Friendship in Service of Testing

Gábor Márton, martongabesz@gmail.com
Zoltán Porkoláb, zoltan.porkolab@ericsson.com

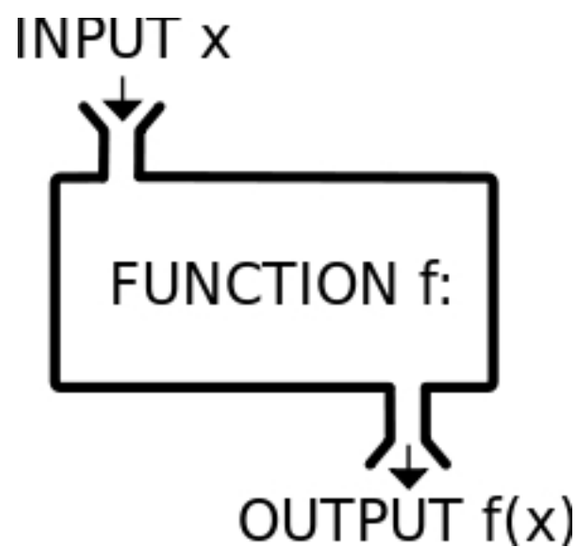
Agenda

- Principals
- Case study (running example)
- Vision

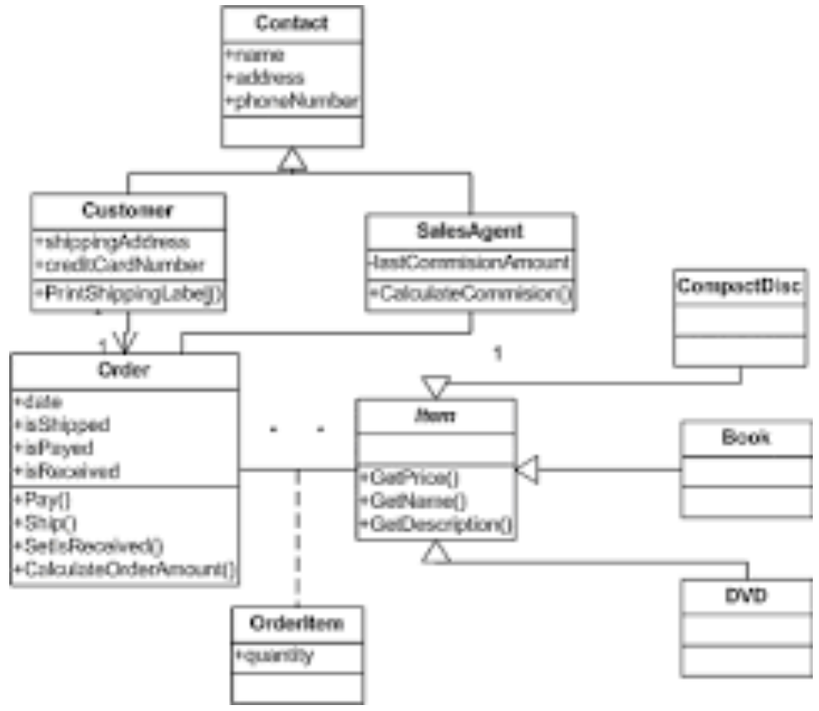
FP

- Immutability
  - Thread safe
  - Easy to test!

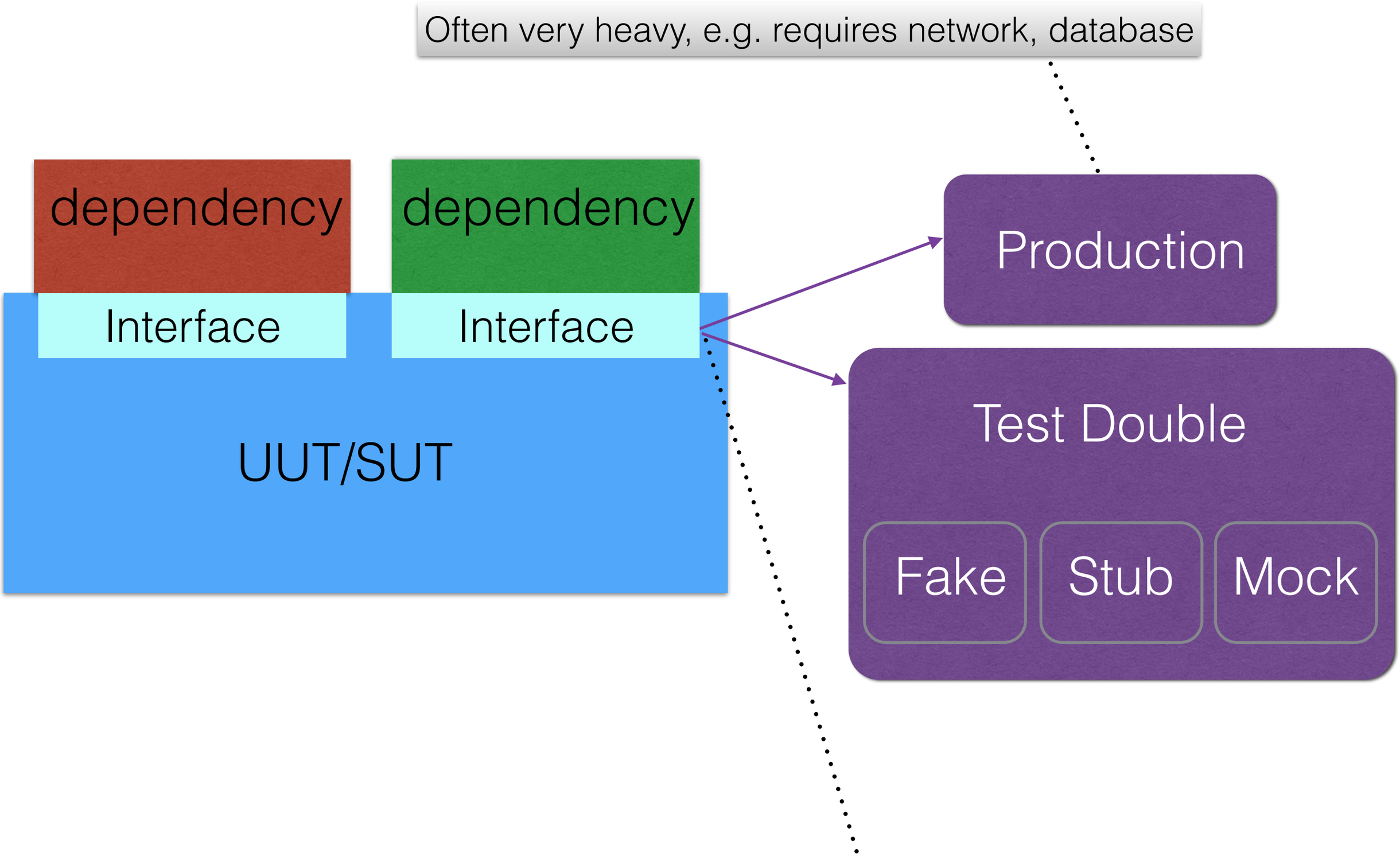
```
std::size_t
fibonacci(std::size_t);
ASSERT(fibonacci(0u) == 0u);
...
ASSERT(fibonacci(7u) == 13u);
ASSERT(fibonacci(8u) == 21u);
```



OOP



- States and behaviour
  - ✤ Encapsulation
- Dependencies (e.g Strategy pattern)
- Bad patterns -> side effects (Singleton)
- *Loosely coupled systems*
  - Interface (ptr or ref)

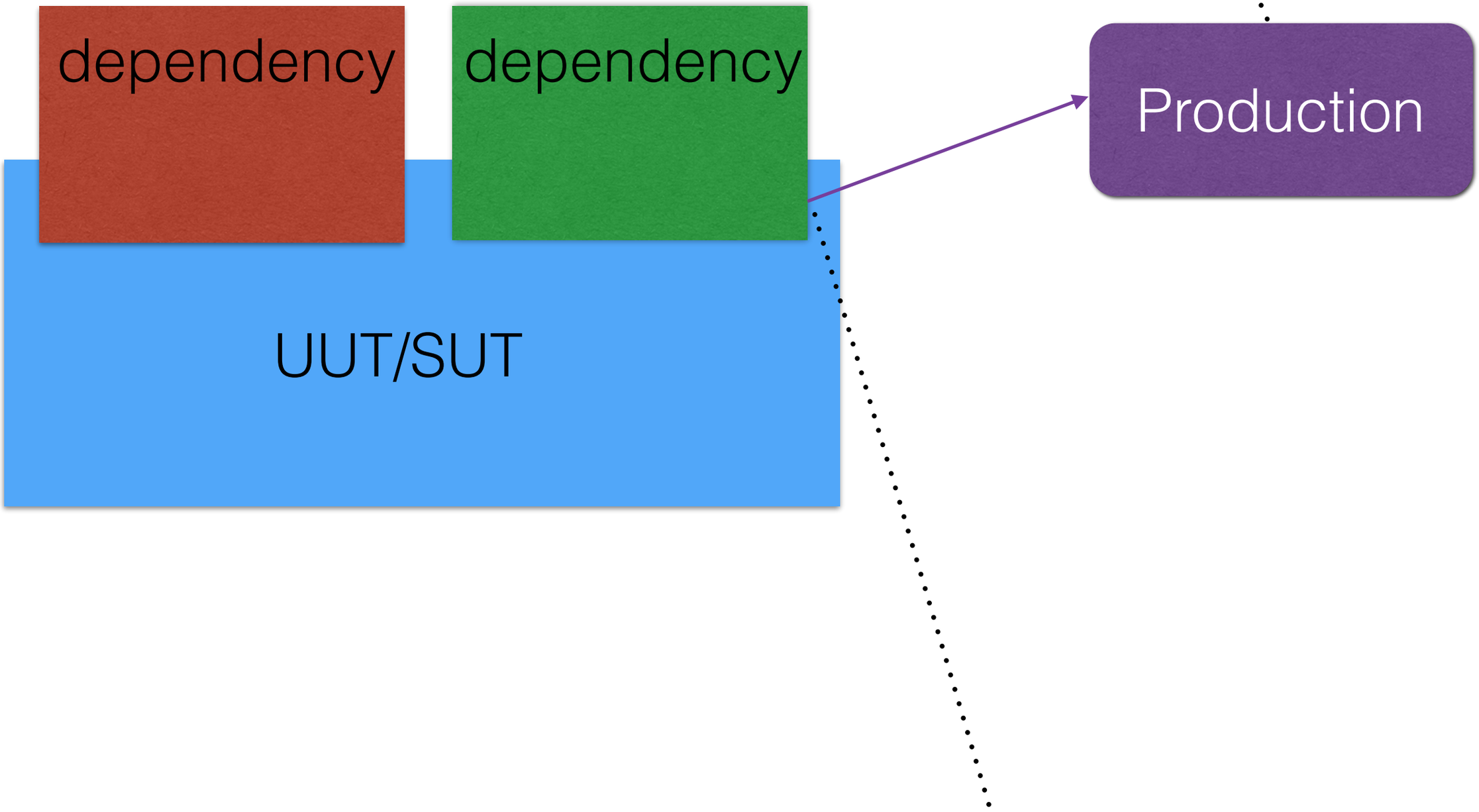


Dependency Replacement - DR

OOP - Testing

- Replace the dependency
- Access the private dependency (white-box tests)
  - To exercise them
  - Make assertions on them

Often very heavy, e.g. requires network



No interface in the source code,
What are the options for testing in C++?

Note:
For perfection, check the
definition of Seams by M. Feathers

- Linker
  - Inline code?
    - Header only code?
    - Templates?
  - Requires build system support

Note:
For perfection, check the definition of Seams by M. Feathers

- Linker
  - Inline code?
    - Header only code?
    - Templates?
      - Requires build system support
- Preprocessor
  - namespaces?

Note:
For perfection, check the definition of Seams by M. Feathers

- Linker
  - Inline code?
    - Header only code?
    - Templates?
      - Requires build system support
- Preprocessor
  - namespaces?
- Refactor
  - Add that interface

Note:
For perfection, check the definition of Seams by M. Feathers

Case Study

Problem Definition 1/2

```
class Entity {
public:
  int process(int i) {
    return accumulate(v.begin(),v.end(),i);
  }
  void add(int i) { v.push_back(i); }
private:
  vector<int> v;
};

void test1() {
  Entity e;
  e.add(1); e.add(2);
  ASSERT(e.process(0) == 3);
}
```

Problem Definition 1/2

```
class Entity {
public:
  int process(int i) {
    return accumulate(v.begin(),v.end(),i);
  }
  void add(int i) { v.push_back(i); }
private:
  vector<int> v;
};

void test1() {
  Entity e;
  e.add(1); e.add(2);
  ASSERT(e.process(0) == 3);
}
```

"Can we make it work in a multithreaded environment?"

Problem Definition 2/2

```
class Entity {
public:
  int process(int i) {
    if(m.try_lock()) {
      auto result = std::accumulate(…);
      m.unlock();
      return result;
    } else { return -1; }
  }
  void add(int i) { m.lock(); … m.unlock(); }
private:
  std::mutex m;
  …
};
```

Objective

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

Extend the Public Interface

```
class Entity {
public:
  // Add a getter
  auto& getMutex() { return m; }
  int process(int i) {
    if(m.try_lock()) { … } else { … }
  }
  …
};
```

```
void test() {
  Entity e;
  auto& m = e.getMutex();
  m.lock();
  ASSERT(e.process(1) == -1);
}
```

Unit:

- ❌ Encapsulation?

Test:

- ❌ Using a wider interface than the minimal needed
  - Test code coupled with std::mutex::lock()
- ❌ We want to fake our dependencies
  - They might be really heavy

Runtime Interface

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

```
struct RealMutex : Mutex { … };
struct StubMutex : Mutex {
  bool try_lock_result = false;
  virtual bool try_lock() override {
    return try_lock_result;
  }
  …
};
```

```
class Entity {
public:
  Entity(Mutex& m) : m(m) {} // DI
  int process(int i) { … }
private:
  Mutex& m;
};
```

```
void production() {
  RealMutex m;
  Entity e(m);
  // Real usage of e
}
```

```
void test() {
  StubMutex m;
  Entity e(m);

  m.try_lock_result = false;
  ASSERT(e.process(1) == -1);
  m.try_lock_result = true;
  ASSERT(e.process(1) == 1);
}
```

```
void production() {
  RealMutex m;
  Entity e(m);
  // Real usage of e
}
```

❌ Encapsulation?

```
void test() {
  StubMutex m;
  Entity e(m);

  m.try_lock_result = false;
  ASSERT(e.process(1) == -1);
  m.try_lock_result = true;
  ASSERT(e.process(1) == 1);
}
```

✅ Decoupled Test code

```
class Entity {
public:
  Entity(Mutex& m) : m(m) {}
  int process(int i) { … }
private:
  Mutex& m;
};
```

- Feels so unnatural!

```
class Entity {
public:
   Entity(Mutex& m) : m(m) {}
   int process(int i) { … }
private:
   Mutex& m;
};
```

- Feels so unnatural!

- ❌ Ownership?
- ❌ Extra ctor/setter
- ❌ Extra interface
  - More code (LoC)
  - Virtual functions
    - Cache locality?
- ❌ Pointer semantics
  - Cache locality?

Ownership

```
class Entity {
public:
  Entity(std::unique_ptr<Mutex> m) : m(std::move(m)) {}
  int process(int i) {
    if(m->try_lock()) { … } else { … }
  }
  Mutex& getMutex() { return *m.get(); }
private:
  std::unique_ptr<Mutex> m;
};

void test() {
  Entity e(std::make_unique<StubMutex>());
  auto& m = e.getMutex();
  // assertions …
}
```

- ✅ Ownership OK
- ❌ As before: Extra ctor, virtuals, pointer semantics, etc
- ❌ Extra getter

Compile time Interface

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

```
void production() {
  Entity<std::mutex> e;
  // Real usage of e
}
void test() {
  struct StubMutex{
    void lock() {}
    void unlock() {}
    bool try_lock_result = false;
    bool try_lock() {
      return try_lock_result; }
  };

  Entity<StubMutex> e;
  auto& m = e.getMutex();
  // assertions …
}
```

```
void production() {
  Entity<std::mutex> e;
  // Real usage of e
}
void test() {
  struct StubMutex{ ..... 
    void lock() {}
    void unlock() {}
    bool try_lock_result = false;
    bool try_lock() {
      return try_lock_result; }
  };

  Entity<StubMutex> e;
  auto& m = e.getMutex();
  // assertions …
}
```

- No inheritance
- No virtual functions

- ✅ Ownership
- ✅ Locality
- ✅ No Extra ctor/setter
- ✅ Implicit Interface (compile time)
  - Less LoC
- ❌ Extra Getter
- ❌ Extra Compilation Time

Extra Compilation Time - Combine with Pimpl

```
// detail/Entity.hpp
namespace detail {

// as before
template <typename Mutex>
class Entity { … };

} // detail
```

```cpp
// Entity.hpp
class Entity {
public:
   Entity();
   ~Entity();
   int process(int i);
private:
   struct Impl;
   std::unique_ptr<Impl> pimpl;
};
```

```cpp
// Entity.cpp
#include “detail/Entity.hpp”
using ProdEntity =
  detail::Entity<std::mutex>;
//using Entity::Impl = ProdEntity; // ERROR

struct Entity::Impl : ProdEntity {
  // delegate the consturctors
  using ProdEntity:: ProdEntity;
};

Entity::Entity() :
  pimpl(std::make_unique<Impl>()) {}
Entity::~Entity() = default;
int Entity::process(int i) {
  return pimpl->process(i);
}
```

```cpp
// Production.cpp
#include “Entity.hpp”
void production() {
  Entity e;
  // Real usage of e
}
```

```cpp
// Test.cpp
#include “detail/Entity.hpp”
void test() {
  struct StubMutex { … };
  detail::Entity<StubMutex> e;
  // Test code as before
}
```

Compile time Interface + Pimpl

- ✅ No extra compile time
  - But at least two compiles if `detail::Entity` changes
    - Entity.cpp
    - Test.cpp
- Performance? Fast pimpl.
- ❌ Extra complexity

Compile time Interface + `extern` Template Declarations

- ✅ A bit better compile time
  - All included headers must be parsed
    - Precompiled headers, modules?
- ❌ Extra complexity

```
class Entity {
public:
   int process(int i) { … }
private:
   std::mutex m;
};
```

```
template <typename Mutex> class Entity {
public:
   Mutex& getMutex() { return m; }
   int process(int i) { … }
private:
   Mutex m;
};
```

```
class Entity {
public:
   int process(int i) { … }
private:
   std::mutex m;
};
```

```
template <typename Mutex> class Entity {
public:
   Mutex& getMutex() { return m; }
   int process(int i) { … }
private:
   Mutex m;
};
```

Encapsulation?

Eliminate the Extra Getter

```
template <typename Mutex>
class Entity {
   …
#ifdef TEST
   Mutex& getMutex() { return m; }
#endif
   …
};
```

Eliminate the Extra Getter

```
template <typename Mutex>
class Entity {
   …
#ifdef TEST
   Mutex& getMutex() { return m; }
#endif
   …
};
```

- ❌ Worse
  - Readability
  - Maintainability

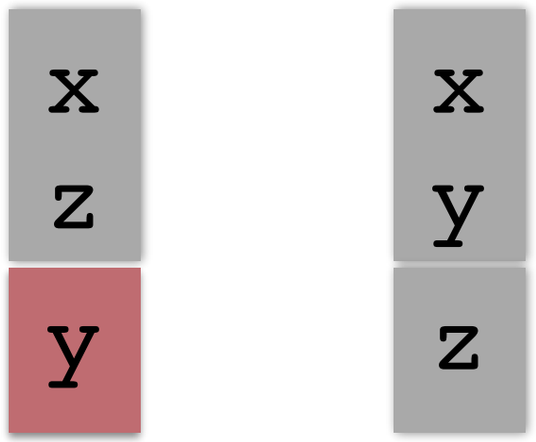
```
#define private public
#define class struct
#include "Entity.hpp"
#undef private
#undef class
// Test code comes from here
```

```
#define private public
#define class struct
#include "Entity.hpp"
#undef private
#undef class
// Test code comes from here
```

❌ Undefined behaviour

- The order of allocation of non-static data members with different access control is unspecified

```
class X {
public: int x;
private: int y;
public: int z;
};
```



❌ Danger, everything included from Entity.hpp is now public

Eliminate the Extra Getter - Access via a Friend Function

```
template <typename Mutex> class Entity {
public:
   friend Mutex& testFriend(Entity &e);
   int process(int i) { … }
private:
   Mutex m;
};
```

```cpp
// Test.cpp
struct StubMutex { … };

StubMutex& testFriend(Entity<StubMutex>& e){
  return e.m;
}
```

```cpp
void test() {
  Entity<StubMutex> e;
  auto &m = testFriend(e);
  // assertions …
}
```

Eliminate the Extra Getter - Access via a Friend Class

```
template <typename Mutex> class Entity {
public:
   friend struct EntityTestFriend;
   int process(int i) { … }
private:
   Mutex m;
};
```

```cpp
// Test.cpp
struct StubMutex { … };

struct EntityTestFriend {
  template<typename Mutex>
  static auto& getMutex(Entity<Mutex>& e) {
    return e.m;
  }
};

void test() {
  Entity<StubMutex> e;
  auto &m = EntityTestFriend::getMutex(e);
  // assertions …
}
```

Attorney - Client Idiom

```
class EntityTestFriend {
  template<typename Mutex>
  static auto& getMutex(Entity<Mutex>& e) {
    return e.m;
  }
  friend void test_try_lock_succeeds();
  friend void test_try_lock_fails();
};

void test_try_lock_succeeds() {
  Entity<StubMutex> e;
  auto &m = EntityTestFriend::getMutex(e);
  // assertion …
}
// …
```

Attorney - Client Idiom

```
class EntityTestFriend {
  template<typename Mutex>
  static auto& getMutex(Entity<Mutex>& e) {
    return e.m;
  }
  friend void test_try_lock_succeeds();
  friend void test_try_lock_fails();
};

void test_try_lock_succeeds() {
  Entity<StubMutex> e;
  auto &m = EntityTestFriend::getMutex(e);
  // assertion …
}
// …
```

- Attorney <—> `EntityTestFriend`
- Client <—> `Entity`

```
class Entity {
public:
   int process(int i) { … }
private:
   std::mutex m;
};
```

```
template <typename Mutex> class Entity {
public:
   friend struct EntityTestFriend;
   int process(int i) { … }
private:
   Mutex m;
};
```

```
class Entity {
public:
   int process(int i) { … }
private:
   std::mutex m;
};
```

Intrusive

```
template <typename Mutex> class Entity {
public:
   friend struct EntityTestFriend;
   int process(int i) { … }
private:
   Mutex m;
};
```

Eliminate the Extra Getter - Non-intrusive Access

```
template <typename Mutex> class Entity {
public:
   int process(int i) { … }
private:
   Mutex m;
};
```

```cpp
// Test.cpp
struct StubMutex { … };

ACCESS_PRIVATE_FIELD(
  Entity<StubMutex>, StubMutex, m)

void test_try_lock_fails() {
  Entity<StubMutex> e;
  auto& m = access_private::m(e);
  m.try_lock_result = false;
  ASSERT(e.process(1) == -1);
}
```

Access a private static member

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

Access a private static member

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

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

Access a private static member

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

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

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

Access a private static member

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

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

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

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

Access a private static member

```
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>;
```

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

Access a private static member

```
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);
}
```

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

Access a private non-static member

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

Access a private non-static member

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

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

Access a private non-static member

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

using PtrType = int A::*;
template<PtrType PtrValue>
struct private_access {
    friend PtrType get() { return PtrValue; }
};

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

Access a private non-static member

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

using PtrType = int A::*;
template<PtrType PtrValue>
struct private_access {
    friend PtrType get() { return PtrValue; }
};

PtrType get();
template struct private_access<&A::i>;
```

Access a private non-static member

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

using PtrType = int A::*;
template<PtrType PtrValue>
struct private_access {
    friend PtrType get() { return PtrValue; }
};

PtrType get();

template struct private_access<&A::i>;

void usage() {
    A a;
    PtrType ip = get();
    int& i = a.*ip;
    ASSERT(i == 42);
}
```

- ✅ Can access private member fields, functions
- ❌ Can’t access private types
- ❌ Can’t access private ctor/dtor
- ❌ Link error with in-class defined private `static const`

Eliminate the Extra Getter - Out-of-class Friend

```
class Entity {
  public: int process(int i) { … }
  private: std::mutex m;
};
```

Eliminate the Extra Getter - Out-of-class Friend

```
class Entity {
  public: int process(int i) { … }
  private: std::mutex m;
};
```

```
// Test.cpp
friend for(Entity) void test() {
  Entity e;
  auto& m = e.m; // access the private
  // assertions …
}
```

Eliminate the Extra Getter - Out-of-class Friend

```
template <typename Mutex> class Entity {
  public: int process(int i) { … }
  private: Mutex m;
};
```

Eliminate the Extra Getter - Out-of-class Friend

```
template <typename Mutex> class Entity {
  public: int process(int i) { … }
  private: Mutex m;
};
```

```
// Test.cpp
friend for(Entity<StubMutex>)
void test() {
  Entity<StubMutex> e;
  auto& m = e.m; // access the private
  // assertions …
}
```

- ✅ Clear intentions, self describing code
- ✅ No cumbersome accessor patterns
- ✅ Can access all private (types, ctor, …)
- ✅ Proof-of-concept implementation in clang
- ✅ Validate 3rd party, read-only code

- Encapsulation (?)
  - The language should prevent accidental failures
  - “friend for” cannot be accidental
  - searchable / “greppable”
  - compile the tests with -enable-ooc-friend switch

Vision

```
class Entity { … };
```

Vision

```
class Entity { … };


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

Vision

```
class Entity { … };
```

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

- Compile time reflection
  - reification
  - access private
- Header only
  - modules?
- Replace only the member?
  - replace local variables of a member?

Thank you!

- Gábor Márton
  martongabesz@gmail.com
- https://github.com/martong/access_private
- https://github.com/martong/clang/tree/out-of-class_friend_attr
- Questions?

Black-box testing

```
struct RealMutex : Mutex { … };
struct FakeMutex : Mutex { //Simulates RealMutex
  bool locked = false;
  virtual lock() override { locked = true; }
  virtual unlock() override { locked = false; }
  virtual bool try_lock() override {
    return !locked
  }
};
class Entity {
public:
  Entity(Mutex& m) : m(m) {} // DI
private:
  Mutex& m;
};
```

Black-box testing

```
void production() {
  RealMutex m;
  Entity e(m);
  // Real usage of e
}
```

```
void test() {
  StubMutex m;
  Entity e(m);

  m.lock();
  ASSERT(e.process(1) == -1);
  m.unlock();
  ASSERT(e.process(1) == 1);
}
```

Black-box testing

```
void production() {
  RealMutex m;
  Entity e(m);
  // Real usage of e
}
```

- Encapsulation?

```
void test() {
  StubMutex m;
  Entity e(m);

  m.lock();
  ASSERT(e.process(1) == -1);
  m.unlock();
  ASSERT(e.process(1) == 1);
}
```

- Decoupled Test code
- Black-box testing
  - Only public functions are used
  - We know a mutex's
    - behaviour
    - state: locked/unlocked

Tension: Design vs Test

```
class Entity {
public:
   int process(int i) {
     if(m.try_lock()) {
       auto result = std::accumulate(…);
       m.unlock();
       return result;
     } else { return -1; }
   }
   void add(int i) { m.lock(); … m.unlock(); }
private:
   std::mutex m;
   …
};
```

Different public interface towards
- production (don't care of the mutex)
- test (interested in the mutex, to exercise).

two diff aspects of the design

Tension: Design vs Test

```
class Entity {
public:
   int process(int i) {
     if(m.try_lock()) {
       auto result = std::accumulate(…);
       m.unlock();
       return result;
     } else { return -1; }
   }
   void add(int i) { m.lock(); … m.unlock(); }
private:
   std::mutex m;
   …
};
```

Different public interface towards
- production (don't care of the mutex)
- test (interested in the mutex, to exercise).

two diff aspects of the design

must be private, encapsulation!

Tension: Design vs Test

```
class Entity {
public:
   int process(int i) {
     if(m.try_lock()) {
        auto result = std::accumulate(…);
        m.unlock();
        return result;
     } else { return -1; }
   }
   void add(int i) { m.lock(); … m.unlock(); }
private:
   std::mutex m;
   …
};
```

Different public interface towards
- production (don't care of the mutex)
- test (interested in the mutex, to exercise).

The existence of std::mutex infiltrates into the public interface

two diff aspects of the design

must be private, encapsulation!

Tension: Design vs Test

```
class Entity {
public:
   int process(int i) {
     if(m.try_lock()) {
        auto result = std::accumulate(…);
        m.unlock();
        return result;
     } else { return -1; }
   }
   void add(int i) { m.lock(); … m.unlock(); }
private:
   std::mutex m;
   …
};
```

Different public interface towards
- production (don't care of the mutex)
- test (interested in the mutex, to exercise).

The existence of std::mutex infiltrates into the public interface

two diff aspects of the design

must be private, encapsulation!

Reference wrapper ?

```
template <typename Mutex>
class Entity {
public:
   Entity(const Mutex& m) : m(m) {}
   int process(int i) {…}
   …
private:
   Mutex m;
   …
};
```

```
void test() {
  StubMutex m;
  using RefM =
    std::reference_wrapper<StubMutex>;
  Entity<RefM> e(m); // OK
  set_try_lock_fails(m);
  ASSERT(e.process(1) == -1);
  // ERROR          ^ ref wrapper has no
  // try_lock men fun.
}

// Would work if std::mutex
// would be copyable
void production() {
  std::mutex m;
  Entity<std::mutex> e(m); // error
}
```