Mockator User Manual

Michael Rüegg

18.03.2014

0.1 Introduction

Breaking dependencies is an important task in refactoring legacy code and putting this code under tests. Feathers' seams (Feathers (2004)) 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 a plug-in for the Eclipse C/C++ development tooling project. Mockator 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.

0.2 Seams

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 which are discussed by using Mockator in the following sections.

0.2.1 Object Seams

Object seams are probably the most common seam type. To start with an example, consider the following code where the class `GameFourWins` has a hard coded dependency to `Die`:

```
// Die.h
struct Die {
  int roll() const ;
};
// Die.cpp
int Die::roll() const {
  return rand() % 6 + 1;
}
// GameFourWins.h
```

```cpp
struct GameFourWins {
  void play(std::ostream& os);
private:
  Die die;
};
// GameFourWins.cpp
void GameFourWins::play(std::ostream& os = std::cout) {
  if (die.roll() == 4) {
    os << "You won!" << std::endl;
  } else {
    os << "You lost!" << std::endl;
  }
}
```

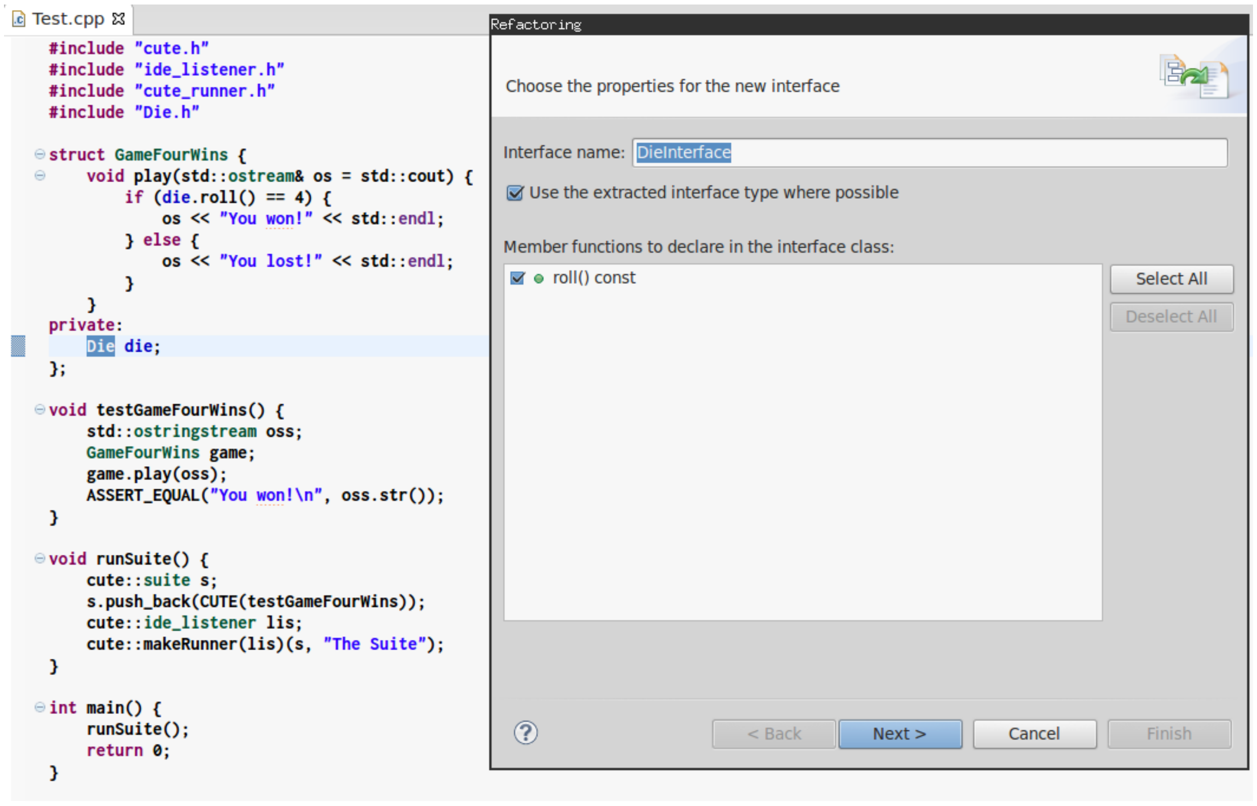
According to Feathers definition, the call to `play` is not a seam because it is missing an enabling point. We cannot alter the behaviour of the member function `play` without changing its function body because the used member variable `die` is based on the concrete class `Die`. Furthermore, we cannot subclass `GameFourWins` and override `play` because `play` is monomorphic (not virtual).

This fixed dependency also makes `GameFourWins` hard to test in isolation because `Die` uses C's standard library pseudo-random number generator function `rand`. Although `rand` is a deterministic function since 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. The injected class inherits from a base class, thus enabling subtype polymorphism.

To achieve an object seam, the first step is to extract an interface. For this, Mockator provides a new refactoring called *Extract Interface*. Select the class to extract an interface from (e.g., `Die`) and click "Refactor->Extract Interface" (see figure 1).

As a result, a new interface with pure virtual member functions is created:

```cpp
struct IDie {
  virtual ~IDie() {}
  virtual int roll() const =0;
};
struct Die : IDie {
  int roll() const {
    return rand() % 6 + 1;
  }
};
struct GameFourWins {
  GameFourWins(IDie& die) : die(die) {}
  void play(std::ostream& os=std::cout) {
    // as before
  }
private:
  IDie& die;
};
```



```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include "Die.h"

struct GameFourWins {
    void play(std::ostream& os = std::cout) {
        if (die.roll() == 4) {
            os << "You won!" << std::endl;
        } else {
            os << "You lost!" << std::endl;
        }
    }
private:
    Die die;
};

void testGameFourWins() {
    std::ostringstream oss;
    GameFourWins game;
    game.play(oss);
    ASSERT_EQUAL("You won!\n", oss.str());
}

void runSuite() {
    cute::suite s;
    s.push_back(CUTE(testGameFourWins));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main() {
    runSuite();
    return 0;
}
```

Choose the properties for the new interface

Interface name: DieInterface

✓ Use the extracted interface type where possible

Member functions to declare in the interface class:

✓ roll() const

Figure 1: Extract interface refactoring for applying object seams.

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`.

0.2.2 Compile Seams

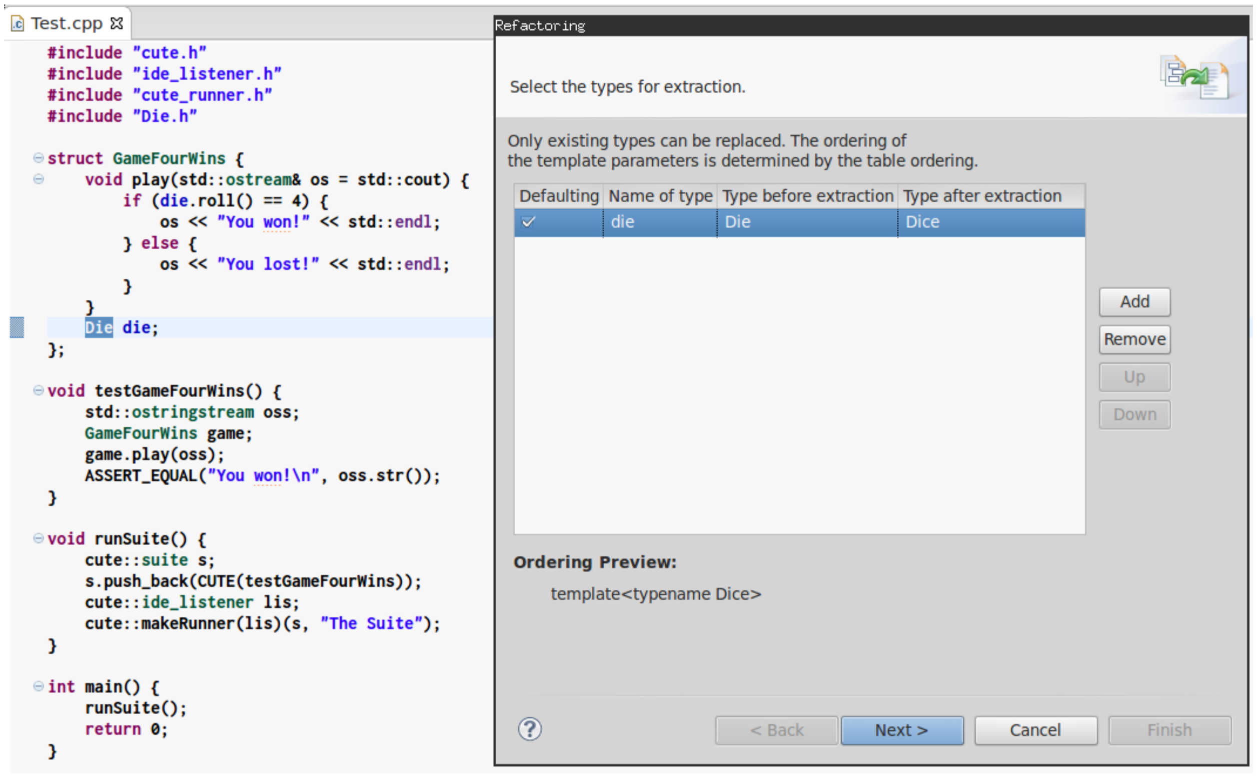
Although object seams are the classic way of injecting dependencies, we think there is often a better solution to achieve the same goals. C++ has a tool for this job providing static polymorphism: template parameters. With template parameters, we can inject dependencies at compile-time. We therefore call this seam compile seam.

The use of static polymorphism with template parameters has several advantages over object seams with subtype polymorphism. It does not incur the run-time overhead of calling virtual member functions that can be unacceptable for certain systems. 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 (providing compile-time duck typing). This can ease the burden of an otherwise wide interface that one might need to implement in case of an object seam.

The essential step for this seam type is the application of a the refactoring *Extract Template Parameter* through the menu "Refactor->Extract Template" (see figure 2) which comes with the Cute plug-in (Sommerlad (2011)).

The result of this refactoring can be seen here:

```
template <typename Dice=Die>
struct GameFourWinsT {
```


Test.cpp

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include "Die.h"

struct GameFourWins {
    void play(std::ostream& os = std::cout) {
        if (die.roll() == 4) {
            os << "You won!" << std::endl;
        } else {
            os << "You lost!" << std::endl;
        }
    }
    Die die;
};

void testGameFourWins() {
    std::ostringstream oss;
    GameFourWins game;
    game.play(oss);
    ASSERT_EQUAL("You won!\n", oss.str());
}

void runSuite() {
    cute::suite s;
    s.push_back(CUTE(testGameFourWins));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main() {
    runSuite();
    return 0;
}
```

Refactoring

Select the types for extraction.

Only existing types can be replaced. The ordering of the template parameters is determined by the table ordering.

Defaulting	Name of type	Type before extraction	Type after extraction
✓	die	Die	Dice

Ordering Preview:

template<typename Dice>

Figure 2: Extract template refactoring for applying compile seams.

```
  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;
```

The enabling point of this seam is the place where the template class `GameFourWinsT` is instantiated.

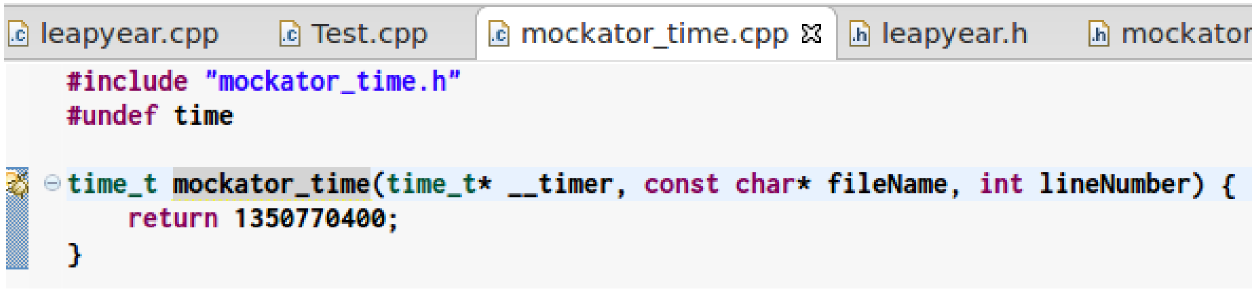
0.2.3 Preprocessor Seams

C and C++ offer another possibility to alter the behaviour of code without touching it in that place using the preprocessor. 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 next where we trace calls to C's `time` function with the help of Mockator:

```
/* leapyear.h */
```

```
#ifndef TODAYSTIME_H_
#define TODAYSTIME_H_

bool isLeapYear();

#endif /* TODAY_H_ */

/* leapyear.cpp */

#include "leapyear.h"
#include <ctime>

unsigned int thisYear() {
    time_t now = time(0);
    tm* z = localtime(&now);
    return z->tm_year + 1900;
}

bool isLeapYear() {
    unsigned int year = thisYear();
    if ((year % 400) == 0) {
        return true;
    }
    if ((year % 100) == 0) {
        return false;
    }
    if ((year % 4) == 0) {
        return true;
    }
    return false;
}

/* Test.cpp */

#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include "leapyear.h"

void testLeapYear() {
    //1350770400 2012 is a leap year; date -d "Oct 21 2012" +%s
    ASSERT(isLeapYear());
}

void runSuite(){
    cute::suite s;
    s.push_back(CUTE(testLeapYear));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main(){
    runSuite();
    return 0;
}
```

To do this, select the function call for `time` and execute the source action `trace function call` via the context menu "Source->Trace Function Call". Now you can toggle the activation of this feature within the resolution of an Eclipse quickfix marker (see figure 3).



```
#include "mockator_time.h"
#undef time

time_t mockator_time(time_t* __timer, const char* fileName, int lineNumber) {
    return 1350770400;
}
```

Figure 3: Toggle activation of the traced function call.

The enabling point for this seam are the options of our compiler to choose between the real and our tracing implementation. We use the option `-include` of the GNU compiler here to include the header file `malloc.h` into every translation unit. With `#undef` we are still able to call the original implementation of `malloc`.

0.2.4 Link Seams

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 gives us another kind of seam called link seam. We show three kinds of link seams here:

- Shadowing functions through linking order (override functions in libraries with new definitions in object files)
- Wrapping functions with GNU's linker option -wrap (GNU Linux only)
- Run-time function interception with the preload functionality of the dynamic linker for shared libraries (GNU Linux and Mac OS X only)

0.2.4.1 Shadow Functions In this type of link seam we make use of the linking order. 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 yield a duplicate definition error when considering the object file. Mockator helps in shadowing functions and generates code and the necessary CDT build options to support this kind of link seam:

```
// shadow_roll.cpp
#include "Die.h"
int Die::roll() const {
  return 4;
}
// test.cpp
void testGameFourWins () {
  // ...
}
```

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

The order given to the linker is exactly as we need it to prefer the symbol in the object file since 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. This type of link seam 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 function call.

0.2.4.2 Wrap Functions The GNU linker ld provides a lesser-known feature which helps us to call the original 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."

As an example, we compile GameFourWins.cpp. 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 GameFourWins.o | grep roll
U _ZNK3Die4rollEv
```

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. Mockator helps in applying this seam type by creating the following code and the corresponding build options in Eclipse CDT:

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

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

To prevent the compiler from mangling the mangled name again, we need to define it 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.

Alas, this feature is only available with the GNU tool chain on Linux. GCC for Mac OS X does not offer the linker flag -wrap. A further constraint is that it does not work with inline functions but this is the case with all link seams presented here. Additionally, when the function to be wrapped is part of a shared library, we cannot use this option.

0.2.4.3 Intercept Functions If we have to intercept functions from shared libraries, we can use this kind of link seam. It is based on the fact that it is possible to alter the run-time linking behaviour of the 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 that the loader ld.so interprets.

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. As an example, consider the following code and the CDT build options which is generated by Mockator to intercept function calls to Die::roll:

```
#include <dlfcn.h>
int rand(void) {
  typedef int (*funPtr)(void);
  static funPtr origFun = 0;
  if (!origFun) {
    void* tmpPtr = dlsym(RTLD_NEXT, "rand");
    origFun = reinterpret_cast<funPtr>(tmpPtr);
  }
  int notNeededHere = origFun();
  return 3;
}
```

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

The advantage of this solution compared to the first two link seams is that it does not require re-linking. It is solely based on altering the behaviour of ld.so. A disadvantage is that this mechanism is unreliable with member functions, because the member function pointer is not expected to have the same size as a void pointer.

0.3 Using Test Doubles

0.3.1 Creating Mock Objects

0.3.2 Move Test Double to Namespace

0.3.3 Converting Fake to Mock Objects

0.3.4 Toggle Mock Support

0.3.5 Registration Consistency

0.3.6 Mock Functions

0.3.7 Using Regular Expressions

0.4 References

Feathers, Michael C. 2004. *Working Effectively with Legacy Code*. Prentice Hall PTR.

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