Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Introducing Mock framework for Unit Test in a modeling environment

by

Joakim Braaf

LIU-IDA/LITH-EX-G--14/004--SE

2014-03-10

Linköping University
Department of Computer and Information Science

Final Thesis

Introducing Mock framework for Unit Test in a modeling environment

by

Joakim Braaf

LIU-IDA/LITH-EX-G--14/004--SE

2014-03-10

Supervisor: Kristian Sandahl
Examiner: Kristian Sandhahl

Abstract

Testing is an important part in the software development process. Unit tests aim to test individual units in isolation. These units may have dependencies to their surroundings that make the units hard to test in isolation without also testing the surrounding units. A technique to help isolate these units is to replace the surrounding units with mock objects. This work investigates how a C++ mock framework can be integrated into a modeling environment's unit test framework. Several mock frameworks are evaluated, and a proof of concept is created to show that integration is possible. Additionally, ideas for how to use mocks in a model environment are presented.

Acknowledgements

I would like to start by saying thanks to Pontus Sandberg and Johan Wibeck for giving me the opportunity to do this thesis at Ericsson. It has been both a fun and educational experience.

Many thanks to Bjerker Andersson and my supervisor Johan Westling. Both of you have been of great help and it is fun to work with people who are really interested in the subject and comes with their own ideas and input. I would also like to say thanks to my supervisor at the university, Kristian Sandahl. Even if I have not been asking many questions during the work it self, you have provided valuable feedback while I have been writing this report.

Thanks is also in order to the personal at Ericsson who have been both friendly and helpful, and shown interest in my work. Doing a work becomes much more fun when others around you show interest.

Last but not least I would like to say thanks to my opponent Kristina Ferm for providing valuable feedback on my work.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Purpose . . . 1
1.3 Limitations . . . 2
1.4 Methodology . . . 2

2 Theory framework **3**
2.1 Modeling environment . . . 3
2.2 Software testing . . . 3
2.2.1 Stubs . . . 3
2.2.2 Mock objects . . . 4

3 Framework comparison **6**
3.1 Google Mock . . . 6
3.2 Turtle . . . 7
3.3 Hippo Mock . . . 8
3.4 Comparison . . . 8
3.4.1 Integration with other test environments . . . 9
3.4.2 Documentation . . . 9
3.4.3 Verbosity . . . 9
3.4.4 Features . . . 10
3.5 Conclusion . . . 10

4 Integration **11**
4.1 Objectives . . . 11
4.2 Proof of concept . . . 11
4.2.1 Verdict . . . 13
4.3 Mock utility . . . 13

5 Mock usage **14**
5.1 When to use mocks . . . 14
5.2 Workflow . . . 15
5.3 Dependency injection . . . 15
5.3.1 Hi-pref dependency injection . . . 16

5.3.2 Beautiful dependency injection 17
5.3.3 Curiously recurring template pattern 18
5.3.4 Code generation 18

6 Closing **19**
6.1 Discussion and future work 19
6.2 Conclusions 20

References **22**

Chapter 1

Introduction

1.1 Background

Testing is an important part in the software development process to help catch bugs and find errors when the code gets refactored. In test-driven development tests are also used as a way to design and help form the software.

One way of testing software is to test individual parts (units) in isolation, this is called unit testing. Unit tests aim to test the individual parts of the software and ensure that the unit is working properly.

Usually units have dependencies to its surroundings which makes the unit hard to isolate for a unit test. A technique to help isolate units is to replace the surrounding dependencies with mock objects. Even if the concept of mock objects is more than ten years old, it has become a more and more popular technique in software testing [1].

1.2 Purpose

At Ericsson LTE RBS, significant parts of the software is developed using a modeling environment where C++ code is generated from models. The unit test environment for the models currently lack a mock object framework.

The goal of this work is to learn about existing third party mocking frameworks and analyze how they work. Based on this, integrate a mocking framework, analyze and suggest how mocks can be used with the unit test environment.

1.3 Limitations

The focus of this work will be on mocking of class model elements. Other elements such as capsules contain more internal parts and are considered more complex, and the time frame is limited which does not give enough time to look further into more complex elements.

Additionally, because of the limited time frame, the investigation and comparison of C++ mock frameworks will be limited in depth and only focus on a few key requirements. A comparison of mock frameworks is a whole thesis on its own.

1.4 Methodology

To learn about different C++ mock frameworks the official documentation which can be found on the homepage for each framework will be used as a base and first hand source for information. Using the official documentation as a primary source provides first hand information from the developers and also shows how they intended the library to be used. Third party tutorials and web articles or blogs may be using another style than the intended one. Using third party web pages may on the other hand provide information which is not available in the official documentation e.g solutions to more uncommon pitfalls not covered in the documentation.

The literature study of mock objects will consist of paper and on-line articles about their usage, and experience from using mock objects. Today many hobby and professional developers choose to use the web to publish their findings and solutions to problems. Because of this much information can be found on-line. Information learned from this study can then help form a suggestion of how mocks can be used and how they are best used in tests.

Chapter 2

Theory framework

2.1 Modeling environment

In a modeling environment the architecture of the software is designed and modeled using Unified Modeling Language (UML). This gives the developer a visual representation of different parts of the software and allows the developer to design the software at a higher level of abstraction, removing the need to mentally visualize how code interacts.

These models that can represent classes, state machines and other constructs are then used to generate source code for a specified language e.g C++ or Java.

2.2 Software testing

Software testing is an important part of software development. Extreme programming focus a lot on testing [2]. in particular testing of individual units which is called unit testing. A unit can be a function or an entire class, the definition of how large a unit may be varies.

The goal of unit tests is to test individual units in isolation to give an indication that the unit is working as intended, and also help verify that the units keeps working as intended later in the development when the code may have been refactored. When multiple units are tested together it is called integration testing.

2.2.1 Stubs

Usually in unit tests the unit has dependencies to surrounding units which are outside of the scope of the test. This may require the surrounding units to be set in a particular state to perform the test. The units may be slow or complex (e.g a database object) which increases the runtime for the test.

To solve this problem the dependencies can be replaced using a technique called stubs. Stubs are objects that behaves (in the eye of the code calling the stub) like the object it replaces but uses predetermined results to calls [3] [4]. The use of stubs allows easier set up of scenarios that that may be hard to set up with the real object. An example scenario is simulating the loss of a database connection. A stub would in this case throw an exception or return the correct error code while the real object requires an actual connection loss during the test, which can be hard to reproduce during test execution.

Setting up the surrounding objects can be a tedious task if the object has states that require multiple state changes before achieving the desired state, as this requires the test to first go through all these state changes in the setup phase before running the actual test case. Another similar technique are fakes [4] which are a replacement but fakes have an actual implementation that takes a shortcut (e.g in-memory database).

2.2.2 Mock objects

Mock objects are an extension of stubs that allow verification of the unit's behavior (e.g verifying that a function is called, or that a set of functions are called in a certain order) [4] [1].

The concept of mock objects was introduced at the extreme programming conference XP2000 in the paper *Endo-Testing: Unit Testing with Mock Objects* by Tim Mackinnon, Steve Freeman and Philip Craig [1].

Since the introduction further studies has been put into the usage of mock objects. The article *Mock Roles, Not Objects* [5] follows up on *Endo-Testing: Unit Testing with Mock Objects*. In the article mocks are discussed as a tool to test interaction and relationship between objects and not just a tool to replace complex objects with a simplified version.

The key to test the interaction and relationship between objects with mocks is expectations. Expectations are set on the mock before a test case and verified after the test case has finished.

A concrete example is a function which uses a database object to perform an operation on a database. The database object requires the user to first initialize the database object, then call the connect method to connect to the database before calling the query method to perform the operation.

When the function is tested in a unit test the database object can be replaced with a mock. The mock can be used to help verify that the function calls are in the correct order (initialization, connect and then query) using expectations.

```
MockFoo mock;
EXPECT_CALL(mock, function()).Times(2)
```

Listing 2.1: Example expectation in Google Mock

Later if the function is rewritten and the developer forget to call the initialization function, the mock will catch this while verifying the expectations and fail the test because the expectations were not met (initialization function was not called). This allows the developer to find changes in the usage of the object and correct them before pushing the changes.

Chapter 3

Framework comparison

This chapter is a brief comparison of three C++ mock frameworks.

There are more C++ mock frameworks available. Because some of them are complete test frameworks which do not support integration, and because of the limited time frame, Google C++ Mocking Framework, Turtle, and Hippo Mock where chosen because they were either suggestions in the thesis description or because they came up regularly during the research phase. These names often come up when searching for phrases such as "C++ mock framework" and "C++ mock framework comparison" using on-line search engines.

A mock framework should provide the utilities needed to create and use mock objects, and the possibility to set expectations to be able to test the interaction between objects. Actions are important so the mock also can be used to remove complexity from the test. All frameworks provide this functionality and more, such as setting limitations on parameters.

3.1 Google Mock

Google C++ Mocking Framework (Google Mock) [6] is as the name suggests, a C++ mocking framework developed by Google. It is primarily developed to be used with their C++ test framework (Google Test), but it still supports integration into other unit test frameworks. How to integrate Google Mock is well documented, and this is one of Google Mock's strong points, it has a well written documentation with a for dummies introduction guide and a cook book with recipes that covers different topics. The for dummies introduction describes basic usage of and how to set up the framework. More topics which are not covered in the introduction are covered in the cook book (e.g how to mock nonvirtual functions, expecting ordered calls and more common usage cases).

In the frequently asked questions (FAQ) section, it is documented how to debug and see why expectations are not met. Additionally there are also

answers to common problems. Out of all three mock frameworks in this comparison, Google Mock has the most comprehensive documentation of its features such as how to add custom actions and validation and how you can work around common problems.

Google mock supports three levels of verbosity which can be set by either passing the *gmock_verbose* flag to your test executable or setting *::testing::FLAGS_gmock_verbose* in your test code. The different levels are info, warning and error, where info is the most verbose level. Information levels are useful for debugging or learning how the mock behaves. Warning is the default level, which allows output of warnings such as when a function is called but no expectations are set. When the error level is set, Google Mock will only inform if expectations were not met and not print any warnings.

3.2 Turtle

Turtle [7] is a C++ mock object library primarily meant to used with Boost Test but it also supports integration with other frameworks.

The quality of the documentation is good and well written, but compared to Google mock's it does not cover as many topics (e.g how to combine actions and returning references from mocked calls). However It is open about the frameworks limitations and suggests workarounds if such exists.

Turtle's feature set is the same as most mock object frameworks. It uses a macro to create a mock class (see *Listing 3.1* and *Listing 3.2*). This is different compared to Google Mock where the user creates a mock by creating a class which can inherit from the class the user want to mock.

```
MOCK_CLASS(name)
{
        ...
};
```

Listing 3.1: Example of how a mock without inheritance is declared with Turtle

```
MOCK_BASE_CLASS(name, base)
{
        ...
};
```

Listing 3.2: Example of how a mock with inheritance is declared with Turtle

3.3 Hippo Mock

Hippo Mock [8] was created by two employees at Topic Automatisering. The goal of Hippo Mock is to create a framework that allows creation of mocks while writing the test without the need of first declaring the mock classes. This is done in a way that slightly resembles the factory design pattern [9] by creating a repository which handles the lifetime of the mock.

```
MockRepository repo;
Foo* fooMock = repo.InterfaceMock<Foo>();
```

Listing 3.3: Hippo Mock MockRepositry example

This is a different approach compared to Google Mock and Turtle where you first have to declare the mock class before writing the test.

The documentation for Hippo Mock is quite sparse. The tutorial covers the creation and how to set expectations and actions of mocks. Instead of long documentation the tutorial is complemented by example tests.

The feature set is of the same as for Google Mock and Turtle, which means it supports expectations and actions.

3.4 Comparison

For this comparison the focus will be on the key requirements listed below. The following four requirements have been chosen to help choose a mock framework that can be used in this work:

- **Supports integration with other test environments**
  This one is the most important requirement because if you can not use the framework with other testing environments, it may involve a lot of hacking to get it to work. Even if the framework excel in all other areas, it may not be worth doing all the hacking and tweaking to get it to work.
- **Documentation**
  When familiarizing with a new framework, having a comprehensive documentation helps you spend less time figuring out how to perform a certain task and more time on the actual writing of the test. This can speed up the learning process for users new to the framework or mocking in general as less time has to be spent looking around for a solution.
- **Verbose output**
  Having different verbosity levels is helpful because this lets the user decide how much information should be displayed. When running a test it may be too much output if the framework tells about everything that happens. If some condition is not met it is useful to be able to

pinpoint why. This helps you debug and see if it is the test that needs to be updated or if the code being tested is not behaving as expected.

- **Features**
  Different frameworks may have the same features but they work a little differently and one way may be better than the other. This report will focus on three features: expectations, actions and validation. Expectations is one of the main features of mock. Expectations and validation can be used to verify behavior of the code being tested. Actions allow the mock object to do actions. This can be used to perform a light version of the real objects work to speed up the test.

3.4.1 Integration with other test environments

Both Google Mock and Turtle supports integration into other test environments even though both are designed to primarily work with a specific test framework (Google C++ Testing Framework and Boost Test). If Hippo Mock is intended for a specific framework is not explicitely mentioned in its documentation. From the tutorial it is possible to come to the conclusion that it is not intended for a specific unit test environment or framework.

All three frameworks use exceptions to signal if all expectations were met or if any expectations were not met.

3.4.2 Documentation

Google Mock has a really comprehensive documentation. The documentation includes a beginners guide and a cookbook with a lot of tips on how to solve different problems and how to set up the mock to perform a certain action. Turtle's documentation is not as comprehensive as Google Mock's but it does still include an introduction about how to create a mock object and set up the mock for a test. The setup includes expectations, actions and verification of parameters. Hippo Mock has a short tutorial with enough information to get you started. There is also test code available that works as examples.

3.4.3 Verbosity

The only framework that offers the option to dynamically set different verbosity levels is Google Mock which has three different levels [10]. Turtle does allow changes to the verbosity if a custom policy is created [11]. For Hippo Mock this is not mentioned in the tutorial and there does not seem to be any indication of this in version 3.1 0x source code (*http://www.assembla.com /spaces/hippomocks/documents/bvmlG4oACr3Oy _eJe5afGb/download /hippomocks.zip*).

3.4.4 Features

All three frameworks support expectations with the possibility to specify how many times a function should be called. Actions can either be a simple return statement or custom functions that perform more complicated calculations, and verification of input parameters. For these three features the only real visible difference is the syntax.

One feature that may normally be overlooked is the way the mock object is created. Hippo Mock with its factory like mock object creation can be useful for a project working with plain text files. This approach does not require the user to write the mock before writing the test. With the approach used by Google Mock and Turtle the user has to write the mock definition at least once, but if the mock definition is saved in a source file it can then be shared between different test cases.

Google Mock's approach of creating the mock object is by defining a C++ class without any macro, like Turtle does. This makes it possible to represent and generate the mock with a class model in the modeling environment. Depending on the modeling environment used this may be a must unless there is support for inclusion of source files.

If a mock is represented with a class model in the modeling environment, it is possible to create tools using the available application programming interface (api) to manipulate the model.

3.5 Conclusion

Taking into account what is discussed in the previous section, the choice of which C++ mock framework to integrate falls on Google Mock. This is motivated by the comprehensive documentation and because of the way the mock is defined, allowing the use of the model api to create tools.

Chapter 4

Integration

4.1 Objectives

The goal with the integration is to have Google Mock running with the existing unit test framework. The mock should be represented in the model environment as a class element. This is partially because the modeling environment currently does not support source files (file artifacts), and because the model also gives the user a visual representation of the mock. From the class model element, a C++ class should be generated which can then be used as a mock with Google Mock in a test case.

If possible, utilities should be created to ease the use of Google Mock in the Ericsson unit test framework. Moving code that is common between tests to make it reusable by other tests to reduce boilerplate code.

4.2 Proof of concept

To test that Google Mock works with the current test framework and identify which components that have to be customized, a proof of concept test was created. The test consists of a simple class named Foo that internally uses another class named Bar, which in this test will be replaced with a mock. To ease the writing of the test, Foo's constructor takes a pointer to Bar in its constructor. Alternate dependency injection options will be discussed later in this report.

Foo has a member function that calls one of Bar's member functions, this is to simulate the dependency real code could have. After creating a class element for Foo and Bar with their respective member functions a new class element representing the mock was created. Because of a limitation that does not allow the user to add a member function and not have it generate any code, the choice was made to create a class without any attached member functions and put the code for functions inside the public declaration field.

```
class Foo
{
private:
    Bar* _b;
public:
    Foo(Bar* b) : _b(b) {}
    int func()
    {
        return _b->other_func();
    }
}
```

Listing 4.1: Example showing the code generated from the Foo class element

After setting up the elements they were transformed into code to verify that the output from the mock model would be the same as handwritten code. This method proved to work as expected.

The Google Mock cookbook advices that the compilation can be speeded up by moving the constructor and destructor declaration into the source file [12]. It is possible to do this by activating the option for generating a default constructor and destructor for the mock's class element.

The next step was to create and run an actual test case. If Google Mock is used with another test framework than Google test the user is required to pass parameters to the *InitGoogleMock* function [13]. The documentation passes the *argc* and *argv* from the main function to InitGoogleMock, because the test framework's main function is abstracted away from the user. The work around to solve the problem is to manually create the arguments and pass the arguments to InitGoogleMock from the test framework's initialization function. The test case created was a minimal test to verify that Google Mock works and consisted of the following steps:

1. Create an instance of the mock for Bar.
2. Create an instance of Foo passing the mock to the constructor.
3. Set an expectation that Bar's function that Foo calls internally should be called once.
4. Call Foo's function that uses Bar.

After compilation the test was ran and returned a success. This was the expected result because Foo calls Bar once and the test only makes one call to this function. For the next run the expectation was set to that Bar was not to be called at all and this should trigger that the expectation were not met. This time after compiling and running the test an exception was thrown indicating that the test failed because Bar's function was called

when it was expected to not be called. This shows that the Google Mock works and it is possible to use it in combination with the test framework.

To make the output from Google Mock work with the frameworks output an event listener had to be created.

4.2.1 Verdict

The proof of concept test case proved successful but there are a few points to note. Google Mock has to be initialized for every test case and the boilerplate code needed for this makes it a bit more cumbersome to write a test. Also for every test case that will use mocks, the transformation configuration (TC) has to be updated with the include path to Google Mock. If the path to Google Mock source gets changed, all tests have to be updated individually. A solution to this is presented in section 4.3.

4.3 Mock utility

To solve the issues with initialization of Google Mock and inclusion path a separate mock utility was created. By setting up the TC that contains the inclusion path to Google Mock, tests that use mock can refer to this TC to get the correct inclusion path. In case the inclusion path changes only the TC in the mock utility project has to be updated. However, this does require that any test that uses mock to also import the mock utility.

The boilerplate code needed to initialize Google Code is the same except that the user may want to change the output level. Because the boilerplate code is the same a function was created in the mock utility that perform the initialization and also allows the user to set the output level. Using this solution reduces the code needed to setup Google Mock to a single line. This solution also removes the need to include Google Mock in the test case.

Chapter 5

Mock usage

5.1 When to use mocks

Like anything, mocks can be abused and overused if the developer does not fully understand the interaction with the surrounding objects. The tests may not properly simulate the real world, resulting in incorrect tests that do not really test what the developer intended to test.

Overusing mocks may also create problems. The reason for this is that the design and creation of mocks has a time overhead. As long as the surrounding objects are not complex or slow there is little overhead to including them in the test. When updating the signature for an object, all mocks for that object has to be updated.

Jeff Langr writes about this in his article *Don't Mock Me: Design Considerations for Mock Objects* [14]. The article mentions five reasons when to use mock objects in tests, which can be summarized in the following points:

- Test takes too long
- Test does not run consistently
- Simulate something that does not yet exist
- Generate events that are hard to trigger
- Write tests for large dependency chains

Using the five guidelines as good motivations for when an object should be mocked and having an understanding of the object's real purpose and how the object interacts with its surroundings should help create tests that emulate the real world more accurately.

5.2 Workflow

When adding mocks to a test case, the developer should consider what was said in section 5.1. If a mock can be motivated the developer first has to create the mock, if no mock already exists. Creating the mock model can be done with the mock creation plug-in developed to ease to create of mocks. The plug-in allows to user to right-click on the class element the user want to create a mock for and choose *Create Mock* in the LTE menu (Figure *5.1*).

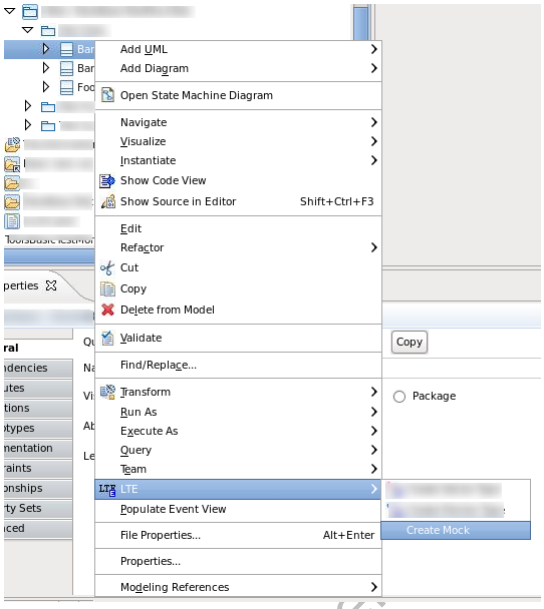

Figure 5.1: Mock creation plug-in right-click menu

Next the mock utility has to be imported. Using this utility the developer can add the TC to get the correct inclusion path and access to the initialization function. The mock utility initialization function should be called from the test frameworks initialization function.

The last step is to set up the mock and set expectations and actions.

5.3 Dependency injection

When using mocks the real object has to be replaced with the mock. Dependency injection is a design pattern which allows objects to be replaced at run-time or compile-time. The pattern can be achieved in different ways. If the code is designed with dependency injection in mind one of the easiest ways is to allow the object to be set via the constructor.

In C++ this is done by taking a pointer to the base as an argument,

then letting the mock derive from the base and then be passed into the constructor. This does require the function to be virtual which in some cases is not desired.

```
class Foo
{
private:
    Bar* _b;
public:
    Foo(Bar* b) : _b(b) {}
    int func()
    {
        return _b->other_func();
    }
};

class MockBar : public Bar
{
    //...
};

MockBar b;
Foo(&b);
```

Listing 5.1: Demonstration of dependency injection via the constructor

Virtual functions have an overhead when accessed via a pointer to the base class because the vtable has to be accessed to determine which function should be called. Accessing the vtable can destroy the cpu cache in the sense that space has to be made for the vtable and then when the code continues the main memory has to be accessed again causing a cache miss.

Because C++ is not a managed language like Java, memory management has to be done manually. This creates a problem when taking a pointer to the object. If the ownership is transfered to the object, the real object has to be created on the heap. Otherwise the memory management has to be elsewhere. This can be a problem in high performance software or software that has to be run with a minimal memory footprint. Therefor alternate ways to inject the mock may have to be used.

5.3.1 Hi-pref dependency injection

When virtual functions is not an option one technique to inject mocks is hi-pref dependency injection [15]. The idea behind this technique is to create a mock class that does not derive from the base object, but copies the base objects signature. With Google Mock this is possible and can be done in the same way as a regular mock is created, except it does not derive from

the base object. Templates are then used to inject the mock into the code at compile-time.

Using this technique has two drawbacks. All functions and objects using the base objects has to be created or modified to use templates. Templates are known to sometimes generate long error messages that are hard to read and understand. This is common to any usage of templates.

The advantages are that virtual functions are not needed and the injection is done at compile-time. Because the injection is done during compilation there should be no overhead during run-time.

5.3.2 Beautiful dependency injection

Beautiful dependency injection[16] is a technique created to avoid the use of virtual functions but still allow the object to be testable.

```
template<class T>
class UsesBase
{
private:
    T base;
public:
    UsesBase(T base) : base(base) {}
};

class Concrete : public UsesBase<Base*>
{
private:
    Base _b;
public:
    Concrete() : UsesBase<Base*>(&_b) {}
};
```

Listing 5.2: Beautiful dependency injection

The idea is to not call delete inside the destructor of the class (UsesBase) which uses the class the developer wants to replace (Base) (*Listing 5.2*). Instead the memory management is moved outside to a derivative of UsesBase. This allows the developer to choose how to manage the lifetime of Base. Note here that the object injected into UsesBase is not a derivative of Base but a class with the same signature.

Using this method has the same drawbacks and advantages as hi-pref dependency injection but may be a pattern that fits better into the code as this method reassembles normal class derivation.

5.3.3 Curiously recurring template pattern

Curiously recurring template pattern[17] is a design pattern that resembles inheritance but uses templates instead. The reader is advised to read about this pattern in more detail on their own as there are quite a few details. In short the base class is a template taking the derived as a parameter (*Listing 5.3*). This allows calls to be made to the derived class without first accessing the vtable because which function to call is determined at compile-time.

```
template<class DerivedType>
class Base
{
    ...
};

class Derived : public Base<Derived>
{
   ...
};
```

Listing 5.3: Curiously recurring template pattern

One drawback with this pattern is that you cannot store different derivatives using a base pointer. Because using pointers was not an option to start with, it is not really a drawback that didn't already exist. Another drawback is circular dependencies as the base has to know about the derived types. This can create problems when a new derived type is created and the developer forgets to include the derived in the base's header-file, which also may create long header-files. One must also keep in mind in what order objects are created, as base cannot access a function of derived inside the constructor because derived is initialized after base.

5.3.4 Code generation

Because the code is generated from models, one possible way to inject mocks into the code could be to do the injection during code generation time. This could work in a similar way as templates but without the need to create templated code. Another benefit is that there would be no complicated compiler template errors. Instead the error should highlight that the injected object does not have a function with a specific signature.

Though this is only possible if the code generator supports some kind of code substitution and different profiles, for production the code would be generated from the real class.

Chapter 6

Closing

6.1 Discussion and future work

In chapter 3, three different C++ mock frameworks were compared. The goal with the comparison is to give a brief overview of different frameworks and show what requirements were looked into. When a framework is to be chosen, one usually looks at how mature the framework is and what features are available. In this case the framework is going to be used in a modeling environment, which in this case also adds the requirement of having the mock represented as a model element. In model environments which supports source files, this may not be a requirement because it is still possible to write the mocks once and reuse them in multiple tests.

Because Google Mock uses class inheritance which is not hidden behind a macro, it was quite easy to see how the mocks could be generated from model elements. Using Turtle would also be possible but would require a less optimal solution as the underlying class inheritance is hidden behind a macro, forcing the code for the model element to be handwritten and not use the code generator.

Using the proof of concept test case to find what could be improved in the integration was a good start. It showed that some code would be common between tests and could be put into a mock utility that works as glue between the current testing framework and Google Mock. Because a real test was not modified to use mock there is a possibility that as time passes, more common operations can be moved into the mock utility.

The same goes for usage of mocks. Hopefully this report, combined with presentation(s) at Ericsson can be used as a base when starting to learn how to use mocks and get a feel of how mocks can be put to the best use. Chapter 5 could work as a good starting point as it covers when to use mocks (section 5.1) and what that has to be done to use mocks (section 5.2). Writing more detailed how-to documentation for the mock creation plug-in and mock utility can help ease the process when starting to use

mocks. Section 5.3 covers different ways and suggestions of how mocks can be injected into both existing and future code.

Future work could start with this information and investigate how existing tests can benefit from mocks and set better guidelines for when to use mocks, as there may be circumstances that are unique to different projects. It should also investigate the possibility of using mocks as a tool in the current workflow to test code when developing new features that may not exist yet.

6.2 Conclusions

The goal of this thesis was to learn about third party C++ mock frameworks and integrate one into the current unit test environment. This thesis has demonstrated how Google Mock can be integrated into the unit test environment and what glue had to be added to make both frameworks work together. Using what has been learned and with the help of other studies in the area, suggestions of how and when to use mocks have been suggested.

References

[1] "Mock objects." `http://www.mockobjects.com/`, January 2014.

[2] "Test first." `http://www.extremeprogramming.org/rules/testfirst.html`, 02 2014.

[3] D. Thomas and A. Hunt, "Mock objects," *IEEE Software*, vol. 19, no. 3, pp. 22–24, 2002.

[4] M. Fowler, "Mocks aren't stubs." `http://martinfowler.com/articles/mocksArentStubs.html`, 2007.

[5] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'04*, pp. 236–246, 24 October 2004 through 28 October 2004 2004.

[6] "Google c++ mocking framework." `http://code.google.com/p/googlemock/`, December 2013.

[7] "Turtle." `http://turtle.sourceforge.net/`, December 2013.

[8] "Home — hippo mocks project — assembla." `https://www.assembla.com/wiki/show/hippomocks`, December 2013.

[9] R. J. Erich Gamma, Richard Helm and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[10] "Google c++ mocking framework cookbook." `http://code.google.com/p/googlemock/wiki/CookBook#Controlling_How_Much_Information_Google_Mock_Prints`, December 2013.

[11] "Turtle." `http://turtle.sourceforge.net/turtle/customization.html\#turtle.customization.logging`, December 2013.

[12] "Google c++ mocking framework cookbook." `http://code.google.com/p/googlemock/wiki/CookBook#Making_the_Compilation_Faster`, December 2013.

[13] "Google c++ mocking framework for dummies." `http://code.google.com/p/googlemock/wiki/ForDummies#Using_Google_Mock_with_Any_Testing_Framework`, December 2013.

[14] J. Langr, "Don't mock me: Design considerations for mock objects," *Agile Development Conference 2004*, 2004.

[15] "Google c++ mocking framework cookbook." `http://code.google.com/p/googlemock/wiki/CookBook#Mocking_Nonvirtual_Methods`, December 2013.

[16] "Beautiful dependency injection in c++." `http://programmaticallyspeaking.blogspot.se/2010/04/beautiful-dependency-injection-in-c.html`, January 2014.

[17] "Curiously recurring template patterns." `http://sites.google.com/a/gertrudandcope.com/info/Publications/InheritedTemplate.pdf`, January 2014.