal[--top];
    else
      return null;
  }
private:
  Object internal[max] = Object[];
  top = 0;
}

class List {
public:
  void insert(Object o) {
    Node next(o);
    head.next = next;
    head = next;
  }

private:
  class Node{
    Node(Object o) {
      held = o;
      next = null;
    }
    Object held;
    Node next;
  }
  Node head = null;
}
```

Figure 6: Typical Data Structure Classes Stack and List

```
class Stack
{
public:
  void push(Object o) {
    lock(mutex);

    if (top != max) {
      internal[top++] = o;
    }

    unlock(mutex);
  }

  Object pop() {
    retVal = null;
    lock(mutex);
    if (top > 0){
      retVal = internal[--top];
    }
    unlock(mutex);
    return retVal;
  }

private:
  Object internal[max] = Object[];
  top = 0;
  Mutex mutex;
}

class List {
public:
  void insert(Object o) {
    lock(mutex);
    Node next(o);
    head.next = next;
    head = next;
    unlock(mutex);
  }

private:
  class Node{
    Node(Object o){
      held = o;
      next = null;
    }
    Object held;
    Node next;
  }
  Node head = null;
  Mutex mutex;
}
```

Figure 7: Stack and List, Synchronized

```
class Stack {
public:
  void push(Object o)
  {
    if (top != max){
      internal[top++] = o;
    }
  }

  Object pop()
  {
    if (top > 0){
      return internal[top--];
    }
    else{
      return null;
    }
  }
private:
  Object internal[max] = Object[];
  top = 0;
}

class List{
public:
  void insert(Object o){
    Node next(o);
    head.next = next;
    head = next;
  }

private:
  class Node{
    Node(Object o){
      held = o;
      next = null;
    }
    Object held;
    Node next;
  }
  Node head = null;
}

aspect Synchronizer{
  advice execution : before{
    lock(mutex);
  }

  advice execution : after{
    unlock(mutex);
  }

private:
  Mutex mutex;
}
```

Figure 8: Stack and List with Aspect