Refactoring towards seams in C++ - How to make your legacy code testable

Michael Rüegg

Institute for Software, University of Applied Sciences Rapperswil

March 5, 2013
Content

▶ Motivation / Contribution
▶ What is a seam?
▶ Object Seam
▶ Compile Seam
▶ Preprocessor Seam
▶ Link Seams
▶ Future Work & Conclusions
Motivation

- Perils of dependencies in software
- **Triggers** for changing existing code (Feathers): adding new functionality, fixing a bug, applying refactorings and code optimisations
- **Legacy code**: code without unit tests
- The Legacy code **dilemma**
- We do **not** want to **change** the code **inline**
- There is hope: **Seams** - but they are hard and cumbersome to create by hand! \(\Rightarrow\) **refactorings and IDE support necessary**
Our Contribution

- **Refactorings** and **toolchain support** for achieving seams in C++
- **Eclipse plug-in** for the Eclipse C/C++ Development Tooling Project (CDT)
- **C++ mock object library** (header-only) with Eclipse support
What is a Seam?

- Term introduced by Michael Feathers in *Working Effectively With Legacy Code*:

  “A place in our code base where we can alter behaviour without being forced to edit it in that place.”

- **Inject dependencies from outside** to improve the design of existing code and to enhance testability

- Every seam has an **enabling point**: the place where we can choose between one behaviour or another

- Different kinds of seam types in C++: **object, compile, preprocessor** and **link seams**
// Real object 'Die' makes it hard to test the system under test (SUT) 'GameFourWins' in isolation

```cpp
struct Die {
    int roll() const { return rand() % 6 + 1; }
};

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;
};
```
Object Seam

- Based on **subtype / inclusion polymorphism**
- Used refactoring: **Extract interface** (Fowler)
- **Enabling point**: DI via ctor / member function:

```cpp
struct IDie { // extracted interface
    virtual ~IDie() {} 
    virtual int roll() const =0;
};

// GameFourWins(IDie& die) : die(die) {}

void testGameFourWins() {
    struct : IDie {
        int roll() const { return 4; }
    } fake;
    GameFourWins game(fake); // enabling point
    std::ostringstream oss;
    game.play(oss);
    ASSERT_EQUAL("You won!", oss.str());
}
```
Extract interface refactoring

▶ “Refactorings always yield legal programs that perform operations equivalent to before the refactoring.” - William F. Opdyke

Functionality preservation

```cpp
struct AlwaysSixDie : Die {
    int roll() const { return 6; }
};
struct Croupier {
    void sixWinsJackpot(Die const & die) {
        if (die.roll() == 6) { /* jackpot */ }
    }
};
// NEW:
struct Die : IDie { /* .. */ };
void quiz() {
    Croupier croupier; AlwaysSixDie die;
    croupier.sixWinsJackpot(die); // huuh?
}
```
Object Seam

- **Tradeoffs:**
  - Run-time overhead of calling virtual member functions
  - Tight coupling
  - Enhanced complexity and fragility

- **Demo:**
Compile Seam

- Based on static / parametric polymorphism
- Compile-time duck typing:
  
  ```cpp
  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` (known as the implicit interface)
- Used refactoring: Extract template parameter
- Enabling point: Template instantiation
template <typename Dice=Die> // compile seam
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;
};

// do not break existing code
typedef GameFourWinsT<> GameFourWins;
void testGameFourWins() {
    struct FakeDie {
        int roll() const {
            return 4;
        }
    };
    GameFourWinsT<FakeDie> game; // enabling point
    std::ostringstream oss;
    game.play(oss);
    ASSERT_EQUAL("You won!\n", oss.str());
}
C++11 Excursion: Local Classes

- With C++98/03: local classes had no linkage $\Rightarrow$ could not be used as template arguments
- With C++11: awkward restriction has been removed
- Still no first-class citizens:
  - Declarations in local classes can only use type names, static and external variables, functions and enums from their enclosing scope $\Rightarrow$ Access to automatic variables prohibited
  - Not allowed to have static members
  - Cannot have template members
Compile Seam

- **Advantages**: No run-time overhead, compile-time duck typing (no interface burden)
- **Disadvantages**: Increased compile-times, (sometimes) reduced clarity
- **Demo**:

![CDT Project](image)
Preprocessor Seam

- Use of the C preprocessor **CPP**
- Useful for tracing function calls with debug information

```
// myrand.h
#ifndef MYRAND_H_
#define MYRAND_H_
int my_rand(const char* fileName, int lineNr);
#define rand() my_rand(__FILE__, __LINE__)
#endif

// myrand.cpp
#include "myrand.h"
#undef rand
int my_rand(const char* fileName, int lineNr){
    return 3;
}
```

- **Enabling point**: compiler options to include header file or to define macros (GCC -include option)
Preprocessor Seam

- **Tradeoffs**: Many!
  - Preprocessor lacks type safety causing hard to track bugs
  - Recompilations necessary
  - Redefinition of member functions is not possible
  - ...

- **Demo**:

  ![CDT Project](image)
Link Seams

- Tweak build scripts by using your **linker’s options**
- Three kinds with GNU toolchain:
  1. Shadowing functions through **linker order**
  2. Wrapping functions with **GNU’s wrap option**
  3. Run-time function interception of **ld**

- **Enabling point**: linker options
- **Constraints**: All link seams do not work with **inline functions**
Shadow Function

- Based on **linking order**: linker takes symbols from object files instead the ones defined in libraries
- Place the object files **before** the library in the linker call
- Allows us to shadow the real implementation:

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

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

- Mac OS X GNU linker needs the shadowed function to be defined as a weak symbol:

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

- **Tradeoff**: No possibility to call the original function
- **Demo:**

![CDT Project](image)
Wrap Function

- Based on GNU’s linker option wrap
- Possibility to call the original / wrapped function
- Useful to intercept function calls (kind of monkey patching)
- Example:

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

- Extract of ld’s manpage: “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.”
Wrap Function

- Watch our for C++ mangled names!
- Example with Itanium’s ABI:

```
$ gcc -c GameFourWins.cpp -o GameFourWins.o
$ nm --undefined-only GameFourWins.o | \ 
  > grep roll
U_ZNK3Die4rollEv

extern "C" {
  extern int __real__ZNK3Die4rollEv();
  int __wrap__ZNK3Die4rollEv() {
    return 4;
  }
}
```

```
$ g++ -Xlinker -wrap=_ZNK3Die4rollEv \ 
  > -o Test test.o GameFourWins.o Die.o
```
Wrap Function

▶ **Tradeoffs:**

▶ Only works with GNU’s linker on Linux (Mac OS X not supported)

▶ Does not work with functions in shared libraries

▶ **Demo:**

![CDT Project](image-url)
Run-time function interception

- **Alter the run-time linking behaviour** of the loader *ld.so*
- Usage of the environment variable *LD_PRELOAD* the loader *ld.so* interprets
- Manpage of *ld.so*: “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.”
- Instruct the loader to prefer our code instead of libs in *LD_LIBRARY_PATH*
- Used by many C/C++ programs (e.g., Valgrind)
Run-time function interception

- **Not done yet:** would not allow us to call the original function
- **Solution:** use `dlsym` to lookup original function by name
- Takes a handle of a dynamic library (e.g., by `dlopen`)
- Use pseudo-handle `RTLD_NEXT`: next occurrence of symbol

```c
#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;
}
```

$ g++ -shared -ldl -fPIC foo.cpp -o libFoo.so
$ LD_PRELOAD=path/to/libRand.so executable
Run-time function interception

- Mac OS X users: Note that environment variables have different names!
- LD_PRELOAD is called DYLD_INSERT_LIBRARIES
- Additionally needs the environment variable DYLD_FORCE_FLAT_NAMESPACE to be set
- Demo:
Run-time function interception

- **Advantages:**
  - Allows wrapping of functions in shared libraries
  - No recompilation/relinking necessary
  - Source code must not be available
  - Linux and Mac OS X supported

- **Disadvantages**
  - Not reliable with member functions
  - Not possible to intercept `dlsym` itself
  - Ignored if the executable is a setuid or setgid binary
  - Not possible to intercept internal function calls in libraries
Seams - What have we achieved?

- With object and compile seams:
  - No fixed / hard-coded dependencies anymore
  - Dependencies are injected instead
  - Improved design and enhanced testability

- Preprocessor seam is primarily a debugging aid

- Link seams help us in replacing or intercepting calls to libraries
Future Work

- Support **other toolchains** beside GCC (Clang, MS)
- Gain more **practical experience** (e.g., embedded software industry)
- Support **other programming languages**
Conclusions

- Seams help in making legacy code testable
- Our refactorings and toolchain support makes them easier to apply
- Next step is often the use of test doubles
- Our plug-in contains a mock object library with code generation for fake and mock objects
Thank you!

http://mockator.com