Testable C++

Michael Feathers
mfeathers@objectmentor.com

Rationale

- Much existing C++ code is hard to test
- There are practices and constructs which make testing hard
- Testability can not be an afterthought

Rationale

```
TEST(dispatcher_tests,remap_reallocates)
{
    TCAddressMap addr_map;
    dispatcher_t dispatcher(addr_map);

    dispatcher.remap();
    CHECK_EQUAL(0, addr_map.size());
}
```

Background

- Not much documented advice on C++ testing
- Resurgence of interest in unit testing with Agile
- Plethora of frameworks: CppUnit, CppUTest,UnitTest++

Concepts

- All testing is *feature* testing
- The terms unit, component and system determine scope, not coverage
- Unit tests greatly enhance understandability

Nomenclature

- CUT - Code Under Test
- TUF - Test Unfriendly Functionality
- TUC - Test Unfriendly Constructs

Code Under Test (CUT)

- The CUT is the actual path you exercise in a test.
- It is not a “unit” or “component”
- It can be big or small
- “Unit” or “component” is the length of the path

Test Unfriendly Feature (TUF)

- Some portion of code that thwarts unit testing
- Often it causes some side effect
- Or prevents sensing of an effect

Test Unfriendly Feature (TUF)

```
TEST(dispatcher_tests,remap_reallocates)
{
    TCAddressMap addr_map;
    dispatcher_t dispatcher(addr_map);

    dispatcher.remap();
    CHECK_EQUAL(0, addr_map.size());
}
```

Test Unfriendly Feature (TUF)

- Use of a third party library
- Use of the file system
- Use of the network
- Use of a database
- Expensive operations (> 100 millisec)

The Goal

- Tests should run fast and help us understand what works and what doesn’t
- TUFs are the biggest impediment to fast testing

Seams

- There are places where it is easier to break dependencies

C++ Seams (in order)

- Preprocessing seam
- Template instantiation seam
- Link Seam
- Object Seam

Test Unfriendly Construct (TUC)

- C++ Language constructs which prevent or impede testing
- They don’t always, but you have to pay attention to their use
- Don't put TUFs inside TUCs!
- Testability involves design choices

TUC1 Bare Classes

- Bare classes are classes without abstract base classes (ABCs)
- ABCs (like interfaces in Java) reduce dependency and allow mocking when needed

TUC1 Bare Classes

```
class ip_connection {
public:
    void release();
    ...
private
    ...
};
```

TUC1 Bare Classes

```
class ip_connection : public connection {
public:
    virtual void release();
    virtual ~ip_connection();
    ...
private
    ...
};
```

TUC1 Bare Classes

```
// Abstract base class (interface)

class connection {
public:
    virtual void release() = 0;
    virtual ~ip_connection() = 0;
    ...
};
```

TUC2 RAII Objects

- RAII is useful, but if the thing that you are acquiring is a TUF, you are out of luck
- Stack-based semantics make them hard to replace

TUC2 RAII Objects

```
void function() {
    ucs_file file(“RS1”);
    ...
    file << pend_message;
}
```

```
void function() {
    ucs_file file(ucs_impl, “RS1”);
    ...
    file << pend_message;
}
```

TUC3 Multi-Purpose Files

- It’s bad form to put more than one class in a file.
- When you do, you make it harder to test classes independently.
- The context of each is polluted by the other.

TUC3 Multi-Purpose Files

```
#include “grog.h”
#include “groger.h”

class Fizzle { .. };
class Floop { ... };
class Fing { ... };
```

TUC4 Free Functions

- Functions that are not associated with any class
- They can help encapsulation but never use them to hide a TUF (or potential TUF)
- Impossible to replace unless in their own linkage unit

TUC4 Free Functions

```
int perimeter(const std::vector<point>& points) {
    ...
}

double arctan(double value) { ...}

void update_host(char *id, const udp_packet& packet) {
    ...
}
```

TUC5 Unnamed Namespace

- The unnamed namespace is another place to hide things.
- Scope is limited to the file.
- Unable to mock for testing

TUC5 Unnamed Namespace

```
namespace {
    device_descriptor startup_descs;
};

void init() {
    ...
    conn->send(startup_descs.align_parms);
    ...
}
```

TUC6 Non-Virtual Functions

- We often make functions non-virtual for performance reasons and for documentary reasons
- In many languages, all functions are virtual by default
- Better choice. Makes mocking easier
- Reference objects that hide TUFs should be virtual-enabled

TUC6 Non-Virtual Functions

```
class ip_connection {
public:
    void release() {
        // (TUF) We're caught!
    }
    ...
private
    ...
};
```

TUC7 Internal Instantiation

- Objects directly instantiated in other objects are hard to replace.
- Factories are better
- Dependency Injection Pattern is gaining acceptance

TUC7 Internal Instantiation

```
void router::dispatch(const char *msg,
                      const host_info& host)
{
    ...
    addr_resolver resolver;
    ip = resolver.get_iptrans(host);
    ...
}
```

TUC8 Internal Definition

- C++ let you define nested classes
- Not a good idea if you have a TUF
- No way to replace except via template hoisting

TUC8 Internal Definition

```
namespace boost {

    class any {
    public:
        class placeholder {
            ...
        };
    private:
        ...
    };
}
```

TUC8 Internal Definition

```
class module {
public:
    ...
private:
    class module_builder {
        ...
    };
};
```

TUC8 Internal Definition

```
template <typename BUILDER> class module_impl {
public:
    ...
private:
    ...
};

typedef module_impl<module_builder> module;
```

TUC9 Long Functions

- The biggest testability problem
- Long Functions don’t just hide TUFs, they are TUFs.
- Functions should have a single responsibility

TUC9 Long Functions

```
void process_t::inner_action() {
    ...
    addr1 = bind(addr2);
    ...
    routing_table[addr2].push_back(fair_opt);
    ...
    routing_table.clear();
    ...


};
```

TUC10 Templated Member Functions

- A sneaky hole in the semantics of templates
- Template member functions can not be replaced

TUC10 Templated Member Functions

```
struct foo {
    template<typename T> bool do_it(T& t) {
        ...
    }
};
```

TUC11 Static Variables

- They are like glue
- They make repeatability of tests a real problem

TUC11 Static Variables

```
router_t router_t::get_instance() {
    static router_t router;
    return router;
}
```

TUC12 Lack Of Unit Tests

- All of these problems can be easily solved if we simply write tests as we develop our code
- If a test is hard to write, that means that we have to find a different design which is testable
- It’s always possible