basv@odd-e.com

Test-driven Development in C (and embedded)

Agenda

Introduction

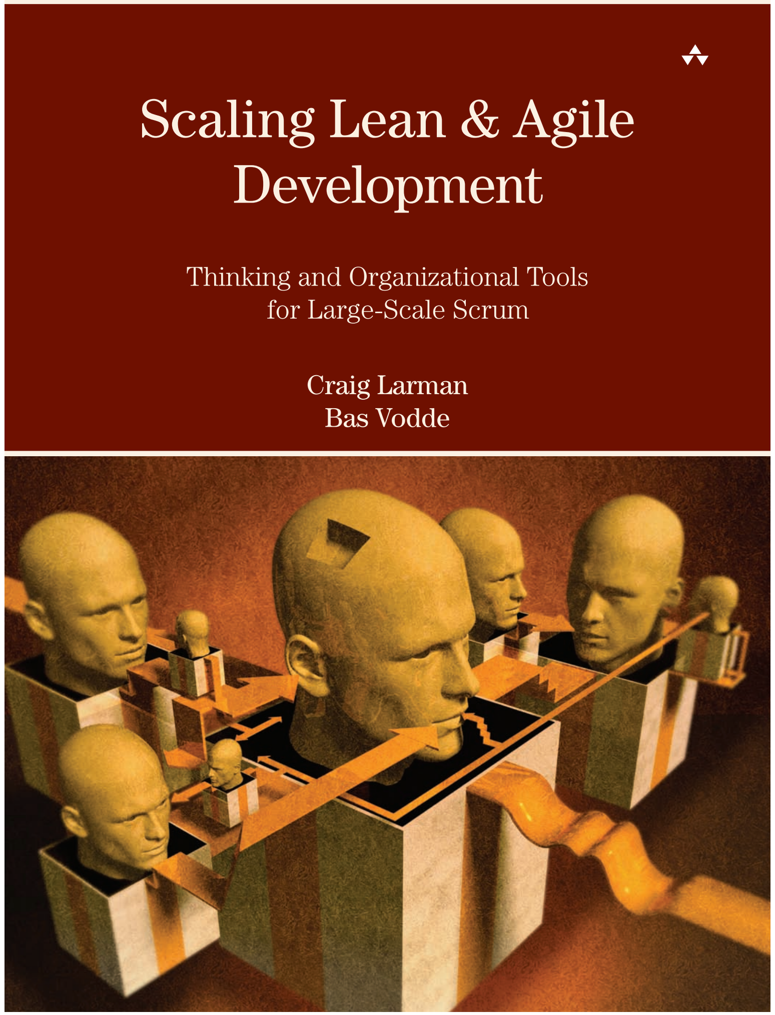
Test-Driven Development

CppUTest

TDD in C

TDD in Embedded systems

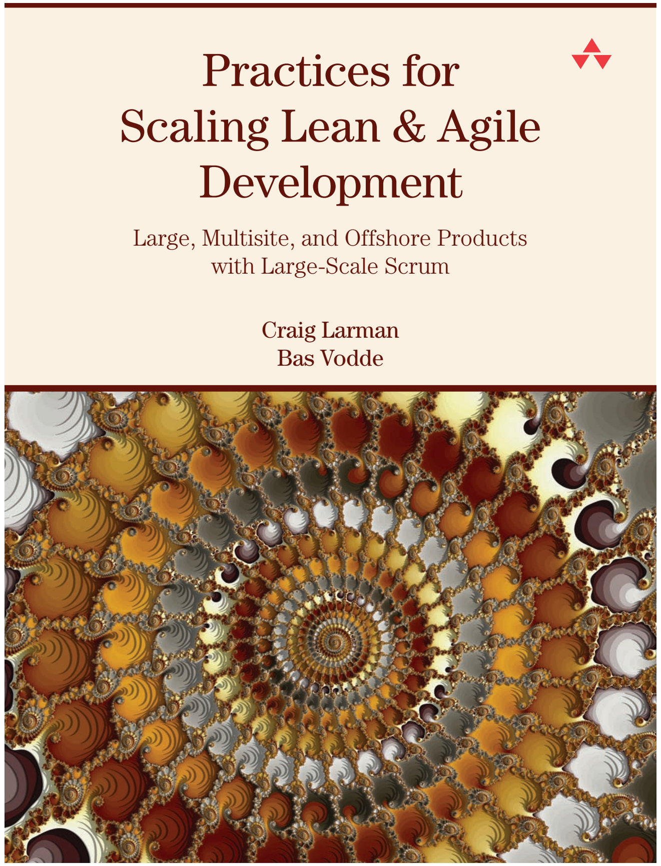
Introduction



Scaling Lean & Agile Development

Thinking and Organizational Tools for Large-Scale Scrum

Craig Larman
Bas Vodde



Practices for Scaling Lean & Agile Development

Large, Multisite, and Offshore Products with Large-Scale Scrum

Craig Larman
Bas Vodde

Odd-e

Test-Driven Development

TDD

The single rule of Test-Driven Development (or test-first programming) :

Only ever write code to fix a failing test

- Write a test (which fails -> “red”)
- Write the code (to make test pass -> “green ”)
- Refactor the code and test (you’re still “green ”)

Not a unit test when...

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment
  (such as editing config files) to run it.

CppUTest

What is CppUTest?

- sUnit -> JUnit -> xUnit variant
- Framework for unit tests in C and C++
- Not need any external scripting.

First test

TestFirst.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest)
{
};

TEST(FirstTest, First)
{
   LONGS_EQUAL(1,0);
}
```

First test

TestFirst.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest)
{
};

TEST(FirstTest, First)
{
   LONGS_EQUAL(1,0);
}
```

Main CppUTest header

Most CppUTest functionality is included in this header

First test

TestFirst.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest)
{
};

TEST(FirstTest, First)
{
   LONGS_EQUAL(1,0);
}
```

Declaration of a new TEST_GROUP.

All tests have to belong to one test group.

First test

TestFirst.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest)
{
};

TEST(FirstTest, First)
{
   LONGS_EQUAL(1,0);
}
```

Do not forget this semi-column

It will lead to strange compiler errors.

First test

TestFirst.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest)
{
};

TEST(FirstTest, First)
{
    LONGS_EQUAL(1,0);
}
```

First test.

First parameter is group name

Second parameter is test name

First test

TestFirst.cpp

```
#include "CppUTest/TestHarness.h"

TEST_GROUP(FirstTest)
{
};

TEST(FirstTest, First)
{
    LONGS_EQUAL(1,0);
}
```

LONGS_EQUAL compares ints

First parameter is expected value

Second parameter is actual value

First test

Main.cpp

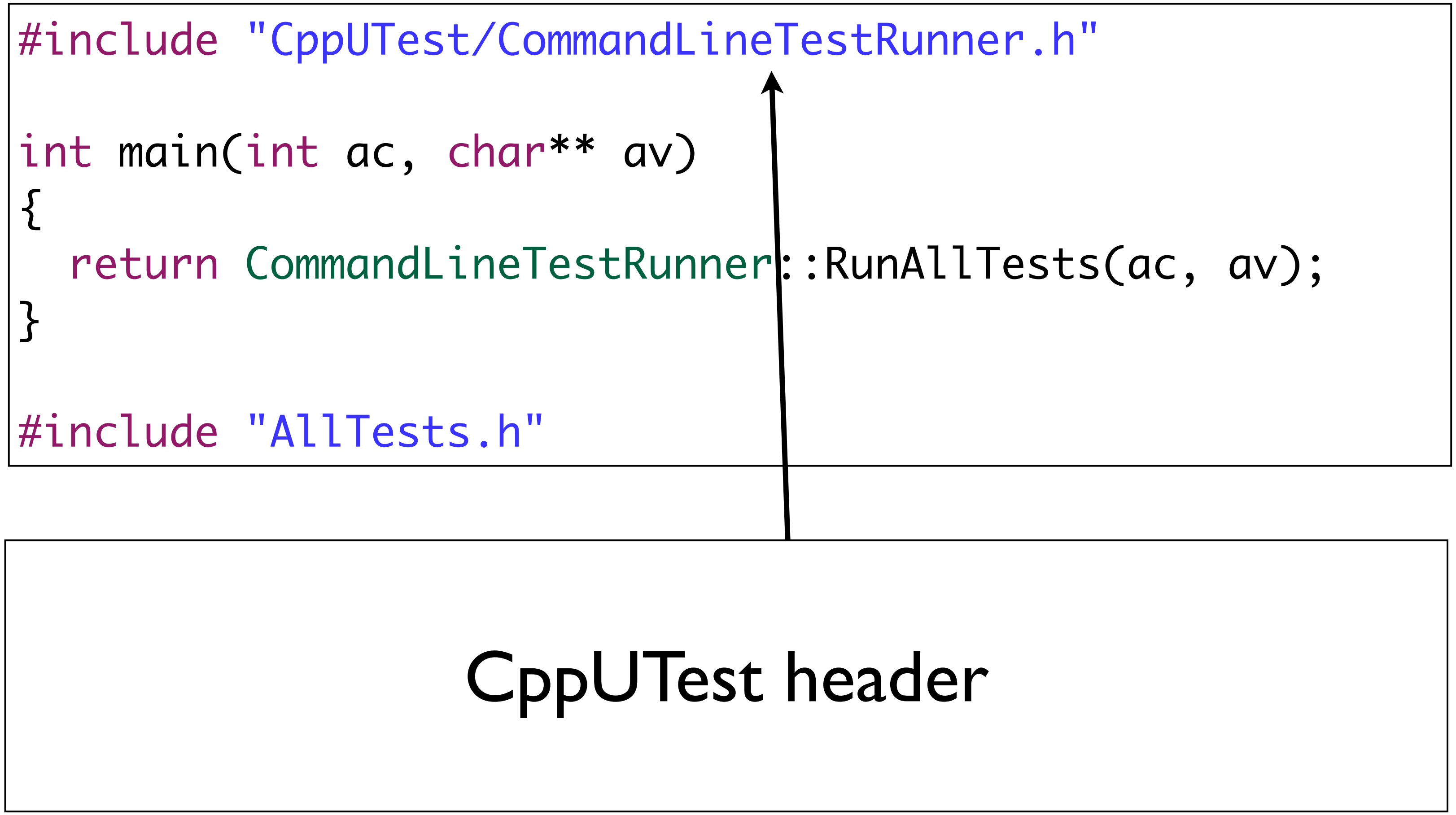
```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, char** av)
{
  return CommandLineTestRunner::RunAllTests(ac, av);
}

#include "AllTests.h"
```

First test

Main.cpp



```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, char** av)
{
  return CommandLineTestRunner::RunAllTests(ac, av);
}

#include "AllTests.h"
```

CppUTest header

First test

Main.cpp

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, char** av)
{
  return CommandLineTestRunner::RunAllTests(ac, av);
}

#include "AllTests.h"
```

Call CommandLineTestRunner::RunAllTests class. This will execute all tests and return an error if one fails.

First test

Main.cpp

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, char** av)
{
  return CommandLineTestRunner::RunAllTests(ac, av);
}

#include "AllTests.h"
```

Pass command line arguments to CppUTest.

First test

Main.cpp

```
#include "CppUTest/CommandLineTestRunner.h"

int main(int ac, char** av)
{
  return CommandLineTestRunner::RunAllTests(ac, av);
}

#include "AllTests.h"
```

Include all the test groups via AllTests.h

First test

AllTests.h

```
IMPORT_TEST_GROUP(FirstTest);
```

Import test group using IMPORT_TEST_GROUP

This is only needed when tests are in a separate library, but it is a good habit to always do this.

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
	./TestFirst

TestFirst: TestFirst.o Main.o
```

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
   ./TestFirst

TestFirst: TestFirst.o Main.o
```

Define a variable with the location of CppUTest

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
   ./TestFirst

TestFirst: TestFirst.o Main.o
```

Add the include path to the default compilation options.

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
   ./TestFirst

TestFirst: TestFirst.o Main.o
```

Add CppUTest and STD C++ to the default linker option

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
	./TestFirst

TestFirst: TestFirst.o Main.o
```

Create a default target

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
	./TestFirst

TestFirst: TestFirst.o Main.o
```

Run the tests
(makefiles require tabs! Be careful with your IDE settings)

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
   ./TestFirst

TestFirst: TestFirst.o Main.o
```

Declare which object files to link

First test

Makefile

```
CPPUTEST = ../../CppUTest

CPPFLAGS += -I $(CPPUTEST)/include
LDFLAGS += $(CPPUTEST)/lib/libCppUTest.a -lstdc++

all: TestFirst
   ./TestFirst

TestFirst: TestFirst.o Main.o
```

Make default targets will take care of the test!

First test

Output

```
./TestFirst

TestFirst.cpp:10: error: Failure in TEST(FirstTest, First)
make: *** [all] Error 1
    expected <1 0x1>
    but was  <0 0x0>


.
Errors (1 failures, 1 tests, 1 ran, 1 checks, 0 ignored, 0 filtered out, 0 ms)
```

Exercise

- Make a first test.
- Let it fail.
- Make it pass.

Cheat Sheet

```
/* in CheatSheetTest.cpp */
#include "CppUTest/TestHarness.h"

/* Declare TestGroup with name CheatSheet */
TEST_GROUP(CheatSheet)
{
/* declare a setup method for the test group. Optional. */
    void setup ()
    {
/* Set method real_one to stub. Automatically restore in teardown */
        UT_PTR_SET(real_one, stub);
    }

/* Declare a teardown method for the test group. Optional */
    void teardown()
    {
    }
}; /* Do not forget semicolumn */

/* Declare one test within the test group */
TEST(CheatSheet, TestName)
{
    /* Check two longs are equal */
    LONGS_EQUAL(1, 1);

    /* Check a condition */
    CHECK(true == true);

    /* Check a string */
    STRCMP_EQUAL("HelloWorld", "HelloWorld");
}
```

```
/* In allTest.cpp */
IMPORT_TEST_GROUP(CheatSheet);

/* In main.cpp */

#include "CppUTest/CommandLineTestRunner.h"
#include "AllTests.h"

int main(int ac, char** av)
{
  return CommandLineTestRunner::RunAllTests(ac,
av);
}
```

TDD in C

Use C or C++?

- Why C++ (e.g. gcc):
  - Able to use C++ unit test framework
  - Able to use C++ features in tests
- Why C:
  - Not annoyed by the small differences
  - Able to use a C compiler.
    - E.g. run tests in “real environment”

Compilation

- Fast build:
  - Limit dependencies - Especially header dependencies!
  - Incremental build - Generate dependency files
  - Compile modules/subsystems
- Execute tests in Makefile!
- Without fast compile: TDD very hard

Refactoring

- All manual -> Almost no refactoring tools
  - Eclipse CDT has some support
    - But be careful. They break stuff.
  - xRefactory emacs plugin (not tried it)
- Function to function pointer refactoring.

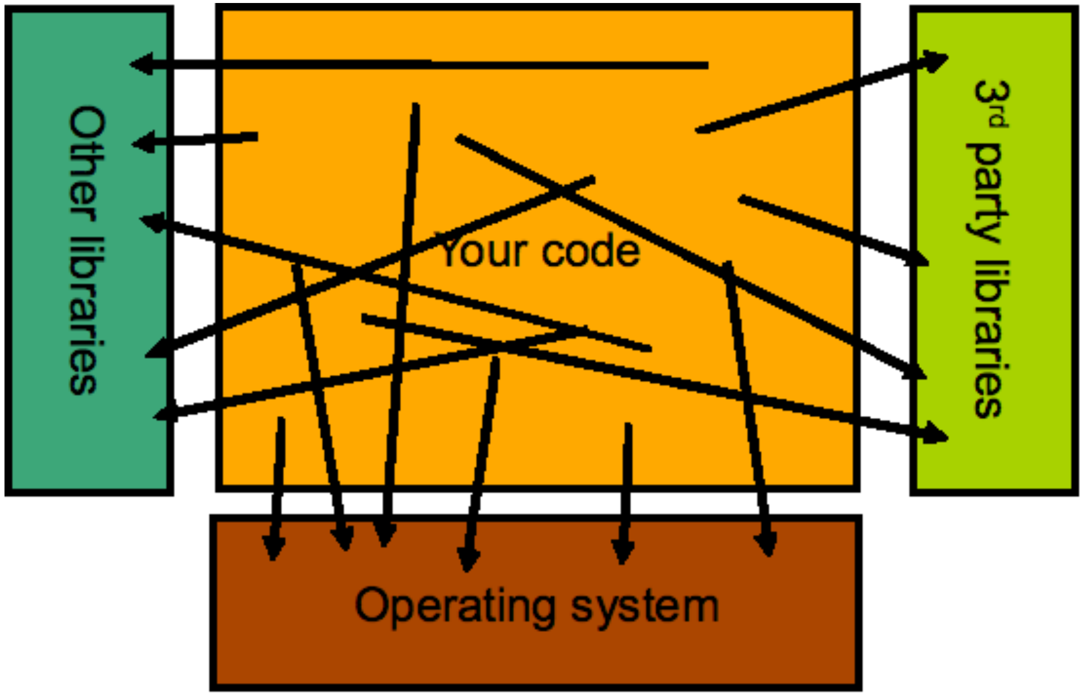
TDD Cycle

- Same cycle
- Biggest problems:
  - Lack of refactoring
  - Slow cycle
    - use prediction

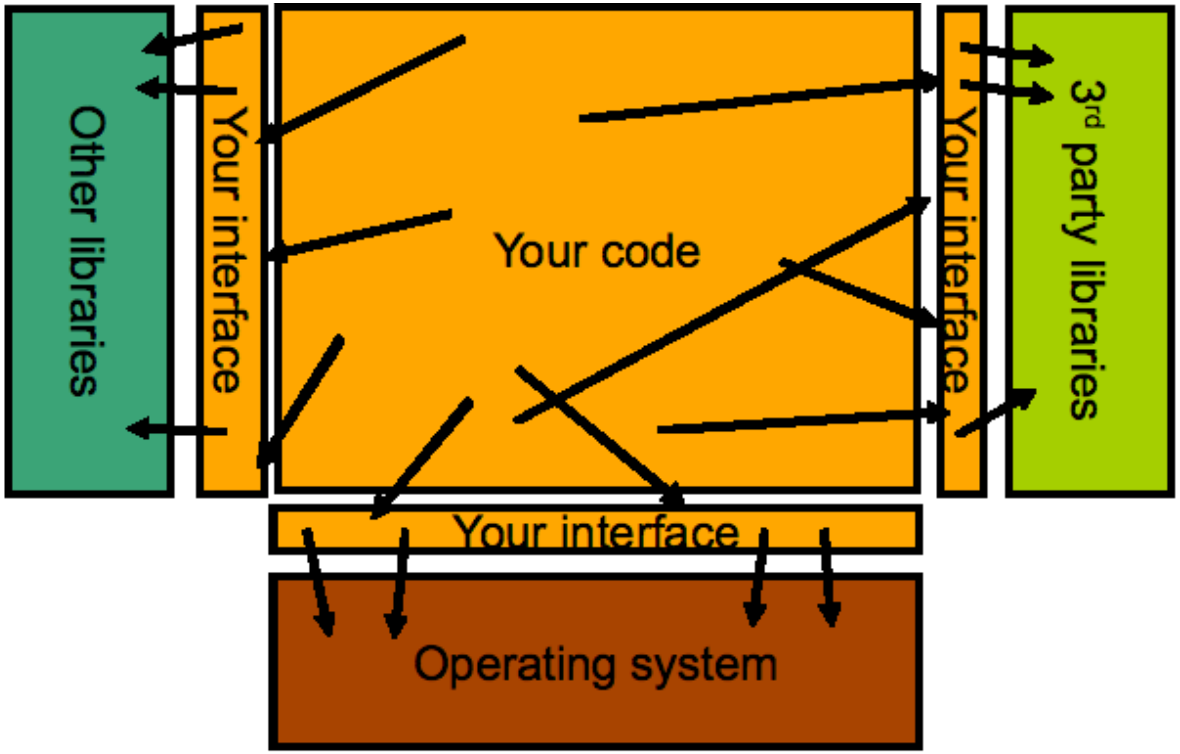
Object Storage

- Exercise:
    - Object Storage module.
    - Allocates a block of memory of “object number” times “object size”
    - Reserves and releases objects
    - Provides fast index-ed find based on 2 integers

Badly structured



Dependencies separated



C Design

- C can be used as OO language!
  - Good written C is OO
- OO techniques
  - Structs with Function Pointers
  - Class-structs
  - Global function pointers

Structs with function pointers

```
struct A
{
    void (*openA)(struct A* a);
    void (*closeA)(struct A* a);

// Private
    int member;
    int anotherMember;
};
```

Takes much memory per object

Class struct

```
struct classA
{
    void (*open)(struct A* a);
    void (*close)(struct A* a);
};

struct A
{
    struct classA * cls;
// Private
    int member;
    int anotherMember;
};

#define A_open(a) (((struct A*)a)->cls->open(a))
#define A_close(a) (((A*)a)->cls->close(a))
```

Better. Much work though

Global function pointers

Header

```
struct A
{
    int member;
    int anotherMember;
};

extern void (*a_open)(struct A*);
extern void (*a_close)(struct A*);
```

Source

```
void a_open_imp(struct A*)
{
    printf("A Open\n");
}

void (*a_open)(struct A*) = a_open_imp;
```

Simple and allows dynamic stubbing and objects.

Very limited though

f2fp refactoring

Header

Source

```
void function (int para);
```

```
void function (int para)
{
    do_implementation ();
}
```

before

```
extern void (*function) (int para);
```

```
void function_imp (int para)
{
    do_implementation ();
}

void (*function)(int para) = function_imp;
```

after

Stubbing

- Stub level:
  - Preprocessor (rare)
  - Function pointer
  - Link
- Stub type
  - Recording
  - Generic
  - Exploding

Level: preprocessor

Source

```
#include "stubs.h"

void something ()
{
    function(100);
}
```

Stub header

```
#define function(a) function_stub(a, b)
```

Advantages:

- Creates lots of flexibility.

  -> for example. Can stub out just one call.

Disadvantages:

- Changes the production code

- Requires different build configurations

Level: function pointer

```
TEST_GROUP(group)
{
    void setup ()
    {
        UT_PTR_SET(real_one, stub);
    }
```

Advantages:

- Ability to runtime change

- Pretty safe

Disadvantages:

- Requires f2fp refactoring

- One extra call

- Dangling function points (use UT_PTR_SET)

Level: link stubs

```
void function ()
{
    /* Do stubbed things */
}
```

Advantages:

- Can give a lot of flexibility!

- No change in production code

- Easy to re-use stubs

Disadvantages:

- Be careful of different configurations. Just have one stub (use generic link stubs)

- Difficult (impossible) to call “the real thing”

Type: Exploding

```
void function ()
{
   FAIL("stub: function was called");
}
```

Especially useful when starting. When it explodes something went wrong or you need to implement it.

Type: Dynamic

```
void (*function_stub) () = NULL;

void function ()
{
    if (function_stub)
        function_stub ();
}
```

Benefits of both function pointer and link level stubs!

Type: Recording

```
struct function_call
{
    static int num_calls;
    int in_parameter1;
    int return_value;
    function_call* next;
};

function_call* g_function_call;

int function (int parameter)
{
    if (g_function_call) {
        g_function_call->num_calls++;
        g_function_call->in_parameter1 = parameter;
        int ret_value = g_function_call->return_value;
        g_function_call = g_function_call->next;
        return ret_call;
    }
}
```

Very generic usage.
More work.

Type: Combination

Very flexible.

```
int function (int parameter)
{
    if (function_stub)
        return function_stub(parameter);

    if (g_function_call) {
        g_function_call->num_calls++;
        g_function_call->in_parameter1 = parameter;
        int ret_value = g_function_call->return_value;
        g_function_call = g_function_call->next;
        return ret_call;
    }

    FAIL("Forgot to set stub or function_call struct.");
}
```

Hello World!

- Exercise:
  - Test-drive “Hello World!”

Embedded TDD

The Real Thing?

- Run unit tests on real hardware?
  - Probably not. Too slow.
  - Every now and then it could be possible.
- Use the real compiler?
  - Often not.
- Do run higher level tests on real HW every now and then!

Design

- Separate hardware dependencies
- Separate OS
- Separate asm from C
- Function pointer vs link stubs
- Static vs dynamic vs ‘dynamic static’ memory allocation

Yes, it IS different

- Integer size
- Endian
- Different compiler -> different binary code
- Bottlenecks (profile on dev env... but be careful)

Exercises

Chat Client-Server

- Exercise:
  - Multiple-clients connect to server
  - Message send to server are distributed to clients
  - Messages are printed
  - POSIX sockets or IPC or ... ?

Lines of Code

- Exercise:
    - Count “lines of code” of C program
    - Ignore the pre-processor