Acta Cybernetica **00** (0000) 1–28.

Unit Testing in C++ with Compiler Instrumentation and Friends

Gábor Márton*, Zoltán Porkoláb†

Abstract

In C++, test code is often interwoven with the unit we want to test. During the test development process we often have to modify the public interface of a class to replace existing dependencies; e.g. a supplementary setter or constructor function is added for dependency injection. In many cases, extra template parameters are used for the same purpose. All existing solutions have serious detrimental effects on the code structure and sometimes on the run-time performance as well. In this paper, we overview existing dependency replacement techniques of C++ and we evaluate their advantages and disadvantages. We introduce our non-intrusive, compiler instrumentation based testing approach that does not have such disadvantages. All non-intrusive testing methods (including our new method) require access to an object's internal state in order to setup a test. Thus, to complement our new solution, we also present different approaches to conveniently access private members in C++. To evaluate these techniques, we created a proof-of-concept implementation which is publicly available for further testing.

Keywords: C++, Unit Test, Instrumentation, Friend, Access Control

1 Introduction

Testing is essential in modern software development [1, 13, 5, 18] to improve the quality of a system and reduce the cost of maintenance. There are different layers of testing from unit tests to stability, functional and integration tests. In this paper we focus on unit testing, which is the most language-specific method. However, some of the findings we discuss might be extended to different/higher level tests as well.

During a unit test we check the behaviour of the unit under test. If we do functional programming and work with pure functions alone (where all functions are free from side-effects) then testing is easy because we just provide a specific input and assert for the desired output. However, in the object-oriented paradigm,

*Department of Programming Languages and Compilers, Eötvös Loránd University, E-mail: martongabesz@gmail.com

†E-mail: gsd@elte.hu

we have objects in some kind of state. This means the object is dependent on other objects that represent the internal state. Testing our object using these dependencies may be problematic; e.g. the dependency may represent a database or a network connection, whose behaviour can be hard or expensive to simulate. In order to create independent, resilient and efficient tests [45] in most cases we need to substitute some (or even all) of the dependencies with test doubles. We refer to this substitution of dependencies as *dependency replacement*.

In object-oriented programming languages, the dependency replacement often requires the modification of the original public interface of the unit under test. For instance, new setter or constructor functions have to be added to a class, otherwise dependency replacement would not work. Nevertheless, there are cases where these new functions are not intended to be used in production code. We refer to all those testing approaches which require source code modification as *intrusive* testing. Moreover, in C++, source code modification for testing could result in performance degradation, e.g. introducing a new runtime interface and virtual functions just because of testing might worsen the performance of the production code. Also, in legacy code bases often there are no unit tests. Refactoring such legacy code in order to provide tests is almost impossible because we cannot verify correctness without having unit tests; hence it is a vicious circle. We can break the circle with non-intrusive tests, though all of the existing non-intrusive testing methods for C++ have some drawbacks.

In this paper, we investigate a new, non-intrusive, compiler instrumentation based testing approach that does not have these disadvantages. All non-intrusive testing methods (including our new method) often require access to an object's internal members in order to set up a test. Thus, to complement our new method, we present different approaches to conveniently access private members in C++.

This paper is organized as follows. In Section 2, we describe principles of dependency replacement in object-oriented programming and we discuss the existing techniques of dependency replacement in C++. We show how we can replace dependent C++ functions with compiler instrumentation in Section 3. Then the access of private members is discussed in Section 4. Here, we inspect how we can access members with our alternative approaches without intrusively changing the unit we wish to test. We present how we could enhance the use of friends with our extension idea to friends. After, we overview related work in Section 5. In Section 6, we outline future work and possible directions for better testing experience in C++. Our paper concludes in Section 7.

2 Dependency Replacement in C++

Figure 1 shows a typical object under test, its dependencies and their possible replacements. If A and B are objects and "A depends on B", then we say that A is a *dependant* of B and B is a *dependency* of A. As for dependency replacement the dependant object is referred as the *system under test* (SUT). Sometimes we refer to that as the *unit* under test. In this study, we use the following definitions

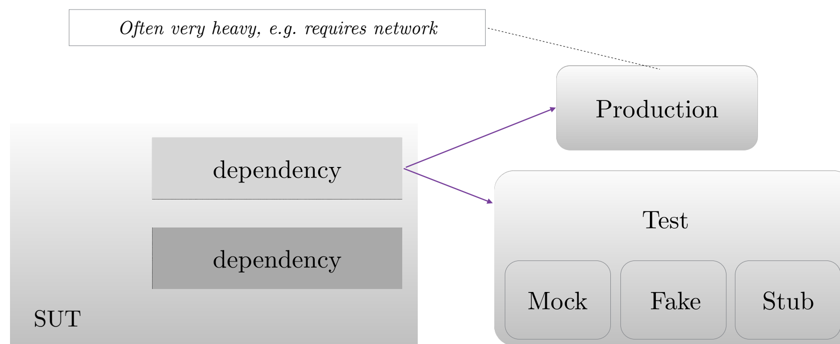

Figure 1: Dependency Replacement

for test doubles: *Fake* classes provide empty definitions of functions in a way that the unit tests can pass. Fakes are the simplest doubles to cut down dependencies. *Stub* classes implement additionally some very basic behaviour, therefore they may be more complex than fakes. We can set up a stub to return with a specific value. *Mock* classes are used to formulate expectations, such as how many times a member function is called with a certain value.

There are several design patterns for dependency replacement like the Factory Method and Abstract Factory [7], the Service Locator [44, 6] and the Dependency Injection (DI) [37, 38, 43].

It is important to emphasize that dependency injection is different from the abstract concept of dependency replacement. Dependency injection (DI) is one realization – amongst many others – of dependency replacement. DI is used mostly in object-oriented languages with runtime reflection, like Java and C#. All of these patterns can be used in the popular, managed languages and in C++ as well. Java and C# provides well documented DI frameworks (like the *Unity Container* in C# [29], and the Spring framework in Java [19]), therefore DI is the widespread method for performing dependency replacement in these managed languages. However, there is no generally accepted DI framework for C++.

Objects in the context of OOP are represented by `class`es in C++. Since C++ is not a strict object oriented language, we must investigate other language constructs like free functions and function templates from the viewpoint of dependency replacement. Generally speaking, a dependant C++ entity (class, function, class template or function template) can have different kinds of dependencies. For instance, it may have a dependency on

- a global object (e.g. via a singleton).
- a global function (via a function call),
- an object via a pointer or reference,
- a type (e.g. via a type template parameter, or the type of a member),

2.1 C++ Seams

A *seam* is an abstract concept introduced by Feathers [4] as an instrument via we can alter behaviour without changing the original unit. Dependency replacement is done via seams in C++. Actually, there are four different kinds of seams in C++ [35, 28]:

1. *Link seam*: Change the definition of a function via some linker specific setup.
2. *Preprocessor seam*: With the help of the preprocessor, redefine function names to use an alternative implementation.
3. *Object seam*: Based on inheritance to inject a subclass with an alternative implementation.
4. *Compile seam*: Inject dependencies at compile-time through template parameters.

The *enabling point* of a seam is the place where we can make the decision to use one behaviour or another. Different seams have different enabling points. For example, replacing the constructor argument for the implementation of an interface with a mock implementation when a unit test is set up is an object seam with the constructor as an enabling point.

2.1.1 Link Seams

We can use a link seam e.g. to replace the implementation of a free function or a member function. For instance:

```
// A.hpp
void foo();
// A.cpp
void foo() { ... };
// MockA.cpp
void foo() { ... };
// B.cpp
#include "A.hpp"
void bar() { foo(); ... }
```

On the one hand, when we need to test the `bar()` function then we should link the test executable to the `MockA.o` object file. On the other hand, we should link the production code with `A.o`. Link-time dependency replacement is not possible if the dependency is defined in a static library or in the same translation unit where the SUT is defined. It is also not feasible to use link seams if the dependency is implemented as an inline function [35]. This makes the use of this seam cumbersome or practically impossible when the dependant unit is a template or when the dependency is a template. The enabling point for a link seam is always outside of the program text. This makes the use of link seams quite difficult to identify. On top of all, link-time substitution requires strong support from the build system we are using. Thus, we might have to specialize the building of the tests for each and every unit. This does not scale well and can be really demanding regarding to maintenance.

2.1.2 Preprocessor Seams

Preprocessor seams can be applied to replace the invocation of a global function to an invocation of a test double [27]. Let us consider the following code snippet:

```
void *my_malloc(size_t size) {
  //...
  return malloc(size);
}

void my_free(void *p) {
  //...
  return free(p);
}

#define free my_free
#define malloc my_malloc

void unitUnderTest() {
  int *array = (int *)malloc(4 * sizeof(int));
  // do something with array
  free(array);
}
```

We can replace the standard `malloc()` and `free()` functions with our own implementation. One example usage may be to collect statistics or do sanity checks in `my_malloc` and `my_free` functions. These seams can be applied conveniently in C, but not in C++. As soon as we use namespaces, the preprocessor might generate code which cannot be compiled because of the ambiguous use of names. Hazardous side effects of macros are also well known.

2.1.3 Object Seams

Object seams are realized by introducing a runtime interface. For example, let us consider the following C++ class (note, using a `const` qualifier on the `process()` member function and a RAII lock guard instead of explicit locking would make the code safer, but it would also make our message less visible):

```
class Entity {
public:
  int process(int i) {
    if(m.try_lock()) {
      auto result = std::accumulate(v.begin(),v.end(),i);
      m.unlock();
      return result;
    }
    else { return -1; }
  }
  void add(int i) {
    m.lock();
    v.push_back(i);
    m.unlock();
  }
private:
  mutable std::mutex m;
  std::vector<int> v;
};
```

We would like to test the `Entity::process()` function for both possible return values of `try_lock`. Our objective is to have a test like this:

```
void test() {
  Entity e;
  set_try_lock_fails(e);
  ASSERT_EQUAL(e.process(1), -1);
  set_try_lock_succeeds(e);
  ASSERT_EQUAL(e.process(1), 1);
}
```

We introduced an interface and we can use it to change the behaviour of the mutex object in runtime. For this, the production specific and test specific implementations of the interface need to be provided:

```
struct Mutex {
  virtual void lock() = 0;
  virtual void unlock() = 0;
  virtual bool try_lock() = 0;
  virtual ~Mutex() {}
};

struct RealMutex : Mutex { //used in production code
  void lock() override { m.lock(); }
  void unlock() override { m.unlock(); }
  bool try_lock() override { return m.try_lock(); }
private:
  std::mutex m;
};

struct StubMutex : Mutex { //used in test code
  // defintion of lock() and unlock() as before
  bool try_lock_result = false;
  bool try_lock() override {
    return try_lock_result;
  }
};
```

Now our class should use the interface:

```
class Entity {
public:
  Entity(Mutex& m) : m(m) {}
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  Mutex& m;
  //...
};
```

We can see that the enabling point of this seam is the newly added constructor. The production and the test code might look like this:

```
void productionClient() {
  RealMutex m;
  Entity e(m);
  // some usage of e
}
void testClient() {
  // test setup
  StubMutex m;
  Entity e(m);
  // assertions ...
}
```

There is a severe problem with object seams that is illustrated via this specific example: the mutex object which was exclusively owned by the `Entity` now is moved outside. There is nothing to prevent any caller from reusing (misusing) this mutex. Regarding encapsulation this is fatal. Also, it is not clear who should create/destroy this object and when. The same problems arise if we replace the `Mutex&` with a raw pointer or a smart pointer. Though passing a `unique_ptr` in constructor and getting a reference to it via a getter might be an option, but then we would need to have a getter function for `m` (to set up the test). Generally speaking, the following problems may arise when we replace dependency objects:

- Either we deprive the unit under test from the ownership of the dependency or we use a superfluous getter function.
- We add an otherwise unnecessary constructor or setter function.
- We introduce superfluous pointer semantics via a reference or smart pointer, which is harmful to cache locality, hence it reduces overall performance [41].
- We have to introduce an interface just for testing. This interface has virtual functions. Calling them requires extra pointer indirections and this might result in cache misses and it loses the possibility of inlining, thus it harms the overall performance [3]. Adding an extra interface makes the program more complex, hence the program is harder to understand. Note that in some cases it might be possible to get rid of the additional explicit interface definition with type erasure [31], but the virtual function calls cannot be avoided even in this case.

2.1.4 Compile Seams

Our `Entity` and mutex example would be more natural if we make `Entity` a template and we use the `Mutex` type as a template parameter:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  Mutex m;
  //...
};
```

However, because of testing we need to access the mutex outside of the `Entity` class. Therefore, one simple approach is to define a getter function:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  Mutex& getMutex() { return m; } // Use only from tests
  //...
private:
  Mutex m;
  //...
};
```

The enabling point of this seam is the template parameter itself. Client code may use our `Entity` with the appropriate type parameter:

```
void productionClient() {
  Entity<std::mutex> e;
  // some usage of e
}
void testClient() {
  struct StubMutex {
    //...
    bool try_lock_result = false;
    bool try_lock() {
      return try_lock_result;
    }
  };
  Entity<StubMutex> e;
  auto& m = e.getMutex();
  m.try_lock_result = false;
  ASSERT_EQUAL(e.process(1), -1);
  m.try_lock_result = true;
  ASSERT_EQUAL(e.process(1), 1);
}
```

We do not need to add an additional runtime interface this time, so the test client can use a `StubMutex` which does not have any virtual functions. Of course, the implicit compile-time interface [25, item 41] of `std::mutex` and `StubMutex` must match.

With this approach, we introduced a template parameter just because of testing. The original `Entity` however was perfectly natural to be a simple class, now it became a class template. Also, we added an extra getter function to be able to drive the dependency externally from our class. Needless to say, we increased code complexity and compilation time [2]. There are some methods with which we could decrease compile time, but they would further complicate the source code (use of pimpl [26, item 22]) or the build system setup (using extern templates [12, 14.7.2]).

All four seams have some disadvantages that prevents us from using them or make us reluctant to use them. Link seams do not work with inline functions and require patching the build system. Preprocessor seams are problematic with classes and namespaces. Object seams and compile seams are intrusive and often demand that we widen the public interface. Also, object seams might introduce additional performance penalty in the production code. Therefore, we seek for a new seam which does not have the above-mentioned disadvantages. Our contribution is to show that such a seam can be implemented and applied in software testing.

3 Compiler Instrumentation for Testing

In software development, it is normal that the compiler generates different code for verification purposes. In most integrated development environments (IDEs), a project has two build configurations: namely debug and release. The debug profile usually does not define the `NDEBUG` macro; this results in a runtime binary which does check all the assertions passed to the `assert` macro [11, 7.2 Diagnostics]. Also, in the debug profile, the debug symbols are in general attached to the binary and

the compiler optimizations are turned off. There is a new proposal for C++ that proposes a minimal system for expressing interface requirements as contracts [34]. Contracts are requirements that an operation places on its arguments for successful completion and a set of guarantees that it provides upon successful completion. In this proposal, the authors recommend that compiler implementations offer switches to select a level of contract checking: on, off, pre-condition only, post-condition only. Moreover, the LLVM/Clang compiler itself supplies switches to turn on different verification tools, like the address or the thread sanitizer functionality [17] to report faulty memory accesses and race conditions.

Our idea here is to change the compiled code to provide the ability to replace specified functions with their designated test doubles upon a compiler switch. The test code of our previous example with the `Entity` and `mutex` might look like this:

```
#include "Entity.hpp"

bool try_lock_result;
bool fake_mutex_try_lock(std::mutex* self) { return try_lock_result; }

TEST_F(Fixture, Mutex) {
    SUBSTITUTE(&std::mutex::try_lock, &fake_mutex_try_lock);
    Entity e;
    try_lock_result = false;
    EXPECT_EQ(e.process(1), -1);
    try_lock_result = true;
    EXPECT_EQ(e.process(1), 1);
}
```

With the `SUBSTITUTE` macro, we simply replace the `try_lock` member function of `std::mutex` with a free function named `fake_mutex_try_lock`. The first parameter of this free function holds a pointer to the object on which the original `try_lock` member function has been called.

We have to compile the test binary with the appropriate compiler flag that enables this kind of instrumentation. In our proof-of-concept implementation [21] we use the `-fsanitize=mock` switch with the LLVM/Clang compiler for this purpose. The production code shall be compiled without this flag. Actually, the idea is quite similar to Clang's thread sanitizer. Thread sanitizer is a race condition detector. It instruments each and every memory load and store so it can update some bookkeeping information to aid the detection [39].

By switching on the `-fsanitize=mock` flag we instruct the compiler to replace each and every function call expression with the following pseudo code (let us suppose that the callee is the `foo` function):

```
char* funptr = __fake_hook(&foo);
if (funptr) {
    funptr(args...);
} else {
    foo(args...);
}
```

The return value of the call to `__fake_hook` is a function pointer to the test double which will be called instead of the original function. If that return value is a `nullptr`, then the original function shall not be replaced. If the callee does return anything other than `void`, then the expression is transformed in the following way:

```
char* funptr = __fake_hook(&foo);
auto ret = result_of(&foo);
if (funptr) {
    ret = funptr(args...);
} else {
    ret = foo(args...);
}
return ret;
```

The `__fake_hook` function is defined in a shared library that we have to link to the test binary. Its implementation searches for the function pointer which has been set up by the `SUBSTITUTE` macro. Our realization uses a simple hash map to store the pointers for the original and the test double functions.

It is possible to call the replaced function from the test double. For this purpose, we need to mark the test double with a special attribute so as to avoid its instrumentation. Otherwise, the call site of the original function inside the test double would also be substituted and lead to an infinite recursion.

It should be mentioned that we can replace a callee function in a call expression only if the translation unit which holds that particular call expression has been compiled with the special flag. Quite often, the call expression where we want to substitute the callee is in the source code of the unit under test; i.e it is in the source of our active project. Thus, in most cases we do not have to recompile dependency libraries. We have to recompile dependency libraries only if we wish to substitute a non-inline callee in the library code itself.

Namespaces and member functions are handled similarly to regular free functions. Despite of the fact that member function pointers are really different from pointers to free functions, internally the compiler assigns a unique identifier to all member functions (the address of the function).

A `constexpr` [12, 5.19] function cannot be replaced with this method when it is used in a compile-time expression. However, it can be replaced whenever it is used within a runtime context since this is a runtime instrumentation.

GCC and Clang do not inline any functions when not optimizing unless we specify the `always_inline` attribute for the function [8]. Always inline functions are inlined even when the `-fno-inline` compiler switch is present [9]. Our instrumentation forces the compiler to emit the code of the specific function even when it is explicitly marked with the `always_inline` attribute; hence we can replace inline and always inline functions as well.

3.1 Evaluation

We implemented the above-presented method based on the LLVM/Clang compiler infrastructure. The modified C++ compiler and the corresponding runtime library is publicly available online [22, 21]. We measured the performance of the compilation process itself by compiling its own source code with and without the instrumentation enabled. In both cases, we started a timer to evaluate the wall clock time of the build process and we could not discover any significant difference between the two results. This is quite similar to the compile-time performance results of other sanitizers (e.g. the thread sanitizer).

We found in practice that the runtime performance of the generated binary is slowed down by a factor of 5 to 60. It depends on the number of inlined functions and call expressions. Inlining is an optimization that is turned off by this instrumentation. Our method costs a lookup and a branching for each and every call whether it is substituted or not. Our evaluation demonstrates that most of the reduction in the performance is caused by the lookup, hence it is an important future work to improve it. We shall use a shadow memory [39] (an offset address in the program's virtual memory) to access the test double's address. This way, we could achieve a similar performance to the thread sanitizer runtime performance, which is a slow down by a factor of 5 to 15.

By using our new instrumentation technique, we can write tests without intrusively changing the original unit itself. We do not have to add a new constructor to setup a dependency, and we do not have to add a new getter. Since we do not have to change the original unit at all, the production code should have the same performance as it had before adding tests. Furthermore, this method works with inline and always inline functions. In contrast to link seams, it is really obvious to identify the use of this seam (via the SUBSTITUTE macro in the test code).

4 Access Private Members

Non-intrusive testing requires the ability to access private data as well. Imagine a situation where a replaced member function has to access the internal state of the object to be able to assert on that. For instance, we may wish to check that the `mutex` is unlocked when the `try_lock` member function is called:

```
bool fake_mutex_try_lock(std::mutex* self) {
  EXPECT_EQ(self->locked, true);
}

TEST_F(FooFixture, Mutex) {
    SUBSTITUTE(&std::mutex::try_lock, &fake_mutex_try_lock);
    // ... as before
}
```

There are various existing methods available for accessing private members in C++, but all have certain drawbacks. In this section we overview the existing approaches and we introduce two new procedures that attempt to overcome the difficulties.

4.1 Access via a shared reference or pointer

In this case, the unit that we wish to test has a constructor or a setter function with a reference or a pointer parameter through which we can inject the dependency. An example is:

```
Entity(Mutex& m) : m(m) {}
```

We cannot use a `unique_ptr` this way, since we need to access the dependency from the test as well; however, `unique_ptr` provides exclusive ownership only for the owner.

4.2 Access via getter

As we saw previously, it might be more natural to use a getter when the unit has exclusive ownership:

```
// Use only in tests !
Mutex& getMutex() { return m; }
```

The disadvantage here is that it violates encapsulation, i.e. it exposes an internal member.

4.3 Use preprocessor and getter

To protect the internal member, it is possible to define the getter function only when the unit is built for testing.

```
#ifdef TEST
  Mutex& getMutex() { return m; }
#endif
```

This requires support from the build system, so during the compilation of each translation unit of the test executable this flag needs to be defined. Also, test specific preprocessing is hard coded, which makes it more difficult to see through the unit's overall structure. Though it is quite obvious that the getter is used only for testing, this is actually a benefit.

4.4 Change the access level with the preprocessor

A frequently applied trick is to use the preprocessor to access private members:

```
#define private public
#include "Unit.h"
#undef private
// Test code comes from here
```

Although the standard forbids us from redefining keywords [12, 17.6.4.3.1 Macro names/2], most preprocessors accept this. The obvious drawbacks are easy to see however. All the other classes that are directly or indirectly included from `Unit.h` now expose all their internals. This opens the possibility for errors in the test code via accidentally accessing members of the dependencies, which we shall not know about (violating encapsulation).

Also, this approach does not always work. To see this, consider the following `class`:

```
class X { int a; };
```

The default access specifier is `private` in the case of C++ classes, therefore the define directive has no effect. Still, this can be circumvented with an additional define: `#define class struct`.

What is more, this is undefined behaviour because the C++ standard specifies that the order of allocation of non-static data members with different access control is unspecified [12, 9.2 Class members/13].

4.5 Friend function or class

In C++, friends have right to access private and protected members. With this approach, we declare a concrete test function inside the unit to be a friend:

```
class Entity {
public:
  friend void testClient(Entity& e);
  Entity(std::unique_ptr<Mutex> m) : m(std::move(m)) {}
  int process(int i) { if(m->try_lock()) { ... } else { ... } }
  //...
private:
  std::unique_ptr<Mutex> m;
  //...
};
void testClient(Entity& e) {
  // access e.m here
}
```

We can also declare an in-between class to be a friend; then in the tests we can use the different member functions of the friend class to access the private members.

4.6 Access via explicit instantiation

So far we have considered only intrusive methods to access private members. But there are other approaches which do not require intrusive changes in the unit we wish to test. In this section, we present an interesting non-intrusive technique then in the forthcoming subsection we present our new solution for accessing private members as a generalization of this technique.

We can access outside of the declaring class any private member if we exploit the fact that C++ allows us to pass the address of a private member in explicit instantiation [12, 14.7.2 Explicit instantiation/12]. The standard permits this behaviour, because otherwise specializing traits for private types would not be possible. Besides private members, we can access private `static` variables and functions as well with this technique.

To understand how we can exploit this fact, consider the following class:

```
class A { static int i; };
int A::i = 42;
```

We would like to access the static private variable `i`. Normally, accessing that private variable results in a compiler error:

```
int x = A::i; // Error, i is private
```

Yet, there is an exceptional case, namely when we provide a template argument in an explicit template specialization. Let us assume that we have a class template defined, so we can explicitly specialize that:

```
template struct private_access<&A::i>;
```

The template argument `&A::i` has a compile-time available value and it has the type `int*`. In this context, `&A::i` is a completely valid expression, which has the address of the private variable as the value. We need to expose this address somehow, so we define the class template `private_access` as follows:

```
template <int* PtrValue> struct private_access {
  friend int* get() { return PtrValue; }
};
```

The template parameter of `private_access` is a non-type template parameter, which is a pointer value of type `int*` known at compile-time. We define the `get()` function to return the actual compile-time value of this template parameter. It returns the address of the private static variable, since the template is instantiated with that value as an argument. By defining the `get()` function as a friend it becomes part of the enclosing namespace scope. Even so, its name is not found by normal lookup (qualified or unqualified) [12, 7.3.1.2 Namespace member definitions/3]. Therefore, we need to provide an additional *declaration* outside of the class:

```
int* get();
```

Putting this all together, our code with a usage example is the following:

```
class A { static int i; };
int A::i = 42;

template <int* PtrValue> struct private_access {
  friend int* get() { return PtrValue; }
};

int* get();

template struct private_access<&A::i>;

void usage() {
  int* i = get();
  assert(*i == 42);
}
```

Accessing private, non-static members is quite similar:

```
1  class A { int i = 42; };
2
3  template<int A::* PtrValue> struct private_access {
4      friend int A::* get() { return PtrValue; }
5  };
6
7  int A::* get();
8
9  template struct private_access<&A::i>;
10
11 void usage() {
12     A a;
13     int A::* ip = get();
14     int& i = a.*ip;
15     assert(i == 42);
16 }
```

The only difference is in the type of the template argument, which is now `int A::*`, a pointer to member. Values of this type may be pointers to any `int` data member of the class `A`. Once we get the pointer to the member in line 13, we can bind this pointer to an object, and this way, we get a reference to the data member (in line 14).

4.6.1 Generalized private access

As for our contribution, we generalized the above-presented techniques. We have created a library which automates the generation of the helper constructs to access private data members and to call private member functions (both static and non-static) [20]. So accessing private data members becomes straightforward:

```
class A { int m_i = 3; };

ACCESS_PRIVATE_FIELD(A, int, m_i)

void foo() {
  A a;
  auto &i = access_private::m_i(a);
  assert(i == 3);
}
```

Similarly, calling private functions can be achieved like so:

```
class A {
  int m_f(int p) { return 14 * p; }
};

ACCESS_PRIVATE_FUN(A, int(int), m_f)

void foo() {
  A a;
  int p = 3;
  auto res = call_private::m_f(a, p);
  assert(res == 42);
}
```

We deliberately do not use pointer-to-members in the public interface of this macro library. We think that their use is just an implementation detail that we do not wish to expose to the user.

As a first design decision, we place all components of this library into an unnamed namespace to prevent multiple definition linker errors. For instance, we want the following 3 files (`a.hpp`, `x.cpp`, `y.cpp`) to be linkable into an executable file:

```
// a.hpp
class A { int m_i = 3; };

// x.cpp
#include "A.hpp"
#include "access_private.hpp"
ACCESS_PRIVATE_FIELD(A, int, m_i)

// y.cpp
#include "A.hpp"
#include "access_private.hpp"
ACCESS_PRIVATE_FIELD(A, int, m_i)
int main() { return 0; }
```

Then we commence with the generic definition of `private_access`. We use the nested namespace `private_access_detail` as a safeguard, because we wish to avoid name clashing as the user code might have additional names defined in unnamed namespace:

```
namespace {
  namespace private_access_detail {
    template <typename PtrType, PtrType PtrValue, typename TagType>
    struct private_access {
      friend PtrType get(TagType) { return PtrValue; }
    };
  } // namespace private_access_detail
} // namespace
```

By introducing the `PtrType` type template parameter, we generalize the type of the pointer we wish to use. This might be `int*` or `int A::*` if we take our examples from the previous section. We also bring in the `TagType` type template parameter, which we use to define different instances of the `get()` function. This is achieved implicitly by instantiating the `private_access` class template with different concrete tag types.

Next, we define some helper macros for concatenation:

```
#define PRIVATE_ACCESS_DETAIL_CONCATENATE_IMPL(x, y) x##y
#define PRIVATE_ACCESS_DETAIL_CONCATENATE(x, y)                    \
  PRIVATE_ACCESS_DETAIL_CONCATENATE_IMPL(x, y)
```

We use the `PRIVATE_ACCESS_DETAIL` prefix for all the implementation macros that are supposed to be hidden from the clients of this macro library.

Afterwards, we introduce a macro which contains all those things that are common in the implementation of accessing a static or a non-static member:

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name, \
2                                               PtrTypeKind)           \
3    namespace {                                                       \
4      namespace private_access_detail {                               \
5        struct Tag {};                                                \
6        /* Explicit instantiation */                                  \
7        template struct private_access<decltype(&Class::Name),        \
8                                       &Class::Name, Tag>;            \
9        /* Define the PtrType alias */                                \
10       using PRIVATE_ACCESS_DETAIL_CONCATENATE(Alias_, Tag) = Type;   \
11       using PRIVATE_ACCESS_DETAIL_CONCATENATE(PtrType_, Tag) =       \
12           PRIVATE_ACCESS_DETAIL_CONCATENATE(Alias_,                  \
13                                             Tag) PtrTypeKind;        \
14       /* declare the get() function */                              \
15       PRIVATE_ACCESS_DETAIL_CONCATENATE(PtrType_, Tag) get(Tag);     \
16     }                                                               \
17   }
```

The macro parameter `Tag` is the name of the tag class we want to define and we also use it as a suffix for the name of the type aliases. `Class` denotes the qualified or unqualified name of the class we wish to provide access to. `Type` is the type of the member variable. The parameter `PtrTypeKind` describes what kind of pointer are we dealing with, namely a simple pointer or a pointer-to-member. For instance, it may have the strings `*` or `A::*`. First, we define the tag type (line 5), then comes the explicit instantiation with the type and address of the member and with the recently defined tag type (line 7-8).

Then, we define a type alias (with `PtrType_` prefix) for the concrete type of the pointer (line 9-13). Basically, this alias is formed from the concatenation of the `Type` and `PtrTypeKind` parameters. For example, in the case of a pointer-to-member, the canonical type of the type alias might be `int A::*`. The twist here is

that we need to add two type aliases, because pointer-to-member-functions cannot be expressed generically with one alias, e.g.:

```
using PtrType1 = int(int) *; // ERROR
using Alias = int(int);
using PtrType2 = Alias *; // OK
```

Next, we declare the `get()` function to make it available for finding by normal name lookup (line 15).

Following this, we define the specific macro for non-static member fields.

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FIELD(Tag, Class, Type, \
2                                                     Name)            \
3    PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name,       \
4                                         Class::*)                     \
5    namespace {                                                        \
6      namespace access_private {                                       \
7        Type &Name(Class &&t) {                                        \
8          return t.*get(private_access_detail::Tag{});                 \
9        }                                                              \
10       Type &Name(Class &t) {                                         \
11         return t.*get(private_access_detail::Tag{});                 \
12       }                                                              \
13       using PRIVATE_ACCESS_DETAIL_CONCATENATE(X, Tag) = Type;        \
14       using PRIVATE_ACCESS_DETAIL_CONCATENATE(Y, Tag) =              \
15           const PRIVATE_ACCESS_DETAIL_CONCATENATE(X, Tag);           \
16       PRIVATE_ACCESS_DETAIL_CONCATENATE(Y, Tag) &                    \
17           Name(const Class &t) {                                     \
18         return t.*get(private_access_detail::Tag{});                 \
19       }                                                              \
20     }                                                                \
21   }
```

The macro parameters `Tag`, `Class`, `Type` and `Name` have the exact same meanings as before. In line 3, we call the previously described macro to generate all the generic code we need. We pass "Class::*" as a macro argument, since we are dealing with non-static members. If we were dealing with static members, then the argument would be "*". Then, we define two overloaded functions in the enclosing `access_private` namespace with the name which is equal to the name of the member we are exposing, e.g. "m_i" (line 7-12). These overloads are for those cases where the class instance is bound to an rvalue reference or to a non-const lvalue reference. We bind the result of `get()` function to the object of the class; and then we return with a reference to the resulting member. Later, (in lines 13-19) we add another overload that handles the cases where the object is bound to a const lvalue reference. In this case we would like to preserve the constness of the object, therefore we should return with a const reference to the member. So, we create a type alias for this const reference type (lines 13-15). Here, once again we need to use two type aliases because we would like to avoid warnings that arise from duplicated const qualifiers. If we used just one alias, then we would get this warning if the `Type` macro parameter already contains a const qualifier. After defining the type alias, we use this in the definition of the third overload (lines 16-19).

The implementation of accessing static fields is very similar to the implementation of accessing non-static members.

The realization of calling private member functions, however, requires an explanation:

```
1  #define PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FUN(Tag, Class, Type,   \
2                                                    Name)             \
3    PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE(Tag, Class, Type, Name,       \
4                                         Class::*)                     \
5    namespace {                                                        \
6      namespace call_private {                                         \
7        template <typename Obj,                                        \
8                  std::enable_if_t<std::is_same<                       \
9                      std::remove_reference_t<Obj>, Class>::value> * = \
10                     nullptr,                                         \
11                 typename... Args>                                    \
12       auto Name(Obj &&o, Args &&... args) -> decltype(               \
13           (std::forward<Obj>(o).*get(private_access_detail::Tag{}))( \
14               std::forward<Args>(args)...)) {                        \
15         return (std::forward<Obj>(o).*                               \
16                 get(private_access_detail::Tag{}))(                  \
17             std::forward<Args>(args)...);                            \
18       }                                                              \
19     }                                                                \
20   }
```

Here, we again call the common macro that does the explicit instantiation (lines 3-4). Then we perfect forward both the object and the parameters of the private function we wish to call (lines 7-17). We bind the pointer-to-member-function (result of the `get()` function) to the perfect forwarded object and then we call the resulting member function with the forwarded arguments (lines 15-17). We use the same expression's type as the trailing return type in the header of the function (lines 12-14). (Note that in C++14 there is no need to specify the trailing return type.) We also restrict the set of function template instantiations that can participate in the overload resolution with the `enable_if`. The goal here is to exclude a template function when the type of the object is different from the type of the `Class` parameter. By doing this, we get a more compact error message if we misuse the library for some reason. Otherwise we would get error messages originating from the body of the function template.

The implementation of calling static member functions is very similar to the implementation of calling non-statics, but we do not need the `enable_if` there, since we do not have an object in that case.

Somehow we need to generate unique tag types, so for this we use the built-in `__COUNTER__` macro which returns an integer and is incremented by the preprocessor each time it is referenced. `__COUNTER__` is not a standard macro, but it is available on most mainstream compilers (GCC, Clang, MSVC).

```
#define PRIVATE_ACCESS_DETAIL_UNIQUE_TAG                               \
  PRIVATE_ACCESS_DETAIL_CONCATENATE(PrivateAccessTag, __COUNTER__)
```

The macro `PRIVATE_ACCESS_DETAIL_UNIQUE_TAG()` will generate a unique name with the prefix `PrivateAccessTag`. Finally, we can define the main macros of the library with the help of the unique tag generator:

```
#define ACCESS_PRIVATE_FIELD(Class, Type, Name)                        \
  PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FIELD(                          \
      PRIVATE_ACCESS_DETAIL_UNIQUE_TAG, Class, Type, Name)
```

```
#define ACCESS_PRIVATE_FUN(Class, Type, Name)                        \
  PRIVATE_ACCESS_DETAIL_ACCESS_PRIVATE_FUN(                          \
      PRIVATE_ACCESS_DETAIL_UNIQUE_TAG, Class, Type, Name)
```

During the compilation of one translation unit, each invocation of these macros generates different tag types. Since these tag types are defined in an unnamed namespace, we will not have any linkage errors of duplicate symbols when linking multiple translation units together.

Now we have seen how we can access private member fields and how we can call private member functions, regardless of whether if they are static or not. However, this library has some limitations. We cannot access private types, because the only valid context of using that private type is inside the explicit instantiation. We cannot call private constructors nor destructors. This is because a pointer to member cannot bind to a constructor (since we do not have the object unless the constructor is called). Nor can a pointer to member bind to a destructor because there is no valid expression in C++ to grab the address of a destructor. We have a link time error in the case of in-class declared `const static` variables (without an out-of-class definition). This is because we would take the address of that variable, and if that is not defined (i.e the compiler does a compile-time insert of the const value), we would be trying to dereference an undefined symbol. Owing to all of these limitations we were motivated to come up with a more sophisticated solution.

Note that the Java language has a built in support to achieve something similar. With `setAccessible` we can indicate that the reflected object should suppress access checking when it is used [30].

4.7 Out of class friend

As we saw above, private access via explicit instantiation does not work for all kinds of private entities. So, our other contribution is to explore the idea of a new lingual element with which we would be able to access all kinds of private members. In our non-intrusive approach, we define a function or a class as `friend` out of the befriending class:

```
template <typename Mutex>
class Entity {
public:
  int process(int i) { if(m.try_lock()) {} else {} }
  //...
private:
  Mutex m;
  //...
};

friend for(Entity<StubMutex>) void test_try_lock_fails() {
  Entity<StubMutex> e;
  auto& m = e.m; // access the private member
  // set up try_lock result value to false and do the assertions ...
}
```

Based on the LLVM/Clang compiler (version 3.6.0) [17], we created a proof-of-concept implementation for out-of-class friends and it is now available online (see

[23]). The goal of this implementation is to demonstrate that the idea is indeed feasible, though it is not our objective to provide a full featured perfect realization. Therefore, we add some restrictions to the functionality and we do not implement proper error handling.

To ease the implementation, we use C++ attributes [12, 7.6 Attributes] instead of modifying the existing grammar. More specifically, we use the GCC `__attribute__` syntax because the standard `[[attribute]]` syntax implementation was not complete in the Clang version we used. By using attributes, we skip the problem of parsing and we can focus on the new semantic actions. So, the above definition of `test_try_lock_fails` with attributes is the following:

```
__attribute__((friend(Entity<StubMutex>)))
void test_try_lock_fails() {
  //...
}
```

However, prior to this definition we need to explicitly instantiate the `Entity` class template.

```
template class Entity<StubMutex>;
```

This is required because the attribute's associated semantic action attempts to access all the details of its type parameter (`Entity<StubMutex>`). In a future study, it might be possible to modify the realization so as to implicitly do the instantiation during the semantic action of the friend attribute. The instantiation could be triggered just before accessing the details of the type parameter.

The definition with the attribute behaves exactly as any other in-class defined friend definition. As such, it is not found by normal lookup unless we declare it explicitly. Of course we wish it to be found by normal lookup, otherwise we will not be able to call the function. Overall, this means that our test code should have the form:

```
template class Entity<StubMutex>;

__attribute__((friend(Entity<StubMutex>)))
void test_try_lock_fails() {
  //...
}
void test_try_lock_fails();

// part of the test framework
void testDriver() {
  test_try_lock_fails();
  // ... call other test functions
}
```

Here `testDriver` is the function which embodies the test framework, whose task is to execute each test case (or test suite) one-by-one. Another restriction of this particular realization of out-of-class friends is to allow only functions to be declared friends in this way.

After defining the constraints of such an attribute-based implementation we can explore the concrete realization steps. First, we define our new attribute in Clang's `Attr.td` file:

```
def OutOfClassFriend : InheritableAttr {
  let Spellings = [GCC<"friend">];
  let Args = [TypeArgument<"Host">];
  let Subjects = SubjectList<[Function]>;
  let Documentation = [Undocumented];
}
```

`Spellings` defines the list of the supported attribute syntaxes, but this time it is only the GCC style. The attribute syntax also defines the name of the attribute, in our case it is `friend`. `Args` specifies the list of the attribute arguments. Our friend attribute has only one argument which is a type. This type argument refers to the type that we would like to be the host class (the befriending class), i.e. the class for which we define the additional friend function. `Subobjects` describes the list of the lingual elements that might have this attribute. In this case, we only allow functions to have it. Note that implementing this attribute for classes is an important issue for future research.

Once we have the attribution definition in place, the Clang infrastructure will generate all the necessary parsing code. What is left is for us to define the semantic action for the new attribute and to hook that action into the existing complier machinery. As for the hooking, we need to add a new function call in the `ProcessDeclAttribute` function. This function is dedicated to apply a specific attribute to the specified declaration if the attribute applies to declarations. (Our attribute applies to function declarations.)

```
static void ProcessDeclAttribute(Sema &S, Scope *scope, Decl *D,
                                 const AttributeList &Attr,
                                 bool IncludeCXX11Attributes) {
  //...
  case AttributeList::AT_OutOfClassFriend:
    handleOutOfClassFriendAttr(S, D, Attr);
    break;
  //...
}
```

The semantic action for the new attribute is defined as follows:

```
1  static void handleOutOfClassFriendAttr(Sema &S, Decl *D,
2                                         const AttributeList &Attr) {
3    // Get the attribute type argument as QualType
4    ParsedType PT;
5    if (Attr.hasParsedType())
6      PT = Attr.getTypeArg();
7    else { // TODO error handling
8    }
9    TypeSourceInfo *QTLoc = nullptr;
10   QualType QT = S.GetTypeFromParser(PT, &QTLoc);
11   if (!QTLoc)
12     QTLoc = S.Context.getTrivialTypeSourceInfo(QT, Attr.getLoc());
13
14   // The type argument must be a CXXRecordDecl
15   RecordDecl *RD = getRecordDecl(QT);
16   assert(RD);
17   CXXRecordDecl *CRD = cast<CXXRecordDecl>(RD);
18   // The attribute is subject of a FunctionDecl
19   FunctionDecl *FD = cast<FunctionDecl>(D);
20   // Set this function as a friend function
21   FD->setObjectOfFriendDecl();
22   // Create a new friend decl for the befriending class
```

```
23   FriendDecl::Create(S.Context, CRD, D->getLocation(),
24                      cast<NamedDecl>(D), Attr.getLoc());
25   // For the record, Add the attribute to the Decl
26   D->addAttr(::new (S.Context) OutOfClassFriendAttr(
27       Attr.getRange(), S.Context, QTLoc,
28       Attr.getAttributeSpellingListIndex()));
29 }
```

The `S` parameter holds a reference of the monumental `Sema` class which is responsible for semantic analysis and AST building in the Clang compiler. The `D` parameter represents the declaration which has the attribute. The attribute itself is described with the `Attr` parameter. The first step is to get the type parameter of the attribute as a `QualType` (line 3-12). A `QualType` holds the basic type (e.g. `int`) and all the qualifiers – if any – on that type. For this, we get the `ParsedType` from the `Attr` with the `getTypeArg()` function (lines 4-6). A `ParsedType` is an opaque pointer for `QualType`s, i.e. this is a type erased generic holder, this is something similar to `void*`. Next, we get the underlying `QualType` from the `ParsedType` and we set the location of it (lines 10-12). If we cannot get the location information for the type, we simple set it to the location of the attribute (lines 11-12).

Afterwards, we get the `RecordDecl` instance from the `QualType` instance with the help of the `getRecordDecl` function (line 15). This function returns a null pointer if the `QualType` does not represent a record declaration. In Clang, a `RecordDecl` is the type of the AST node that is created for C structs and unions. Similarly, a `CXXRecordDecl` is specifically for C++ classes, structs and unions. This means that we can safely cast the record declaration to a `CXXRecordDecl` (line 17). The cast expression used here is a Clang specific cast, which is a "checked cast" operation. It converts a pointer or reference from a base class to a derived class, causing an assertion failure if it is not really an instance of the right type [16].

Next, we get the more specific function declaration (`FunctionDecl`) from the parameter `Decl` (line 19). The conversion from `Decl` to `FunctionDecl` must succeed, since we explicitly specified in the `Attr.td` file that this attribute is valid only for function declarations. So we use the checked cast again. Then we set up this function declaration as a friend declaration (line 21).

Later, we create the AST node for this new friend declaration (lines 23-24). This friend declaration references the previously synthesized `CRD` pointer as the befriending class, and the `D` parameter as the friend declaration. Once we have the friend declaration in place, the access checking mechanism will assess the target function like any other regular friend function.

As a last step, we register the attribute for the declaration (lines 26-28). This step is not essential, but it makes the whole procedure complete. We did this because in some future static analysis or other tool might want to process this information.

5 Related Work

There are many unit test frameworks available for C++ [40, 10, 32]. These frameworks provide only the basic tooling for creating test suites, test cases and assertions within the test cases. They do not provide any device for creating tests for legacy code without refactoring.

The experimental Boost DI framework places an emphasis to ease the creation of object trees [14]. However this framework requires that all the dependencies be injected via a constructor, i.e. we should refactor our legacy code intrusively in order to use it.

Non-intrusive testing can be done by the means of certain seams. Seams were introduced by Feathers [4]. In his book, he describes what a seam is in the context of legacy object-oriented code and he defines the three basic seams, these being preprocessor, link and object seams. He does not focus on C++, for instance he explains link seams with Java and classpath settings. He presents the refactoring method called *Extract Interface* for breaking dependencies.

Rüegg and Sommerlad elaborated the concept of seams in C++ [35]. They added a new seam for C++, called the *compile seam*. Their study expanded on the advantages, disadvantages and usability of the four seams. They presented three different techniques for linker seams, namely:

- shadow functions through linking order (original function cannot be called),
- wrapping functions with GNU's linker `wrap` command line option (the original function can be called),
- runtime function interception with `LD_PRELOAD` (the original function can be called and does not require relinking).

They also created a refactoring tool which is a plug-in for the Eclipse CDT platform including a C++ based mock object library [28]. Their tool supports refactoring towards all four mentioned C++ seams. Refactoring towards compile seams is possible via the technique they call the *Extract Template Parameter*.

Accessing private *non-static* data members via static pointers was first presented by Johannes Schaub [36]. Later it was extended by Chandra Shekhar Kumar [15] so as to use friend functions instead of static pointers. We also extended Kumar's approach to make it simpler and cleaner and we based our generic macro library implementation on the simplified version. To the best of our knowledge, accessing private *static* variables had never been presented before.

6 Vision/Future

With the help of compile-time reflection it would be possible to not change the `Entity` only for the simple reason that we wish to test it.

```
class Entity {
public:
  int process(int i) { if(m.try_lock()) { ... } else { ... } }
  //...
private:
  std::mutex m;
  //...
};

void testClient() {
  using EntityUnderTest =
    test::ReplaceMemberType<Entity, std::mutex, StubMutex>;
  EntityUnderTest e;
  auto& m = e.get<StubMutex>();
  // Test code as before
}
```

Here, `EntityUnderTest` is a type alias to such a type, which is equivalent to the `Entity` type except that all of its members with type `std::mutex` are replaced by the `StubMutex` type. Also, this type could give access to its internal mutex instance via its `get` function template. The above code requires the language capability of being able to declare variables and functions based on reflected names and types. Unfortunately, current C++ compile-time reflection proposals do not handle this language capability [42, 24]. This kind of reflection is sometimes called intercession (aka reification). For this technique to work, the given class has to be header-only, because the compiler has to know its internal layout and types to be able to replace some of them with another type. This header-only requirement might sound frightening, but with C++ modules it might be less painful to use header only classes [33].

7 Conclusion

The interweaving of the original code with the test code is an everyday problem in C++. Testing object-oriented systems frequently requires the replacement of objects representing the state of unit under test, either for exercise them or to mock them with a test double. While in managed programming languages dependency injection is supported by language features and libraries, in C++ the programmer often has to apply techniques which spoil the program structure, weaken encapsulation, and degrade performance.

In this paper, we overviewed the most frequent seams and their enabling points to alter the original behaviour of C++ programs for test purposes. We discussed the detrimental effects they may cause in the code structure and their compile-time and run-time performance.

To overcome these difficulties, we decided to introduce a new, non-intrusive compiler instrumentation-based approach for dependency replacement. The compiler collects information on the designated (member)functions at compile-time and creates a hook to make it possible to replace the called function during run-time. This technique could be used even in the case of inline functions and header only classes too, in a way that test code could be independent entirely from the unit

under test. The idea may be viewed as a new kind of seam that avoids most of the problems with the existing seams. We implemented this method as a patch for the LLVM/Clang compiler infrastructure, and evaluated its compile-time and run-time performance.

All non-intrusive testing methods require access to the internal state of the objects under test. Our method is of course no exception. Therefore, for the sake of accessing private members we discussed different techniques available for C++. Exploiting an exceptional language rule concerning explicit template instantiation provides an interesting way of accessing private non-static data members. We generalized the technique to access static members and member functions as well. Then we created a library to automate the access.

Since our new technique above had still some shortages, we presented a more generic method to access private or protected members. Friend declarations added outside of a class could provide a full, non-intrusive solution to separate test related code from the source of the unit under test. This new language element has the capability to work on every private assets, data, function, or type. We also realized a prototype based on C++ attributes to demonstrate the feasibility of the idea.

The out-of-class friends together with the compiler instrumented dependency replacement solution not only prevent the degradation of the code structure, but also avoid performance penalties. In a future study we would like to replace whole types based on the future standardized compile-time reflection of C++.

References

[1] Bertolino, Antonia and Marchetti, Eda. 1 a brief essay on software testing, 2004.

[2] Bright, Walter. C++ compilation speed, August 2010. http://www.drdobbs.com/cpp/c-compilation-speed/228701711.

[3] Driesen, Karel and Hölzle, Urs. The Direct Cost of Virtual Function Calls in C++. In Anderson, Lougie and Coplien, James, editors, *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 306–323. ACM, 1996.

[4] Feathers, Michael. *Working Effectively with Legacy Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[5] Fowler, Martin. The new methodology. http://www.martinfowler.com/articles/newMethodology.html.

[6] Fowler, Martin. Using a service locator. http://martinfowler.com/articles/injection.html#UsingAServiceLocator.

[7] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] gcc.gnu.org. An inline function is as fast as a macro, 2016. https://gcc.gnu.org/onlinedocs/gcc/Inline.html.

[9] gcc.gnu.org. Options that control optimization, 2016. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[10] Google. Google test. https://github.com/google/googletest.

[11] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[12] ISO. *ISO/IEC 14882:2014 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2014.

[13] Khan, Mohd. Ehmer and Khan, Farmeena. Importance of software testing in software development life cycle, 2014.

[14] Krzysztof, Jusiak. [boost].di. http://boost-experimental.github.io/di/index.html.

[15] Kumar, C.S. *Advanced C++ Faqs: Volumes 1 & 2*. Createspace Independent Pub, 2014.

[16] llvm.org. Llvm programmer's manual. http://releases.llvm.org/3.6.0/docs/ProgrammersManual.html.

[17] llvm.org. clang: a c language family frontend for llvm, 2016. http://clang.llvm.org.

[18] Majchrzak, Tim A. *Improving Software Testing: Technical and Organizational Developments*. Springer Publishing Company, Incorporated, 2012.

[19] Mane, Dashrath, Ojha, Namrata, and Chitnis, Ketaki. The spring framework: An open source java platform for developing robust java applications. *International Journal of Innovative Technology and Exploring Engineering*.

[20] Márton, Gábor. Access private, 2016. https://goo.gl/ynaZv5 https://github.com/martong/access_private.

[21] Márton, Gábor. Instrumentation for testing, 2016. https://goo.gl/FmNT5J https://github.com/martong/finstrument_mock.

[22] Márton, Gábor. Modified clang++ for intrumentation for testing, 2016. https://goo.gl/zutxKc https://github.com/martong/clang/tree/finstrument_-mock.

[23] Márton, Gábor. Out-of-class friend, 2016. https://goo.gl/kJgnzG https://github.com/martong/clang/tree/out-of-class_friend_attr.

[24] Márton, Gábor and Porkoláb, Zoltán. C++ compile-time reflection and mock objects. *Studia Univ. Babes–Bolyai Informatica*, LIX(2), 2014.

[25] Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. Item 41, Understand implicit interfaces and compile-time polymorphism.

[26] Meyers, Scott. *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, Inc., 1st edition, 2014.

[27] Mihalicza, József, Porkoláb, Zoltán, and Gabor, Abel. Type-preserving heap profiler for C++. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*, pages 457–466. IEEE Computer Society, 2011.

[28] mockator.com. An eclipse cdt plug-in for c++ seams and mock objects. http://mockator.com/.

[29] Network, Microsoft Developer. Unity container. https://goo.gl/YQwWlE https://msdn.microsoft.com/en-us/library/ff647202.aspx.

[30] Oracle. Java platform, standard edition 6 api specification, class accessibleobject. https://goo.gl/rfKFRA https://docs.oracle.com/javase/6/docs/-/api/java/lang/reflect/AccessibleObject.html.

[31] Parent, Sean. Inheritance is the base class of evil, 2013.

[32] Peter, Sommerlad and Emanuel, Graf. Cute: C++ unit testing easier. In *OOPSLA '07: Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 783–784, New York, NY, USA, 2007. ACM. 548071.

[33] Reis, Gabriel Dos. A module system for c++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4047.pdf.

[34] Reis, Gabriel Dos, Daniel García, J., and Logozzo, Francesco. Simple contracts for c++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4415.pdf.

[35] Rüegg, Michael and Sommerlad, Peter. Refactoring towards seams in c++. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 117–123, Piscataway, NJ, USA, 2012. IEEE Press.

[36] Schaub, Johannes. Access to private members: Safer nastiness. https://goo.gl/aG6HEv http://bloglitb.blogspot.hu/2010/07/access-to-private-members-thats-easy.html.

[37] Schwarz, Niko, Lungu, Mircea, and Nierstrasz, Oscar. Seuss: Decoupling responsibilities from static methods for fine-grained configurability. *Journal of Object Technology*, 11(1):3:1–23, April 2012.

[38] Seemann, Mark. *Dependency Injection in .NET*. Manning, 2011.

[39] Serebryany, Konstantin and Iskhodzhanov, Timur. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM.

[40] sourceforge.net. Cppunit - c++ port of junit. https://sourceforge.net/projects/cppunit/.

[41] Stroustrup, Bjarne, Sutter, Herb, et al. C++ core guidelines, 2016. https://goo.gl/7ZXiov https://github.com/isocpp/-/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rr-scoped.

[42] Tomazos, Andrew and Kaeser, Christian. Type property queries. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4428.pdf.

[43] wikipedia.org. Dependency injection. https://goo.gl/OjlpOY http://en.wikipedia.org/wiki/Dependency_injection.

[44] wikipedia.org. Service locator pattern. https://goo.gl/1KFxKj https://en.wikipedia.org/wiki/Service_locator_pattern.

[45] Winters, T. and Wright, H. All your tests are terrible, 2015.