University of Applied Sciences Rapperswil

Mockator Pro

Seams and Mock Objects for Eclipse CDT

Master Thesis: *Fall and Spring Term 2011/12*

Author
Michael Rüegg

Supervisor
Prof. Peter Sommerlad

"To arrive at the simple is difficult."
— Rashid Elisha

Abstract

Breaking dependencies is an important task in refactoring legacy code and putting this code under tests. Feathers' seams help us here because they enable us to inject dependencies from outside. Although seams are a valuable technique, it is hard and cumbersome to apply them without automated refactorings and tool chain configuration support. We provide sophisticated support for seams with Mockator Pro, a plug-in for the Eclipse C/C++ development tooling project. Mockator Pro creates the boilerplate code and the necessary infrastructure for the four seam types object, compile, preprocessor and link seam.

Although there are already various existing mock object libraries for C++, we believe that creating mock objects is still too complicated and time-consuming for developers. Mockator provides a mock object library and an Eclipse plug-in to create mock objects in a simple yet powerful way. Mockator leverages the new language facilities C++11 offers while still being compatible with C++98/03.

Management Summary

In this report we discuss the development of an Eclipse plug-in to refactor towards seams in C++ and the engineered mock object solution. This master's thesis is a continuation of a preceding term project at the University of Applied Sciences Rapperswil by the same author.

Motivation

High coupling, hard-wired and cyclic dependencies lead to systems that are hard to change, test and deploy in isolation. Unfortunately, legacy code often has these attributes. Feathers' seam model helps us in recognising opportunities to inject dependencies from outside, thus getting rid of fixed dependencies. There are different kinds of seam types. In C++ we have object, compile, preprocessor and link seams. While object seams based on inheritance are well-established, the others are lesser-known. Although seams are a valuable technique, they are hard to apply without refactorings and tool chain support. The latter is because preprocessor and link seams are highly dependent on the chosen tool chain and require a significant amount of manual work to specify the required options for the used tools. We are convinced that our IDE of choice Eclipse CDT would benefit from supporting seams.

After we have applied seams in a legacy code base, we have an improved design and remarkably enhanced the testability of our code base. When we want to test a piece of code in isolation, we often cannot inject the real objects because they might have non-deterministic behaviour, are slow or communicate with subsystems which we want to avoid in our unit tests. This is where we use test doubles instead like fake and mock objects. Although there are already various existing mock object libraries for C++, we believe that creating mock objects is still too complicated and time-consuming for developers. Most often, common mock object libraries tend to overuse macros and hide the code from the developer which might lead to challenging debugging sessions when problems arise. We think that it is better to have the code necessary for mock objects beside our unit tests to enhance transparency and to increase the possibilities of the programmer to adapt the code if necessary when our library does not provide a solution for a particular problem. Beside this, we also believe that current mock object libraries overuse inheritance and are lacking IDE support.

Goals

The primary goal of this master's thesis is to provide support for the four seam types object, compile, preprocessor and link seam for Eclipse CDT. The other goals are all related to our mock object solution. The existing C++11 based library should be adapted in order to support C++03. To make the handling of mock objects easier, we have to develop functionality to make the creation and adaption of mock objects more comfortable.

Results

The engineered Eclipse plug-in is able to generate code and the necessary infrastructure for all four seam types object, compile, preprocessor and link seam. To offer object seams, we have developed a new refactoring to extract an interface. Mockator Pro is able to recognise missing member functions in both object and compile seam classes. We developed a useful implementation of preprocessor seams which are especially handy for tracing system function calls. For our link seam implementation we provide three types to shadow and wrap functions which work with static and shared libraries. We tried to support both Linux and Mac OS X for our link seams wherever these platforms and their corresponding GCC tool chain support them themselves. All implemented seams can be disabled or removed easily when the need arises.

Our header-only mock object library now also supports C++03 beside C++11. The Eclipse plug-in is able to generate code for both standards which can be chosen in the project settings. Because we recognised that it is often beneficial to just mock a single function instead of extracting an interface or a template parameter, we implemented mocking of functions including wizard support for CUTE. We implemented various convenience functions to make working with mock objects easier like moving them to a namespace, converting fake to mock objects, toggle recording on a member function level and recognising inconsistent expectations. Beside missing member functions and constructors we now also recognise missing operators.

To make our work more popular, we presented Mockator Pro at the ICSE workshop AST2012 located at University of Zürich Irchel. Additionally, we wrote an article for the magazine Overload about refactoring towards seams with our Eclipse plug-in.

Contents

1. Introduction

The present master's thesis is a continuation of the preceding term project *Mockator* [Rü11] at the University of Applied Sciences Rapperswil. Its goal is to help C++ developers in applying seams in their code base and in creating mock objects for our IDE of choice, Eclipse C/C++ Development Tooling (CDT).

In this chapter we will explain the problems of unwanted software dependencies, how to get rid of them by making use of seams, and how we can use test doubles when the real objects would be a burden for our unit tests. At the end of this chapter we will also discuss the motivation and the goals of this thesis.

1.1. The Perils of Software Dependencies

A critical problem in software development are unwanted dependencies. Obviously, dependencies are necessary to make software components work together. But high coupling, hard-wired and cyclic dependencies between classes and subsystems are the perils of software dependencies. They lead to applications that are difficult to change, test and deploy in isolation.

Breaking unwanted dependencies is often a precondition to be able to change existing code. Feathers has identified four triggers for changing existing code [Fea04]: adding new functionality, fixing a bug, applying refactorings and code optimisations. What all four triggers have in common is the fact that we should have tests in place before we apply the changes.

According to Feathers' definition, legacy code is code without unit tests [Fea04]. An important preliminary to introduce unit tests is to break existing dependencies. This is especially hard because before we change existing code we should have tests in place. But in order to apply them, we first have to change the code. This dilemma is called *The Legacy Code Dilemma* [Fea04].

Feathers' *seam model* helps us to reason about the several possibilities that exist to break dependencies.

1.2. Breaking Dependencies With Seams

The goal in breaking dependencies is to have a place where we can alter the behaviour of a program without modifying it in that place — which is exactly the definition of a *seam*. This is important because editing the source code is often not an option (e. g., when a function the code depends on is provided by a system library). Moreover, it is often better to avoid making the changes inline because the code will otherwise get longer and will never be properly tested[1].

```
#include "creditcard.h"
void CreditCard::charge(double amount) {
  if (!isValidAmount(amount)) {
     throw InvalidAmountException();
  }
  verifyCredentialsOnline(cardHolder);
  cardHolder.charge(amount);
}
```

Listing 1.1: *Example of a C++ member function with a call to a global function which we want to avoid in our unit tests. The question is if this code contains a seam.*

Consider the code in listing 1.1 where the global function `verifyCredentialsOnline` is used to communicate with another subsystem. We would like to avoid this during our unit tests because communicating across a network is something we generally do not want to do in our unit tests [Fea04]. What if we would like to circumvent the call to `verifyCredentialsOnline` without changing the existing code? This leads to the question if this code contains a seam. And indeed, there is a seam in the place where we call `verifyCredentialsOnline`.

```
struct CreditCard {
  // as before
  virtual void verifyCredentialsOnline(CardHolder const&);
};
void CreditCard::verifyCredentialsOnline(CardHolder const& ch) {
  ::verifyCredentialsOnline(ch);
}
struct TestingCreditCard : CreditCard {
  void verifyCredentialsOnline(CardHolder const&) {
  }
};
```

Listing 1.2: *Object seam to avoid the call to the global function* `verifyCredentialsOnline`*.*

[1]This is why Feathers recommends the two techniques *Sproud Method* and *Sproud Class* in [Fea04] to extract the necessary changes and delegate to them in the existing code instead of applying the changes inline.

We can make use of this seam by providing a new virtual member function and delegate to the global function by using C++'s scope operator `::` [Fea04]. To avoid the call in our unit tests, we subclass `CreditCard` and override the member function `verifyCredentialsOnline`, as it is shown in listing 1.2.

This type of seam is called *object seam*. C++ offers a great facility of language mechanisms to create seams. Beside this classic way of using subtype polymorphism which relies on inheritance, C++ also provides static polymorphism through template parameters. With the help of the preprocessor or the linker we can use additional ways of seams.

1.3. Unit Testing With Test Doubles

Once we have achieved to break dependencies in our legacy code base, our code is not relying on fixed dependencies anymore, but instead asks for collaborators trough dependency injection. Not only our design has improved much, but we are now also able to write unit tests for our code. Sometimes however, it is impractical or impossible to exercise our code with real objects. In these situations, we might use test doubles, a term used by Meszaros [Mes07].

A specific type of test doubles are *mock objects*. Mackinnon et al. introduced them in the paper "Endo-Testing: Unit Testing with Mock Objects" [MFC01] presented at the XP 2000 conference. They called it Endo-Testing[2] because the mock objects are passed to the target code which they test from inside. Mock objects are used as placeholders for real objects and provide implementations to verify collaboration with other classes. They have the same interface as real objects, which leads to the fact that client code does not have to be aware if it works together with real or mock objects.

Mock objects are used in unit testing whenever it is impractical or impossible to exercise real objects. If a real object possesses one or more of the following criterias, mock objects might help in testing objects in isolation [Wik11]:

- supplies non-deterministic results (e. g., the current time)
- contains states that are difficult to create or reproduce (e. g., network errors)
- is slow (e. g., databases)
- does not exist yet or may change behaviour
- would need to have additional behaviour only necessary for testing purposes

According to Meszaros, we replace a component on which the target code depends with a "test-specific equivalent" [Mes07]. Other types of test doubles are dummies, fake objects and stubs. Test doubles come from the world of test-driven development (TDD) and are closely related to class / responsibility / collaboration (CRC) cards [WBM03].

[2]"endo": prefix derived from Greek, means "inside".

The idea is that unit tests validate the responsibilities of a class whereas test doubles take the part of the collaborators. To better understand these concepts, we are now going to take a deeper look how unit testing with fake and mock objects works.

Figure 1.1 shows an example of a class relationship where `LaserPrinter` satisfies the property of being slow, difficult to setup and hence hard to test. Consider that we would like to test `CheckIn` in isolation. We somehow have to break the dependency to `LaserPrinter` because we want to have fast and independent unit tests, otherwise they cannot be considered as being *real* unit tests [Fea04]. Note that changing `CheckIn` is hard because we are violating the open-closed principle [Mey00]. This is because if we wish that `Checkin` uses another kind of printer it must be changed to use the name of the new printer class — hence it is not closed for modifications.

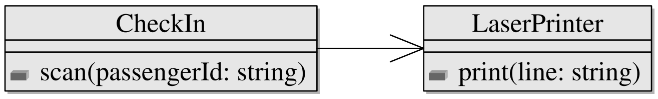


Figure 1.1.: *The hardwired dependency between `CheckIn` and `LaserPrinter` leads to a violation of the open-closed principle.*

One of the many possibilities to break hardwired dependencies is to extract an interface and to inject the dependency into `CheckIn`, thus making use of an object seam. This leads us to the design presented in figure 1.2.

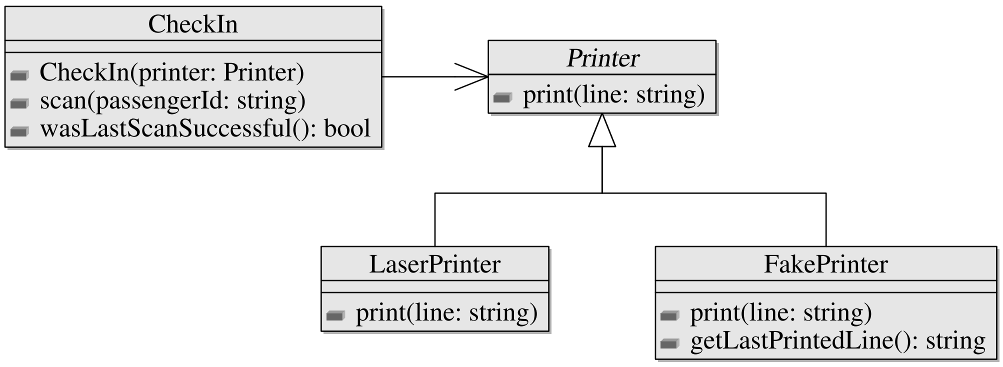

CheckIn
CheckIn(printer: Printer)
scan(passengerId: string)
wasLastScanSuccessful(): bool
Printer
print(line: string)
LaserPrinter
print(line: string)
FakePrinter
print(line: string)
getLastPrintedLine(): string

Figure 1.2.: *Extraction of the new interface `Printer` leads to this class hierarchy where we now make use of an object seam.*

We now have a new interface `Printer` and `CheckIn` only depends on this. The dependency is injected via the constructor of `CheckIn` and is not fixed anymore. To mimic the behaviour of `LaserPrinter` in our unit tests, we introduce a fake object called `FakePrinter`[3]. To verify that `CheckIn::scan` calls `Printer::print` we also provide a supplementary member function called `FakePrinter::getLastPrintedLine`. We are now ready to test the functionality of `CheckIn` in isolation as shown in listing 1.3[4].

[3]Note that the definitions of fakes and stubs vary greatly in the relevant literature. [Mes07] would call `FakePrinter` a stub, whereas [Fea04] names this a fake.

[4]We will use CUTE [Som11] for writing unit tests in all code examples of this report.

```
void testScanPassenger() {
  // setup
  FakePrinter printer;
  CheckIn checkin(printer);
  // exercise
  checkin.scan("132-763-945");
  // verify
  ASSERT(checkin.wasLastScanSuccessful());
  ASSERT_EQUAL("132-763-945", printer.getLastPrintedLine());
}
```

Listing 1.3: *State verification using the classic xUnit testing approach.*

The unit test follows the typical work flow proposed by the xUnit pattern: setup, exercise, verify and teardown ([Fow11]). In this example, we do not need a separate teardown phase because we already got rid of the dependency to `LaserPrinter` which would have made it necessary to close the printer connection — something we would normally handle in C++ with *Resource Acquisition is Initialisation* (RAII) and the help of automatic objects which are destroyed automatically at the end of the function providing the possibility to release resources.

To make further explanations easier, we introduce a few important terms: `CheckIn` is commonly referred as the *system under test* (SUT) whereas `FakePrinter` is a *collaborator* or a replacement for a *depended-on component* (DOC) [Mes07][5]. Because we are examining the state of the SUT and its collaborators after the exercising phase, Fowler calls this kind of testing *state verification* [Fow11]. In contrast, he describes testing with mock objects as *behaviour verification*.

The code in listing 1.4 is used to test the same example, but this time with the help of mock objects. We use the mock object library Google Mock [Goo12] here because we do not want to write the boilerplate code ourselves and we also want to show how common mock object libraries work. The collaborator in this example is called `MockPrinter`. As Google Mock is based on using inheritance, we inherit from the base class `Printer`. Google Mock expects that we provide macros for all member functions we want to get called during the exercise phase. The format of the mock definitions is `MOCK_METHODn()`, where `n` stands for the number of function arguments.

The important part of this example is the specification of the expectations on the mock object. These indicate which member functions should be called how many times and which argument values are expected during the SUT is exercised. Google Mock allows us to specify these expectations through a fluent interface API. In this example, we specify that the method `print` has to be called exactly once with the argument `"132-763-945"`. After this, we specify the actions to be performed on the SUT. Here,

[5]A DOC is generally replaced by a test double whenever we cannot use the former.

we call the member function `scan`. The verification that all expectations have been satisfied is automatically done by Google Mock when the mock object is destroyed.

```cpp
#include "gmock/gmock.h"
struct Printer {
  virtual ~Printer() {}
  virtual void print(std::string) = 0;
};
struct MockPrinter : Printer {
  MOCK_METHOD1(print, void(std::string));
};
void testScanPassenger() {
  // setup
  MockPrinter printer;
  // expectations
  EXPECT_CALL(printer, print("132-763-945")).Times(1);
  // exercise
  CheckIn checkin(printer);
  checkin.scan("132-763-945");
}
```

Listing 1.4: *Behaviour verification with Google Mock. Note the use of inheritance and the macro we need to provide for every member function.*

The key difference between state and behaviour verification is how we verify if `CheckIn` did the right thing in its interaction with `Printer` [Fow11]: With state verification, we executed asserts against the state of `CheckIn` and `FakePrinter` whereas with behaviour verification we verified if `CheckIn` called the right member functions on `Printer`.

Behaviour verification is what makes mock objects different from its test double colleagues: Mock objects test interactions between objects which is done by recording function calls and verifying that all expected communication happened as specified. Both mock and fake objects stand in for the collaborators of the SUT, but only the mock objects test the collaboration. Fake objects — on the other hand — just simulate the collaborators and only implement a subset of their functionality [Mes07], e. g., to use a fake database with hash tables in lieu of a real database connection. Therefore, they do not need to be configured with collaboration expectations as it is the case with mock objects.

TDD in combination with mock objects often results in compositions of objects communicating through narrowly defined interfaces and flat class hierarchies [FPM+04]. Nevertheless, we have to be aware that mock objects can couple unit tests closely to the actual implementations when developers specify too detailed expectations about object interactions which might result in brittle (i. e., unstable) unit tests.

1.4. Motivation

Making use of seams is a valuable technique, but it is tedious to apply them without automated refactorings and tool chain configuration support. As we will show in chapter 2, applying seams often needs changes in the build settings of the project and code that is hard and cumbersome to create by hand. So far, no C++ IDE is available to support the developer in the extend we describe it here.

Although there are already various existing mock object libraries for C++, we believe that creating mock objects is still too complicated and time-consuming for developers. The overwhelming part of the existing C++ mock object libraries [Goo12, Pou11] relies on inheritance which has the well-known software engineering problems which we will discuss in section 2.2.1. Beside this, these libraries often lack an integration into an IDE. This is especially cumbersome because they require us to write a lot of boilerplate code.

As we have seen in section 1.3 with Google Mock, a programmer has to learn a new domain-specific language (DSL) to specify behaviour and collaboration expectations. Beside the time to get accustomed to the DSL the mock object library provides, the programmer also needs to write the code for the interface for every new mock object with all drawbacks more code brings with it.

Although Google Mock contains a Python tool that is able to generate the interface of the mock object shown in listing 1.4, this would result in a disruption of the development flow — especially with its lacking integration into an IDE. Beside this, Google Mock also relies heavily on preprocessor macros, which makes it hard to understand and debug. Furthermore, as the code is hidden behind them, we cannot easily adapt it to our needs if the library does not offer a functionality we are looking for.

There are other C++ mock object libraries like amop [unk11a], MockitoPP [Pou11] and Hippo Mocks [LB] which do not force us to derive from a base class as Google Mock does. But these often depend on certain compiler-dependent features like run-time type information (RTTI) .

Because of the mentioned drawbacks, we are really convinced that there is a need for an integrated solution which is able to create fake and mock objects for Eclipse CDT. Furthermore, the use of new C++11 language features will allow us to greatly reduce the need for preprocessor tricks used by most of the existing mocking libraries. One example is the application of variadic templates as we will see in chapter 3.

We also believe that both fake and mock objects are valuable and both need to be supported to assist users practising TDD or putting their legacy code base under tests. Sometimes we only want to specify the behaviour of objects for testing purposes. This would call for fake objects, as it is the easiest thing to do. When we really want to verify the collaborations of objects, we go for mock objects.

1.5. Thesis Goals

Based on the results of the foregoing term project, the goal of this master's thesis is to make Mockator able to support the C++ developer in applying seams and creating test doubles. This thesis can therefore be grouped into the two main objectives "Refactoring Towards Seams" and "Enhanced Mock Object Support", resulting in the new *Mockator Pro*[6].

1.5.1. Refactoring Towards Seams

The term project focused on compile seams only. C++ offers three other seam types beside compile seams: object, preprocessor and link seams. Because all types of seams can be valuable depending on the concrete scenario of breaking dependencies in existing code, Mockator has to support these seam types as well.

For object seams a new refactoring to extract an interface from a given class has to be written. This is the first step in applying object seams. The creation of a test double making use of subtype polymorphism has to be offered for convenience. The existing infrastructure to collect missing member functions in the injected test doubles via template parameters has to be enhanced for subtype polymorphism.

Preprocessor seam is an important technique for debugging purposes like tracing function calls and should be supported as well.

With link seams we alter the build system to prefer injected code instead of the production one at link-time. Current solutions focus on the C programming language and require a significant amount of manual work. Mockator should be able to automate these tasks for the C++ programming language. Support for both static and shared libraries for the code to be replaced is demanded.

For all supported seam types it is important to be able to temporarily deactivate them.

1.5.2. Enhanced Mock Object Support

The created mock object library of the term project was focused on C++11 and does not work with the older C++ standards 98/03. Because the bulk of existing code is written in C++98/03 and there will also be new projects still going to use these older standards, Mockator needs to support these beside C++11.

Mockator was so far only able to mock classes and its member functions. Sometimes, mocking a free function is easier and sufficient and should therefore be supported as well.

[6]Note that we will usually apply the name Mockator in this thesis and only refer to Mockator Pro for direct comparisons.

For the convenience of our users we should provide the ability to toggle mock support on a member function level in the test double. If the developer manually adjusted the registration of the function calls and forgot to adapt the expectations accordingly, Mockator should be able to detect and correct this.

Mockator creates the test doubles in the function of the unit test as a local class. Although this has the advantage of the enhanced locality that makes it easier to keep the expectations and the test double in sync, this comes at the price of more code that is placed in the unit test function compared to other mocking libraries where this is often hidden behind macros. We therefore have to provide a functionality to move the test doubles out of the functions. As a target we think it is appropriate to move them to a namespace in the translation unit of the unit test.

The current solution detects missing member functions and constructors of the injected test doubles which are used by the SUT. It does not recognise missing operators which is a further goal for this thesis.

Other mock object libraries like Google Mock provide sophisticated instruments to specify expectations. Mockator currently only supports comparisons based on equality of the function arguments. To make the specification of expectations more powerful and convenient, we have to implement the possibility to use regular expressions.

The current mock object library of Mockator is not thread-safe. This issue has to be analysed in this thesis.

Because Mockator Pro will probably be bundled together with the CUTE testing framework in the near future, its collaboration has to be improved. The project wizard of CUTE has to be adapted via its extension points to add mock support to a new project. Although we improve the collaboration with CUTE, Mockator should still be able to work without it.

1.6. About This Report

In this report we describe goals, implementation details, solutions to encountered problems and the outcome of this master's thesis. In chapter 2 we describe the four kinds of seam types object, compile, preprocessor and link seam. Myers and Bazinet discussed link seams to intercept functions for the programming language C [MB11]. Our contribution is to show how link seams can be accomplished in C++ where name mangling comes into play and how it is possible to shadow functions. Feathers presented object, preprocessor and link seams [Fea04], while the latter two are only discussed briefly and we have chosen another approach to accomplish them. We also introduce a fourth kind of seam called compile seam. For every seam we also discuss how we implemented its support for Eclipse CDT.

In chapter 3 we present the C++ mock object library. We first introduce its basic functionality, will then discuss the new support for the older C++ standards, specifying expectations with regular expressions and the current thread-safety issues. The mock object support for Eclipse CDT is discussed in chapter 4. In chapter 5 we show the architecture of the developed Eclipse plug-in and a few important implementation details. The final chapter 6 presents the project results, describes open problems and possible enhancements, contributions made to conferences and magazines, and concludes with a personal review and acknowledgement.

2. Refactoring Towards Seams

In this chapter we present four kinds of seams that exist in C++ and the corresponding refactorings we created for Eclipse CDT to achieve them.

2.1. Object Seam

In this section we discuss object seams which are probably the most common seam type as they are based on inheritance.

2.1.1. Introduction

To start with an example, consider Listing 2.1 where `GameFourWins` has a hard coded dependency to class `Die`.

```cpp
// Die.h
#include <cstdlib>
struct Die {
  int roll() const { return rand() % 6 + 1; }
};
// GameFourWins.h
#include "Die.h"
#include <iosfwd>
struct GameFourWins {
  void play(std::ostream& os);
private:
  Die die;
};
// GameFourWins.cpp
#include "GameFourWins.h"
#include <iostream>
void GameFourWins::play(std::ostream& os = std::cout) {
  if (die.roll() == 4)
    os << "You won!" << std::endl;
  else
    os << "You lost!" << std::endl;
}
```

Listing 2.1: *Code with fixed dependencies harming testability.*

According to Feathers, the call to `play` is not a seam because it is missing the essential property every seam has: an *enabling point*. We cannot alter the behaviour of the member function `play` without changing its function body. One reason for this restriction is that the member variable `die` is based on the concrete class `Die`. Furthermore, we cannot subclass `GameFourWins` and override `play` since `play` is monomorphic[1]. This fixed dependency also makes it hard to test `GameFourWins` in isolation because `Die` uses C's standard library pseudo-random number generator function `rand`. Although `rand` is a deterministic function because calls to it will return the same sequence of numbers for any given seed, it is hard and cumbersome to setup a specific seed for our purposes.

The classic way to alter the behaviour of `GameFourWins` is to inject the dependency from outside by using subtype polymorphism[2]. We first have to apply the refactoring *Extract Interface* [Fow99] to enable subtype polymorphism. Then we provide a constructor to inject the dependency from outside[3]. The resulting code is shown in Listing 2.2.

```
//IDie.h
struct IDie {
  virtual ~IDie() {}
  virtual int roll() const =0;
};
//Die.h
#include "IDie.h"
#include <cstdlib>
struct Die : IDie {
  int roll() const { return rand() % 6 + 1; }
};
//GameFourWins.cpp
#include "Die.h"
#include <iostream>
struct GameFourWins {
  GameFourWins(IDie& die): die(die) {}
  void play(std::ostream&) { /* as before */ }
private:
  IDie& die;
};
```

Listing 2.2: *Code after applying the refactoring extract interface to achieve an object seam.*

This way we can now inject a different kind of `Die` depending on the context we need. This is a seam because we now have an enabling point: The instance of `Die` that is passed to the constructor of `GameFourWins`.

[1]Monomorphic as the opposite of polymorphic, hence not virtual. This is a term used by Martin in [Mar08].
[2]Because of the way functions are resolved through the vtable at run-time, this type of polymorphism is often also called run-time polymorphism. In type theory it is named inclusion polymorphism [CW85].
[3]We could also have passed an instance of `Die` to `play` directly.

2.1.2. Use Case

The following use case describes the steps necessary to refactor towards an object seam with Mockator:

UC 1	*Create object seam*
Primary Actor	C++ Developer
Precondition	The developer has a fixed dependency in the code which needs to be replaced by an alternative implementation.
Postcondition	An object seam exists that can be used with the alternative implementation in the newly created subclass.
Main sequence	1. *User:* Extracts an interface from the chosen class (see 2.1.4). 2. *System:* Creates a new interface class which the chosen class inherits from. 3. *User:* Provides a constructor or a member function in the SUT with the interface parameter type to inject the dependency. 4. *User:* Creates an instance of the SUT and injects a new class name. 5. *System:* Shows marker with quick fix to create the object seam class. 6. *User:* Applies the quick fix. 7. *System:* System creates the new local subclass and shows a marker for every pure virtual member function that needs to be implemented. 8. *User:* Applies one of the quick fixes (with or without recording call support). 9. *System:* Creates the missing member functions. 10. *User:* Implements an alternative implementation for the object seam in the member functions.

2.1.3. Implementation

After extracting an interface and providing a constructor or member function to inject a dependency as shown in listing 2.2, we can use Mockator to create a local test double class. Whenever an unresolved identifier is used as an argument for passing it to a reference or pointer parameter of a constructor or member function and the class type of the parameter can be considered as a valid base class, Mockator creates a marker with a quick fix (see listing 2.3[4]). Obviously, we cannot use pass by value when we want to make use of subtype polymorphism in C++ because of object slicing. Valid base classes are all classes that have a virtual non-private destructor, as it is created by our extract interface refactoring.

```
void testGameFourWins() {
  GameFourWins game(die);
}
```

Listing 2.3: *Using an unresolved identifier to the reference parameter of the constructor of the SUT yields a CodAn marker.*

After applying the quick fix corresponding to the marker, Mockator creates a local class which inherits from the class type of the parameter it is injected to. Note that we define an instance of the newly created class named according to the unresolved identifier, as can be seen in listing 2.4.

```
void testGameFourWins() {
  struct Die : IDie {
  } die;
  GameFourWins game(die);
}
```

Listing 2.4: *After applying the quick fix to create a local subclass that is injected into the SUT Mockator detects that an abstract class is instantiated and creates another marker.*

Actually, there would be no need to name the local class in this example because after the creation of the instance we never use it anymore. Therefore, we could create an anonymous local class here. Because we need to define a constructor in the case we have to call the constructor of the base class (when there is no default constructor in the base class), we need a class name and therefore specify one.

Mockator detects that the abstract class `Die` is instantiated and therefore creates another marker. After applying the marker, Mockator creates default implementations[5] for all pure virtual member functions of the base classes. Note that Mockator by default only marks incomplete classes when they are part of or referenced by a test function. This reduces the amount of analysis work that has to be done. A test function is defined as a niladic function (i. e., a function with zero parameters [Mar08]). Which functions to consider as test functions can be changed in the project settings where it is also possible to recognise all functions as test functions. Note that we do not restrict test functions to be free functions and therefore also support test functors from CUTE.

[4]Note the use of the red wave in the listing which we use to document code analysis (CodAn) markers in this thesis.

[5]This happens in the case the user has not chosen to use recording support for mock objects, as we will discuss in chapter 4.

Listing 2.5 shows an example where we additionally have added a non-default constructor to the base class `IDie`, hence Mockator needs to create code to call the base class constructor in the new local subclass to have no compile errors. If there are several base class constructors, Mockator always calls the one with the least number of parameters. Note the use of the default instantiations by using the new C++11 initialiser syntax.

```
struct IDie {
  // as before
  IDie(int i);
};
void testGameFourWins() {
  struct Die : IDie {
    Die() : IDie{int{}} { }
    int roll() {
      return int{};
    }
  } die;
  GameFourWins game(die);
}
```

Listing 2.5: *After applying the quick fix Mockator creates default implementations for all pure virtual member functions.*

2.1.4. Extract Interface Refactoring

When using object seams we want to apply the Liskov substitution principle and subtype polymorphism and inject our test doubles as subclasses of the dependencies the SUT works with. A preliminary for subclassing is to have a base class or interface which the SUT works with. To achieve this, we implemented the new refactoring *Extract Interface* [Fow99], as there is no such support in Eclipse CDT yet.

Introduction

In contrast to Java, C++ does not have a dedicated keyword to express interfaces. An interface in C++ can be emulated by a class having nothing but pure virtual function declarations. As there is not first class support for interfaces, we have to be aware of a few specialities that otherwise can lead to problems, which we will discuss in this section.

The use of interfaces in object-oriented designs has numerous advantages. When our code depends on interfaces instead of concrete implementations, we normally have to change our code less often because interfaces typically change far less [Fea04]. The use of interfaces helps in applying the dependency inversion principle [Mar02], because high-level and low-level modules depend upon abstractions. This decreased coupling again results in shorter build times because changes in the implementation classes are separated from the interface and therefore code that is only working with the interface is ideally not affected by these.

However — as it is often the case in programming — it is a matter of using the right tool for the given job. In base classes with increased costs of change (e. g., in libraries and frameworks), it is often advisable to make virtual functions non-public, making use of the *non-virtual interface pattern* (NVI) instead of the approach with public virtual functions presented here [Mey05, SA04].

Because we often encounter designs where interfaces are used sparingly in legacy code and instead high coupling dominates, we need a refactoring for this job. Fowler describes the mechanics of an extract interface refactoring in [Fow99] in four steps:

1. Create an empty interface
2. Declare the common operations in the interface
3. Declare the relevant class(es) as implementing the interface
4. Adjust client type declarations to use the interface

Use Case

UC 2	*Extract interface*
Primary Actor	C++ Developer
Precondition	The developer wants to extract an interface from a given class due to hardwired dependencies to a collaborator.
Postcondition	A class interface with the chosen member functions is generated in a new header file. If desired, existing references are adapted to use the interface class instead of the concrete one.
Main sequence	1. *User:* Selects a class name to extract an interface from. 2. *System:* Starts the refactoring process. 3. *System:* Asks the user for the name of the new interface, which member functions should be specified in the interface class and if existing references should be adapted. 4. *User:* Makes the decisions for the new class interface. 5. *System:* Shows a preview for the refactoring. 6. *User:* Accepts changes. 7. *System:* Creates an interface in a new header file and uses the interface type wherever possible.

Implementation

From the concrete class we extract an interface from, we collect all public non-static member functions to use them as pure virtual member function declarations in the interface class. Additionally, we create an empty implementation of a virtual destructor. The latter is needed in order to let the destructors of derived classes be invoked properly when a polymorphic object is deleted through a base class pointer, otherwise possibly resulting in a resource leak [Mey05].

For the newly created interface class, we use the keyword `struct` because its default visibility is public which allows us to avoid the clutter of defining a public visibility label. For the concrete class, we add public inheritance to the newly created interface. If we are extracting an interface from a class that uses the `class` keyword, we have to explicitly use public inheritance to make it work.

```
//Foo.h
#include <map>
#include <string>
class A;
typedef std::map<std::string, std::string> KeyValStore;
namespace Ns {
struct Foo {
  void bar(KeyValStore const& s);
  void bar(KeyValStore const& s) const;
private:
  void ignoreNonPublicMemFuns();
  A* a;
};
}
//SUT.h
#include "Foo.h"
struct SUT {
  SUT(Ns::Foo const& foo) : foo(foo) {}
  void doit() {
    KeyValStore kv;
    foo.bar(kv);
  }
private:
  Ns::Foo const& foo;
};
```

Listing 2.6: *Starting position to show several aspects of the extract interface refactoring. The initial text selection is shown in yellow.*

To assist the user in creating narrow interfaces, we only select member functions in the refactoring dialog by default that are used in the SUT class for the selected class name[6]. As an example, consider the code in listing 2.6 where only the `const` version of `bar` is used in the class `SUT`. In the refactoring dialog, only this member function will therefore be selected by default. Listing 2.7 shows the result of the refactoring process if the user accepts this suggestion.

```
#ifndef FOOINTERFACE_H_
#define FOOINTERFACE_H_
#include <map>
#include <string>
class A;
typedef std::map<std::string, std::string> KeyValStore;

namespace Ns {
struct FooInterface {
  virtual ~FooInterface() { }
  virtual void bar(KeyValStore const& kv) const =0;
};
}
#endif
```

Listing 2.7: *Newly created interface class `FooInterface` with the used public member function `bar` and the moved includes, forward declarations and the `typedef`.*

For every newly created interface, a header file including appropriate include guards is created, as can be seen in listing 2.7. If the class the interface is extracted from is already part of a namespace as in this example, the newly created interface is put in the same one too. In order to have necessary type definitions available, we move all includes, forward declarations and `typedef`'s from the place where the concrete class is defined to the newly created header file. This makes the header file of the interface class self-sufficient. Although this might move dependencies that are only used in the implementation to the interface file, we value correctness over optimisation here and let other tools [Fel12] clean up unnecessary includes.

```
#include "Foo.h"
struct SUT {
  SUT(Ns::FooInterface const& foo) : foo(foo) {}
  void doit();
private:
  Ns::FooInterface const& foo;
};
```

Listing 2.8: *Replaced references to concrete class by using the new interface name `FooInterface`.*

As a last step, we have to adapt all references to the chosen class and use the name of the interface if possible. Wherever the chosen class name is used for a reference or

[6]This selection is visualised in the code listing by a yellow marker. We will use this to emphasise all selections in this thesis.

pointer type, we us the name of the new interface. Listing 2.8 shows how the class SUT looks like after applying the refactoring.

Non-virtual Functions

In C++ we need to be aware of the potential side effects of extracting interfaces in terms of non-virtual member functions. If the class we extract an interface from has a subclass and we add the signature of a non-virtual member function to the newly created interface, we can break existing client code. Consider listing 2.9 with the Die class known from previous examples and AlwaysSixDie which — as its name implies — always returns 6. Note that AlwaysSixDie::roll shadows Die::roll because it has the same function signature and the latter is not defined virtual.

```cpp
#include <cstdlib>
struct Die {
  int roll() const {
    return rand() % 6 + 1;
  }
};
struct AlwaysSixDie : Die {
  int roll() const {
    return 6;
  }
};
```

Listing 2.9: *Subclassing with non-virtual member functions leads to shadowing.*

If we now extract an interface for Die, Croupier in listing 2.10 is broken if we pass it an instance of AlwaysSixDie. Behaviour changes because now subtype polymorphism comes into play where before only the static type of Die was considered for resolving member function calls (i. e., the call was statically bound). This happens in C++ because if we make a member function virtual in a base class, all member functions that override it in subclasses become virtual too[7].

Of course, creating a member function in a derived class with an equal signature as a non-virtual one in the base class is bad practice[8] and should be avoided [Mey05]. But we need to consider this case because preserving behaviour during refactoring is a very important goal we try to achieve. We therefore create a warning in our refactoring with an indication that it might be better to add a new virtual member function with a different name and delegate to the non-virtual one [Fea04].

[7]This is in contrast to Java where all member functions are virtual by default.

[8]Eclipse CDT creates a marker for shadowing functions since version 8.0.0 that — when clicked — jumps to the shadowed function.

```
struct DieInterface {
  virtual ~DieInterface() { }
  virtual int roll() const = 0;
};
struct Die : DieInterface {
  // as before
};
struct Croupier {
  void doJob(Die const& die) {
    if (die.roll() == 6) {
      // jackpot
    }
  }
};
```

Listing 2.10: *Change of behaviour in* `Croupier` *caused by the extract interface refactoring.*

Public Virtual vs. Protected Non-virtual Destructor

It does not always make sense to have a virtual destructor. There are situations where it is more beneficial for a class to have a non-virtual and non-public (i. e., protected) destructor. This is the case when we do not intend to have pointers to a class interface passed around our program, therefore the interface is not intended to be a *usage type* [Rad11]. Listing 2.11 shows an example of a class interface that is not intended to be a usage type. This is a typical scenario for *mixin interfaces* [Rad11].

```
#include <iosfwd>
struct Serializable {
  virtual void load(std::istream& in) = 0;
  virtual void save(std::ostream& out) = 0;
protected:
  ~Serializable();
};
struct DieInterface {
  virtual ~DieInterface() { }
  virtual int roll() const = 0;
};
struct Die : DieInterface, Serializable {
  //...
};
```

Listing 2.11: `Serializable` *is not a usage type and is better served with a protected non-virtual destructor.*

Although we originally planned to let the user decide if the destructor should be public and virtual or protected and non-virtual, we decided against this approach because it may be too hard for a programmer to make this decision upfront.

Prohibit Incidental Assignments

We again consider the interface class `DieInterface` from listing 2.10. Written like this, the compiler will provide a copy assignment operator, a default and a copy constructor. Although interface classes are stateless and allowing assignment therefore is harmless, prohibiting it would be a safety contribution to avoid errors. We think that a refactoring should also enforce best practices where possible and reasonable and it would be easy for us to do it here. Despite of this, an interface classes assignment semantics can be considered inappropriate and irrelevant [Rad11]. We could make a proposal in the refactoring dialog to enforce this which would result in the code shown in listing 2.12[9].

```
struct DieInterface {
  // as before
private:
  DieInterface& operator=(const DieInterface&);
};
```

Listing 2.12: *`DieInterface` with a private assignment operator to prohibit incidental assignments.*

However, we decided against this approach because it might again be a design decision that is too hard for a programmer to make at the beginning. An additional disadvantage would be that it has the side effect of prohibiting the use of the assignment in subclasses.

Internal Design

In this section we describe the internal design of the implemented refactoring. To explain the most important classes that an LTK (Language Toolkit) based refactoring in CDT uses, we provide the class diagram of figure 2.1. The delegate for our UI action to start the extract interface refactoring via the Eclipse menu is contained in the class `EIDelegate`[10] which is registered in our `plugin.xml`. The delegate creates an instance of `EIAction` and calls `run` on it. This action then creates an `EIRunner` which provides the wizard and the refactoring to run and finally initiates the wizard. Note that its parent class `RefactoringRunner2` provides the functionality to create and finally dispose the `RefactoringASTCache` which is a CDT provided class to cache abstract syntax trees (AST) for given translation units.

Our provided class `EIRefactoring` is responsible for checking pre- and postconditions and the actual change generation for the refactoring. `EIWizard` controls the single page for our refactoring which is provided by the class `EIWizardPage`. This class is responsible for creating all the UI elements for the refactoring dialog.

[9]Note that in C++11 we would use the new `delete` keyword instead of declaring the assignment operator private.

[10]We use the shortform EI for "ExtractInterface" in this section because of space reasons.

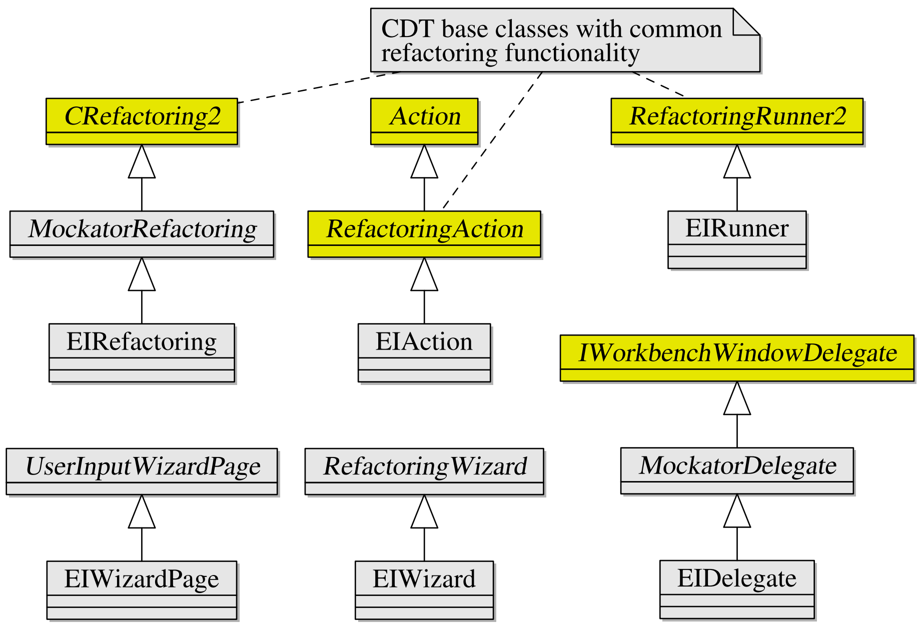


Figure 2.1.: *Extract interface refactoring class hierarchy. Yellow tagged classes are provided by Eclipse.*

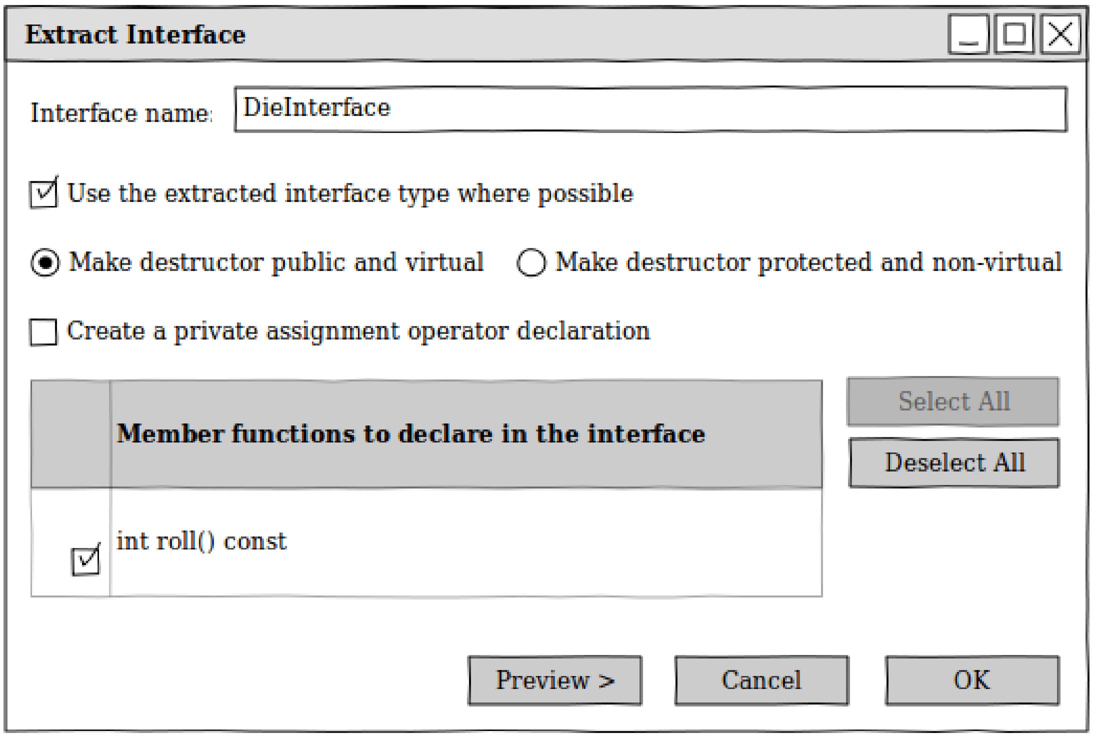
External Design

Figure 2.2 shows a screen mock-up for the new extract interface refactoring dialog. The user can specify the name of the interface, if existing references to the concrete type should be replaced by the interface class wherever possible and the member functions to declare in the interface where the used member functions in the SUT are pre-selected.

Note that as explained in section 2.1.4, we do not offer the possibilities to choose the visibility of the destructor and to create a private assignment operator because of the complexity the user would have to cope with.

2.2. Compile Seam

Although object seams are the classic way of injecting dependencies, we think there is often a better solution to achieve the same goals called *compile seam*. In this section we will discuss the necessary steps and how Mockator can help to refactor towards this seam type.



Interface name: DieInterface

[✓] Use the extracted interface type where possible

[● Make destructor public and virtual] ○ Make destructor protected and non-virtual

☐ Create a private assignment operator declaration

	Member functions to declare in the interface
✓	int roll() const

Figure 2.2.: *UI mock-up for the extract Interface refactoring dialog.*

2.2.1. Introduction

Beside supporting subtype polymorphism with object seams, C++ also provides static polymorphism[11] with template parameters. They allow us to inject dependencies at compile-time. We therefore call this seam type compile seam. Its essential precondition is the application of the refactoring *extract template parameter* [Thr10]. The result of this refactoring can be seen in Listing 2.13 where the class `GameFourWins` from Listing 2.1 was used to extract a template parameter. The enabling point of this seam is the place where the template class `GameFourWinsT` is instantiated.

One might argue that the intrusion of testability into our production code is ignored here: we have to template a class in order to inject a dependency, where this might only be an issue during testing. The approach taken by the extract template parameter refactoring is to create a type definition which instantiates the template with the concrete type that has been used before applying the refactoring (here through the use of a default template parameter). This has the advantage that we do not break existing code.

The use of static polymorphism with template parameters has some advantages over object seams with subtype polymorphism. It does not incur the run-time overhead of

[11]Static polymorphism [SA04] is not a term one can usually find in type theory books. We used it here in contrast to dynamic polymorphism. Other comparisons employ compile-time vs. run-time polymorphism. In type theory, static polymorphism is referred to as parametric polymorphism.

calling virtual member functions that can be unacceptable for certain systems. This overhead results due to pointer indirection, the necessary initialisation of the vtable (the table of pointers to its member functions) and the fact that virtual functions usually cannot be inlined by compilers. Note that there is also an increased space-cost because of the additional pointer per object that has to be stored for the vtable [DH96]. There are also scenarios where it makes sense to do calculations during compile-time evaluations upfront in order to avoid them being done at run-time.

```cpp
#include <iostream>
class Die;
template <typename Dice=Die>
struct GameFourWinsT {
  void play(std::ostream &os = std::cout){
    if (die.roll() == 4) {
      os << "You won!" << std::endl;
    } else {
      os << "You lost!" << std::endl;
    }
  }
private:
  Dice die;
};
typedef GameFourWinsT<> GameFourWins;
```

Listing 2.13: *Code after applying the refactoring extract template parameter to inject dependencies through template parameters.*

Beside performance and space considerations, inheritance brings all the well-known software engineering problems like tight coupling[12], enhanced complexity and fragility with it [Szy98, SA04, Mey05]. Most of these disadvantages can be neglected with the use of templates. Probably the most important advantage of using templates is that a template argument only needs to define the members that are actually used by the instantiation of the template. This can ease the burden of an otherwise wide interface that one might need to implement in case of an object seam. The concept is known as *compile-time duck typing*.

Although duck typing is mainly used in the context of dynamically typed programming languages, C++ offers duck typing at compile-time with templates. Instead of explicitly specifying an interface our type has to inherit from, the template argument just has to provide the components that are used in the template definition. If we have a `template <typename T> void foo(T t)`, `t` can be of any type as long as it provides the operations executed on it in `foo`. This is known as the *implicit interface*, whereas with inheritance one has an *explicit interface* to implement.

Of course, there are also drawbacks of using templates in C++. They can lead to increased compile-times, the known export and inline issues, code bloat when used

[12]Inheritance is the second strongest relationship in C++, only after friendship [Sut00].

naively and (sometimes) reduced clarity. The latter is because of the missing support for concepts even with C++11, therefore solely relying on the implicit interface with the naming and documentation of template parameters. Furthermore, the use of new types will force recompilation and redeployment, and types cannot change at run-time which might be interesting for plug-in architectures. Apart from these objections, we are convinced that compile seams should be often preferred over object seams in C++[13].

2.2.2. Use Case

This use case describes the steps necessary to refactor towards compile seams:

UC 3	*Create compile seam*
Primary Actor	C++ Developer
Precondition	The developer has a fixed dependency in the code which needs to be replaced by an alternative implementation.
Postcondition	A compile seam exists that can be used with an alternative implementation in the newly created class.
Main sequence	1. *User:* Extracts a template parameter from the SUT. 2. *System:* Creates a template parameter and a `typedef` for clients that still want to use the class with the default collaborator. 3. *User:* Instantiates the SUT with an unknown identifier. 4. *System:* Shows marker with quick fix to create the compile seam class. 5. *User:* Applies the quick fix. 6. *System:* System creates the class (either a local one for C++11 or one in a namespace for C++03) and shows a marker indicating all missing member functions that are used in the SUT on the template parameter. 7. *User:* Applies one of the quick fixes (with or without recoding call support). 8. *System:* Creates the missing member functions. 9. *User:* Implements an alternative implementation for the compile seam in the member functions.

[13]Of course, it is not all black or white. There are interesting applications of mixing the two paradigms as stated in [Ale01]. Examples include command and visitor design patterns or discriminated union implementations.

2.2.3. Implementation

After applying the extract template parameter refactoring, the compile seam exists and we can inject an alternative implementation. Mockator detects the usage of an unknown identifier as a template argument and creates a marker with a corresponding quick fix for this situation, as can be seen in Listing 2.14. The quick fix creates an empty class with the name of the unresolved identifier. The target for this new class depends on the chosen C++ standard.

```cpp
template<typename T>
struct ShapePainter {
  void paint() {
    T shape(4);
    int area = shape.area();
    //...
  }
};
void testShapePainterWithSquare() {
  ShapePainter<SquareFake> painter;
  painter.paint();
}
```

Listing 2.14: *Dependency injection with template parameter and an unknown identifier used as template argument yielding a marker.*

In case C++11 is used, we create a local class right in front of the template instantiation. Otherwise, the local class is wrapped in a namespace which name depends if the current test function is registered in a CUTE suite or not. In case it is a registered test function, we create a namespace with the name of the test suite and an additional one for the test function. If not, we only create the latter. We use namespaces to encapsulate the test doubles from the surrounding code and to avoid naming collisions. Both situations are shown in Listing 2.15. Note that we create a `using namespace` declaration to have access to our test double in the unit test function.

The creation of the test doubles as local classes in the same function as the unit test code leads to an increased locality that makes it easier to keep them in sync with the unit test code. Although local classes can simplify implementations and especially enhance the locality of symbols [Ale01], they are a lesser known and used language feature for most C++ programmers. One of the reasons for this is that local classes had no linkage and therefore could not be used as template arguments [ISO03, §14.3.1.2]. With C++11, this has changed and the awkward restriction has been removed.

Nevertheless, local classes are still not first-class citizens even in C++11. Declarations in local classes can only use type names, static and external variables, functions and enums from their enclosing scope. Access to automatic variables is therefore prohibited [ISO11, §9.8.1]. Furthermore, they are not allowed to have static members [ISO11, §9.8.4]. Unfortunately, they can also not have template members.

```
namespace testSuite {
  namespace testWhenCpp03IsUsed_Ns {
    struct SquareFake {
    };
  }
}
void testWhenCpp03IsUsed() {
  using namespace testSuite::testWhenCpp03IsUsed_Ns;
  ShapePainter<SquareFake> painter;
  painter.paint();
}
void testWhenCpp11IsUsed() {
  struct SquareFake {
  };
  ShapePainter<SquareFake> painter;
  painter.paint();
}
void runSuite() {
  cute::suite testSuite;
  testSuite.push_back(CUTE(testWhenCpp03IsUsed));
}
```

Listing 2.15: *Creation of classes for compile seam for both C++ standards. Note the markers which are created due to the missing member functions.*

The next and final step for the programmer is to create the missing member functions by using a quick fix for the marker shown in listing 2.15. The result of this quick fix is presented in listing 2.16 for the case when C++11 is used.

```
void testWhenCpp11IsUsed() {
  struct SquareFake {
    SquareFake(int const&) {
    }
    int area() const {
      return int{};
    }
  };
  ShapePainter<SquareFake> painter;
  painter.paint();
}
```

Listing 2.16: *After applying the quick fix all missing member functions are provided.*

2.2.4. Recognising Missing Concept Implementations

So far we have only talked about missing member functions and constructors. But there are more language items that could be used in the SUT which we should provide in our injected test doubles. To formalise this discussion, we use the well-known idea of *concepts* and also explain them here.

Introduction to Concepts

C++ supports generic programming with templates. A template can be either a function or a class that is parametrized with one or more *template parameters*. The actual types that are used for a template are called *template arguments*. The compiler creates a so called *template instantiation* with the given arguments for a template. Beside this, the compiler is also able to infer template arguments for function templates in case they are not given.

Listing 2.17 shows the template definition of the function template `min`. `T` is the template parameter and `a` and `b` are so called *dependent types* [VJ02] because they are of the type of the template parameter `T`. The compiler is only able to verify the template definition for syntactical correctness based on non-dependent names. Dependent names can only be checked when the template is instantiated, which is the case in the given example.

```
template <typename T>
T const& min(T const& a, T const& b) {
  return a < b ? a : b;
}
int typeInferenceWithCallingTemplateFunction() {
  return min(3, 7);
}
```

Listing 2.17: *Template function `min` which returns the smaller of the two given arguments as its result including an instantiation making use of type inference.*

`operator<` is used by the template to compare the given values and is expected to be implemented for the template argument type given to `min`. Because `operator<` is applied on values of type `T`, it is a dependent name. The example in listing 2.17 also shows that the compiler is able to infer the type of `T` based on the arguments given.

Although its usefulness and its advantages over inheritance and virtual functions to achieve polymorphism, templates also have a few subtle drawbacks as described in section 2.2.1. One of them is the lack of an explicit contract between the template and its users. Stroustrup and Dos Reis therefore conclude in [SR05]: “The near-optimal performance offered by ISO C++ templates comes at the price of a very weak separation between template definitions and their uses.” This problem manifests itself in the fact that template definitions and their uses cannot be verified separately.

This is one of the reasons why we are confronted with long and hard to grasp compiler error messages when templates are used wrongly. The canonical example is the incorrect use of STL algorithms, as one example is shown in listing 2.18.

```
#include <list>
#include <algorithm>
int main() {
  std::list<int> numbers = {3, 1, 2, 5, 4};
  std::sort(numbers.begin(), numbers.end());
}
```

Listing 2.18: *Typical STL algorithm template function usage error.*

Compiling this example with the GNU Compiler Collection (GCC) 4.7 yields an error message chain of 46 text lines[14]. Of course, a developer familiar with the STL will find the source of the problem quickly in the last part of the compiler error (see listing 2.19)[15]. `sort` expects the passed iterators to be of the type `RandomAccessIterator` and therefore providing direct access to the underlying elements of the container with the help of constant-time member function for moving forward and backward (as it is shown in the error message, `operator-` is missing). This is clearly not the case with the iterators `std::list` offers, which are of type `BidirectionalIterator`[16].

```
c++/4.7.0/bits/stl_algo.h:5476: error: no match for
'operator-' in '__last - __first'
c++/4.7.0/bits/stl_iterator.h:329: note: template<class _Iterator> typename
std::reverse_iterator::difference_type std::operator-(const
std::reverse_iterator<_Iterator>&, const
std::reverse_iterator<_Iterator>&)
```

Listing 2.19: *Last part of error message yielded by GCC 4.7 when given the code of Listing 2.18.*

Although this constraint is implicitly remarked in the code of the function template `sort` with a naming schema used by the STL for template parameters (see listing 2.20), library designers cannot enforce that these rules are followed by their users in a simple and compiler verifiable way.

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

Listing 2.20: *Template function signature of `std::sort` with argument requirements based on a naming schema for STL iterators.*

Concepts are able to make these requirements explicit. They basically offer a type system for templates, thus make it possible for compilers to verify the types of the

[14]Note that GCC is not very good in producing concise error messages. With clang we often get much better results which somehow lessen the need for concepts.

[15]Tools like `gccfilter` [Li12] can help a lot here in reducing the visual clutter of the error message and by adding coloring. We tried this example with `gccfilter --colorize --remove-instantiated-from --remove-path --remove-template-args --remove-namespaces g++ -std=c++11 test.cpp` and could spot the error quickly.

[16]`std::list` has a special member function called `sort` which can be used instead of the STL algorithm.

template definitions separately from their uses [GS06]. They are not a part of C++11 anymore, but may be included in a future C++ standard.

We now refer back to listing 2.17 again. With concepts, we can make the constraints explicit that there must be a `operator<` present for the arguments of type `T const&` and that the return type of `operator<` has to be convertible to `bool`. The original proposal [GS06] offers a new syntax to write concepts as it is shown in listing 2.21.

```
auto concept LessThanComparable<typename T> {
  bool operator<(T const&, T const&);
};
```

Listing 2.21: *Concept definition `LessThanComparable` enforcing a corresponding `operator<`.*

Instead of expecting an arbitrary type as it is denoted with the syntax `typename T`, a library designer is now able to explicitly state that `T` must follow the constraints given by the concept `LessThanComparable`, as it is shown in Listing 2.22.

```
template<LessThanComparable T>
T const& min(T const& a, T const& b) {
  return a < b ? a : b;
}
```

Listing 2.22: *Concept `LessThanComparable` applied to template function `min`.*

Although concepts are not part of the new standard C++11 and are therefore not included in compilers implementing the standard like GCC, we can still test our `min` function with ConceptGCC [ea11b]. This is an enhanced GNU C++ compiler which supports concepts. Listing 2.23 shows the same example using the concept `LessThanComparable` which is defined in the `concepts` header. If we compile this example, ConceptGCC yields an error message like shown in listing 2.24 which clearly states that class `X` does not fulfil the requirement given by the concept `LessThanComparable`.

```
#include <concepts>
template<std::LessThanComparable T>
T const& min(T const& a, T const& b) {
  return a < b ? a : b;
}
struct X { };
void foo() {
  X x1, x2;
  X result = min(x1, x2);
}
```

Listing 2.23: *Code for using `min` with ConceptGCC.*

An important goal of using concepts is to make it easier for compilers to produce better and shorter error messages. This can be achieved through early error detection and the obsolete creation of an instantiation stack [GS06]. The difference is that the missing `operator<` was detected while instantiating `min`, while with concepts the error is recognised at the call of `min` when the concept `LessThanComparable` could not be satisfied.

```
min.cpp:20: error: no matching function call to "min(X&, X&)"
min.cpp:11: note: candidates are: const T& min(const T&,
const T&) [with T = X] <requirements>
min.cpp:20: note: no concept map for requirement
"LessThanComparable<X>"
```

Listing 2.24: *Error message produced by ConceptGCC for the `min` example.*

In the last example, this was not a big deal because even without concepts the error message clearly indicated that class `X` was missing `operator<`. But if we compile the code in listing 2.18 with ConceptGCC, we only get four lines (see listing 2.25) compared to the 46 before and it is clearly described that `std::list`'s iterator type does not satisfy the requirement of being a `RandomAccessIterator`.

```
sort.cpp: In function "int main()":
sort.cpp:6: error: no "sort(std::_List_iterator<long int>,
std::_List_iterator<long int>)"
stl_algo.h:2673:note:candidates are: void std::sort(_Iter, _Iter)
[with _Iter = std::_List_iterator<long int>] <requirements>
iterator_concepts.h:174: note: no concept map for requirement
"std::RandomAccessIterator<std::_List_iterator<long int> >"
```

Listing 2.25: *Error message of ConceptGCC when compiling the code of listing 2.18.*

Recognising Missing Concept Implementations in Mockator

The recognition of missing concept implementations and the creation of default implementations for them is the main part Mockator does in its support for compile seams. Consider listing 2.27 which shows a SUT using the template parameter `T` to create an object and calling member functions on it. The template class is instantiated with `Fake` which is a local class lacking the implementation of the used constructor and member functions.

```
auto concept FakeRequirements<typename T> {
  T::T(std::string);
  void T::foo1();
  double T::foo2(char);
  static void T::foo3();
  char const* T::foo4(int);
}
```

Listing 2.26: *Concept definition for class `Fake` of listing 2.27.*

To better understand the requirements `Fake` has to satisfy, we present these in concept notation in listing 2.26. As we can see, `Fake` needs to provide a constructor taking a `std::string` as its parameter and four member functions called `foo1`, `foo2`, `foo3` (static) and `foo4`.

```
template<typename T>
struct SUT {
  char const* bar() {
    T t("test");
    t.foo1();
    double d = 3.1415 + t.foo2('c');
    T::foo3();
    return t.foo4(42);
  }
};
void testSUT() {
  struct Fake {
  };
  SUT<Fake> sut;
  sut.bar();
}
```

Listing 2.27: *Local class used as template argument to instantiate the template class `SUT` lacking one constructor and four member functions*

Mockator needs to recognise the missing concept implementations and provide default implementations. Listing 2.28 exhibits how `Fake` needs to look like in order to compile successfully.

```
#include <string>
void testSUT() {
  struct Fake {
    Fake(std::string const&) { }
    void foo1() const { }
    double foo2(char const&) const {
      return double{};
    }
    static void foo3() { }
    char const* foo4(int const&) const {
      return nullptr;
    }
  };
  // rest as before
}
```

Listing 2.28: *Local class adhering to the requirements of the concept `FakeRequirements`.*

Note that we have to deduce the return type from the context where the member functions on the template parameter are called. Mockator uses the default value of the return type for the return value with C++11's new initialiser syntax using curly brackets. For pointer return types, we use the new `nullptr`. Also note that we currently make all member functions and their parameters `const`, thus enforcing good coding rules by using `const` whenever possible [Mey05].

```cpp
template <typename T>
int sut(T const& t) {
  return t.foo();
}
void testSUT() {
  struct Fake {
    int foo() const {
      return int{};
    }
  };
  sut(Fake());
}
```

Listing 2.29: *Example of a template function used to inject a local class.*

Beside template classes, Mockator also supports template functions to inject test doubles. An example is shown in listing 2.29. There are several more complex cases we have to consider while detecting missing concept implementations. Listing 2.30 presents some of them, which we will discuss next.

(1) refers to the case where we need to resolve a `typedef` to recognize that `Fake_type` actually is the template parameter `T` and therefore has to be considered in the analysis of missing concept implementations.

(2) is a `typedef` used as the type of an argument to a member function of the local class. To access the `typedef`, we could have used an explicit instantiation of the SUT with the class `Fake` as its template argument type, but we do not do this as we will explain when `this` is discussed. Therefore, we currently resolve the `typedef` to the underlying type and use it as the type of the parameter (i. e., `unsigned int`).

In (3) we have to recognise the type of a temporary object. We also have to discover function calls made on instances of the injected classes in complex expressions. (4) is a situation where the call is done inside of an `if` statement and negated with the boolean operator `!`, which leads to the possible return type `bool` for `foo3`.

(5) depicts the situation where the template member function is defined outside the class declaration. In (6) we have to duduce the return type of a called function as the type of the argument for the function call on the local class. Furthermore, its return type must use the fully qualified name of `A` because it is located in a namespace.

In (7) we have to create a default constructor because there would be no default generated one by the compiler. Note that we would not create a solely default constructor for fake objects in contrast to mock objects where the situation is different because we always have to do necessary registration work in the mock object constructors as we will discuss in chapter 3.

```
#include <string>
#include <map>
namespace NS { struct A { }; }

template <typename T>
struct SUT {
  typedef T Fake_type; // (1)
  Fake_type fake;
  typedef unsigned int Positive; // (2)
  void bar() {
    std::map<std::string, int> m;
    T fake2(m);
    fake.foo1(NS::A()); // (3)
    fake.foo2(this);
    Positive p = 42;
    if (!fake2.foo3(p)) { return; }; // (4)
    bar2();
  }
  NS::A bar2();
};
bool isPrime(int) { return false; }

template <typename T> // (5)
NS::A SUT<T>::bar2() {
  return fake.foo4(true, isPrime(42)); // (6)
}

void handlingOfMoreComplexCases() {
  struct Fake {
    Fake() { } // (7)
    Fake(std::map<std::string, int> const&) { } // (8)
    void foo1(NS::A const&) const { }
    void foo2(SUT<Fake> const*) const { } // (9)
    bool foo3(unsigned int const&) const {
      return bool{};
    }
    NS::A foo4(bool const&, bool const&) const {
      return NS::A();
    }
  };
  SUT<Fake> sut;
  sut.bar();
}
```

Listing 2.30: *More complex cases to consider when creating missing concept implementations.*

(8) presents the case where we use a `std::map` that uses two default template arguments (comparator and allocator) which we do ignore when creating the parameter types to have a shorter type. Note that we also always take the shortest (the one with the least amount of characters) `typedef` that exists when creating parameter types. Instead of using the real type `std::basic_string<char>`, we use the found `typedef std::string` as the key of the map.

(9) Because we cannot have template member functions in local classes in C++, we have to use the name of the local class to instantiate the class template for the template-id of the function parameter. This has not been implemented and is discussed in section 6.3.2.

2.2.5. Recognising Operators

So far Mockator was not able to detect missing operators in injected test doubles. This has changed with Mockator Pro which now supports them. In listing 2.31 we present a few usages of operators and how we create implementations for them. As can be seen in this example, we have to handle prefix and postfix operators differently by adding a parameter of type `int` in case of a postfix operator. For function calls, we have to deduce the return type from the context of the call. Comparison operators like the equality operator are created as class members, although the preferable way of implementing these is as free functions because every additional member function decreases encapsulation of the class and using a free function has the benefit that comparisons work in both directions. But with Mockator we just create member functions in the chosen test double and therefore ignore this issue.

A further important thing is that we declare the implemented operators only as `const` if they are not supposed not change the internal state of the object. As an example, compound assignment operators should not be declared `const` because of that reason. Note that we return a reference to `this` in the case of operators that modify their object.

2.3. Preprocessor Seam

C and C++ offer another possibility to alter the behaviour of code without touching it in that place using the preprocessor, which we will discuss in this section.

2.3.1. Introduction

Although we are able to change the behaviour of existing code as shown with object and compile seams before, we think preprocessor seams are especially useful for debugging purposes like tracing function calls. An example of this is shown in listing 2.32 where

```
template<typename T>
struct SUT {
  bool bar() {
    T fake1;
    T fake2 = -fake1;
    fake1 = fake2;
    ++fake1; fake1++;
    fake1(42); &fake1;
    fake1 == fake2; fake1 /= fake2;
    if (!fake1 && !fake2)
      return fake1[3];
    return !(fake1 < fake2);
  }
};
void createMissingOperators() {
  struct Fake {
    Fake operator -() const             { return Fake{};  }
    Fake& operator ++()                 { return *this;   }
    Fake operator ++(int)               { return Fake{};  }
    int operator ()(const int&)         { return int{};   }
    Fake* operator&()                   { return nullptr; }
    bool operator ==(const Fake&) const { return bool{};  }
    Fake& operator /=(const Fake&)      { return *this;   }
    bool operator !() const             { return bool{};  }
    bool operator [](const int&)        { return bool{};  }
    bool operator <(const Fake&) const  { return bool{};  }
  };
  SUT<Fake> sut; sut.bar();
}
```

Listing 2.31: *Example for the use of operator overloading when used on an injected test double.*

we exhibit how calls to C's `malloc` function can be traced for statistical purposes. This is additionally supported by making use of the file name and line number which are available through the macros `__FILE__` and `__NAME__` of the C preprocessor `cpp`.

The enabling point for this seam are the options of our compiler to choose between the real and our tracing implementation. The file `malloc.h` must be included into every translation unit where `malloc` is used in order to redefine the call to our own version. Note that we use `#undef` to still be able to call the original implementation of `malloc`.

We strongly suggest to not using the preprocessor excessively in C++. The preprocessor is just a limited text-replacement tool lacking type-safety that causes hard to track bugs. Nevertheless, it comes in handy for this task. Note that in case the statistical data (file name and line number) is not used, it is better to apply a link seam because preprocessor seams cause re-compilations of every translation unit that includes the header file with the macro if changes in this occur.

```
// malloc.h
#include <cstdlib>
#ifndef MALLOC_H_
#define MALLOC_H_
void *my_malloc(size_t size, const char* fileName, int lineNumber);
#define malloc(size) my_malloc((size), __FILE__, __LINE__)
#endif
// malloc.cpp
#include "malloc.h"
#undef malloc
void *my_malloc(size_t size, const char*, int) {
  // remember allocation size and origin in statistics
  return malloc(size);
}
```

Listing 2.32: *Using the preprocessor to intercept function calls to `malloc`.*

2.3.2. Use Case

This use case describes the steps necessary to refactor towards a preprocessor seam:

UC 4	*Create preprocessor seam*
Primary Actor	C++ Developer
Precondition	The developer wants to replace a fixed dependency by an alternative implementation or wants to trace function calls for statistical purposes.
Postcondition	A preprocessor seam exists that can be used with the alternative implementation.
Main sequence	1. *User:* Selects a function name and performs the trace function call action. 2. *System:* Creates a new source folder `trace` (if not already existing) and two new files (one header and one source file) with the redefinition of the function name and the source file where the alternative implementation can be placed. 3. *User:* Inserts the alternative implementation. 4. *System:* Shows marker with quick fix to disable this seam.

2.3.3. Implementation

To discuss the implementation of this seam, we use another example that uses the well-known algorithm to detect leap years shown in listing 2.33. Note the use of `time` to get the current time on the system. This call makes this code hard to test. Because we do not intend to make changes in-place and use some global state to differentiate between production and testing environment, we apply a seam here.

```
#include "leapyear.h"
#include <ctime>
unsigned int getYear() {
    time_t now = time(0);
    tm* z = localtime(&now);
    return z->tm_year + 1900;
}
bool isLeapYear() {
    unsigned int year = getYear();
    if ((year % 400) == 0) return true;
    if ((year % 100) == 0) return false;
    if ((year % 4) == 0) return true;
    return false;
}
```

Listing 2.33: *Implementation of an algorithm to detect leap years using* `time` *making it hard to test.*

To apply the refactoring, the user selects the call to `time` and Mockator creates the translation unit presented in Listing 2.34. To include the header file into every translation unit of the chosen project, we make use of the option `-include` of GCC and add this option to the configuration of the current project. To make it convenient to temporarily disable the preprocessor seam, Mockator creates an info marker to toggle the activation of the referred `-include` option.

```
//mockator_time.h
#ifndef MOCKATOR_TIME_H_
#define MOCKATOR_TIME_H_
#include <ctime>
time_t mockator_time(time_t* __timer, const char* fileName, int lineNumber);
#define time(__timer) mockator_time((__timer), __FILE__, __LINE__)
#endif
//mockator_time.cpp
#include "mockator_time.h"
#undef time
time_t mockator_time(time_t* __timer, const char*, int) {
  return time(__timer); // replace with a hardcoded value like
  // 986725194 = Sun Apr 08 2001 12:19:54 GMT+0200 (CEST)
}
```

Listing 2.34: *Preprocessor seam to provide an alternative implementation of* `time`*.*

Note that in this example it might have been better to shadow the function through a link seam because we do not need the statistical data, but instead just want to replace the time with some hard coded value. In case the function to be traced is part of a namespace, we pack the new function definition into the same and qualify the function declaration accordingly. If the translation unit where the original function is defined is part of the same project as the translation unit where the call occurs, we add an `#undef` in order to not replace the definition with our replacement. One disadvantage of this seam type is that we cannot redefine member functions. We therefore yield an error if a user attempts to do this.

2.4. Link Seam

Beside the separate preprocessing step that occurs before compilation, we also have a post-compilation step called linking in C and C++ that is used to combine the results the compiler has emitted. The linker provides another kind of seam named *link seam* [Fea04].

We show three possibilities of using the linker to shadow or wrap function calls. Although all of them are specific to the used tool chain and platform, they have one property in common: Their enabling point lies outside of the code, i. e., in our build scripts. A further commonality of link seams is that it is often preferable to create separate libraries for code we want as a replacement. This allows us to adapt our build scripts to either link to those for testing rather than to the production ones [Fea04].

Because we extensively work with the build system of Eclipse CDT in this section, we give a short introduction here. CDT provides two different kinds of build systems for C and C++ projects. The *standard build* system relies on the user supplied Makefiles to build a project. For the second called *managed build*, CDT generates the Makefiles based on the project type and tool chain (e. g., GCC Linux) for us. Because we are not in charge of the user's customised build, we will concentrate on the managed build here. Note that there are three different types of managed build projects: *executable*, *shared library* and *static library*.

2.4.1. Shadow Functions Through Linker Order

Introduction

In this link seam type we make use of the linking order. Although from the language standpoint the order in which the linker processes the files given is undefined, it has to be specified by the tool chain. For GCC this is described in [GS04] as follows: "The traditional behaviour of linkers is to search for external functions from left to right in the libraries specified on the command line. This means that a library containing the definition of a function should appear after any source files or object files which use it."

The linker incorporates any undefined symbols from libraries which have not been defined in the given object files. If we pass the object files first before the libraries with the functions we want to replace, the GNU linker prefers them over those provided by the libraries. Note that this would not work if we placed the library before the object files. In this case, the linker would take the symbol from the library and would then yield a duplicate definition error when considering the object file.

```
// GameFourWins.cpp and Die.cpp as in shown in listing 2.1
// shadow_roll.cpp
#include "die.h"
int Die::roll() const {
   return 4;
}
// test.cpp
void testGameFourWins() {
  // code to test the GameFourWins class
}
```

Listing 2.35: *Code for shadowing the member function `Die::roll`.*

To demonstrate this kind of link seam, consider the commands used in listing 2.36 for the code shown in listing 2.35. The order given to the linker is exactly as we need it to prefer the symbol in the object file because the library comes at the end of the list. This list is the enabling point of this kind of link seam. If we leave `shadow_roll.o` out, the original version of `roll` is called as defined in the static library `libGame.a`.

```
$ ar -r libGame.a Die.o GameFourWins.o
$ g++ -Ldir/to/GameLib -o Test test.o shadow_roll.o -lGame
```

Listing 2.36: *GNU tool chain commands for shadowing functions. Note that object files are passed before the libraries to make this work.*

As we have noticed, the GNU linker of Mac OS X needs the shadowed function to be defined as a weak symbol; otherwise the linker always takes the symbol from the library. Weak symbols are one of the many function attributes GCC offers. If the linker comes across a strong symbol (the default) with the same name as a weak one, the latter will be overridden. In general, the linker uses the following rules to select a symbol in case of naming conflicts [BO10]: 1. Not allowed are multiple strong symbols. 2. Choose the strong symbol if given a strong and multiple weak symbols. 3. Choose any of the weak symbols if given multiple weak symbols.

```
struct Die {
  __attribute__((weak)) int roll() const;
};
```

Listing 2.37: *Weak declaration to shadow functions in Mac OS X which uses a different linker for GCC.*

With the to be shadowed function defined as a weak symbol, the GNU linker for Mac OS X prefers the strong symbol with our replacement code. Listing 2.37 shows the function declaration with the attribute `weak`.

Shadowing functions has one big disadvantage: It is not possible to call the original function anymore. This would be valuable if we just want to wrap the call for logging or analysis purposes or do something additional with the result of the real function.

Use Case

This use case describes the steps necessary to shadow a function with Mockator:

UC 5	*Shadow function*
Primary Actor	C++ Developer
Precondition	The developer has a fixed dependency in the code which needs to be replaced by an alternative implementation.
Postcondition	A new translation unit with the shadowed function is created where the developer can place an alternative implementation.
Main sequence	1. *User:* Selects a function name in a translation unit of a library project and performs the shadow function action. 2. *System:* Creates a new source folder `shadows` (if not already existing) in the referencing executable project and creates a new translation unit with the shadowing function definition where the alternative implementation can be placed. 3. *User:* Inserts an alternative implementation.

Implementation

Consider listing 2.38 where the code is shown that Mockator creates for the example introduced in listing 2.33. Note that we first have to check if the translation unit of the selected function is a library project and has a referencing executable project. Otherwise, we yield an error to the user.

```
#include <ctime>

time_t time(time_t*) throw() {
  return time_t { };
}
```

Listing 2.38: *Shadowed time function generated in new translation unit by Mockator.*

As we have discussed in section 2.4.1, a fundamental precondition of shadowing functions is that the linker is called with the object files containing the shadowed function definitions *before* the libraries with the to be replaced function definitions. The generated `Makefile` of CDT's managed build supports this, as can be seen in Listing 2.39 for the case where the GCC tool chain is used and `g++` is called with the object placeholder `$(OBJS)` before `$(LIBS)`.

```
g++ -L"workspace/ShadowFunctionLib/Debug" -o ShadowFunction
$(OBJS) $(USER_OBJS) $(LIBS)
```

Listing 2.39: *Linker call with placeholders in the Makefile for a managed executable project* `ShadowFunction` *and a referencing shared library project* `ShadowFunctionLib`*.*

An important restriction of all three link seams in this chapter is that they do not work with inline functions. Obviously, inline functions do not have a separate symbol that is accessible for the linker because the compiler eliminates the call to the function[17]. We therefore yield an error to the user if an attempt to shadow or wrap an inline function is done.

In case Mac OS X is used, we add `__attribute__((weak))` before the function declaration if it is part of a translation unit in the current workspace to make shadowing work. Note that this could have potential side effects on production code we currently do not warn the user about.

Removing the shadowed function feature is just a matter of deleting the translation unit in the folder `shadow`. But note that it is not possible in Eclipse CDT to remove the source folder itself as this messes up the include paths (known bug in Eclipse CDT, see section J).

2.4.2. Wrapping Functions With GNU's Linker

Introduction

The GNU linker `ld` provides a lesser-known feature which helps us to call the original redefined function. This feature is available as a command line option called `wrap`. The man page of `ld` describes its functionality as follows: "Use a wrapper function for symbol. Any undefined reference to symbol will be resolved to `__wrap_symbol`. Any undefined reference to `__real_symbol` will be resolved to symbol."

`ld` allows us to call the real function by using `__real_symbol` which is useful to intercept function calls (e. g., for logging purposes) and then call the real function. For shadowing we do not need this, as we want to circumvent the "real" functionality. But sometimes it can be handy to call custom code before and after the execution of

[17]Note that we could make use of GCC's `-fno-inline` option which is turned on by default when no optimisations are used (`O0` compiler option) to disable the inlining of functions to avoid this restriction.

a certain function. This is a form of code instrumentation which can be used e. g. to perform performance analysis or to try out some new API functions.

```
FILE* __wrap_fopen(const char* path, const char* mode) {
  log("Opening %s\n", path);
  return __real_fopen(path, mode);
}
```

Listing 2.40: *Intercepting `fopen` with the `__real_symbol` functionality.*

Consider listing 2.40 which shows an example of an instrumented call to `fopen`. Note that we do not have to mangle `fopen` as this is the symbol name of a C function. As an example for a C++ function, we compile `GameFourWins.cpp` from listing 2.1. If we study the symbols of the object file, we see that the call to `Die::roll` — mangled as `_ZNK3Die4rollEv` according to Itanium's Application Binary Interface (ABI) that is used by GCC v4.x — is undefined (`nm` yields `U` for undefined symbols).

```
$ gcc -c GameFourWins.cpp -o GameFourWins.o
$ nm --undefined-only GameFourWins.o | grep roll
U _ZNK3Die4rollEv
```

Listing 2.41: *Undefined symbol in object file `GameFourWins.o` enabling GNU linker's wrapping function feature.*

This satisfies the condition of an undefined reference to a symbol. Thus we can apply a wrapper function here. Note that this would not be true if the definition of the function `Die::roll` would be in the same translation unit as its calling origin. If we now define a function according to the specified naming schema `__wrap_symbol` and use the linker flag `-wrap`, our function gets called instead of the original one.

Listing 2.42 presents the definition of the wrapped function. To prevent the compiler from mangling the mangled name again, we need to define it as a C code block.

```
extern "C" {
  extern int __real__ZNK3Die4rollEv();
  int __wrap__ZNK3Die4rollEv() {
    // our functionality here
    return __real__ZNK3Die4rollEv();
  }
}
```

Listing 2.42: *Definition of the wrapped function for `Die::roll` and the declaration for calling the original version in a C code block.*

Note that we also have to declare the function `__real_symbol` which we delegate to in order to satisfy the compiler. The linker will resolve this symbol to the original implementation of `Die::roll`. Listing 2.43 demonstrates the command line options necessary for this kind of link seam.

```
$ g++ -Xlinker -wrap=_ZNK3Die4rollEv -o Test test.o GameFourWins.o Die.o
```

Listing 2.43: *GNU linker call with `-wrap` option for the wrap function link seam type.*

Alas, this feature is only available with the GCC tool chain of Linux. The GNU linker of Mac OS X does not provide this option. Additionally, when the function to be wrapped is part of a shared library, we cannot use this seam type. Finally, because of the mangled names, this type of link seam is much harder to achieve by hand compared to shadowing functions which makes tool support necessary.

Use Case

This use case describes the steps necessary to wrap a function with Mockator:

UC 6	*Wrap function*
Primary Actor	C++ Developer
Precondition	The developer has a fixed dependency in the code which needs to be replaced by an alternative implementation.
Postcondition	The code to wrap the function and to call the original one including the necessary linker options is created.
Main sequence	1. *User:* Selects a function name and performs the wrap function action. 2. *System:* Checks that the project of the chosen function has a GCC Linux tool chain. Creates the code for wrapping functions and adds the necessary linker options to the project. Additionally, an information marker is created with a quick fix to toggle the activation of the wrapped function and to delete it. 3. *User:* Inserts an alternative implementation.

Implementation

Because all three link seams also support shadowing or intercepting operators, we present an example in listing 2.44 where we wrap `operator new`. Mockator's wrap function refactoring creates the code in listing 2.45. We first verify that the function definition is not in the same translation unit as its declaration because this seam type does not allow this. If this is the case, we abort the refactoring with an error. Note that we also create the delegate to the original function so that the user does not need to do this manually.

```
struct Bird {
  void* operator new(size_t);
};
void createBird() {
  Bird* bird = new Bird;
  // ...
}
```

Listing 2.44: *Wrapping* `operator new` *of class* `Bird`*. The current selection is shown in yellow.*

Due to the fact that wrapping functions only works with the GNU linker on Linux, we verify the tool chain of the chosen project. In case the tool chain is not supported, we create a dialog with a corresponding information message that can be suppressed for further attempts in the same project and where we store the decision of the user. Note that — instead of checking this precondition as part of the refactoring and aborting it — we used this approach to make it possible for other tool chain clients to see the generated code.

```
#ifdef WRAP__ZN4BirdnwEm
extern "C" {
  extern void* __real__ZN4BirdnwEm(size_t);
  void* __wrap__ZN4BirdnwEm(size_t sz) {
    return __real__ZN4BirdnwEm(sz);
  }
}
#endif
```

Listing 2.45: *Wrapping of the operator new mangled according to Itanium's ABI name mangling rules.*

Because wrapping functions does not work with shared libraries, we create a run-time function interception as explained in section 2.4.3 when the user applies this action in a shared library project.

Name Mangling for Itanium C++ ABI

For wrapping functions we have to mangle C++ functions. Because we did not want to call the compiler and analyse the result with a tool like `nm` which would lead to both performance problems and unnecessary tool dependencies, we decided to implement name mangling according to the Itanium ABI [ea12] which is used by GCC 3.x and 4.x. Despite its name, the Itanium C++ ABI is not limited to Itanium hardware platforms [Fog12].

Beside name mangling, an ABI also specifies the representation of member pointers, exception handling, function calling conventions, virtual tables, RTTI and a few other aspects. The Itanium ABI was created in order to reduce vendor coupling and — except Microsoft and a few others — a lot of compiler companies use it.

Name mangling is used to generate unique names [Kef12]. C++ is by far not the only language which makes use of it. Java uses name mangling to create unique names for anonymous and inner classes. Python mangles class attributes with names starting by two underscores. In C++, name mangling is used because of historical reasons and its compatibility with the C programming language. Linkers in general do not support C++ symbols. This is why language features that have an impact on naming symbols and that are unique to C++ and not available in C like function overloading need to be specially addressed.

To make it possible for the linker to differentiate between different functions with the same name, but different parameters, C++ compiler mangle their name and include the type information of their parameters. As discussed before, this process is highly compiler-dependent and there exist various schemes how this can be done.

For Mockator, we implemented most of the rules the Itanium ABI has. We ignored mangling of function templates because they are not supported anyway by our link seams (see section 6.2.1). Beside the basic mangling rules, we also have implemented the abbreviation rules that are part of the compression rules which are used to minimise the length of external names [ea12]. This was important to support names from the `std` namespace. We also implemented the substitution encodings to eliminate repetition of equal types in function parameters. To give an example how name mangling is implemented in the Itanium ABI, we mangle the function shown in listing 2.46.

```cpp
void foo(int&) { }
```

Listing 2.46: *Example of a simple function `foo` with a reference to `int` as parameter type.*

To demonstrate the process, we use the following reduced set of mangling rules of the Itanium ABI written in Extended Backus–Naur Form (EBNF):

⟨mangled-name⟩ → **Z_** ⟨encoding⟩
⟨encoding⟩ → ⟨name⟩ ⟨bare-function-type⟩
⟨bare-function-type⟩ → ⟨type⟩ {⟨type⟩}
⟨type⟩ → **R** ⟨type⟩ | ⟨builtin-type⟩
⟨builtin-type⟩ → **i**
⟨name⟩ → ⟨unscoped-name⟩
⟨unscoped-name⟩ → ⟨unqualified-name⟩
⟨unqualified-name⟩ → ⟨source-name⟩
⟨source-name⟩ → ⟨number⟩ ⟨identifier⟩
⟨number⟩ → [n] ⟨digit⟩ {⟨digit⟩}
⟨identifier⟩ → ⟨unqualified source code identifier⟩
⟨digit⟩ → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

The deduction of the grammar rules finally leads to the mangled name `_Z3fooRi`. Note that `bare-function-type` consists of at least one type because empty parameter lists are encoded with the parameter specifier `v` for a void parameter.

2.4.3. Run-time Function Interception

Introduction

If we have to intercept functions from shared libraries, we can use this kind of link seam instead of wrapping functions as explained in section 2.4.2. Intercepting library calls with our own provided wrapper code is useful in many situations, especially during testing. As an example, we can use our own implementations of `malloc` or `free` to log memory use without the need to recompile or relink with this link seam type.

This link seam is based on the fact that it is possible to alter the run-time linking behaviour of the dynamic linker loader `ld.so` in a way that it considers libraries that would otherwise not be loaded. This can be accomplished by the environment variable `LD_PRELOAD` the loader `ld.so` interprets[18]. Its functionality is described in the man page of `ld.so` as follows: "A white space-separated list of additional, user-specified, ELF shared libraries to be loaded before all others. This can be used to selectively override functions in other shared libraries." With this we can instruct the loader to prefer our function instead of the ones provided by libraries normally resolved through the environment variable `LD_LIBRARY_PATH` or the system library directories[19].

Now consider we want to intercept a function `foo` which is defined in a shared library. We have to put the code for our intercepting function into its own shared library (e. g., `libFoo.so`). If we call our program by appending this library to `LD_PRELOAD` as shown in listing 2.47, our definition of `foo` is called instead the original one.

```
$ LD_PRELOAD=path/to/libFoo.so executable
```

Listing 2.47: *Call of an executable with `LD_PRELOAD` set to our intercepting shared library.*

Note that environment variables have different names in Mac OS X. The counterpart of `LD_PRELOAD` is called `DYLD_INSERT_LIBRARIES`. This additionally needs the environment variable `DYLD_FORCE_FLAT_NAMESPACE` to be set [MB11].

The solution is not perfect yet because it would not allow us to call the original function. To accomplish this, we can use the following four library functions `ld.so` provides to manually load and access symbols from a shared library: `dlopen`, `dlclose`, `dlsym` and `dlerror`. With `dlsym` we can look up our original function by a given name. It takes a handle of a dynamic library we normally get by calling `dlopen` and yields a void

[18]Valgrind makes use of this feature to load itself with any dynamically linked library [Dev11].

[19]Note that we can have a similar effect in Java by tweaking the environment variable `CLASSPATH` to prefer certain directories for looking up Java classes.

pointer for the symbol as its result. Because we try to achieve a generic solution and do not want to specify a specific library here, we can use a pseudo-handle that is offered by the loader called `RTLD_NEXT`. With this, the loader will find the next occurrence of a symbol in the search order *after* the library the call resides [MB11].

```
#include <dlfcn.h>
int foo(int i) {
  typedef int (*funPtr)(int);
  static funPtr orig = nullptr;
  if(!orig) {
    void *tmp = dlsym(RTLD_NEXT, "_Z3fooi");
    orig = reinterpret_cast<funPtr>(tmp);
  }
  // our functionality here
  return orig(i);
}
```

Listing 2.48: *Calling the original function with POSIX' `dlsym`.*

As an example, consider listing 2.48 which shows the definition of the intercepting function `foo` and the code necessary to call the original function. Note that we cache the result of the symbol resolution to avoid the process being made with every function call. Because we call a C++ function, we have to use the mangled name `_Z3fooi` as the symbol.

As it is not possible in C++ to implicitly cast the void pointer returned by `dlsym` to a function pointer, we use an explicit cast. Furthermore, `dlfcn.h` has to be included and the compiler flag `-ldl` is necessary for linking to make this work as can be seen from the GCC command in listing 2.49.

```
$ g++ -std=c++0x -shared -ldl -fPIC foo.cpp -o libFoo.so
```

Listing 2.49: *GCC command necessary to create a shared library that makes use of `ld.so`'s symbol handling functions.*

The advantage of this solution compared to the first two link seams is that it does not require relinking. It is solely based on altering the behaviour of `ld.so`. A disadvantage is that this mechanism is unreliable with member functions, because member function pointers are not expected to have the same size as a void pointer. There is no reliable, portable and standards compliant way to handle this issue. Even the conversion of a void pointer to a function pointer was not specified in C++03. This has changed now with C++11 where it is implementation-defined [ISO11, §5.2.10.8][20]. Due to this problems, `dlsym` will change its interface in a future version to return function pointers instead of a void pointer, as it is promised in its manpage [IG12].

[20]GCC 4.7 yields the warning "ISO C++ forbids casting between pointer-to-function and pointer-to-object" when the compiler options `-std=c++0x -pedantic` are used.

Intercepting function calls with `ld.so` has a few important limitations. It is not possible to intercept `dlsym` itself. A further constraint is given due to security concerns: the man page states that `LD_PRELOAD` is ignored if the executable is a setuid or setgid binary. It is also not possible to intercept internal function calls in libraries. One example might be that if a function in the GNU C library calls `time` we cannot wrap it with our own version.

Use Case

This use case describes the steps necessary to intercept a function with Mockator:

UC 7	*Intercept function*
Primary Actor	C++ Developer
Precondition	The developer has a fixed dependency in a shared library which needs to be replaced by an alternative implementation.
Postcondition	The code to intercept the function and to call the original one is created in a new shared library project including the necessary run-time configuration changes for the executable project.
Main sequence	1. *User:* Selects a function name in a shared library project and performs the intercept function action. 2. *System:* Checks that the project of the chosen function has a GCC Linux or Mac OS X tool chain and that the selected function name is part of a shared library. Creates the code for intercepting functions in a new shared library project and adds the necessary changes to the run-time configuration of the associated executable project. 3. *User:* Inserts the alternative implementation.

Implementation

If the user attempts to intercept a function call, Mockator first checks if the function is part of a shared library and that the tool chain is either Linux or Mac OS X GCC. If the latter is not the case, we yield an error dialog with decision memory as discussed in section 2.4.2. If all these checks are successful, we create a new shared library project and the code as shown in listing 2.48 in a new translation unit. Because the shared library project makes use of the dynamic linker functions, we have to add the library `dl` to its CDT project configuration.

The final step is to alter the run-time configuration of the referencing executable project to add the `LD_PRELOAD` environment variable to the newly created shared library. This makes it possible that this code is used instead of the one from the shared library where the original code relies. Additionally, we also need to add the new shared library to the environment variable `LD_LIBRARY_PATH`. Because we also support Mac OS X for this seam type, we set the environment variables `DYLD_INSERT_LIBRARIES` and `DYLD_FORCE_FLAT_NAMESPACE` in case a Mac OS X GCC tool chain is used.

Although intercepting member functions is not reliable, we have decided to experimentally support them as well. The code in listing 2.50 shows how we achieve that. Note that we again need the mangled name of the function to be called when using `dlsym`. Furthermore, we cannot cast the void pointer yielded by `dlsym` to a member function pointer and therefore copy the address with the help of `memcpy`.

```cpp
//foo.h
struct Foo {
  int getValue(int);
};
//bar.cpp
#include "foo.h"
int bar() {
  Foo foo;
  return foo.getValue(3);
}
//getValue.cpp
#include "foo.h"
#include <dlfcn.h>
#include <cstring>
int Foo::getValue(int i) {
  typedef int (Foo::*funPtr)(int);
  static funPtr origFun = nullptr;
  if(!origFun){
    void *tmpPtr = dlsym(RTLD_NEXT, "_ZN3Foo8getValueEi");
    memcpy(&origFun, &tmpPtr, sizeof(&tmpPtr));
  }
  return (this->*origFun)(i);
}
```

Listing 2.50: *Code for intercepting the member function `Foo::getValue(int)`.*

Because we allow the user to easily deactivate or remove all discussed seams, we create a context menu entry for executable projects that have referencing shared library projects that make use of function intercepting. This context menu is filled with all these shared library projects and every entry has a checkbox to toggle its activation status.

3. C++ Mock Object Library

In this chapter we will first discuss the basics and inner workings of the original, C++11 based header-only mock object library of Mockator. We will then explain how we achieved to support C++03, regular expressions and thread-safety with Mockator Pro.

3.1. Basics And Inner Workings

This section discusses the basics of the engineered C++ mock object library which includes how we record function calls, what the requirements are for the test double classes to make this work, how expectations can be specified and how these can be made order-independent.

3.1.1. Recording Function Calls

An important part of a mock object implementation is the recognition of the function calls the SUT makes on the mock object while the unit test runs. Beside the sequence and number of calls, we are also often interested in their argument values. We therefore have to store these facts to be able to later compare the calls with the users expectations.

In Mockator, we use an abstraction named `call` for this purpose which represents a call of a member function. Its basic functionality is shown in listing 3.1. A function call consists of the signature of the function and its argument values. Because we have to allow arguments of any type, we use a template parameter for the arguments in the constructor of `call`. Due to the fact that we do not want to restrict the number of arguments, we use a variadic template parameter pack.

The constructor of `call` uses the variadic template member function `record` to recursively process the arguments of the function call. `record(Head const&, Tail const&)` is used as the recursion step whereas `record()` handles the basic case of the recursion. Note the use of template parameter unpacking in the `sizeof` call to separate the argument values with commas and for the recursive call in `record`.

As can be seen, `call` uses a `std::string` object to store the function signature and the argument values. This is used to remember the values of any possible argument type and to give the user as much information as possible when a comparison fails.

```
#include <string>
#include <sstream>
struct call {
  template<typename ... Param>
  call(std::string const& funSig, Param const& ... params) {
    record(funSig, params ...);
  }
  template<typename Head, typename ...Tail>
  void record(Head const& head, Tail const& ...tail) {
    std::ostringstream oss;
    oss << toString(head);
    if (sizeof...(tail)) {
      oss << ",";
    }
    trace.append(oss.str());
    record(tail ...);
  }
  void record() {
  }
  std::string trace;
};
```

Listing 3.1: `call` *class in Mockator used to record function calls on mock objects.*

On the client side, the code for using mock objects with Mockator is presented in listing 3.2. We think it is worthwhile to have the code for the mock object in the unit test without hiding it behind macros as other mock object libraries do. This yields more transparency and exploits the full power of the host language when the library does not provide a desired feature.

We use the macro `INIT_MOCKATOR` to import a couple of names from the namespace `mockator` and from Boost in case C++03 is used. Note that we have to make the vector `allCalls` static because of the shortcomings local classes still have (see section 2.2.3). We create the vector initially with a size of one. Index `0` is reserved for calls of static member functions on the mock object. Note that every mock object has a mock ID which is used to access the calls made by the SUT on every instance of the mock object class.

Another important thing to explain is the use of the function `reserveNextCallId`. This is used to initialise the ID of the mock object and to add another call vector to the `allCalls` vector which collects all calls made on all instances of the mock object class. In the registrations of the function calls, we use the ID of the mock object to store the call for the corresponding mock object instance. Finally, we assert the calls made with the index `1` (we only have one instance of `MockExchange`) with our expectations.

In the classic mock object approach the unit test does not exercise any assertions [Mes07].

```cpp
#include "mockator.h"
#include "nasdaq.h"
template<typename STOCKEXCHANGE=Nasdaq>
struct Trader {
  void sell(std::string const& symbol, unsigned int amount) {
    Share share = lookupBy(symbol);
    STOCKEXCHANGE s;
    s.sell(share, amount);
  }
};
void testSellShare() {
  INIT_MOCKATOR();
  static std::vector<calls> allCalls{1};
  struct MockExchange {
    MockExchange() : mock_id{reserveNextCallId(allCalls)} {
      allCalls[mock_id].push_back(call{"MockExchange()"});
    }
    void sell(Share const& share, unsigned int amount) const {
      allCalls[mock_id].push_back(call{"sell(Share const&, unsigned int)
          const", share, amount});
    }
    const size_t mock_id;
  };
  Trader<MockExchange> trader;
  trader.sell("FB", 1000000);
  calls expected = {{"MockExchange()"}, {"sell(Share const&, unsigned int)
      const", "FB", 1000000}};
  ASSERT_MATCHES(expected, allCalls[1]);
}
```

Listing 3.2: *Example of a test case where the mock object library is used.*

This is entirely handled by the mock object which — when called during SUT execution — compares the actual arguments received with the expected arguments using equality assertions and fails the test if they do not match. We have decided against this common approach and exercise the assertions in the unit test itself because we want to be independent of the underlying unit testing framework. We therefore do not assert for equality in the mock object member functions, but instead compare the string traces in the unit test.

3.1.2. Requirements on Function Parameter Types

To store the argument values in a string, we expect that types used for the function arguments implement a corresponding `operator<<(ostream&, Type)`. To prevent compiler errors if this is not the case, we use some template meta programming tricks taken from the Boost library [ea11a][1]. This is done in the function `mockator::toString` which uses the stream output operator in case it is defined and otherwise writes a message into the stream to inform the user about the missing operator, as both can be seen in listing 3.3.

```
void missingStreamOperator() {
  struct Person { unsigned int age; };
  Person p = { 42 };
  std::ostringstream oss;
  oss << call{"foo(int)", 42}; // yields "(foo(int), 42)"
  oss << call{"foo(Person p)", p}; // yields "(foo(Person p), operator<<
  // not defined for type missingStreamOperator()::Person)"
}
//This is how we would implement an ostream operator:
std::ostream& operator<<(std::ostream& os, Person const& p) {
  return os << p.age;
}
```

Listing 3.3: *This code shows the result of calling the stream operator with the `call` instances. Note the warning text that is yielded in case a type does not implement a corresponding stream operator.*

3.1.3. Specifying Expectations With Initialiser Lists

When unit testing our objects, we want to compare a list of function calls against our expectations. C++ always allowed to initialise plain old data (POD) types and arrays with initializer lists, i. e., to give a list of arguments in curly brackets. But it was not possible in the old standard to use initialiser lists with regular (non-POD) classes. This has changed with C++11 where we are now able to instantiate regular classes with initialiser lists. This is especially useful for initialising STL container types.

```
calls expected = {
  {"foo(int i)", 42},
  {"bar(char c)", 'x'},
  {"foo(std::string s, double d)", "mockator", 3.1415}
};
```

Listing 3.4: *Specifying expectations with initializer lists.*

In Mockator we use initialiser lists for specifying expectations the SUT has to fulfil. Listing 3.4 shows this with a simple example. Note that `calls` is defined as a `typedef` for `std::vector<call>`. In order to make comparisons work, we have to provide an equality operator for `call`. This is shown in listing 3.5. `operator==` just delegates the work to the equality operator of `std::string` to compare the traced function call.

[1]The interested reader might want to have a look at the file `is_output_streamable.hpp` in a recent Boost library version to see how this works.

```
bool operator==(call const& lhs, call const& rhs) {
  return lhs.getTrace() == rhs.getTrace();
}
```

Listing 3.5: *`operator==` based on function call string traces.*

CUTE prints a string representation of the object under consideration if a comparison fails. To support this, we provide a stream operator for `call` as shown in listing 3.6.

```
std::ostream& operator<<(std::ostream& os, call const& c) {
  os << c.getTrace();
  return os;
}
```

Listing 3.6: *`operator<<` for `call` as a debugging help when comparisons fail.*

3.1.4. Support For Order-independent Comparisons

Mockator is order-sensitive and therefore uses strict mock objects [Mes07] by default. But sometimes we do not care about the order the function calls happened on our mock objects[2]. To allow to compare function calls order-independent, we offer a helper function which is presented in listing 3.7.

```
bool equalsAnyOrder(calls const& expected, calls const& actual) {
  return std::multiset<call>(expected.begin(), expected.end())
      == std::multiset<call>(actual.begin(), actual.end());
}
bool operator==(call const& lhs, call const& rhs) {
  return lhs.getTrace() == rhs.getTrace();
}
bool operator<(call const& lhs, call const& rhs) {
  return lhs.getTrace() < rhs.getTrace();
}
```

Listing 3.7: *Helper function to compare calls order-independent.*

A STL `multiset` helps us here because it has two important properties for our purposes [Jos99]: It is ordered and it allows — in contrast to an ordinary set — duplicates. The first property we need in order to get the same result using two call vectors with different ordering. The second is necessary because the same calls can happen multiple times which we need to record. Note that in order to use `call` instances with a `std::multiset`, we have to provide an `operator<` which allows to compare and hence order elements based on their string trace.

[2]Meszaros calls these *lenient mock objects* [Mes07].

3.2. Support for C++03

As we have mentioned in the preceding section, the existing mock object library uses various features of the new C++11 standard. Because there is a tremendous amount of existing code written in the old standards C++98 and C++03 and a lot of developers are not able to upgrade to the new standard yet, we also want to support them. In this section we will therefore analyse how we provide the same functionality with the old language features.

3.2.1. s/Variadic Templates/Boost Preprocessor Library/

We use variadic templates to record the function calls made by the SUT on the mock object. Variadic templates basically address two limitations we have with the old C++ standards [GJP06] to implement the recording: The impossibility to instantiate class and function templates with arbitrary long parameter lists and to pass an arbitrary amount of arguments to a function in a type-safe manner.

```cpp
struct call {
  template<typename T0>
  call(T0 const& t0) {
  }
  template<typename T0, typename T1>
  call(T0 const& t0, T1 const& t1) {
  } //etc.
};
```

Listing 3.8: *Attempt to mimic the behaviour of variadic template functions by manual overloads.*

To simulate the behaviour of variadic templates, we might offer overloaded template functions as shown in listing 3.8. Of course, an implementation like this would have various drawbacks compared to variadic templates: It leads to code repetition, long type names in error messages and long mangled names [GJP06]. Apart from this, it is obvious that it has a fixed upper limit on the number of template arguments.

Unfortunately, we are not able to solve all these issues with the old C++ standards, but we can at least get rid of the tedious code repetition with the help of the Boost Preprocessor library [Unk11b]. Consider listing 3.9 which shows the use of the three Boost Preprocessor macros `BOOST_PP_REPEAT`, `BOOST_PP_ENUM_TRAILING_PARAMS` and `BOOST_PP_ENUM_TRAILING_BINARY_PARAMS` to generate a fixed amount of template constructors.

We will now have a closer look at the three used macros with examples and also show the expanded result generated with the run of the GNU preprocessor (see listing 3.10).

```
#include <boost/preprocessor/repetition.hpp>
#include <string>
#include <sstream>
#ifndef MAX_NUM_OF_PARAMETERS
#  define MAX_NUM_OF_PARAMETERS 10
#endif
#define PRINT_CALL(z, n, name) << "," << toString(name ## n)
struct call {
#define MAKE_CALL(z, n, unused) \
  template<typename T BOOST_PP_ENUM_TRAILING_PARAMS(n, typename T)> \
  call(T t BOOST_PP_ENUM_TRAILING_BINARY_PARAMS(n, T, t) ) { \
    trace.append("("); \
    std::ostringstream oss; \
    oss << toString(t) BOOST_PP_REPEAT(n, PRINT_CALL, t); \
    trace.append(oss.str()); \
    trace.append(")\n"); \
  }
  BOOST_PP_REPEAT(MAX_NUM_OF_PARAMETERS, MAKE_CALL, ~)
  std::string trace;
};
```

Listing 3.9: *Use of the Boost Preprocessor library to avoid the code repetition.*

The first macro `BOOST_PP_ENUM_TRAILING_PARAMS(count, param)` is used to generate a comma-separated list of parameters with a leading comma. `count` is used to define the number of parameters to generate and `param` describes the text of the parameter. Listing 3.11 shows an example of this macro.

```
g++ -P -E -I/usr/include/boost example.cpp
```

Listing 3.10: *Run of the GNU preprocessor* `cpp`.

The `BOOST_PP_ENUM_TRAILING_BINARY_PARAMS(count, p1, p2)` macro is used to generate a comma-separated list of binary parameters. This list starts with a comma too. `count` describes the number of parameters to generate, `p1` is the first and `p2` the second part of the parameter. Listing 3.12 shows that this is especially useful for setting default template parameters. In order to intercept numeric concatenation for the default parameter, we use the macro `BOOST_PP_INTERCEPT` which expands to nothing.

```
#include <boost/preprocessor/repetition/enum_trailing_params.hpp>

template <typename First BOOST_PP_ENUM_TRAILING_PARAMS(3, typename T)>
// expands to:
template <typename First, typename T0, typename T1, typename T2>
```

Listing 3.11: *Example usage of the Boost Preprocessor macro* `BOOST_PP_ENUM_TRAILING_PARAMS`.

`BOOST_PP_REPEAT(count, macro, data)` has one single purpose: It repeatedly invokes a macro named by its second argument (we use `PRINT_CALL`). It could therefore be described as a higher-order macro in terms of functional programming. The first argument `count` denotes the number of these repetitions and `data` is used to pass data to the invoked macro. In our case we do not need this and use ~ because it is part of the basic character set of C++ implementations and not subject to macro expansion. Listing 3.13 presents a simple scenario for `BOOST_PP_REPEAT` where `##` is used for concatenating strings.

```
#include <boost/preprocessor/facilities/intercept.hpp>
#include <boost/preprocessor/repetition/enum_trailing_binary_params.hpp>
struct NullType {};

template<typename X BOOST_PP_ENUM_TRAILING_BINARY_PARAMS(4, typename
T, = NullType BOOST_PP_INTERCEPT)>
struct call {};
// expands to:
template<typename X , typename T0 = NullType , typename T1 =
NullType , typename T2 = NullType , typename T3 = NullType >
struct call {}
```

Listing 3.12: *Example usage of the macro* `BOOST_PP_ENUM_TRAILING_BINARY_PARAMS`*.*

It must be noted that this solution makes Mockator dependent on Boost which was not the case before. Because Mockator will be bundled with CUTE which has a dependency to Boost anyway and because we use Boost Assign as a replacement for initializer lists, this is not a serious drawback. Furthermore, the dependency will only affect users of the old C++ standard[3].

```
#include <boost/preprocessor/repetition/repeat.hpp>
#define DECL(z, n, text) text ## n = n;

BOOST_PP_REPEAT(5, DECL, int x)
// expands to:
int x0 = 0; int x1 = 1; int x2 = 2; int x3 = 3; int x4 = 4;
```

Listing 3.13: *Example of the Boost Preprocessor macro* `BOOST_PP_REPEAT` *[Unk11b].*

3.2.2. s/Initialiser Lists/Boost Assign/

With C++03, we are not able to use initialiser lists to specify expectations. Instead, we use Boost to improve the readability of our code. It offers a library called `Assign` [Ott11]

[3]But note that we will introduce a mandatory dependency to Boost Regex for both C++ standards as explained in section 3.3.

which facilitates the addition of elements to STL containers. Consider listing 3.14 which shows how easily we can set our expectations with its help.

```
using namespace boost::assign;
calls expected;
expected += call("SquareMock(int)", 4),
            call("area() const");
```

Listing 3.14: *Specifying expecations with the help of Boost Assign.*

Sometimes we want to be able to specify that a SUT is expected to call a member function a specific number of times. Instead of adding it manually the necessary number of times, we can use Boost Assign's `repeat` function, as presented in listing 3.15.

```
calls expected;
expected += call("SquareMock(int)", 4),
            repeat(5, call("area() const"));
```

Listing 3.15: *Using Boost Assign's* `repeat` *function.*

Consider the benefit this has compared to the classic way of using the STL `vector` member functions in listing 3.16.

```
calls expected;
expected.push_back(call("SquareMock(int)", 4));
expected.insert(expected.end(), 5, call("area() const"));
```

Listing 3.16: *Specifying expectations with the classic member functions the STL vector class provides.*

Note that we can achieve a similar effect with C++11 and handle this like presented in listing 3.17.

```
calls expected = {5, {"area() const"}};
```

Listing 3.17: *Specifying repeating expecations with C++11.*

3.2.3. Setup of Mockator Infrastructure

The Mockator library consists of one header file called `mockator.h`. We do not want to have separate header files for C++11 and the old standard because this would blow up our setup and would make things more complicated than necessary. We prefer the approach of verifying which standard is used to differentiate which code to use by checking if a special GCC compiler define is set called `__GXX_EXPERIMENTAL_CXX0X__` or the standard `__cplusplus` is set to a value greater than `201103L`[4] as shown in

[4]The GNU C++ compiler defined `__cplusplus` to be 1 which was a long-time bug first reported in 2001 [Mau11]. This has been fixed with GCC 4.7.

listing 3.18. The latter is only the case if GCC is called with the program argument `std=c++0x` or `std=c++11`.

```
#if defined(__GXX_EXPERIMENTAL_CXX0X__) || __cplusplus >= 201103L
// use call class with variadic templates
#else
#include <boost/preprocessor/repetition/repeat.hpp>
#include <boost/preprocessor/repetition/enum_trailing_binary_params.hpp>
#include <boost/preprocessor/repetition/enum_trailing_params.hpp>
// use the call class with boost preprocessor library
#endif
```

Listing 3.18: *Precompiler conditions to verify if the compiler support for C++11 is enabled or not.*

3.3. Specifying Expectations With Regular Expressions

With the existing Mockator library it was only possible to do exact comparisons based on string equality of the expectations and the actual arguments of the calls. Sometimes however, we do not want to be that strict, e. g. because our tests would get fragile otherwise. With Mockator Pro, the user can specify regular expressions for the expressions instead. An example can be seen in listing 3.19 which uses the code of listing 3.2.

```
calls expected = {{"MockExchange()"}, {"^^sell(Share const&, unsigned
int) const,[A-Z]\\{2\\},[0-9]\\{1,10\\}$"}};
ASSERT_MATCHES(expected, allCalls[1]);
```

Listing 3.19: *Example of using regular expressions to specify expectations for the code shown in listing 3.2.*

We use POSIX Basic Regular Expressions (BRE) for this because in that case the function signature can stay the same and does not need to be escaped — with one exception: asterisks for pointers still need to be escaped, but this can be neglected because they are used far less[5] than parentheses in the function signatures. This is because parentheses are no special characters in BRE's compared to the Extended Regular Expressions (ERE) where they are used to refer to matched sub-expressions.

We interpret all expectations with a leading ^ as a regular expression. Note that we could also have used the character classes `[:alpha:]` and `[:digit:]` in this example. When using regular expressions, we have to use the macro `ASSERT_MATCHES` that is provided by Mockator instead of CUTE's `ASSERT_EQUAL` for comparisons based on equality.

Mockator Pro yields a `mockator::RegexMatchingFailure` in case a regular expression would not match the given function signature. For debugging purposes, we also pass

[5]At least, we advocate for this very much.

the expected and actual value with the exception, as can be seen in the example given in listing 3.20 where we have used a one-letter symbol instead of the required two symbol characters.

```
'^sell(Share const&, unsigned int) const,[A-Z]\{2\},[0-9]\{1,10\}$' did
not match 'sell(Share const&, unsigned int) const,F,1000000'
```

Listing 3.20: *Message yielded when a regular expression does not match a function signature.*

Note that we use Boost's regular expressions even in the case when C++11 is activated. This is because we experienced problems with GCC's regex support even with version 4.7. We therefore add the Boost regex library to the CDT project in our Eclipse plug-in.

3.4. Multi-threading Support

Mockator's mock object library is not thread-safe. The registration of the function calls are read from and written to an unprotected `std::vector`. If the SUT uses multiple threads which access the injected test double, inconsistent registrations due to race conditions might appear. To prevent this, we use the classic way of dealing with problematic race conditions [Wil12] and protect the initialisation of the mock ID and the registration of the calls with a mutex. The most important code parts for supporting thread-safe registrations is shown in listing 3.21.

```
namespace mockator {
#if defined(USE_STD11)
  using std::lock_guard; using std::mutex;
#else
  using boost::lock_guard; using boost::mutex;
#endif
  struct NullMutex {
    void lock() { }
    bool try_lock() { return true; }
    void unlock() { }
  };
  template<typename T>
  size_t reserveNextCallId(std::vector<calls>& allCalls, T& _calls_mutex) {
    lock_guard<T> lock(_calls_mutex);
    size_t counter = allCalls.size();
    allCalls.push_back(calls());
    return counter;
}}
#define NULL_MUTEX NullMutex _calls_mutex
#define REAL_MUTEX mutex _calls_mutex
#define INIT_MOCKATOR_MT() USE_MOCKATOR_NS USE_BOOST_NS REAL_MUTEX
```

Listing 3.21: *Thread-safe call registrations by using protected access to the underlying call vector.*

For C++11, we use the new std::mutex and when using C++03 we rely on boost::mutex. We introduce a new macro called INIT_MOCKATOR_MT the user should apply instead of INIT_MOCKATOR when multiple threads are used. The reservation of the next available mock id in reserveNextCallId is protected because when another thread is simultaneously modifying the calls vector race conditions might occur. Note that we also provide a NullMutex that makes use of the Null Object pattern [Mar02]. This mutex is used when single threading is requested through the use of the macro INIT_MOCKATOR. We do this because we do not want to duplicate code for single- and multi-threading-support.

```
static const unsigned int NUM_OF_THREADS = 100;
template<typename T>
struct SUT {
  void bar() {
    std::vector<std::thread> threads;
    for (int i = 1; i < NUM_OF_THREADS; ++i) {
      threads.push_back(std::thread([]() {
        T mock;
        mock.foo();
      }));
    }
    for (auto &t : threads) { t.join(); }
  }
};
void test_multi_threaded() {
  INIT_MOCKATOR_MT();
  static std::vector<calls> callsMock{1};
  struct Mock {
    Mock() : mock_id{reserveNextCallId(callsMock, _calls_mutex)} {
      lock_guard<mutex> lock{_calls_mutex};
      callsMock[mock_id].push_back(call{"Mock()"});
    }
    void foo() const {
      lock_guard<mutex> lock{_calls_mutex};
      callsMock[mock_id].push_back(call{"foo() const"});
    }
    const int mock_id;
  };
  SUT<Mock> sut;
  sut.bar();
  calls expected = { {"Mock()"}, {"foo() const"} };
  for (unsigned int i = 1; i < NUM_OF_THREADS; ++i) {
    ASSERT_EQUAL(expected, callsMock[i]);
  }
}
```

Listing 3.22: *Example of using a SUT that makes use of threads and a unit test that asserts the calls being made.*

To test the new library changes, we use the unit test in listing 3.22. We start the threads in the SUT and make calls on the injected test double. With the help of the member function `thread::join` we wait for all threads to end their processing. In the mock object, we protect the registration with the help of a mutex.

As one can see, there would be a couple of code changes necessary for the client to achieve thread-safety. We think that a better approach would be to use a concurrent data structure like Intel's `tbb::concurrent_vector`[Cor12] for the calls vector instead of enforcing this work on the client. Although we could generate this code in our plug-in, there is still a significant amount of visual clutter that comes with this. For Mockator Pro, we therefore do not provide this and hence thread-safety is not given yet.

4. Mock Object Support

The mechanisms discussed in chapter 2 are used to make legacy code testable. We are now able to inject dependencies from outside. When we intend to inject mock objects, Mockator Pro supports us in creating and maintaining them which is the topic of this chapter.

4.1. Creating Mock Objects

In chapter 2 we discussed how to create object and compile seams and how to inject a test double. We then used Mockator to create the missing member functions, but we did not use the recording function calls support. In this section we will show how our Eclipse plug-in assists the user in this process.

```
template<typename T>
struct ShapePainter {
  void paint() {
    T shape(4);
    int area = shape.area();
    //...
  }
};
void testWhenCpp11IsUsed() {
  struct ShapeMock {
  };
  ShapePainter<ShapeMock> painter;
  painter.paint();
}
```

Listing 4.1: *Starting point for creating member functions with recording calls support. Note the marker that is created due to the missing member functions.*

Consider listing 4.1 which uses the `ShapePainter` example from chapter 2 together with a compile seam. The marker is created by Mockator because the injected test double is missing a constructor and a member function. Mockator offers three types of quick fixes for this situation. The first one called "Add missing member functions" is used to create the missing member functions with an empty implementation — except the case where a return statement is necessary where we create one with the default value of the return type. While this quick fix is used for applying fake objects, the next two come into play when we need mock objects.

The first of these two is called "Record calls by choosing function arguments" and is the default when order dependent assertions are used. The result of this quick fix is shown in listing 4.2 when C++11 is used. Mockator creates default values for all arguments of the missing functions and starts a linked mode for the user where these defaults can be adapted. The second kind of quick fix for using mock objects just creates the function signatures with no default arguments. When this quick fix is applied, the linked mode allows to alter the ordering of the functions as we expect them to occur.

```
#include "painter.h"
#include "cute.h"
#include "mockator.h"
void testWhenCpp11IsUsed() {
  INIT_MOCKATOR();
  static std::vector<calls> allCalls{1};
  struct ShapeMock {
    const size_t mock_id;
    ShapeMock(const int& i) : mock_id{reserveNextCallId(allCalls)} {
      allCalls[mock_id].push_back(call{"ShapeMock(const int&)", i});
    }
    int area() const {
      allCalls[mock_id].push_back(call{"area() const"});
      return int {};
    }
  };
  ShapePainter<ShapeMock> painter;
  painter.paint();
  calls expectedShapeMock = {{"ShapeMock(const int&)", int {}},
                             {"area() const" }};
  ASSERT_EQUAL(expectedShapeMock, allCalls[1]);
}
```

Listing 4.2: *Result of applying quick fix "Record calls by choosing function arguments" for a compile seam when C++11 is used.*

Note that we create an include for the mock object header file. For both quick fixes the linked mode also offers to choose between the three assertion kinds `ASSERT_EQUAL`, `ASSERT_ANY_ORDER` and `ASSERT_MATCHES`. We consistently use C++11's initialiser list syntax where possible. Because we cannot access automatic variables from a local class, we have to make the vector `allCalls` static. Furthermore, in case there are only static member functions in the mock object, we create the assertion by using index `0` of the vector `allCalls` because this is where static calls reside with Mockator's mock object library.

In listing 4.3 we show a similar example, but use an object seam instead of a compile seam and have requested support for C++03 in the project settings of Mockator which can be seen by the fact that we do not use C++11's initialiser list syntax anymore and apply Boost Assign's vector initialisation. Note that we do not register calls of the constructors in the test double anymore because — in contrast to compile seams — the instances of the mock objects are always created outside of the SUT when object seams are used.

```
#include "cute.h"
#include "mockator.h"
struct Shape {
  virtual ~Shape() {}
  virtual int area() const =0;
};
struct ShapePainter {
  void paint(Shape const& shape) {
    shape.area();
  }
};
void testWhenCpp11IsUsed() {
  INIT_MOCKATOR();
  static std::vector<calls> allCalls(1);
  struct ShapeMock : Shape {
    const size_t mock_id;
    ShapeMock() : mock_id(reserveNextCallId(allCalls)) { }
    int area() const {
      allCalls[mock_id].push_back(call("area()"));
      return int();
    }
  } shapeMock;
  ShapePainter painter;
  painter.paint(shapeMock);
  calls expectedShapeMock;
  expectedShapeMock += call("area()");
  ASSERT_EQUAL(expectedShapeMock, allCalls[1]);
}
```

Listing 4.3: *Result of applying quick fix "Record calls by choosing function arguments" for an object seam when C++03 is used.*

4.2. Move Test Double to Namespace

The test doubles created by Mockator in the unit tests are very flexible because the user can alter the code as necessary and does not need to use macros to specify their behaviour, but instead can apply the full power of C++. However, this comes at the price of more code that is placed in the unit test functions compared to other mocking libraries where this is hidden behind macros. Because of that, we provide a source action to move a test double out of the function to a namespace. Note that this is done automatically whenever compile seams are used together with C++03 because of the restrictions mentioned in section 2.2.3.

```
#include "painter.h"
#include "cute.h"
#include "mockator.h"
namespace testWhenCpp11IsUsed_Ns {
  namespace ShapeMock_Ns {
    INIT_MOCKATOR()
    std::vector<calls> allCalls{1};
    struct ShapeMock {
      const size_t mock_id;
      ShapeMock(const int& i) : mock_id{reserveNextCallId(allCalls)} {
        allCalls[mock_id].push_back(call{"ShapeMock(const int&)", i});
      }
      int area() const {
        allCalls[mock_id].push_back(call{"area() const"});
        return int{};
      }
    };
  }
}
void testWhenCpp11IsUsed() {
  using namespace testWhenCpp11IsUsed_Ns::ShapeMock_Ns;
  ShapePainter<ShapeMock> painter;
  painter.paint();
  calls expectedSquareMock = {{"ShapeMock(const int&)", int {}},
    {"area() const" }};
  ASSERT_EQUAL(expectedSquareMock, allCalls[1]);
}
```

Listing 4.4: *Code of listing 4.2 after applying the move to namespace source action.*

Consider listing 4.4 which shows the result of applying the move to namespace source action on the code in listing 4.2. Note that we remove the `INIT_MOCKATOR` call and the calls vector from the test function and instead create a using namespace declaration. In the newly created namespace we do not make the calls vector static anymore because this restriction is only necessary with local classes.

4.3. Converting Fake to Mock Objects

Sometimes we might start with a fake object to inject into the SUT but then encounter that we actually also need to verify the collaboration between them. For this case, we provide a source action to convert an existing fake to a mock object. This includes the registration of the calls as well as the complete infrastructure that is necessary to use our mock object library.

```
#include "gamefourwins.h"
#include "cute.h"
namespace testGameFourWins_Ns {
  namespace FakeDie_Ns {
    struct FakeDie {
      int roll() const { return 4; }
    };
  }
}
void testGameFourWins() {
  using namespace testGameFourWins_Ns::FakeDie_Ns;
  GameFourWinsT<FakeDie> game;
  std::ostringstream oss;
  game.play(oss);
  ASSERT_EQUAL("You won!\n", oss.str());
}
```

Listing 4.5: *Fake object before applying the source action to convert it to a mock object.*

Listing 4.5 shows the situation before applying the source action to convert the fake to a mock object. After the conversion to a mock object, the code as shown in listing 4.6 is created. Note that we now insert a constructor when a compile seam is used to register the class instantiation.

4.4. Toggle Mock Support For Functions

Because we think it might be useful to enable or disable the recording of function calls for a member function in mock objects, we have implemented a source action to do so. This is not only a matter of removing the recording in the member function, but also to adapt the expectations accordingly. To show an example, we take the code in listing 4.6 and apply the toggling mock support action on the member function `roll`. Listing 4.7 contains the resulting diff of this source action. Note that the registration of the function call has been deleted and the expectation for the member function `roll` is removed.

```
d
<   allCalls[mock_id].push_back(call{"roll() const"});
c
<   calls expectedFakeDie = { { "FakeDie()" }, { "roll() const" } };
---
>   calls expectedFakeDie = { { "FakeDie()" } };
```

Listing 4.7: *Diff of applying toggle mock support on the member function* `roll` *of listing 4.6. Note that we removed the line and column numbers in the diff because they are not of interest here.*

```
#include "gamefourwins.h"
#include "cute.h"
#include "mockator.h"
namespace testGameFourWins_Ns {
  namespace FakeDie_Ns {
    INIT_MOCKATOR()
    std::vector<calls> allCalls{1};
    struct FakeDie {
      const size_t mock_id;
      FakeDie() : mock_id { reserveNextCallId(allCalls) } {
        allCalls[mock_id].push_back(call{"FakeDie()"});
      }
      int roll() const {
        allCalls[mock_id].push_back(call{"roll() const"});
        return 4;
      }
    };
  }
}
void testSUT() {
  using namespace testGameFourWins_Ns::FakeDie_Ns;
  GameFourWinsT<FakeDie> game;
  std::ostringstream oss;
  game.play(oss);
  ASSERT_EQUAL("You won!\n", oss.str());
  calls expectedFakeDie = { { "FakeDie()" }, { "roll() const" } };
  ASSERT_EQUAL(expectedFakeDie, allCalls[1]);
}
```

Listing 4.6: *Resulting mock object after the conversion.*

As the name of this source action implies, it is also possible to add the recording again. This includes the linked edit mode where the user can change the function arguments if this is activated in the project settings.

4.5. Registration Consistency Checker

The user might sometimes manually adapt the registrations in the member functions of the mock object or the expectations which could lead to the situation where the expectations and the actual registrations are not consistent anymore. Because this could lead to tedious debugging sessions, we provide a CodAn checker with a quick fix to correct these inconsistencies.

Note that we sometimes want to test that the SUT does not call a certain member function. In this situation, the user would either use `assert(expected != allCalls`

`[1])` or `ASSERT(expected != allCalls[1])` depending if CUTE is used or not and we will not create a marker. Another situation is the use of regular expressions in the expectations. In this case, we will not analyse for inconsistencies because it is not possible to this without knowing the actual function calls with its arguments executed on the injected mock object.

4.6. Mock Functions

So far we have only mocked classes when we applied object and compile seams. Sometimes it would be tedious to apply one of these two seams where we just want to mock a function to see if the SUT calls it correctly. This is also true if we cannot change the code at all. For these cases, we have implemented mocking of functions by leveraging the shadow function link seam. We used this kind of link seam because it is the easiest one that fulfils our requirements and because we do not intend to call the original function. Instead, we just record the call to have it available for assertion in our mock object implementation.

```
#include "draw_funs.h"
#include "crossplanefigure.h"
void CrossPlaneFigure::rerender() {
  // draw the label
  drawText(m_nX, m_nY, m_pchLabel, getClipLen());
  drawLine(m_nX, m_nY, m_nX + getClipLen(), m_nY);
  drawLine(m_nX, m_nY, m_nX, m_nY + getDropLen());

  if (!m_bShadowBox) {
    drawLine(m_nX + getClipLen(), m_nY, m_nX + getClipLen(), m_nY +
        getDropLen());
    drawLine(m_nX, m_nY + getDropLen(), m_nX + getClipLen(), m_nY +
        getDropLen());
  }

  // draw the figure
  for (int n = 0; n < edges.size(); n++) {
    //...
  }
  //...
}
```

Listing 4.8: *CAD code that is hard to test because it uses a third-party drawing library.*

Feathers discussed the example of a CAD application that uses a separate third-party drawing library in [Fea04]. The code for this example is shown in listing 4.8 where the member function `CrossPlaneFigure::rerender` executes various of these drawing functions. One way of testing this member function would be to look at the computer screen when the figures are redrawn [Fea04]. Of course, this is not what we want. Instead we can use our implemented mock function implementation. If the user selects one of the calls to the function `drawLine` and applies our mock function feature, we generate the code in listing 4.9.

```
//drawline.h
#include "cute_suite.h"
extern cute::suite make_suite_drawLine();

//drawline.cpp
#include "crossplanefigure.h"
#include "draw_funs.h"
#include "cute.h"
#include "mockator.h"

namespace testdrawLine_Ns {
  mockator::calls allCalls;
}

void drawLine(int firstX, int firstY, int secondX, int secondY) {
  testdrawLine_Ns::allCalls.push_back(mockator::call{"drawLine(int,int,int,
      int)",firstX,firstY,secondX,secondY});
}

void testdrawLine() {
  INIT_MOCKATOR();
  calls expected = {{"drawLine(int,int,int,int)",int{},int{},int{},int{}}};
  // call SUT
  ASSERT_EQUAL(expected, testdrawLine_Ns::allCalls);
}

cute::suite make_suite_drawLine() {
  cute::suite s;
  s.push_back(CUTE(testdrawLine));
  return s;
}
```

Listing 4.9: *Generated code of our mock function implementation. Note that the comment has been manually inserted to show where we would need to call our SUT* `CrossPlaneFigure::rerender`.

As one can see from this code, we shadow the original implementation of `drawLine` and register the calls in a vector. What is left to the user is calling the SUT and adapting the expectations in the test function `testdrawLine`.

Note that our implementation registers a test function in a CUTE suite in case the action is executed in a CUTE project. This suite is then linked to a CUTE runner. This process is supported by a dialog that we have taken from the CUTE plug-in where the user can choose the name and destination of the new suite and the runner to link the suite to.

The code for linking the suite to a runner is shown in listing 4.10. Note that we created an include for the header file where the function that creates our suite is declared.

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include "drawline.h"
void runSuite() {
  cute::suite s;
  cute::ide_listener lis;
  cute::makeRunner(lis)(s, "The Suite");
  cute::suite drawLine = make_suite_drawLine();
  cute::makeRunner(lis)(drawLine, "drawLine");
}
int main() {
  runSuite();
  return 0;
}
```

Listing 4.10: *Newly created suite for the mock function test linked to the chosen runner.*

In case the chosen project where the mock function action is applied does not have a CUTE nature, we create the code of listing 4.11.

```
//drawline.cpp
#include "crossplanefigure.h"
#include "draw_funs.h"
#include <cassert>
#include "mockator.h"
namespace testdrawLine_Ns {
  mockator::calls allCalls;
}
void drawLine(int firstX, int firstY, int secondX, int secondY) {
  testdrawLine_Ns::allCalls.push_back(mockator::call{"drawLine(int,int,int,
      int)",firstX,firstY,secondX,secondY});
}
void testdrawLine() {
  INIT_MOCKATOR();
  calls expected = {{"drawLine(int,int,int,int)",int{},int{},int{},int{}}};
  // call SUT
  assert(expected == testdrawLine_Ns::allCalls);
}
//drawline.h
extern void testdrawLine();
```

Listing 4.11: *Generated code for mock function when the action is applied in a project without CUTE nature.*

5. Plug-in Implementation Details

The following sections describe the architecture and some important implementation details of the Mockator Eclipse plug-in. We also discuss a few problems that came up during the development of our plug-in.

5.1. Plug-in Architecture

The Eclipse plug-in of Mockator consists of ten main Java packages. These packages and their relationships are shown in the package diagram of figure 5.1. Note that we do not show dependencies to the package `base` because of space reasons as all other packages dependend on that one.

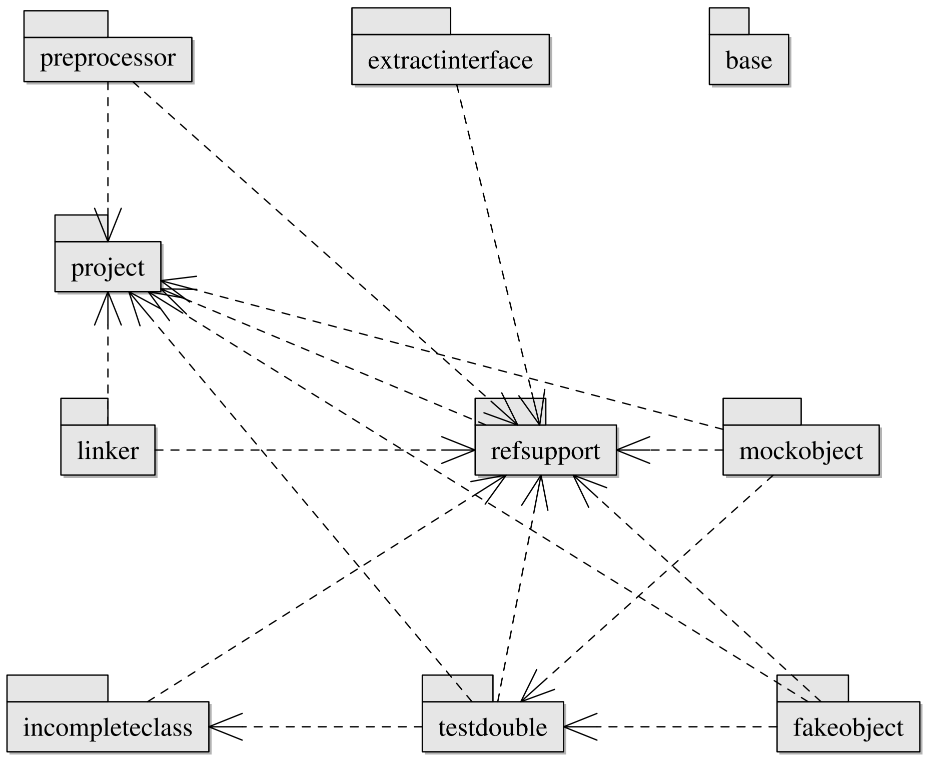


Figure 5.1.: *Java package overview of the Mockator Eclipse plug-in. Note that we do not show dependencies to the package* `base` *because of space reasons.*

`base` provides functionality that is used by all other packages like internationalisation, utility classes for string handling and easier use of Java collections, support for programming with higher-order functions like `map`, `filter` and `fold` and implementations of an option type and a tuple library.

`extractinterface` contains our implementation of the extract interface refactoring. `incompleteclass` handles the process of recognising missing member function implementations and to provide default implementations for both object and compile seams, as it is described in chapter 2. `fakeobject` is used to create the necessary code for providing fake objects as it is discussed in section 2.2.4. `mockobject` includes the functionality to create the infrastructure that is necessary for the Mockator library and all supplementary source actions which we have described in chapter 4. `testdouble` contains common functionality which both fake and mock objects use and the implementations for creating test doubles and moving them to a namespace.

The package `linker` contains our implementations of the three link seam types shadow, wrap and intercept function as explained in section 2.4. `preprocessor` provides the implementation of the preprocessor seam as discussed in section 2.3. In the package `project` we have put all classes that are related to modifying and accessing Eclipse CDT projects. These include altering the options of the compiler and linker, our nature and the necessary classes to extend the CUTE project wizard. Finally, `refsupport` contains classes that are necessary for all refactorings like the handling of includes, the lookup of names, the creation of translation units and the common quick fix infrastructure.

5.2. Used Eclipse Extension Points

Contributions to Eclipse are done through so called *extension points*. In this section we will show the extension points the Mockator Eclipse plug-in uses to provide its functionality.

5.2.1. org.eclipse.cdt.codan.core.checkers

We use this extension point to provide all of our checkers to the CodAn framework. This extension point also allows to offer a new code analysis problem category to CodAn, which we have done as well ("Mockator problems"). The checkers we contribute have different kinds of severities. While the missing object and compile seams and the missing member functions in them yield errors because the code would not compile without applying the quick fix, others like the GNU link seam and the preprocessor checker use an information severity because they are just used to toggle their activation status. For inconsistent expectations we create markers as warnings.

5.2.2. org.eclipse.cdt.codan.ui.codanMarkerResolution

This extension point is used for the various quick fixes Mockator provides. Note that we sometimes have several quick fixes for one specific problem ID. This can be done by providing multiple resolution tags with the same problem ID, but a different class implementing the quick fix (see example in listing 5.1).

```
<resolution
  class="ch.hsr.ifs.mockator.plugin.fakeobject.FakeObjectQuickFix"
  problemId="ch.hsr.ifs.mockator.StaticPolyMissingMemFunsProblem">
</resolution>
<resolution
  class="ch.hsr.ifs.mockator.plugin.mockobject.qf.
      MockObjectByFunArgsQuickFix"
  problemId="ch.hsr.ifs.mockator.StaticPolyMissingMemFunsProblem">
</resolution>
```

Listing 5.1: *Offering multiple resolutions for a given problem ID with CodAn.*

5.2.3. org.eclipse.ui.popupMenus

We use this extension point to provide our source actions in the CDT source context menu, the extract interface refactoring in the CDT refactoring context menu and for the toggling of Mockator support on an Eclipse project.

5.2.4. org.eclipse.ui.bindings

For all our source actions we defined keyboard shortcuts like `Ctrl+Alt+C` to convert a fake to a mock object. Listing 5.2 shows how this can be done in the `plugin.xml`.

```
<key
  commandId="ch.hsr.ifs.mockator.convertToMockObjectCommand"
  contextId="org.eclipse.cdt.ui.cEditorScope"
  schemeId="org.eclipse.ui.defaultAcceleratorConfiguration"
  sequence="M1+M3+C">
</key>
```

Listing 5.2: *Keyboard shortcut for the source action to convert fake to a mock object.*

5.2.5. org.eclipse.core.resources.natures

We provide our own nature with the ID `ch.hsr.ifs.mockator.MockatorNature`. This nature is dependent on the ID `org.eclipse.cdt.core.ccnature` because Mockator does only work with C++ projects.

5.2.6. org.eclipse.ui.propertyPages

For projects that have our nature we also offer a properties dialog where the settings for Mockator can be altered like which C++ standard we should generate code for and a few other settings.

5.2.7. org.eclipse.help.toc

This extension point allows plug-ins to provide their own help documentation which is included in the Eclipse help system [CR09]. Mockator uses this extension point to provide the pages with the links to the screen casts and a changelog.

5.2.8. org.eclipse.ui.intro.configExtension

Our Eclipse plug-in contributes by the use of this extension point to the Eclipse welcome screen. We use both an overview and a tutorials section to link to our help pages.

5.3. Collaboration With CUTE

Although our plug-in is not dependent on CUTE and only has an optional bundle dependency to it, we do want to ease the collaboration with it. We therefore hook our plug-in into the CUTE project creation wizard by using the extension point `ch.hsr.ifs.cute.ui.wizardAddition`. With this, CUTE allows us to provide our own UI elements in this wizard by specifying a class that implements the interface `ICuteWizardAddition`. Our contribution is that a user can add support for Mockator to a new CUTE project and choose which C++ standard we generate code for.

The addition of UI elements to the CUTE project creation wizard is handled by the interface `ICuteWizardAddition` by its method `createComposite`. With the help of this we can add our UI elements to the composite. The handling of the chosen settings in the wizard is done by implementing the interface `ICuteWizardAdditionHandler` and adding our nature in the method `configureProject`. The bridge between these two classes is created by implementing the method `getHandler` in the class implementing the interface `ICuteWizardAddition`.

5.4. CDT Project Settings

Mockator makes heavy use of the various settings Eclipse CDT provides for the tools of a particular tool chain (e. g., GCC). Because of that, we show a few important facts about collecting CDT project information and how these settings can be altered. We use as an example the option “other flags” CDT provides for a compiler and add support for C++11 when GCC is used as shown in listing 5.3.

```java
void addCpp11SupportForGcc(IProject project) throws CoreException {
  IManagedBuildInfo info = ManagedBuildManager.getBuildInfo(project);
  IConfiguration config = info.getDefaultConfiguration();
  String gnuCompilerId = "cdt.managedbuild.tool.gnu.cpp.compiler";
  String otherGnuCompilerFlags = "gnu.cpp.compiler.option.other.other";
  for (ITool tool : config.getToolChain().getTools()) {
    if (gnuCompilerId.equals(tool.getId()) {
      IOption option = tool.getOptionBySuperClassId(otherGnuCompilerFlags);
      String newFlags = flagsOption.getStringValue() + " std=c++0x";
      ManagedBuildManager.setOption(config, tool, option, newFlags);
      ManagedBuildManager.saveBuildInfo(project, true);
    }
  }
}
```

Listing 5.3: *Example of adding support for C++11 to a project with the help of CDT's managed build system. Note that we have omitted exception hanlding and null checks because of space reasons.*

The starting point for all operations is to get the build information for a managed project which is provided by the interface `IManagedBuildInfo`. With its help we can either get all configurations attached to a managed build configuration by using `getManagedProject().getConfigurations()` or — if we are just interested in the default configuration of the project — execute `getDefaultConfiguration()`. With the configuration we can fetch all tools that are associated to the used tool chain of the configuration. As we only want to alter the options of the compiler, we compare its ID with the one of the GCC compiler. An option has different kinds of possibilities to fetch its values depending on its type. For “other flags”, the value is just a plain string which we can get with the help of `getStringValue()`. Finally, we set the option and save it by issuing the corresponding methods on the `ManagedBuildManager` class.

```java
boolean isExecutable(IConfiguration config) {
  return config.getBuildArtefactType().getId().equals(
    ManagedBuildManager.BUILD_ARTEFACT_TYPE_PROPERTY_EXE);
}
```

Listing 5.4: *Example of detecting if the project build artefact is an executable in Eclipse CDT.*

Another often necessary step is to detect the artefact type of a certain project. This can be either an executable, a static or a shared library for a managed build project in CDT. The class `ManagedBuildManager` can help here again as can be seen in listing 5.4. Note that every configuration has a build artefact type which also has an ID. `ManagedBuildManager` also provides constants for the artefact types which we can use here.

5.5. Creating Run-time Configurations

For the run-time function interception link seam described in section 2.4.3 we have to alter the launch configuration of the referencing executable project to tweak the environment variables of the dynamic linker. The process of retrieving these configurations and altering them is described in this section. The first step is to collect all launch configurations which have the launcher type of the CUTE plug-in which is shown in listing 5.5. Note that the necessary interfaces and classes are provided by the plug-in `org.eclipse.debug.core`.

```java
Collection<ILaunchConfiguration> getCuteLaunchConfigs(IProject proj) {
  ILaunchManager manager = DebugPlugin.getDefault().getLaunchManager();
  ILaunchConfigurationType cuteType = manager.getLaunchConfigurationType(
    "ch.hsr.ifs.cutelauncher.launchConfig");
  return filter(manager.getLaunchConfigurations(cuteType),
    new F1<ILaunchConfiguration, Boolean>() {
      @Override public Boolean apply(ILaunchConfiguration config) {
        ICProject cProject = CDebugUtils.getCProject(config);
        return proj.equals(cProject.getProject());
      }
    });
}
```

Listing 5.5: *Example of collecting all CUTE launch configurations for a given project. Note that this example makes use of the `filter` function provided by our `HigherOrderFunctions` class.*

After we have the launch configurations, we want to change them. Listing 5.6 shows the process of adding the dynamic linker environment variable `LD_PRELOAD` with a value of a custom shared library to every CUTE launch configuration of the given project.

```java
for (ILaunchConfiguration config : getCuteLaunchConfigs(proj)) {
  ILaunchConfigurationWorkingCopy wc = config.getWorkingCopy();
  Map<String, String> envs = wc.getAttribute(
    ILaunchManager.ATTR_ENVIRONMENT_VARIABLES,new HashMap<String,String>());
  envs.put("LD_PRELOAD", "libRand.so");
  wc.setAttribute(ILaunchManager.ATTR_ENVIRONMENT_VARIABLES, envs);
  wc.doSave();
}
```

Listing 5.6: *Example of adding an environment variable to all launch configurations.*

Note that the second attribute of the method `getAttribute` is used as the default value in case there is no value with the given key so far.

5.6. Executing Refactorings in Eclipse Jobs

Refactorings are executed by created a change object and calling `perform` on it. The methods necessary to accomplish this are provided by CDT's class `CRefactoring2`. Often we also want to use the change object afterwards, e. g. to fetch the necessary details from it to provide a linked mode. Listing 5.7 shows the basic actions. Note however that this procedure is different when using the LTK framework.

```
RefactoringASTCache astCache = new RefactoringASTCache();
try {
  MyRefactoring refactoring = new MyRefactoring(file, selection, astCache);
  Change change = refactoring.createChange(pm);
  change.perform(pm);
  // do something with the change object, e.g. using it for a linked mode
catch (CoreException e) {
  //error handling omitted
} finally {
  astCache.dispose();
}
```

Listing 5.7: *Basic procedure to execute a refactoring in Eclipse CDT.*

The problem with this approach is that refactorings executed like this from a delegate class are performed in the UI thread. Because every long running action that is run in the UI thread harms the user experience, we decided to execute all our refactorings in separate Eclipse jobs which are run by a job manager in an own thread. This excludes the extract interface refactoring, because this is executed by the runner infrastructure of LTK.

```
<T> void runInDisplayThread(final F1V<T> callBack, final T param) {
  Runnable runnable = new Runnable() {
    @Override public void run() {
      callBack.apply(param);
    }
  };
  Display display = getDisplay();
  if (isDisplayThreadCurrentThread(display))
    runnable.run();
  else
    display.syncExec(runnable);
}
```

Listing 5.8: *Running a function in the UI thread we used for providing linked modes.*

To run actions like the execution of the linked mode in the UI thread, we provide a method to register a callback for this purpose, as can be seen in listing 5.8. Note that we have experienced problems with not releasing the lock the index is protected with when the RefactoringASTCache was not disposed before running the linked mode. We therefore make sure that this happens by executing it in a finally block before the UI callback is executed.

5.7. Missing Preprocessor Support in AstRewriter

Although preprocessor statements like #include or #define are part of the AST in Eclipse CDT (the base interface is IASTPreprocessorStatement), CDT's ASTRewrite class does not support them and yields an IllegalArgumentException when we attempt to insert, replace or delete them. Because of this, we had to insert our includes and #undef's as literal nodes.

This was of course inconvenient, but it is even worse when we try to delete a macro as it is the case with INIT_MOCKATOR in the move test double to namespace refactoring explained in section 4.2. The solution we found was to get the offset and the length of the INIT_MOCKATOR macro and to create a DeleteEdit instance with this information. Furthermore, we had to pack this DeleteEdit into a TextFileChange object to yield it from CRefactoring2's method createChange.

Note that the handling with text edits which are used to apply elementary text manipulation operations on documents is not as convenient to work with as the rewriter. It would be therefore beneficial to support preprocessor nodes in the rewriter.

5.8. Checking Refactoring Pre- and Postconditions

An important part of every refactoring is that all pre- and postconditions are verified correctly. LTK's class Refactoring provides two methods for this process called checkInitialConditions and checkFinalConditions. While the former is used to perform initial checks right after the refactoring has been started like if the selected element is of the correct type, the latter is used to execute checks after the user has provided all necessary information to execute the refactoring.

As we have used these checks often in our refactorings, we show its usage with an example for the extract interface refactoring in listing 5.9. Note that a RefactoringStatus object is used as the result of a condition checking operation in LTK. It consists of a number of RefactoringStatusEntry objects which describe a problem that was detected. Every problem has one of the following serverities: ok, info, warning, error or fatal. In case we are not able to detect the class in the current selection which we should extract an interface from, we cannot perform the refactoring and have to abort it. For a situation like this the status *fatal* is the one to choose.

```
@Override
public RefactoringStatus checkInitialConditions(IProgressMonitor pm)
throws CoreException {
  RefactoringStatus status = super.checkInitialConditions(pm);
  if (!selectionContainsValidClass()) {
    status.addFatalError("No class found to extract an interface from!");
  }
  return status;
}
@Override
protected RefactoringStatus checkFinalConditions(IProgressMonitor pm,
  CheckConditionsContext checkContext) throws CoreException {
  RefactoringStatus status = checkContext.check(pm);
  if (chosenMemberFunctionsContainShadows()) {
    status.addWarning("Member function A::b shadows C::d"); //example
  }
  return status;
}
```

Listing 5.9: *Checking pre- and postconditions of refactorings.*

In the postcondition check we verify if one of the chosen member functions for the new interface shadows a function in one of the subclasses which would lead to behaviour changes in case we make this member function virtual in the base class. Although we have used *warning* here, we could also have yielded *error* which is normally taken when behaviour changing problems arise — but in this case we have decided against it as we think the problem is not that serious.

6. Conclusions

In this chapter we present our achieved project results, show known issues and possible improvements and conclude with personal impressions made during the last couple of months. Beside this, we also write about other contributions we have made for conferences and magazines during this thesis.

6.1. Project Results

Reflecting on the problem outline listed in section 1.5, all goals of the project have been reached. Mockator Pro is able to create all refactorings necessary for the four seam types object, compile, preprocessor and link seam. To achieve an object seam, we have implemented a new extract interface refactoring. Beside compile seams that were already supported with Mockator, we now also recognise missing member functions in object seams. We achieved a useful implementation of preprocessor seams which are especially helpful in tracing function calls. Although not requested as mandatory, we have implemented three kinds of link seams which can be used to shadow or wrap functions in static and shared libraries. For the preprocessor and link seam types which require a significant amount of manual work to deactivate or remove them we implemented quick fixes to automate this task. We provided support for Linux and Mac OS X for our seam types wherever these platforms and their corresponding GCC toolchain support them itself.

The mock object library now also supports C++03 beside C++11. Our Eclipse plug-in is able to generate code for both standards which can be chosen in the project settings. Because we recognised that it is often beneficial to just mock a single function instead of extracting an interface or a template parameter, we implemented mocking of functions including wizard support for CUTE. We implemented various convenience functions to make working with mock objects easier like moving them to a namespace, converting fake to mock objects, toggle recording on a member function level and recognising inconsistent expectations. Beside missing member functions and constructors we now also recognise missing operators.

Because our old implementation was only able to do comparisons of recorded function calls and expectations by equality which can sometimes be limiting, we also implemented support for specifying expectations with regular expressions. We analysed the thread-safety issues of our mock object library and worked out solutions to make it thread-safe. We improved the collaboration with CUTE by extending its project wizard while still not requiring its installation to work with Mockator.

6.2. Possible Improvements

This section outlines possible improvements for Mockator which we think would be useful to implement in the future.

6.2.1. Support For Template Functions in Link Seams

We currently do not support template functions in our implementation of link seams, although from a language and compiler view it would be technically possible. To achieve that for wrapping functions, we would have to create explicit template instantiations and provide support for templates in our name mangling implementation. Listing 6.1 shows how it is possible to wrap the template function `max` with the link seam described in section 2.4.2.

```
//max.h
template<typename T>
T max(T const&, T const&);

//max.cpp
#include "max.h"
template<typename T>
T max(T const& a, T const& b) {
  return a < b ? b : a;
}
template int max(int const&, int const&); // explicit instantiation

//foo.cpp
#include "max.h"
#ifdef WRAP__Z3maxIiET_RKS0_S2_
extern "C" {
  extern int __real__Z3maxIiET_RKS0_S2_(int const&, int const&);
  int __wrap__Z3maxIiET_RKS0_S2_(int const& a, int const& b) {
    return __real__Z3maxIiET_RKS0_S2(a, b);
  }
}
#endif
void foo() {
  max(42, 3);
}
```

Listing 6.1: *Example of wrapping a template function by using an explicit instantiation.*

Because of the reasons we mentioned in section 2.4.2, we cannot use template functions when the inclusion model [VJ02] is used with definitions included in the translation unit where we attempt to wrap the function. Instead, the user has to split the declaration and definition in the usual way as it is done with non-template functions, but with an additional explicit instantiation of the template function which results in a symbol that we can wrap with the linker later.

Although the separation of declaration and definition and the explicit instantiation is out of the scope of Mockator, we should make sure that our name mangling routine supports template functions. With this, we would at least allow this to work when the user does the separation manually.

6.2.2. Include Header File Only Where Necessary in Preprocessor Seam

To make the preprocessor seam work, we add an include of the header file with the macro that redefines a function with our mocked version to the Eclipse CDT project settings which results in the use of the GCC compiler option `-include file`. This has the same effect as if we would add an `#include "file"` into every translation unit of the project as the first line. Although this works, every call to the original function will be mapped and we additionally have to add an `#undef` before the original function definition because otherwise we would violate the one definition rule of C++.

At the end of this thesis, we discovered that it is possible in CDT to specify tool options for single translation units and not only for the containing project. This would allow us to just redefine the function in the translation unit the user has chosen. We think that this would be a useful addition beside the redefinition of functions on a project level.

Altering tool options can be done by creating an `IFileInfo` object and passing this to the `ManagedBuildManager` instead of a `IConfiguration` as explained in section 5.4. An example for adding an include file for the CDT compiler options is shown in listing 6.2. What is left to be analysed is if this still works if translation units are moved to another folder or are simply renamed.

```
IConfiguration config = // config to alter
ITool tool = // GCC compiler
IOption option = // GCC include file option IOption.INCLUDE_FILES
String[] newIncludes = // new includes
IFileInfo fInfo = config.createFileInfo(new Path("path/to/tu.cpp"));
ManagedBuildManager.setOption(fInfo, tool, option, includes);
```

Listing 6.2: *Example of altering CDT tool options for single translation units.*

6.2.3. Support More Tool Chains

Mockator currently only supports the GCC tool chain for its preprocessor and the three link seam types. This is because these seam types rely on the usage of certain compiler and linker options. To achieve a wider acceptance with Mockator, we should try to support other tool chains beside GCC. Important additional tool chains to support would be Clang [unk12] and Microsoft's Visual C++.

We have already prepared support for additional tool chains in our implementation of the `project.cdt.toolchains` package. Note that Micrsoft Visual C++ does not support wrapping functions with the `-wrap` compiler option and does also not have a pendant to `LD_PRELOAD`. But in [MB11] it is written that there is an implementation of intercepting win32 functions with the *Detours* package [Res12] which works by rewriting the in-memory code of the target functions.

Another important aspect is name mangling. If other tool chains provide functionality for wrapping functions and these would use a different name mangling strategy than Itanium ABI, then we also need to enhance our name mangling implementation.

6.2.4. Recognise More Concept Kinds

Mockator recognises missing member functions, constructors and operators the SUT calls on the injected test double and creates default implementations for them. Additionally, it might be also useful to support missing member variables and `typedef`'s. To have an example, consider listing 6.3 where we use the local class `Fake` as the traits template parameter of `std::basic_string` instead of the default `char_traits<char>` and `char_traits<wchar_t>`, respectively.

```
#include <string>
void testWithBasicString() {
  struct Fake {
    typedef char char_type;
    static int compare(char_type const*, char_type const*, size_t) {
      return int{};
    }
  };
  std::basic_string<char, Fake> str1;
  std::basic_string<char, Fake> str2;
  str1.compare(str2);
}
```

Listing 6.3: *Example showing that we would need to create a* `typedef` *which is used by* `basic_string`*.*

Because `basic_string` is using the `typedef` of the traits template parameter as its value type (see listing 6.4), we have to provide this in our local class.

```
template<typename _CharT, typename _Traits, typename _Alloc>
class basic_string {
  typedef typename _Traits::char_type value_type;
  //...
}
```

Listing 6.4: *Extract of `basic_string` showing the use of the `typedef` of the traits template parameter.*

6.2.5. Thread Safety

As we have outlined in section 3.4, Mockator's mock object library is not thread-safe. We discussed the approach of making the call registrations thread-safe by using a mutex. We came to the conclusion that there would be too much code needed in the test doubles to achieve this and that a better solution is to use a thread-safe container like Intel's `tbb::concurrent_vector`. For a future productive release of Mockator, we have to either clearly communicate that our library is not thread-safe or to use this container in our mock object library and declare `tbb` as a dependency of Mockator.

6.3. Known Issues

This section presents some unresolved problems or bugs which could not be fixed during this thesis.

6.3.1. Dead File When Refactoring is Aborted

For our extract interface refactoring, we have to create a new header file where our interface class is declared. CDT provides the class `CreateFileChange` for this purpose which extends the LTK class `Change`. Although `CreateFileChange` returns a `DeleteFileChange` instance in the method `perform` as its undo change object, if the user aborts a refactoring in the dialog before clicking “Finish”, an empty file remains in the workspace. As this is more an issue of CDT, we have not tried to fix this.

6.3.2. Passing this in SUT

As explained in section 2.2.3, local classes are not allowed to have template members. We therefore have to use the concrete type used by the instantiation of the SUT template as the parameter type in the injected test double when `this` is passed in the SUT. Listing 6.5 depicts this situation. Mockator does not support this yet.

```
template <typename T>
struct SUT {
  void bar() {
    T::foo(this);
  }
};
void testSUT() {
  struct Fake {
    static void foo(SUT<Fake> const*) {
    }
  };
  SUT<Fake> sut;
  sut.bar();
}
```

Listing 6.5: *Passing `this` in the SUT on the injected test double should use the concrete type in the template-id of the function parameter type.*

6.4. Articles and Conference Contributions

To make our work more popular, we decided to create a paper for the 7th International Workshop on Automation of Software Test (AST) [Com12] which is a workshop part of the 34th International Conference on Software Engineering (ICSE). Our paper of seven pages which is mainly about the topics discussed in chapter 2 got accepted and will be published in the ICSE 2012 conference proceedings in IEEE Digital Libraries. To achieve this, we also had to present our paper in the workshop which was located at the University of Zürich Irchel. The workshop consisted of the sessions security, surveys, industrial case studies, input generation / selection, GUI testing and designing for test. Our presentation was part of the latter and we had the opportunity to present our work during about half an hour on a Sunday in front of a mostly academical audience. Overall, we think it was a great opportunity to make people aware of the issues of designing code for testability and how this can be achieved with seams and our refactorings.

To reach a wider audience which is more aware of the tricks used to achieve seams in C++, we also decided to submit an article for the magazine *Overload* which was published in its issue 108 [Rü12]. The editor of Overload[1] agreed to a subsequent article which is about using mock objects in C++ and is planned for issue 110.

Beside this, we also had the opportunity to present Mockator at the Workshop on Refactoring Tools 201 which was held at the University of Applied Sciences in Rapperswil. Furthermore, Prof. Peter Sommerlad introduced Mockator beside other topics at a Google talk held in January 2012 and at the ACCU conference in April 2012.

[1]Personal communication with the editor of Overload Ric Parkin.

6.5. Personal Review and Acknowledgements

Looking back to the past couple of months, I think this project was overall a great experience and I am happy with its outcome. This thesis was kind of special to me because of its long duration that is due to the fact that I did it part-time. It was not easy to keep the motivation high during 38 weeks of work without being challenged by team members. But I am proud that I managed to constantly work highly motivated on this thesis for such a long time.

I think I learned a lot about C++, Eclipse CDT and build related topics like name mangling, shared libraries and how linkers work. From my point of view, especially the latter topics are very important and a basic understanding of these is essential for every non-trivial C++ software project.

A great experience was the publication of a scientific paper. This was the first time for me to go through the whole application process for a publication which was both interesting and challenging. From the abstract to the first draft, the corrections based on the reviews of the conference committee, to re-submissions and the camera-ready version and finally the actual presentation in front of an academic audience was an opportunity I would not like to have missed.

I also enjoyed the communication and the feedback of the editors of the Overload magazine. They helped me to constantly improve my article. It was a great feeling when I finally got my copy of the magazine with my article in there.

Personally I think it was the right decision to continue the term project as my master's thesis. The new functionality of supporting seams goes hand in hand with using test doubles and I think they really belong together. The practices of preprocessor and link seams have been used since decades, especially by Unix and Linux programmers, but have so far involved a lot of tedious work which is now done by Mockator. I am really looking forward to apply the refactorings in the practice.

At this point I want to thank my advisor, Prof. Peter Sommerlad, for all the help, valuable remarks and impulses he gave me. I especially appreciate that he motivated me for publishing my work in a magazine and for a scientific conference. I would also like to thank Thomas Corbat for proofreading my paper for the AST conference and for helping me when CDT problems came up. Last but not least I also thank Lukas Felber for the discussions we had about the Eclipse CDT testing infrastructure and for providing me the code to use external files in refactoring tests.

A. User Guide

A.1. Installation of the Plug-in

Mockator needs Eclipse Indigo (3.7) with CDT 8.0.0 installed. This can be done either by installing a bundled "Eclipse IDE for C/C++ developers CDT distribution" from the Eclipse downloads page[1] or by installing CDT from an existing Eclipse installation by using its update site[2].

Afterwards, the Mockator plug-in can be installed. Choose the menu "Help", "Install New Software..." and type the URL of the Mockator update site[3] into the address bar.

A.2. Use of the Plug-in

Due to the fact that some of the processes to achieve seams and mock objects are complex we think that screen casts are better suited as helping material than a textual description with screen shots. We therefore have created two detailed screen casts including audio explanations for both refactoring towards seams (~30 minutes) and mock objects (~20 minutes). We then split the screen casts with the help of `mencoder` (see chapter I) into chunks for every important subtopic. These chunks are only about 3 – 4 minutes long and assist the user in learning a specific application area of Mockator. In the Eclipse help of Mockator, we link to these screen cast chunks. The following list shows the topics we provide as tutorials in our Eclipse help:

- Refactoring Towards Seams (Introduction, object seam, compile seam, preprocessor seam, link seams)
- Mock Objects (How to create mock objects, converting fake to mock objects, moving mock object to namespace, toggling mock support, consistency of registrations, mocking functions, using regular expressions for expectations)

[1] `http://www.eclipse.org/downloads`
[2] `http://download.eclipse.org/tools/cdt/releases/indigo`
[3] `http://sinv-56033.edu.hsr.ch/mockator/repo`. Note that this is a temporary virtual server that will be removed a couple of weeks after this thesis is finished.

B. Organisational

This chapter describes how this project was organised. We present the used environment and tools, show the planning of the project and how much time was invested.

B.1. Project Environment

This section presents the used software components for this project. Beside the ones for the continuous integration server we will also show the development environment used to implement the plug-in and to write the project report.

B.1.1. Continuous Integration Server

This project used the virtual server `sinv-56033.edu.hsr.ch` hosted at the University of Applied Sciences Rapperswil as its continuous integration server. The virtual server runs in a VMWare virtual infrastructure cluster, has Ubuntu 64 Bit 10.04.1 installed, about 512 MB memory and 10 GB disk space. The software installed on this machine can be seen in table B.1.

Tool	Version
Git	1.7.0.4
Trac	0.12.2
Jenkins	1.412
Apache	2.2.14
TeX Live	2009-7
Java Run-time Edition	1.6.0_24
Maven	3.0.2

Table B.1.: *Software installed on the CI environment.*

B.1.2. Development Environment

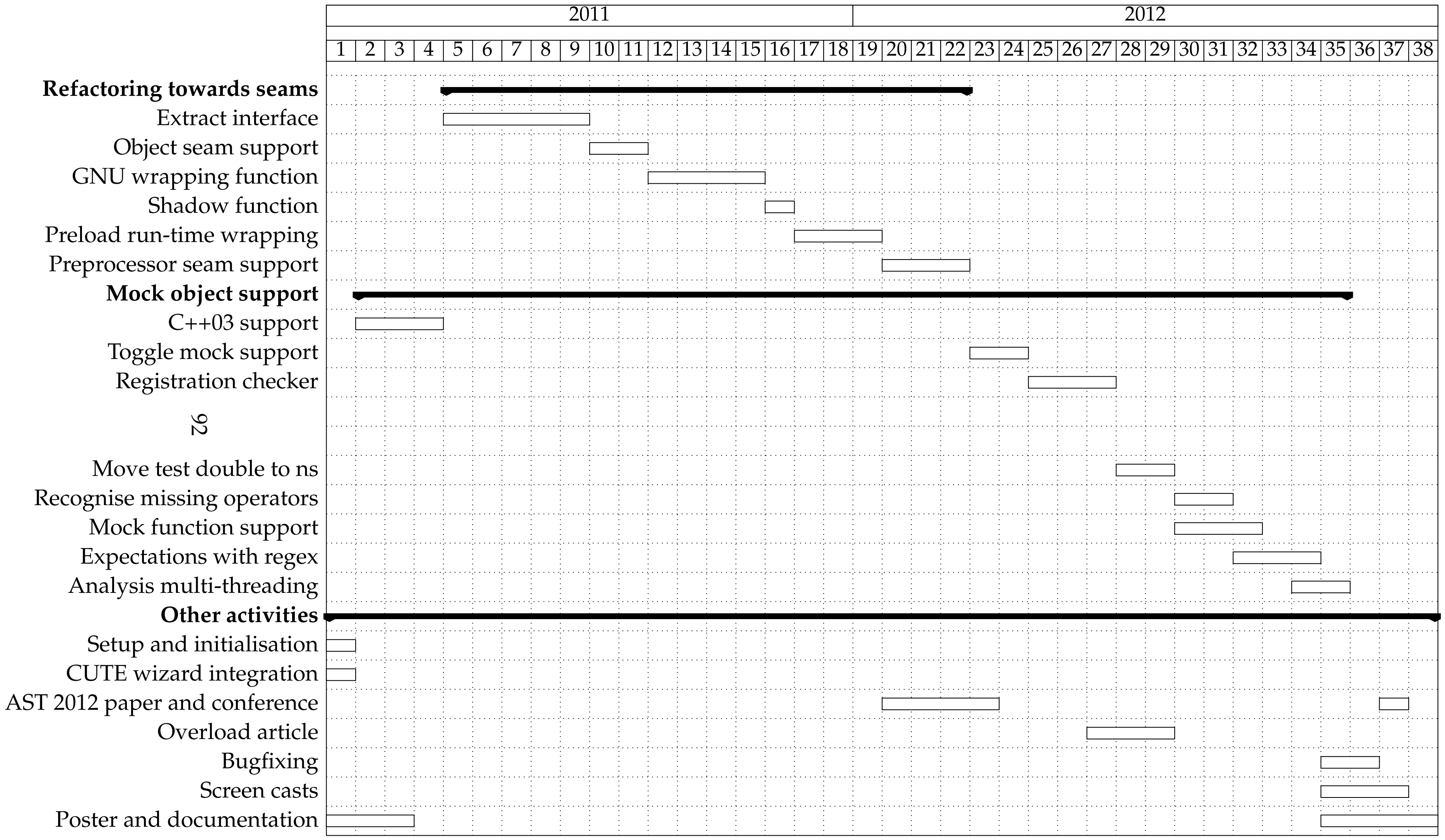
To produce the 28'000 lines of Java, the 502 lines of C++ code[1] and the 34'051 words for this report, a Lenovo T420s notebook (Intel i7-2640M CPU with 2.80GHz, 4 GB physical memory, SSD) with ArchLinux and the following software was used:

- **Eclipse Indigo 3.7.2 with CDT 8.0.2** as the development environment for the plug-ins.
- **Emacs with AUCTeX 11.86** - the best editor mixed with the most compelling LaTeX support in the world.
- **VIM with Trac Plug-in 0.3.6** to maintain the Trac Wiki and the tickets.
- **Git 1.7.10.1** as the source control management system.
- **Inkscape 0.48** to create some of the images for this report.
- **ganttchart 2.1** as a TikZ based LaTeX package used for drawing the Gantt chart in this chapter.
- **LibreOffice 3.5** to fill out the project assignment (which was a MS Word template) and to create the diagram for the work distribution over the weeks and the project plan.
- **MetaUML 0.2.5** to create the class and package diagrams in this report.

B.2. Project Plan

At the start of the project a project plan was created which has been slightly revised several times during the project. Due to the nature of this project which is based on a preceding term project, the actual requirements were quite clear at the beginning and the working packages did not change in a significant way over the time. The resulting version of the project plan which is reduced due to space reasons looks as follows:

[1]Both calculated with `sloccount`.


2011
2012
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38
Refactoring towards seams
Extract interface
Object seam support
GNU wrapping function
Shadow function
Preload run-time wrapping
Preprocessor seam support
Mock object support
C++03 support
Toggle mock support
Registration checker
Move test double to ns
Recognise missing operators
Mock function support
Expectations with regex
Analysis multi-threading
Other activities
Setup and initialisation
CUTE wizard integration
AST 2012 paper and conference
Overload article
Bugfixing
Screen casts
Poster and documentation

B.3. Time Schedule

The recommended duration of a master's thesis in the Master of Science in Engineering program is 810 hours of work (27 ECTS * 30 hours). I worked 996.5 hours over the past 38 weeks. This thesis was done part-time during two terms. The distribution over the project weeks can be seen in table B.2 and as a diagram in figure B.1.

Week	Hours	Week	Hours	Week	Hours	Week	Hours
1	25	11	27.5	21	24	31	28
2	24	12	29	22	25	32	29
3	24	13	27.5	23	28	33	27
4	32.5	14	21	24	24	34	26
5	30	15	13	25	27.5	35	26
6	27	16	11	26	34	36	21
7	29	17	25.5	27	28	37	26
8	25	18	26	28	33.5	38	33
9	30	19	24.5	29	25		
10	26.5	20	25.5	30	28		
Total: 996.5							

Table B.2.: *Distribution of the work time over the project weeks.*

The increased time effort results mainly because of the various activities that have been done beside the actual master's thesis like writing the paper for the AST workshop, the article for the Overload magazine, the presentations held at the ICSE (Refactoring workshop and AST) conferences, the submissions of abstracts for further conferences (ECSE, ACCU). Furthermore, the screen casts took much more time (finally about 50h) than originally planned. Note that the reduced amount of work during the project weeks 15 and 16 resulted due to Christmas holidays.

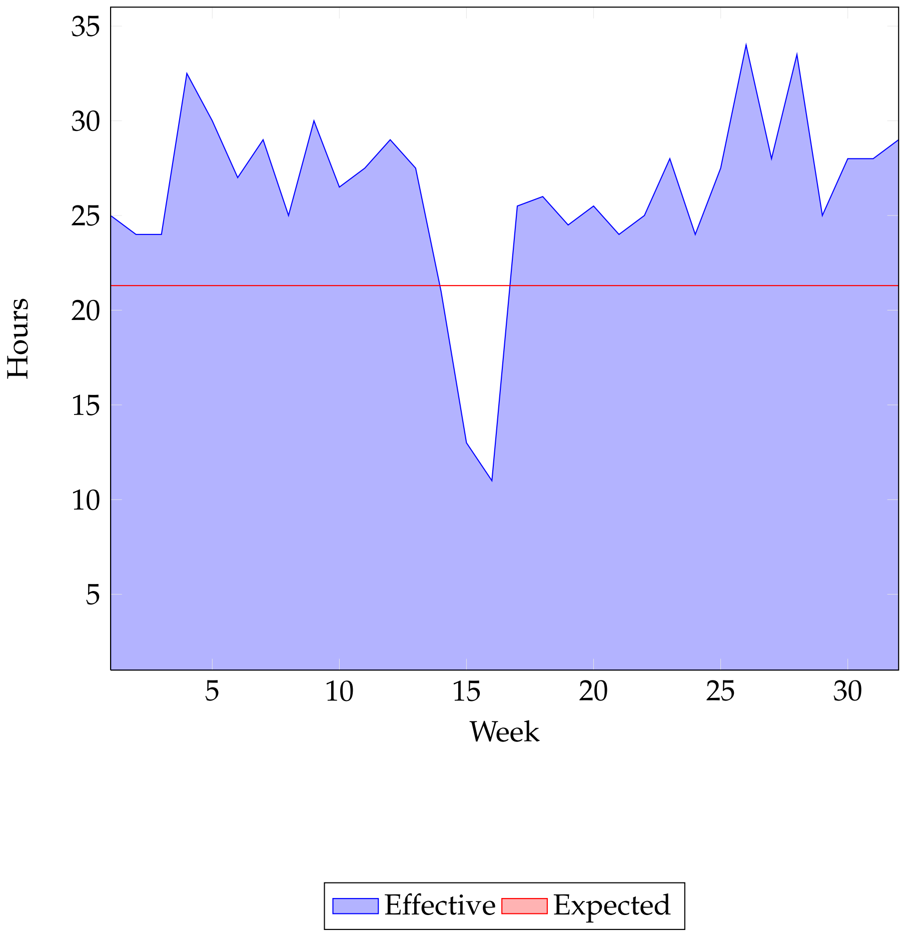


Figure B.1.: *Hours of work per project week. The total amount of 996.5 hours results in an average of 26.25 hours per week.*

C. Continuous Integration Setup

For every development project it is crucial to build and test the software on a regular basis. In this project we used a continuous integration server with several components to guarantee the requested quality. The build process and the execution of our 602 unit tests was controlled by Maven/Tycho and triggered by Jenkins. For source management we used Git. For project management and bug tracking Trac with Apache came into play. This chapter gives an overview about the various steps necessary to setup these components.

C.1. Trac

Trac is a Python based project management and bug tracking tool. In this project it was used as a Wiki, to browse the Git repository and to view the build status of the Jenkins server. Trac runs as an Apache module, therefore we have to install Apache first. Trac can be installed easily as a Python module with the help of Python's installer `pip`:

```
% aptitude install apache2
% aptitude install python-pip
% pip install Trac
```

To be able to run Python applications in the Apache webserver as a module, we have to install `libapache2-mod-python` and enable it:

```
% aptitude install libapache2-mod-python
% a2enmod mod_python
```

The next step is the creation of the Trac project:

```
% mkdir -p /var/lib/trac/mockator
% trac-admin /var/lib/trac/mockator initenv
% chown -R www-data /var/lib/trac/mockator
```

What is left is the installation of the various Trac plug-ins used to browse the Git repository, view the Jenkins build status and to use vim as a front-end for Trac through XML-RPC which can be downloaded from [Rie10], [roa10] and [ea10]. The configuration of Trac can be done in the file `trac.ini` of the project location.

To create a backup of the Trac project, we used the `hotcopy` functionality of `trac-admin` and invoked it with a daily cronjob:

```
% crontab -e
# insert the following in the editor:
# back slashes in the date format field are important: percent
# characters are otherwise interpreted as newlines in a cronjob!
0 1 * * * trac-admin /var/lib/trac/mockator/ hotcopy
          /home/mru/trac-backup/$(date +"\%d\%m\%Y")
```

C.2. Git

We use Git as our source repository tool. The package `git-core` provides everything we need:

```
% aptitude install git-core
```

Then we create a new remote repository:

```
$ mkdir mockator.git && cd mockator.git
$ git init --bare
Initialized empty Git repository in /home/mru/mockator.git
```

To clone the new repository on the client we use this:

```
$ git clone mru@sinv-56033.edu.hsr.ch:mockator.git
```

C.3. Jenkins

Jenkins is used to build our plug-ins, to execute the unit tests and to build the project report and the poster on a daily basis. The installation of Jenkins is easy:

```
$ wget -q -O - \
> http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key | \
> apt-key add -
% vi /etc/apt/sources.list
# insert the following in the editor:
deb http://pkg.jenkins-ci.org/debian binary/
% aptitude update
% aptitude install jenkins
```

The configuration of Jenkins can be done by its convenient web front-end. There, we have defined our build jobs and when and how to run them. To build our plug-ins, Jenkins checks the code out of our Git repository and builds them in its workspace. In order to allow Jenkins to checkout code from our Git repository, we have to define the user name and email address for the repository (run these commands as user `jenkins`):

```
$ git config user.email "mrueegg@hsr.ch"
$ git config user.name "mru"
```

To run our plug-in unit tests on the headless CI environment, we have to install a X virtual screen buffer. We decided to use Xvnc for this. Beside Xvnc, we also need a X11 window manager (e. g., metacity). The following command shows how both are installed:

```
% aptitude install vnc4server metacity
```

Afterwards, we have to set a password for Xvnc. This is done with `vncpasswd` which has to be executed as user `jenkins`:

```
% su jenkins -c vncpasswd
```

Then we have to install the Xvnc Jenkins plugin and configure it for the build, which is both done in the Jenkins web frontend. We have used the following Jenkins plug-ins:

- *Task Scanner*: Shows statistics about `TODO`'s in our code base.
- *Xvnc*: Allows us to run our UI based unit tests in a headless environment with the help of the Unix VNC server Xvnc.
- *ChuckNorris*: To have an inspiring hero to look up to.
- *Various static code analysis tools*: Findbugs, PMD, Checkstyle

Jenkins can be restarted like this:

```
% /etc/init.d/jenkins restart
```

C.4. Apache

Apache is used as our web server which redirects the incoming requests to Jenkins and Trac. The Apache configuration file for Trac `/etc/apache2/sites-enabled/trac` is shown next:

```
 <VirtualHost *:80>
   ServerAdmin mrueegg@hsr.ch
   ServerName sinv-56033.edu.hsr.ch
   DocumentRoot /var/www
   ErrorLog /var/log/apache2/error.trac.log
   CustomLog /var/log/apache2/access.trac.log combined
   DirectoryIndex index.html index.htm

   <Location />
     SetHandler mod_python
     PythonInterpreter main_interpreter
     PythonHandler trac.web.modpython_frontend
     PythonOption TracEnv /var/lib/trac/mockator
     PythonOption TracUriRoot /
     PythonOption PYTHON_EGG_CACHE /tmp
     DirectoryIndex index.html index.htm
   </Location>

   <Location /login>
     AuthType Basic
     AuthName "trac"
     AuthUserFile /etc/apache2/trac.passwd
     Require valid-user
   </Location>

   <Location /mockator>
     SetHandler file
   </Location>
</VirtualHost>
```

The location `/mockator` is used to store the generated project report, the poster, the screen casts and the p2 repository for the Mockator features and plug-ins. After configuration changes we should force Apache to reload its configuration files:

```
% /etc/init.d/apache2 reload
```

A new user for Trac can be created as follows:

```
% cd /etc/apache2
% htpasswd trac.passwd <user>
enter password for <user>
```

D. Building With Maven And Tycho

In Eclipse, the smallest modularization unit is the plug-in. When we talk about plug-ins we also need to discuss what OSGi bundles are because the two terms are almost interchangeable and often both are used to refer to the same entity. An OSGi bundle is basically just a `.jar` file with meta information. This meta information is stored in a file `MANIFEST.MF` in a directory called `META-INF`. Part of this file is the definition of the run-time dependencies a certain bundle has to other bundles. On the other hand, Eclipse projects have compile-time dependencies when it comes to the building process. The big question for all Eclipse build systems is how they manage these dependencies.

To continuously build our plug-ins and to provide a p2 update site where users can install these from, we use Tycho which got a lot of attention recently and is now also an Eclipse project in the incubation phase. Tycho is a bunch of Maven plug-ins to build OSGi bundles with Maven. It uses a so called *Manifest-first* approach [Tho10]. This means that the OSGi manifest is the primary descriptor of the build process. Tycho uses Maven (it needs at least version 3.0) as its underlying build system. The central configuration file of Maven is the POM file. With Tycho, we have a POM file for every project, but this only describes Maven-related concepts. The dependencies are entirely taken from the OSGi manifest. Additionally, Tycho also considers the file `build.properties` where build related issues are defined.

Before we get to an example, we need to define two more important concepts. *Repositories* are used to get the artefacts necessary for the build which can exist locally on the hard disk or somewhere on a web server. The important benefit we gain when using Tycho is that we can access p2 repositories in the same way as we can with Maven based repositories. The second concept is the *target platform* which is the Eclipse installation where the developed plug-ins will be deployed. It is important that the dependencies of the bundles can be resolved through this target platform.

As an example, we will now create the POM for a plug-in with Tycho:

```
$ mvn org.codehaus.tycho:maven-tycho-plugin:generate-poms \
> -DgroupId=ch.hsr.ifs.mockator.plugin \
> -Dtycho.targetPlatform=/opt/eclipse
```

As a result of this command, Tycho generates a `pom.xml` for our project and for the parent project in the current directory. We now have a look at the one generated for our plug-in project (because of space reasons we omitted the namespace and schema header definitions):

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
  <parent>
    <artifactId>ch.hsr.ifs.mockator</artifactId>
    <groupId>ch.hsr.ifs.mockator</groupId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>ch.hsr.ifs.mockator.plugin</artifactId>
  <packaging>eclipse-plugin</packaging>
</project>
```

As we can see, the packaging type of the plug-in is `eclipse-plugin`. Tycho currently supports the following set of packaging types for the various Eclipse artefacts [Son10]:

- *eclipse-plugin*
- *eclipse-test-plugin*
- *eclipse-feature*
- *eclipse-repository*
- *eclipse-update-site*[1]
- *eclipse-application*

To build and deploy our plug-in, we can use the following command:

```
$ mvn -e clean install
```

In this manner we created POM files for all our plug-ins, test fragment projects, features and update sites. The following build script is used to make the update site available on our project server:

```
#!/bin/sh
THIS=$(readlink -f $0)
UPDATE_SITE_DIR="‘dirname $THIS‘/ch.hsr.ifs.mockator.updatesite"
BUNDLE_ROOT="$UPDATE_SITE_DIR/target/repository"
PUBLISH_DESTINATION=/var/www/mockator
cp -r $BUNDLE_ROOT/* $PUBLISH_DESTINATION/repo
cp $UPDATE_SITE_DIR/category.xml $PUBLISH_DESTINATION/repo
```

[1]This packaging type is deprecated and will soon be removed. Instead, *eclipse-repository* should be used.

E. Eclipse Project Setup

Mockator consists of various Eclipse project types which are summarized in figure E.1.

Eclipse project name	Description
ch.hsr.ifs.mockator.feature	Feature for the Mockator plug-in
ch.hsr.ifs.mockator.help	Mockator's Eclipse help with its tutorials
ch.hsr.ifs.mockator.lib	C++ mock object library including its unit tests
ch.hsr.ifs.mockator.plugin	The Mockator Eclipse plug-in
ch.hsr.ifs.mockator.swtbottest	SWTBot UI tests for Mockator
ch.hsr.ifs.mockator.tests	Eclipse plug-in tests
ch.hsr.ifs.mockator.updatesite	Eclipse update site

Table E.1.: *Overview of the various Eclipse projects of Mockator.*

As mentioned in chapter D, Eclipse's most basic modularization unit is the *plug-in*. A *feature* groups together a set of plug-ins in a way that a user can conveniently load, manage and brand these as a single unit [CR09]. Although we could deliver our features as a ZIP file to our users, Eclipse provides a much more elegant solution to solve the distribution problem: *update sites*. An update site is a special website which includes all our plug-ins and features. The content of an update site is described by a manifest file. Clients can use an update site to download our features and keep them up to date.

There are several possibilities in Eclipse projects where to put test code [Pau07]:

- *Place tests and source code into a single plug-in:* This basically corresponds to the project structure Maven uses as default. Beside being a simple solution, this variant has the advantage that all our code is loaded by the same classloader which has the effect that we can access non-public methods from our tests. But there are serious disadvantages with this approach: we would have dependencies in our plug-ins to JUnit and additional used mocking libraries and would need to include the tests into our deployable plug-ins if we want to run them as part of our building process.
- *Use separate plug-ins for source and tests:* The disadvantage of this method is that we have no access anymore to non-public methods of our classes under test. Instead,

we need to export all the packages where we use classes from our tests which is something we typically want to minimise to keep our API as small as possible[1].

- *Place tests in fragment project:* Basically, when a fragment is loaded, its capabilities are merged with those of its host plug-in. Fragments extend the functionality of another plug-in which is most typically used for language packs, maintenance updates and platform-specific implementation classes [CR09]. A fragment and its host plug-in are loaded by the same classloader which allows us to call non-public methods. We also do not need to export packages anymore for our classes under test.

Because of its various advantages we have decided to use an Eclipse fragment project for our tests. Note that we have experienced an issue related to fragment projects when we upgraded from Tycho version 0.11 to 0.14. This is because the handling of optional dependencies has changed in a way that indirect optional bundle dependencies are now always always. This issue came up because we have an optional dependency to CUTE in our fragment host plug-in. With the upgrade, our unit test fragment project did not have access to the CUTE bundle anymore. The solution was to add the dependencies to CUTE in the Manifest file of the unit test fragment project as required bundles.

[1]There is a workaround for this issue: We could define the packages of classes we want to test from as friends of our test plug-in. But this has other disadvantages like the pollution of our manifest file and the fact that we do not want too many friends [Pau07], something which also applies to C++.

F. Build And Execution of Mock Object Library Tests

For our header-only mock object library, we implemented unit tests in CUTE to verify its correct behaviour. To build our unit tests, we used the following Makefile:

```
CC = g++
CXXFLAGS = -Wall -Wextra -Werror -ansi -pedantic
LDFLAGS = -lboost_regex

ifeq ($(USE_STD11), 1)
  CXXFLAGS += -std=c++0x
endif

all: mockator_tests

mockator_tests: mockator_tests.o
  $(CC) $(LDFLAGS) -o $@ $^

mockator_tests.o: mockator_tests.cpp mockator/mockator.h
  $(CC) $(CXXFLAGS) -isystem cute/ -c -o $@ $<

clean:
  rm -f mockator_tests mockator_tests.o test_results*.xml

.PHONY: all clean
```

Note that we use the highest warning levels of GCC including treating warnings as errors and use the flag `pedantic` to issue all the warnings demanded by strict ISO C++. Because we only want to have errors reported in our mock object library, we use the flag `isystem` to treat the CUTE header files as system directories which has the result that warnings in there are not yielded by the compiler.

We run our unit tests with our Jenkins build. In order to have test failures reported in a similar way as we are used to from running JUnit based tests, we applied the JUnit XML output emitter developed by Boris Zweimüller [Zwe12] and specified the path to the generated XML file in the Jenkins configuration.

G. Dependency Analysis With JDepend

There are numerous metric tools to test the quality of Java package designs. One of the most prominent is JDepend [CC09]. JDepend uses the approach described by Martin in [Mar02]. The higher-level goal of his recommendations concerning package design is that low-abstraction (i. e., concrete) packages should depend on high-abstraction packages, therefore inverting package dependencies [Mar02]. This allows to reuse the high-abstraction packages independently, which leads to the conclusion that dependencies on them are considered to be stable and therefore desirable.

To enforce these goals, we have decided to run these metrics as part of our unit tests, leading to a red bar and a broken build if we violate these. JDepend can be used by putting its JAR file into the classpath of the unit tests. Its configuration and start procedure can be packed in the setup code of the unit test. The following code snippet additionally shows how we can use package filters. These are necessary because we only want to measure our code:

```java
@Before protected void setUp() throws IOException {
  PackageFilter filter = new PackageFilter();
  filter.addPackage("java.*");
  filter.addPackage("javax.*");
  filter.addPackage("org.*");
  jdepend = new JDepend(filter);
  jdepend.addDirectory("classes");
  jdepend.analyze();
}
```

Packages that are not expected to change can be specified with a volatility value. This value is either `0` or `1`, whereas `0` means the package is not going to change at all. This leads to the conclusion that it will fall directly on the main sequence, a term also described in Martin's book meaning that the package is optimal with respect to its abstractness and stability. The following code demonstrates the configuration of the volatility value:

```java
JavaPackage stable = new JavaPackage("x.y.z");
stable.setVolatility(0);
```

With the following code we are able to test the conformance of the distance from the mentioned main sequence. Every package will be tested with a specified threshold if it conforms to this important metric by the following test [CC09]:

```java
@Test public void conformanceOfDistanceFromMainSequence() {
  double ideal = 0.0;
  double tolerance = 0.5;
  for (JavaPackage p : packages) {
    assertEquals(ideal, p.distance(), tolerance);
  }
}
```

Another important property of a good package design is that there are no dependency cycles. We can achieve this with the following piece of code that reports all packages that are part of a cycle, finally leading to a broken test if a cycle has been detected:

```java
@Test public void noCyclicPackageDependencies() throws Exception {
  StringBuilder s = new StringBuilder();
  for (JavaPackage p : packages) {
    List<JavaPackage> cycles = new ArrayList<JavaPackage>();
    if (p.collectCycle(cycles)) {
      s.append(String.format("\n$ %s $ [", p.getName()));
      for (int i = 0; i < cycles.size(); i++) {
        if (i > 0) s.append(" -> ");
        s.append(cycles.get(i).getName());
      }
      s.append("]");
    }
  }
  assertEquals("Cycles:" + s.toString(), false, jdepend.containsCycles());
}
```

To enforce these important checks, we decided to run them as part of every commit with the help of Git's commit hooks. As the checks are reasonable fast (about 1-3 seconds), it does not stop us from committing often. The following shell script runs these checks when it is placed in the directory `.git/hooks` and allows commits only if these have been successful:

```sh
#!/bin/sh
SRC_DIR=source/plugins/ch.hsr.ifs.mockator
PLUGIN_DIR=$SRC_DIR/ch.hsr.ifs.mockator.plugin
TESTS_DIR=$SRC_DIR/ch.hsr.ifs.mockator.tests
METRICS_TESTS=ch.hsr.ifs.mockator.plugin.metrics.MetricsTests
CLASS_PATH=$TESTS_DIR/lib/junit4.jar:$TESTS_DIR/lib/hamcrest-core.jar:
    $TESTS_DIR/bin:$TESTS_DIR/lib/jdepend-2.9.1.jar:$PLUGIN_DIR/bin
if [ -f ${TESTS_DIR}/bin/${METRICS_TESTS//.//}.class ]; then
  echo "Running metric tests..."
  java -cp $CLASS_PATH org.junit.runner.JUnitCore $METRICS_TESTS
  exit $?
fi
exit 0
```

H. Style Checking Tools For Writing Good English

Although we use style checkers and static analysis tools for our code since a long time, we have never used tools to check the style of our documentation. We started to think about this also because of the article and the paper we have written. There is active research ongoing in this area like the development of *TextLint*, a rule-based tool to check for common style problems in natural language [PRR12]. They provide various checkers which are based on common style guides like "The Elements of Style" [JW00] and "On Writing Well" [Zin01]. Although we have analysed this tool shortly, we would rather like to use smaller tools that are dedicated to a certain job instead of a huge Smalltalk package.

There are open source Unix tools available to help us here which exist since decades. Two of these tools are called `diction` and `style` [Haa05]. Diction checks for weasel words[1] and misused phrases in a text. It also uses the principles of "The Elements of Style" to mark style problems and is able to detect duplicate words. We have the following two phony targets in the Makefile to build our documentation that make use of it:

```
doubled-words:
  @echo "doubled words: "
  @diction *.tex | egrep -n -i --color 'Double word'

misused-phrases:
  @diction -L en_GB --beginner --suggest *.tex | less
```

To give an example, we execute `diction` on a sentence that contains a weasel word and makes the error of using "it's" instead of "its".

```
$ diction --beginner --suggest
We are mostly convinced that Mockator helps us in writing better unit
tests with it's seam implementation.
(stdin):1: We are [mostly -> (avoid)] convinced that Mockator helps
us in writing better unit tests with [it's -> = "it is" or "its"?]
seam implementation.
```

[1] A weasel word is a vague or ambiguous claim.

With `style` we can analyse our writing style and uses several metrics to yield numbers which can be compared against recommended thresholds. We use the following phony target to find difficult to understand sentences in our report:

```
diffcult-sentences:
  @echo "Difficult sentences (ARI > 18):"
  @style -L en_GB --print-ari 18 *.tex | less
```

Note that we use the *Automated Readability Index* (ARI) here with `style` which produces an approximation of the grade level in the United States of America that is necessary to understand a given text [Sen67].

Finally, we give a few statistics about our project report with the following figures:

```
$ style report.tex
readability grades:
        Kincaid: 13.0
        ARI: 15.1
        Coleman-Liau: 13.1
        Flesch Index: 46.2/100
        Fog Index: 15.6
        Lix: 53.5 = school year 10
        SMOG-Grading: 13.2
sentence info:
        174293 characters
        34051 words, average length 5.12 characters = 1.60 syllables
        1367 sentences, average length 24.9 words
        52% (713) short sentences (at most 20 words)
        17% (236) long sentences (at least 35 words)
        386 paragraphs, average length 3.5 sentences
        0% (2) questions
        53% (732) passive sentences
```

I. How to Make Screencasts With Open Source Command-Line Tools

At the University of Applied Sciences Rapperswil, every student writing a bachelor or master's thesis is required to provide a screen cast about its outcome. We started with using the open source tool `recordMyDesktop` but soon found it limiting. We therefore engineered a script that creates a video by using `ffmpeg` and records audio with `arecord`. We will discuss the most important parts in this chapter.

We record the audio and the video separately which has certain benefits. Among others, it allows us to modify the two parts independently and merge them together later:

```
arecord --quiet -f dat -D hw:0,0 > screencast.wav &
ffmpeg -y -f x11grab -s ${GRAB_W}x${GRAB_H} -r ${FPS} -i
:0.0+${X_OFFSET},${Y_OFFSET} -aspect ${ASPECT} -vcodec libx264
screencast.avi &
```

To reduce noise in the recorded audio, we use `sox` — the Swiss army knife of audio manipulation — which is first used to take a sample of the audio file and to create a noise profile. We then execute it on the recorded audio and it filters the noise:

```
sox screencast.wav noiseaud.wav trim 0 1
sox noiseaud.wav -n noiseprof noise.prof
sox -v 2.0 screencast.wav screencast-clean.wav noisered noise.prof 0.3
```

Finally, we create a mp3 file with `lame` and merge audio and video together:

```
lame -h -m j --vbr-new -b 128 screencast.wav -o screencast.mp3
mencoder -ovc copy -oac copy -audiofile screencast.mp3 \
    screencast.avi -o screencast-final.avi
```

`mencoder` is a very powerful tool. Beside merging, we also used it to split and shorten our screen casts:

```
# Shorten video by only taking the first minute of it:
mencoder -endpos 60 -ovc copy -oac copy video.avi-o shortened.avi
# Split videos after a minute:
mencoder -endpos 60 -ovc copy -oac copy video.avi -o first_half.avi
mencoder -ss 60 -ovc copy -oac copy video.avi -o second_half.avi
```

J. Bugs Raised for CDT, CUTE and CloneWar

During this thesis, we updated the following bugs in Eclipse CDT's Bugzilla:

- *Bug 372807 "Deleting a folder corrupts the Path and symbols"*: We recognised this bug when we tried to remove shadowed functions. Because we create all shadowed functions in a separate source folder, the easiest way of removing them is to just delete the folder. Unfortunately, this does not work properly because after the deletion, include paths in the project settings are messed up.
- *Bug 256763 "Can't build Shared Library project on x86_64 without -fPIC parameter for compiler"*: For our run-time function interception seam we create a shared library project where the intercepting function is defined. Because shared libraries need the `-fPIC` parameter be set when compiled on x86_64 Linux systems with GCC, we do it in Mockator as CDT does not have this as a default which we propose as a solution.

The following feature requests and bug reports have been raised for CUTE:

- *Feature 66 "Feedback of test result when views are minimised"*: While creating our screen casts and also during demos we have noticed that when we minimised the view bar including the CUTE test results view we do not get any visual feedback about our test results. We therefore think it would be great if either the CUTE test results view is shown when a test run is finished or the icon of the view would emphasise if the test run was successful or not (similar to the JUnit plug-in).
- *Bug 65 "Test registration checker does not work together with Boost assign"*: Because we initially used Boost assign to add our tests for the mock object library to the CUTE suite, we experienced that CUTE's test registration checker (which assures that all test functions are properly registered in a test suite) marked all our functions as not being registered. An example is given here:

```
void testGameFourWins {  /* ... */ }
void runSuite() {
  cute::suite s;
  s += CUTE(testGameFourWins);
  cute::ide_listener lis;
  cute::makeRunner(lis)(s, "The Suite");
}
```

- *Bug 64 "Registered test function checker fails if CUTE suite is not fully qualified"*: The following two test suite registrations both yield a CodAn marker for an unregistered test function although the test function is properly registered in both of them (probably due to a not fully qualified CUTE suite):

```
void testGameFourWins {  /* ... */ }
void runSuite1() {
  using cute::suite;
  suite s;
  s.push_back(CUTE(testGameFourWins));
}
void runSuite2() {
  using namespace cute;
  suite s;
  s.push_back(CUTE(testGameFourWins));
}
```

The following bug has been raised for the extract template parameter refactoring (codename *CloneWar* [Thr10]):

- *Bug 63 "Renaming of template parameter breaks generation of default template parameter argument"*: Renaming of the template parameter breaks the generation of a default template argument under these circumstances: 1. Select an instance member variable and choose "Extract Template Parameter". 2. Change type name in the field "type after extraction" 3. Click "Next>" => default template argument is not generated. This is not the case when we click on the table row of the template parameter after the renaming and before we click on "Next>".

K. Content From Term Project

In order to have a more consistent and understandable documentation, we decided to include some sections of the master term project documentation [Rü11] into this report:

- Section 1.3 with some minor adjustment to explain the need for mock objects and to show how competitors (i. e., Google Mock) work.
- Parts of section 2.2.1 to explain the advantages and disadvantages of static polymorphism with templates.
- Parts of section 2.2.4 to formalise the discussion about missing member functions with the help of concepts.
- Parts of section 3.1.1 to explain the inner workings of the C++ mock object library.
- The explanation of the used Eclipse extension points in section 5.2 has been updated with the newly used ones.
- The possible enhancement to recognise more concept kinds discussed in section 6.2.4 as this was already an open point in the term project.
- The problem of passing `this` in the SUT explained in section 6.3.2 as this was already recognised as a bug in the term project.
- Section B.1 and chapter D with the project environment and the build infrastructure including a few minor updates.
- Chapter G was updated with the newly developed git hook.

List of Figures

Nomenclature

ABI	Application Binary Interface
AST	Abstract Syntax Tree
BRE	Basic Regular Expressions
CDT	Eclipse C/C++ Development Tooling
CodAn	Code Analysis, Eclipse CDT's code analysis framework
CRC	Class / Responsibility / Collaboration cards
DOC	Depended-On Component
DSL	Domain Specific Language
EBNF	Extended Backus–Naur Form
ERE	Extended Regular Expressions
GCC	GNU Compiler Collection
LTK	The Language Toolkit: An API for automated refactorings in Eclipse-based IDEs
OSGi	Open Services Gateway initiative framework
p2	Stands for provisioning platform and is the engine used to install plug-ins and manage dependencies in Eclipse
POD	Plain Old Data
RAII	Resource Acquisition Is Initialisation
RTTI	Run-Time Type Information
SUT	System Under Test
TDD	Test-Driven Development

Bibliography

[Ale01] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley, 2nd edition, 2010.

[CC09] Inc. Clarkware Consulting. JDepend. World Wide Web, `http://www.clarkware.com/software/JDepend.html`, 2009. Accessed: 12.06.2011.

[Com12] AST Steering Committee. Ast 2012 7th international workshop on automation of software test. World Wide Web, `http://www.cse.chalmers.se/~rjmh/AST2012/site`, 2012.

[Cor12] Intel Corporation. Intel wiki - concurrent vector. World Wide Web, `http://threadingbuildingblocks.org/wiki/index.php?title=Concurrent_Vector`, 2012.

[CR09] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*, volume Third Edition. Addison-Wesley Professional, 2009.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM COMPUTING SURVEYS*, 17(4):471–522, 1985.

[Dev11] Valgrind Developers. The Design and Implementation of Valgrind. World Wide Web, `http://valgrind.org/docs/manual/mc-tech-docs.html`, 2011.

[DH96] Karel Driesen and Urs Hoelzle. The Direct Cost of Virtual Function Calls in C++. *SIGPLAN Not.*, 31:306–323, oct 1996.

[ea10] Alec Thomas et al. Trac xml-rpc plugin. World Wide Web, `http://trac-hacks.org/wiki/XmlRpcPlugin`, 2010. Accessed: 06.06.2011.

[ea11a] Beman Dawes et al. Boost c++ libraries. World Wide Web, `http://www.boost.org`, 2011. Accessed: 25.05.2011.

[ea11b] Douglas Gregor et al. Conceptgcc. World Wide Web, `http://www.generic-programming.org/software/ConceptGCC`, 2011. Accessed: 06.06.2011.

[ea12] CodeSourcery et. al. Itanium C++ ABI. World Wide Web, `http://sourcery.mentor.com/public/cxx-abi/abi.html#mangling`, 2012.

[Fea04] Michael C. Feathers. *Working Effectively With Legacy Code*. Prentice Hall PTR, 2004.

[Fel12] Lukas Felber. Includator. World Wide Web, `http://www.includator.com`, 2012.

[Fog12] Agner Fog. Calling conventions for different C++ compiles and operating systems. World Wide Web, `http://www.agner.org/optimize/calling_conventions.pdf`, 2012.

[Fow99] Martin Fowler. *Refactoring*. Addison-Wesley, 1999.

[Fow11] Martin Fowler. Mocks aren't stubs. World Wide Web, `http://martinfowler.com/articles/mocksArentStubs.html`, 2011. Accessed: 03.03.2011.

[FPM$^+$04] Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes, and Thoughtworks Uk. Mock roles, not objects. In *In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246. ACM Press, 2004.

[GJP06] Douglas Gregor, Jaakko Järvi, and Gary Powell. Variadic templates (revision 3). World Wide Web, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2080.pdf`, 2006. Accessed: 25.05.2011.

[Goo12] Google. Google mock. World Wide Web, `http://code.google.com/p/googlemock`, 2012.

[GS04] Brian J. Gough and Richard M. Stallman. *An Introduction to GCC*. Network Theory Ltd., 2004.

[GS06] Douglas Gregor and Bjarne Stroustrup. Concepts (revision 1). World Wide Web, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2081.pdf`, 2006. Accessed: 06.06.2011.

[Haa05] Michael Haardt. Style and diction. World Wide Web, `http://www.gnu.org/software/diction/diction.html`, 2005. Accessed: 06.06.2012.

[IG12] The IEEE and The Open Group. dlsym. World Wide Web, `http://pubs.opengroup.org/onlinepubs/009695399/functions/dlsym.html`, 2012. Accessed: 29.05.2012.

[ISO03] ISO/IEC. Programming Language C++, Standard. World Wide Web, October 2003. Accessed: 31.05.2012.

[ISO11] ISO/IEC. Working Draft, Standard for Programming Language C++. World Wide Web, `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf`, February 2011. Accessed: 31.05.2012.

[Jos99] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 1999.

[JW00] William Strunk Jr. and E.B. White. *The Elements of Style*. Longman, 2000.

[Kef12] Fivos Kefallonitis. Name mangling demystified. World Wide Web, `http://www.int0x80.gr/papers/name_mangling.pdf`, 2012.

[LB] Mike Looijmans and Peter Bindels. Hippo mocks. World Wide Web, `http://www.assembla.com/wiki/show/hippomocks`. Accessed: 28.02.2011.

[Li12] Cheng Li. gccfilter. World Wide Web, `http://www.mixtion.org/gccfilter/`, 2012.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, Inc, 2002.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall International, 2008.

[Mau11] Jens Maurer. Bug 1773 - __cplusplus defined to 1, should be 199711l. World Wide Web, `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=1773`, 2011. Accessed: 04.10.2011.

[MB11] Daniel S. Myers and Adam L. Bazinet. Intercepting Arbitrary Functions on Windows, UNIX, and Macintosh OS X Platforms. World Wide Web, `http://www.cs.umd.edu/Library/TRs/CS-TR-4585/CS-TR-4585.pdf`, 2011. Accessed: 29.09.2011.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[Mey00] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2000.

[Mey05] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*. Addison-Wesley, May 2005.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. *Endo-testing: unit testing with mock objects*, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Ott11] Thorsten Ottosen. Boost.assign documentation. World Wide Web, `http://www.boost.org/libs/assign/`, 2011. Accessed: 04.10.2011.

[Pau07] Patrick Paulin. Testing Plug-ins with Fragments. World Wide Web, `http://www.modumind.com/2007/06/20/unit-testing-plug-ins-with-fragments/`, 2007.

[Pou11] Trevor Pounds. Mockitopp - simple mocking for c++. World Wide Web, `http://code.google.com/p/mockitopp/`, 2011. Accessed: 28.02.201.

[PRR12] Fabrizio Perin, Lukas Renggli, and Jorge Ressia. Linguistic style checking with program checking tools. *Computer Languages, Systems and Structures*, 38(1):61 – 72, 2012. <ce:title>SMALLTALKS 2010</ce:title>.

[Rad11] Mark Radford. C++ interface classes - an introduction. World Wide Web, http://accu.org/index.php/journals/233, 2011. Accessed: 30.09.2011.

[Res12] Microsoft Research. Detours. World Wide Web, http://research.microsoft.com/en-us/projects/detours, 2012.

[Rie10] Herbert Valerio Riedel. Git plug-in for trac. World Wide Web, http://trac-hacks.org/wiki/GitPlugin, 2010. Accessed: 06.06.2011.

[roa10] roadrunner. Hudson trac integration plug-in. World Wide Web, http://trac-hacks.org/wiki/HudsonTracPlugin, 2010. Accessed: 06.06.2011.

[Rü11] Michael Rüegg. Mockator — An Eclipse CDT Plug-in for Mock Objects. Master term project, University of Applied Sciences Rapperswil, HSR, 2011.

[Rü12] Michael Rüegg. Refactoring Towards Seams in C++. *overload*, pages 28–32, apr 2012.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards*. Addison-Wesley, November 2004.

[Sen67] E.A. Senter, R.J.; Smith. Automatic readability index. *Wright-Patterson Air Force Base*, 1967.

[Som11] Peter Sommerlad. CUTE - C++ Unit Testing Easier. World Wide Web, http://cute-test.com, 2011.

[Son10] Sonatype. Tycho. World Wide Web, http://tycho.sonatype.org, 2010.

[SR05] Bjarne Stroustrup and Gabriel Dos Reis. Specifying c++ concepts. World Wide Web, http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1886.pdf, 2005. Accessed: 06.06.2011.

[Sut00] Herb Sutter. *Exceptional C++*. Addison-Wesley, 2000.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Tho10] Karsten Thoms. Schaffe, schaffe, Eclipse baue! *Eclipse Magazin*, 2010:54–59, 2010.

[Thr10] Yves Thrier. Clonewar - Refactoring Transformation in CDT: Extract Template Parameter. Master's thesis, University of Applied Sciences Rapperswil, 2010.

[unk11a] unknown. AMOP - Automatic Mock Object for C++. World Wide Web, http://code.google.com/p/amop/, 2011. Accessed: 28.02.2011.

[Unk11b] Unknown. Boost.preprocessor. World Wide Web, http://www.boost.org/doc/libs/1_47_0/libs/preprocessor/doc/index.html, 2011. Accessed: 22.09.2011.

[unk12] unknown. clang: a c language family frontend for llvm. World Wide Web, http://clang.llvm.org, 2012. Accessed: 06.06.2012.

[VJ02] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates - The Complete Guide*. Addison-Wesley, 2002.

[WBM03] Rebecca Wirfs-Brock and Alan McKean. *Object Design - Roles, Responsibilities, and Collaborations*. Pearson Educationl, 2003.

[Wik11] Wikipedia. Mock object. World Wide Web, http://en.wikipedia.org/wiki/Mock_object, 2011. Accessed: 28.02.2011.

[Wil12] Anthony Williams. *C++ Concurrency In Action*. Manning, 2012.

[Zin01] Willoam K. Zinsser. *On Writing Well*. Collins, 2001.

[Zwe12] Boris Zweimüller. Cute plug-in - feature 40: Xml output. World Wide Web, http://cute-test.com/issues/40, 2012.