How we implemented TDD in Embedded C & C++

Bluefruit®

Work for Bluefruit based in Cornwall, England.

Provide an embedded software development service.

Introduced Lean/Agile practices in 2009 and have delivered approximately 30 projects since then.

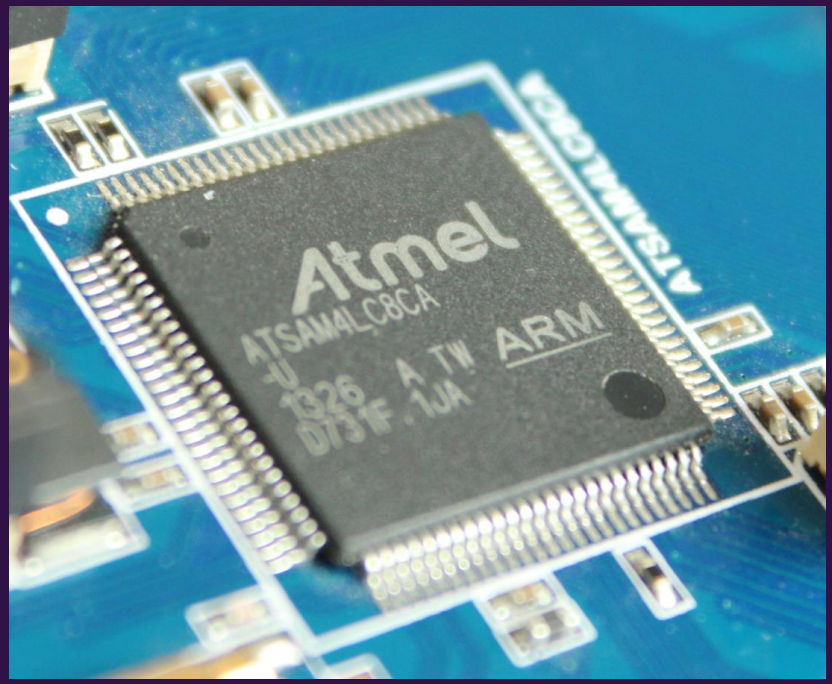
Practices and Patterns we use.

Map data ©Google 2015

What I mean by an Embedded System

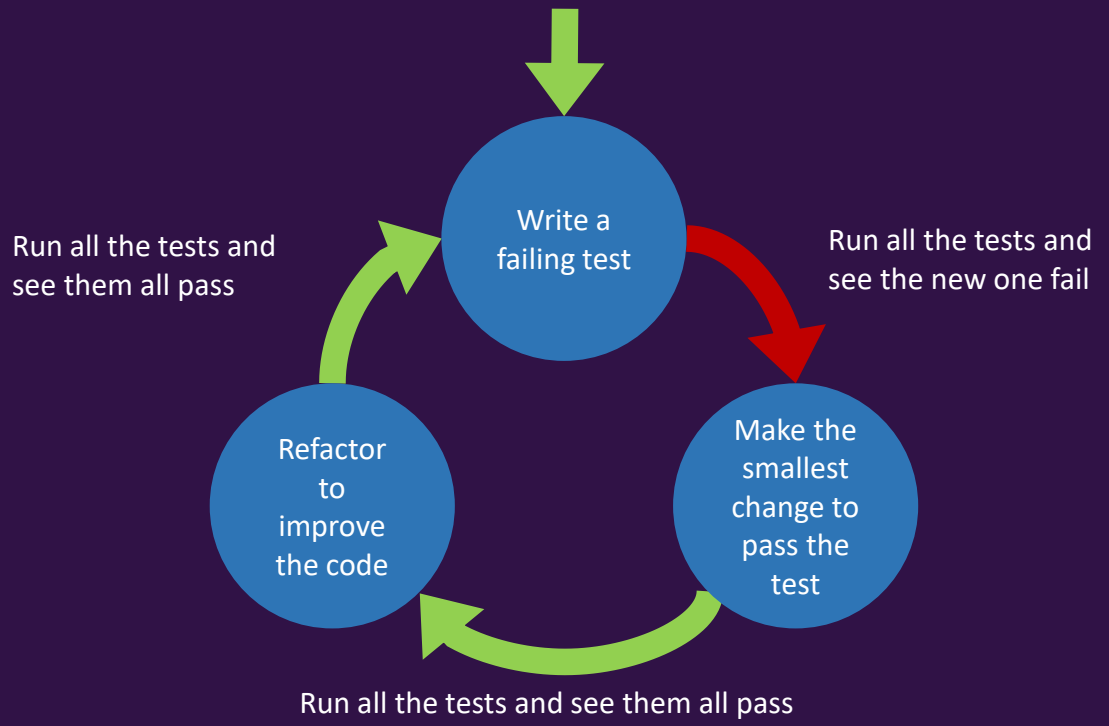
- Embedded System
  - 8MHz - 200MHz Single Core
  - 256B – 512KiB RAM
  - 250KiB Flash

- Samsung Galaxy S7
  - 2.15GHz Dual Core
  - 4GiB RAM
  - Average Android App is 18MiB (32GB Flash on board)



Bluefruit®

Standard TDD Cycle



Fast
Isolated
Repeatable
Self Verifying
Timely

Bluefruit®

Agile in a Flash : http://agileinaflash.blogspot.de/2009/02/first.html

How we achieve FIRST (Contents)

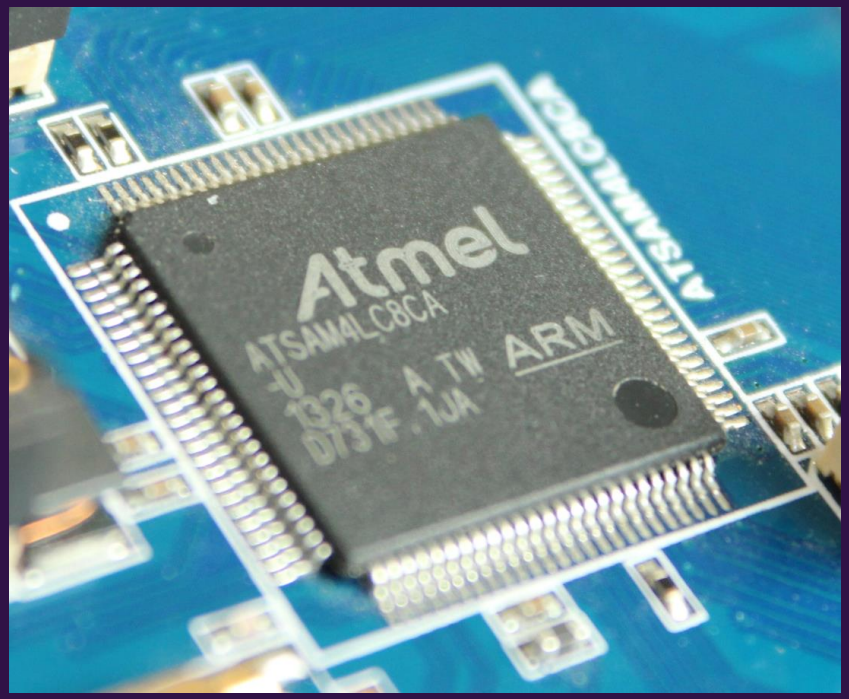
- Where we run our tests to keep them fast
- How TDD Style affects the verification of our tests
- The different methods we use for inserting test doubles to keep our tests isolated and repeatable
- Other practices

Fast
Isolated
Repeatable
Self Verifying
Timely

Where to run the tests?

Test on Target

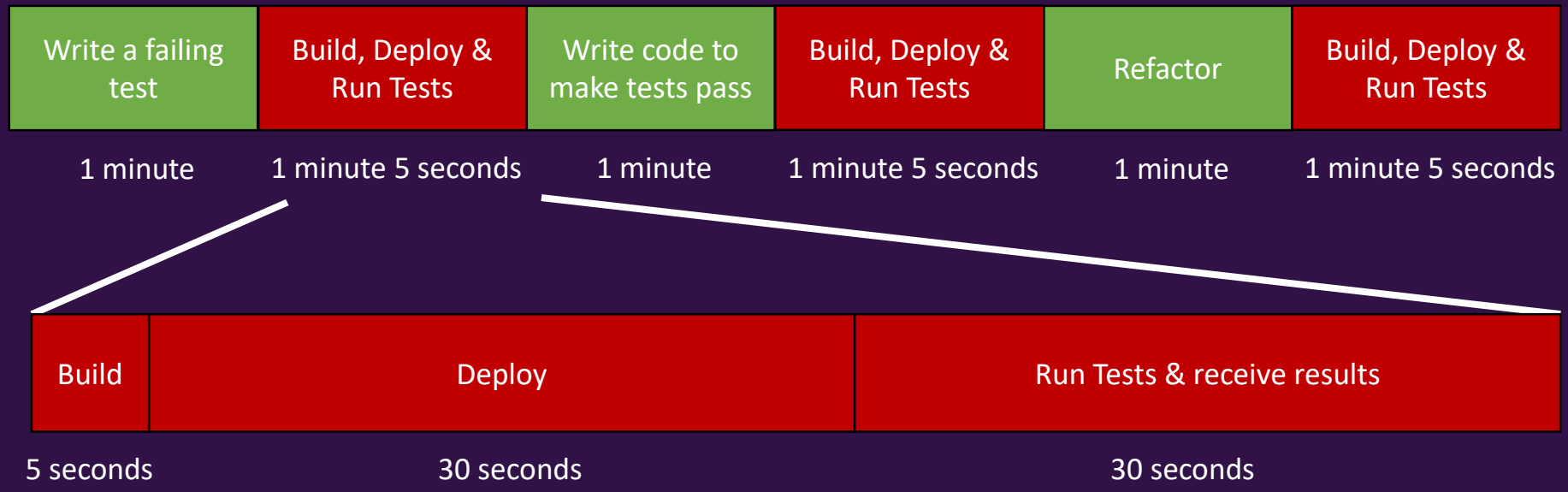
Fast
Isolated
Repeatable
Self Verifying
Timely



Bluefruit®

Analysis of TDD Cycle with Test on Target

Fast
Isolated
Repeatable
Self Verifying
Timely

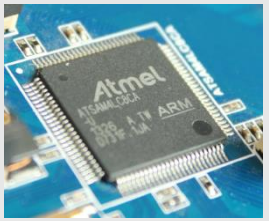


4th Generation Core i7
8GB RAM
SSD

Microchip C32 Compiler
PIC32MX575F512H
MPLAB 8
RealICE

Bluefruit®

Test on Target



Fast
Isolated
Repeatable
Self Verifying
Timely

Advantages

- Accurate test results

Disadvantages

- Slower feedback
  - Programming the target device can be slow
  - The target device is often not fast when compared to modern PCs so the tests will run more slowly
  - Transferring the test results back to the development platform can be slow depending on the method used
  - This will slow down your development process
  - Make you run test less often, leading to bigger changes and more mistakes and missed execution paths
- Limited code space and RAM
  - The tests and the test framework are going to be at least the size of your code if not larger.
- You need target hardware to run the tests
  - Limited hardware – not enough for every development pair
  - Often expensive
  - Sometimes broken

Bluefruit®

We no longer exclusively run tests on the target

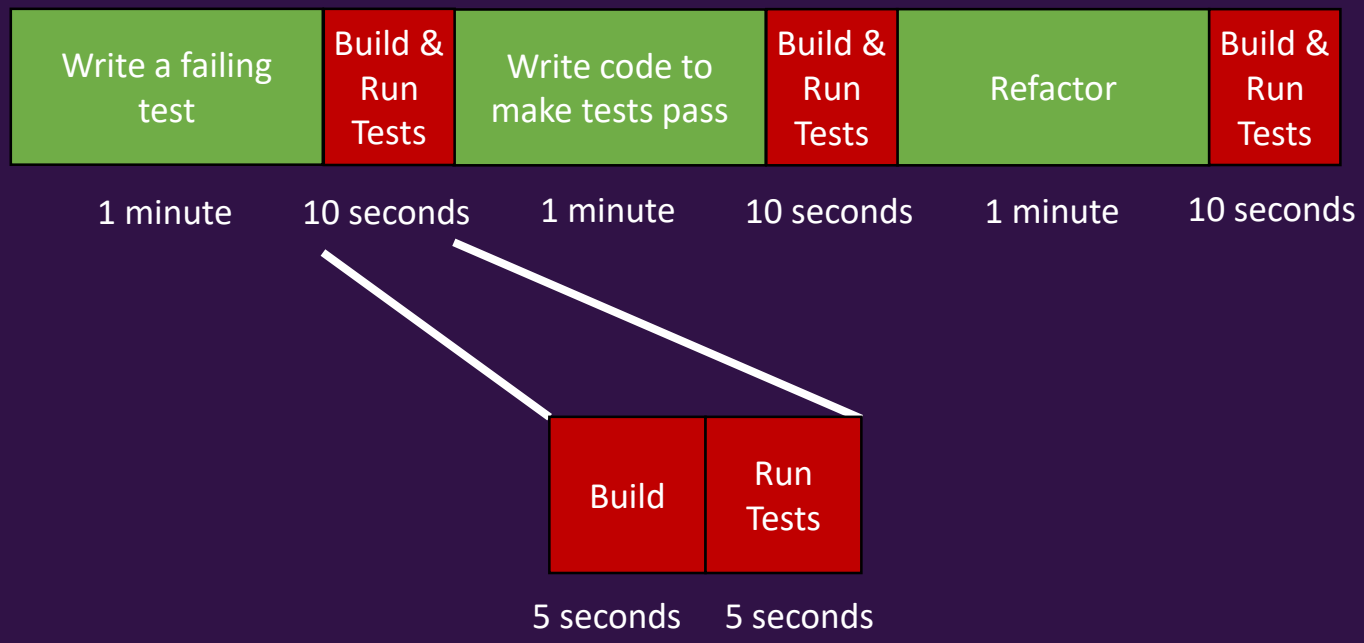
Test on Development Platform

Fast
Isolated
Repeatable
Self Verifying
Timely

Bluefruit®

Analysis of TDD Cycle with Test on Development Platform

Fast
Isolated
Repeatable
Self Verifying
Timely



4th Generation Core i7
8GB RAM
SSD

Visual Studio 2008

Bluefruit®

Test on Development Platform

Fast
Isolated
Repeatable
Self Verifying
Timely

Advantages

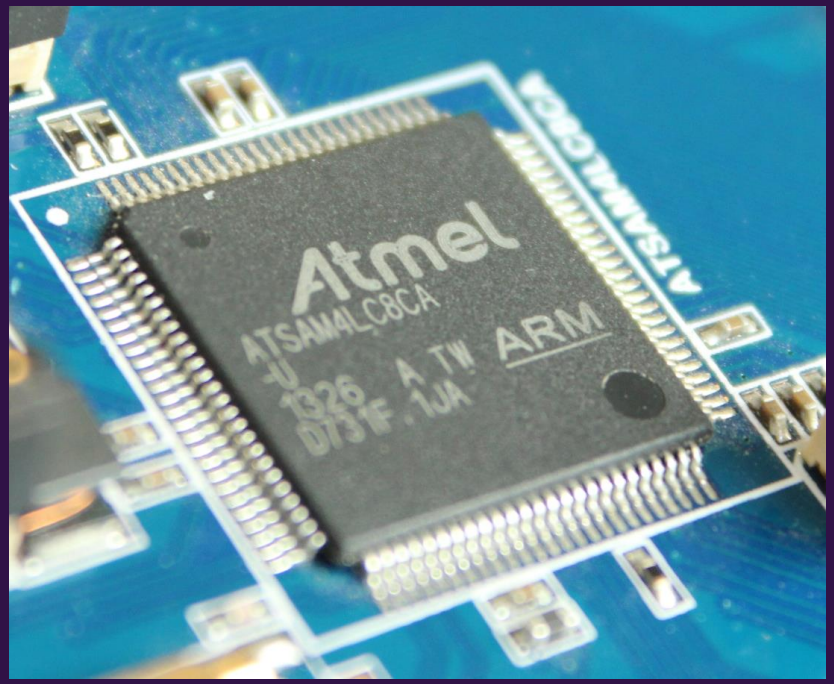
- Fast feedback
- No code space and/or RAM issues
- Reduced the need for target hardware
- More portable code
- Able to write code (in the tests) that may not compile when using the compiler for the target

Disadvantages

- Development platform and target platform are different. Some issues will only happen on the target.
  - E.g. differences in packing, endianness and `sizeof(int)`.
- Able to write code that may not compile when using the compiler for the target

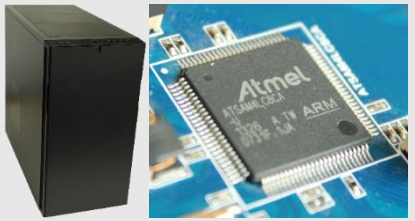
Dual Targeting Tests

Fast
Isolated
Repeatable
Self Verifying
Timely

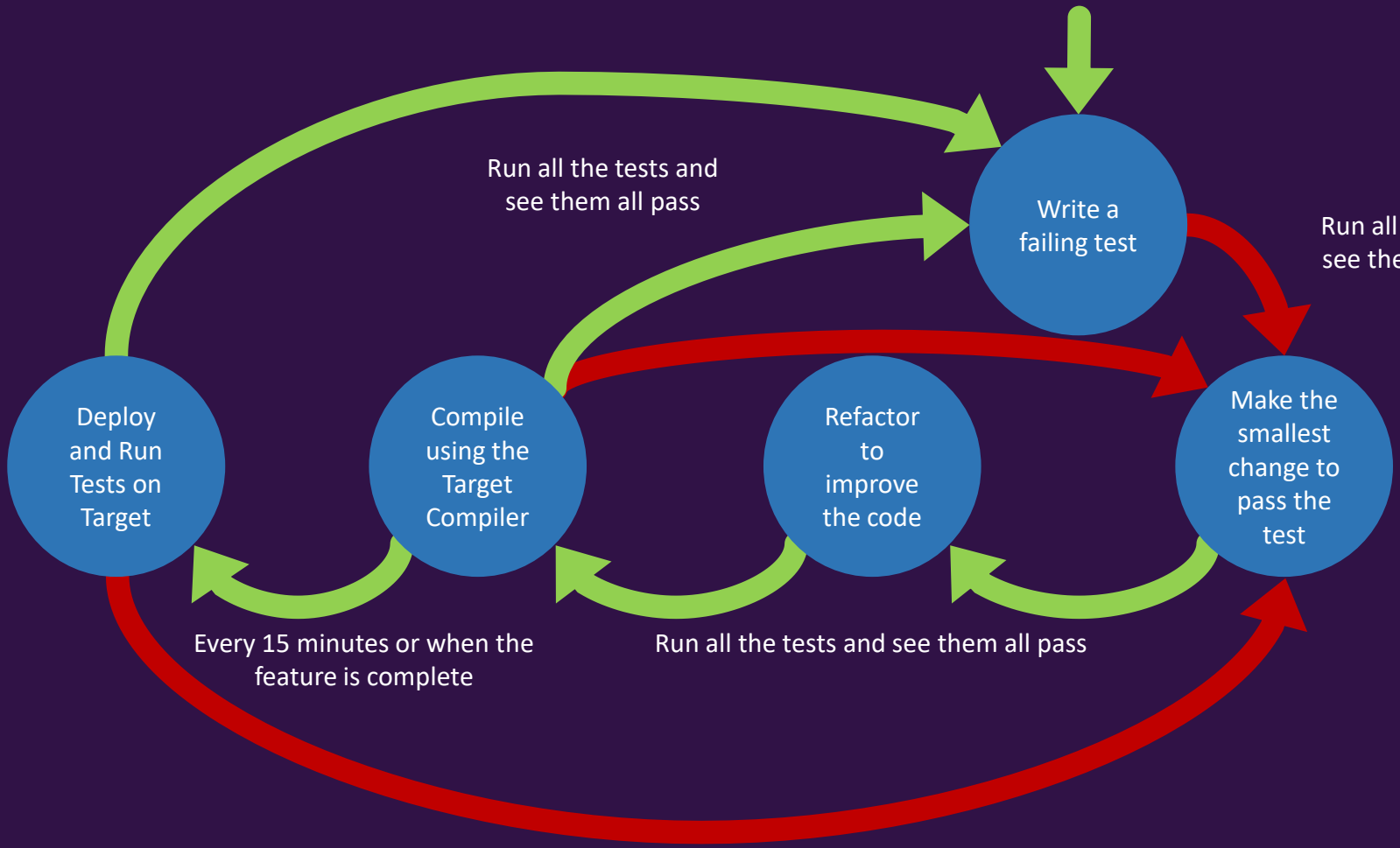


Bluefruit®

Dual Targeting TDD Cycle

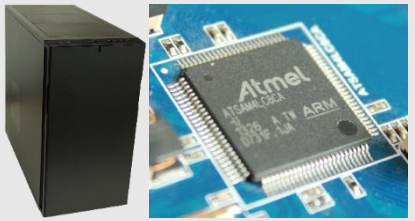


Fast
Isolated
Repeatable
Self Verifying
Timely



Bluefruit®

Dual Targeting



Fast
Isolated
Repeatable
Self Verifying
Timely

Advantages

- Fast feedback
- More portable code
- Compiling on two different compilers increases the chances of catching issues
- Able to run dynamic code analysis (e.g. Memory leak detection & Sanitizers)

Disadvantages

- You need target hardware to run the tests
- You are limited to language features implemented by both compilers
- Maintaining two builds
  - This can be minimised if you can use the same build system and just switch the compiler and linker

Sanitizers

Bluefruit®

Example without Sanitizers

```c
#include <stdlib.h>
#include <stdio.h>

void SortArray(int* array)
{
  free(array);
}

int main(int argc, char** argv)
{
  int* array = calloc(100, sizeof(int));

  SortArray(array);

  printf("%i\n", array[1]);

  return 0;
}
```

```
$ clang -O1 -g code.c && ./a.out
```

Bluefruit

Example output without Sanitizers

```
0
```

Example with Sanitizers

```
 1  #include <stdlib.h>
 2  #include <stdio.h>
 3
 4  void SortArray(int* array)
 5  {
 6    free(array);
 7  }
 8
 9  int main(int argc, char** argv)
10  {
11    int* array = calloc(100, sizeof(int));
12
13    SortArray(array);
14
15    printf("%i\n", array[1]);
16
17    return 0;
18  }
```

```
$ clang -O1 -g -fsanitize=address -fno-omit-frame-pointer
-fno-optimize-sibling-calls code.c && ./a.out
```

Bluefruit

Example output with Sanitizers

```
==4722==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe44 at pc 0x0000004e9483 bp 0x7ffe965baef0 sp 0x7ffe965baee8
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x4e9482 in main /AOTB2016Code/code.c:15:17
    #1 0x7efe3355e82f in __libc_start_main /build/glibc-GKVZIf/glibc-2.23/csu/../csu/libc-start.c:291
    #2 0x417db8 in _start (/AOTB2016Code/a.out+0x417db8)

0x61400000fe44 is located 4 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x4b7d60 in __interceptor_cfree.localalias.0 (/AOTB2016Code/a.out+0x4b7d60)
    #1 0x4e9418 in SortArray /AOTB2016Code/code.c:6:2
    #2 0x4e943f in main /AOTB2016Code/code.c:13:2
    #3 0x7efe3355e82f in __libc_start_main /build/glibc-GKVZIf/glibc-2.23/csu/../csu/libc-start.c:291

previously allocated by thread T0 here:
    #0 0x4b8070 in calloc (/AOTB2016Code/a.out+0x4b8070)
    #1 0x4e9434 in main /AOTB2016Code/code.c:11:15
    #2 0x7efe3355e82f in __libc_start_main /build/glibc-GKVZIf/glibc-2.23/csu/../csu/libc-start.c:291

SUMMARY: AddressSanitizer: heap-use-after-free /AOTB2016Code/code.c:15:17 in main
Shadow bytes around the buggy address:
  0x0c287fff9f70: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f80: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9f90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fa0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fff9fb0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c287fff9fc0: fa fa fa fa fa fa fa fa[fd]fd fd fd fd fd fd
  0x0c287fff9fd0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9fe0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
  0x0c287fff9ff0: fd fd fd fd fd fd fd fd fd fd fa fa fa fa fa fa
  0x0c287fffa000: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c287fffa010: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Heap right redzone:      fb
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack partial redzone:   f4
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
==4722==ABORTING
```

Bluefruit®

Example output with Sanitizers

```
==4722==ERROR: AddressSanitizer: heap-use-after-free on address 0x61400000fe44 at pc
0x0000004e9483 bp 0x7ffe965baef0 sp 0x7ffe965baee8
READ of size 4 at 0x61400000fe44 thread T0
    #0 0x4e9482 in main /AOTB2016Code/code.c:15:17
    #1 0x7efe3355e82f in __libc_start_main /build/glibc-GKVZIf/glibc-2.23/csu/../csu/libc-
start.c:291
    #2 0x417db8 in _start (/AOTB2016Code/a.out+0x417db8)

0x61400000fe44 is located 4 bytes inside of 400-byte region [0x61400000fe40,0x61400000ffd0)
freed by thread T0 here:
    #0 0x4b7d60 in __interceptor_cfree.localalias.0 (/AOTB2016Code/a.out+0x4b7d60)
    #1 0x4e9418 in SortArray /AOTB2016Code/code.c:6:2
    #2 0x4e943f in main /AOTB2016Code/code.c:13:2
    #3 0x7efe3355e82f in __libc_start_main /build/glibc-GKVZIf/glibc-2.23/csu/../csu/libc-
start.c:291

previously allocated by thread T0 here:
    #0 0x4b8070 in calloc (/AOTB2016Code/a.out+0x4b8070)
    #1 0x4e9434 in main /AOTB2016Code/code.c:11:15
    #2 0x7efe3355e82f in __libc_start_main /build/glibc-GKVZIf/glibc-2.23/csu/../csu/libc-
start.c:291
```

Bluefruit®

Fast
Isolated
Repeatable
Self Verifying
Timely

Splitting and testing the solution

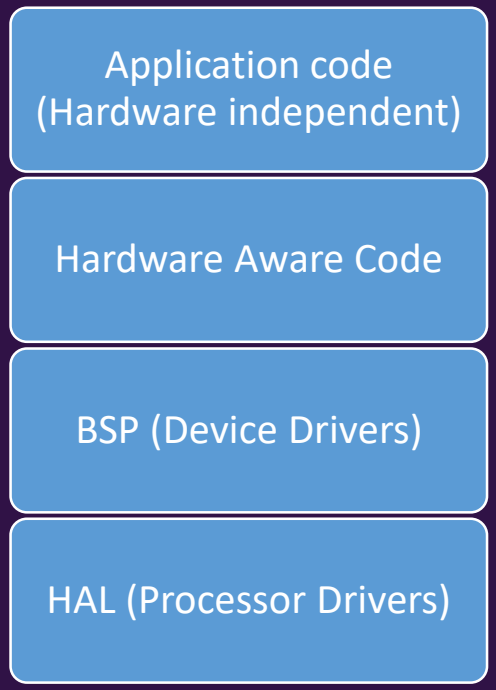
Bluefruit®

A good architecture will make TDD easier

Fast
Isolated
Repeatable
Self Verifying
Timely

We use a simple layered approach

- Low Coupling
- Stick to SOLID principles[1]
  - Single Responsibility Principle
  - Dependency Inversion Principle



Faster to Test

Slower to Test

We have a thin outer (low level) layer that isn't unit tested. This only sets processor registers. (We keep its cyclomatic complexity ≤ 2)

Bluefruit®

1. Agile software development: Principles, Patterns, and Practices – Martin

Running your tests in isolation

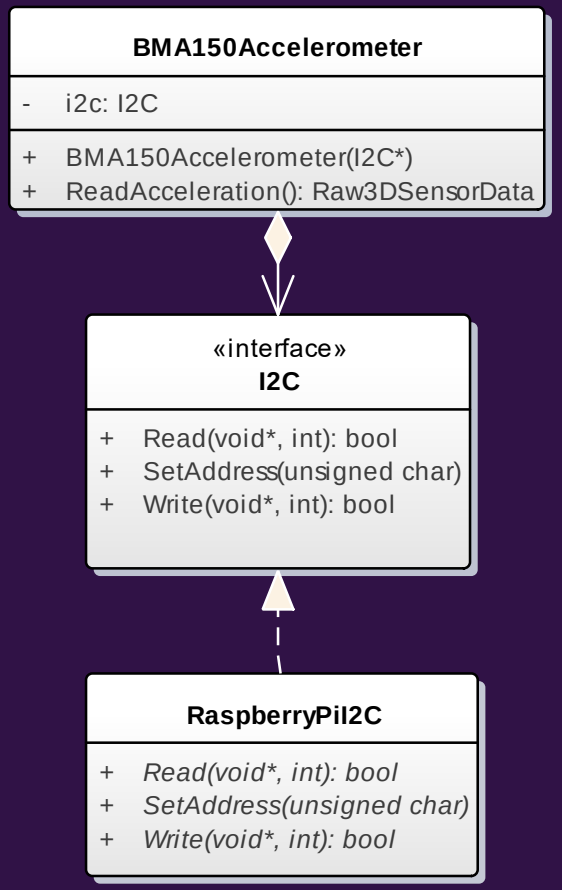
Fast
Isolated
Repeatable
Self Verifying
Timely

To test in isolation your test cannot depend on hardware or something out of your control.

What am I going to replace the dependency with?

Test Double

How am I going to replace the dependency?



Bluefruit®

Test Doubles

Fast
Isolated
Repeatable
Self Verifying
Timely

Stub/Spy – Provide fixed responses to method calls and can record the values they are passed.

Dummy – Used to fulfil a dependency that is not used, they usually consist of empty method definitions.

Fake – Provide a working fake implementation of the dependency. E.g. an in-memory EEPROM

Classical TDD

Mock – Pre-programmed with expected method calls and verifies that they happen.

Mockist TDD

Bluefruit®

Martin Fowler. Mocks aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html

TDD Style

Fast
Isolated
Repeatable
Self Verifying
Timely

I want to fulfil an `Order` object from a `RemovableInventory` that is implemented by a `Warehouse` object

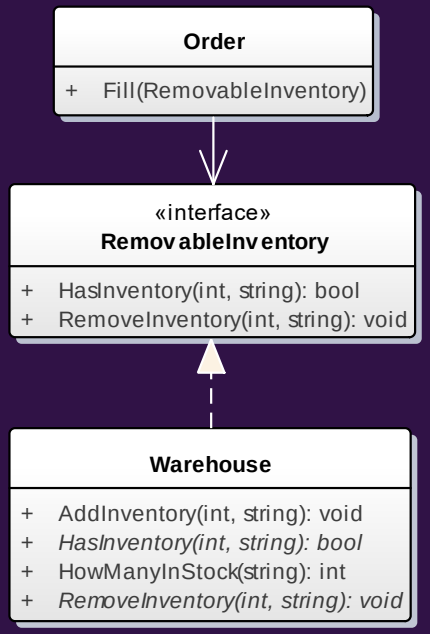
<u>Example Scenario</u>

Given our warehouse has 50 Apples in stock

And an order for 20 Apples

When the order is fulfilled

Then our warehouse has 30 Apples in stock



Bluefruit®

Martin Fowler. Mocks aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html

Classical (Chicago/Detroit) Style State Verification (with Stubs)

Fast
Isolated
Repeatable
Self Verifying
Timely

```
class RemovableInventoryStub : public RemovableInventory {
public:
   int removeNumberOf;
   std::string removeItem;

   RemovableInventoryStub() :  removeNumberOf(0), removeItem("") { }

   virtual bool HasInventory(int numberOf, const std::string &item) const {
      return true;
   }

   virtual void RemoveInventory(int numberOf, const std::string &item) {
      removeNumberOf = numberOf;
      removeItem = item;
   }
};

TEST(Order_ClassicalUsingStub,
     Fulfilling_an_order_removes_the_items_from_the_inventory)
{
   RemovableInventoryStub inventory;

   Order target(20, "Apples");
   target.Fill(inventory);

   EXPECT_EQ(20, inventory.removeNumberOf);
   EXPECT_EQ("Apples", inventory.removeItem);
}
```

Bluefruit®

Martin Fowler. Mocks aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html

Classical (Chicago/Detroit) Style State Verification (using the real object)

Fast
Isolated
Repeatable
Self Verifying
Timely

```
TEST(Order_ClassicalUsingReal,
     Filling_an_order_removes_the_items_from_the_inventory)
{
  Warehouse inventory;
  inventory.AddInventory(50, "Apples");

  Order target(20, "Apples");
  target.Fill(inventory);

  EXPECT_EQ(30, inventory.HowManyInStock("Apples"));
}
```

Bluefruit®

Martin Fowler. Mocks aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html

Mockist (London) Style Behaviour Verification

Fast
Isolated
Repeatable
Self Verifying
Timely

```
class RemovableInventoryMock : public RemovableInventory
{
public:
  MOCK_CONST_METHOD2(HasInventory, bool(int numberOf, const std::string &item));
  MOCK_METHOD2(RemoveInventory, void(int numberOf, const std::string &item));
};

TEST(Order_Mockist, Fulfilling_an_order_removes_the_items_from_the_inventory)
{
  RemovableInventoryMock inventory;

  EXPECT_CALL(inventory, HasInventory(20, "Apples"))
    .Times(1)
    .WillOnce(Return(true));
  EXPECT_CALL(inventory, RemoveInventory(20, "Apples"))
    .Times(1);

  Order target(20, "Apples");
  target.Fill(inventory);
}
```

Bluefruit®

Martin Fowler. Mocks aren't Stubs. http://martinfowler.com/articles/mocksArentStubs.html

TDD Style

Fast
Isolated
Repeatable
Self Verifying
Timely

Classical

Advantages

- Does not specify how the code should work
- Easier to refactor the code

Disadvantages

- Harder to work out what is broken, a single incorrect code change can break many tests
- Can be a trade off between encapsulation and testability. The state might have to be more visible so it can be verified

Mockist

Advantages

- Code changes that break functionality tend to only break the tests that directly relate to them

Advantages/Disadvantages

- You have to think about the implementation when writing tests

Disadvantages

- Tests are coupled to implementation making refactoring harder

How I vary my TDD Style

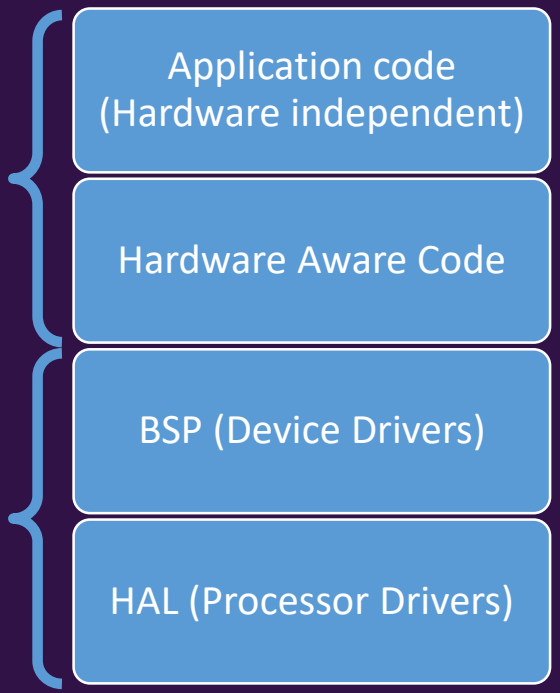
Fast
Isolated
Repeatable
Self Verifying
Timely

I prefer classical testing, because my tests are not coupled to my implementation this allows me to refactor more easily.

Classical Testing – State Verification (Stubs/Fakes/Dummies)

Mockist Testing – Behaviour Verification (Mocks)

The behaviour is usually fixed by the device so using mocks and specifying the behaviour in the tests feels more natural.



Bluefruit®

Running your tests in isolation

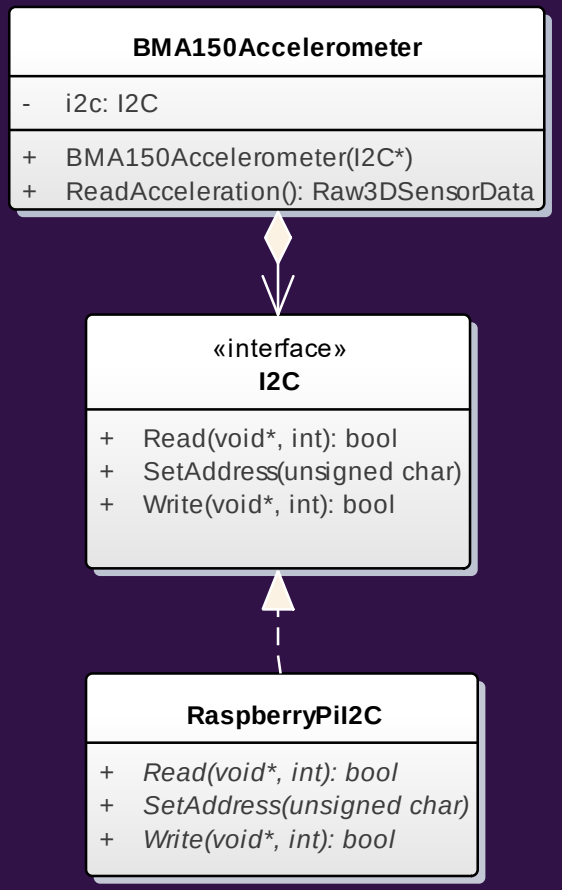
Fast
Isolated
Repeatable
Self Verifying
Timely

To test in isolation your test cannot depend on hardware or something out of your control.

What am I going to replace the dependency with?

Test Double

How am I going to replace the dependency?



Bluefruit®

Where you can insert Test Doubles

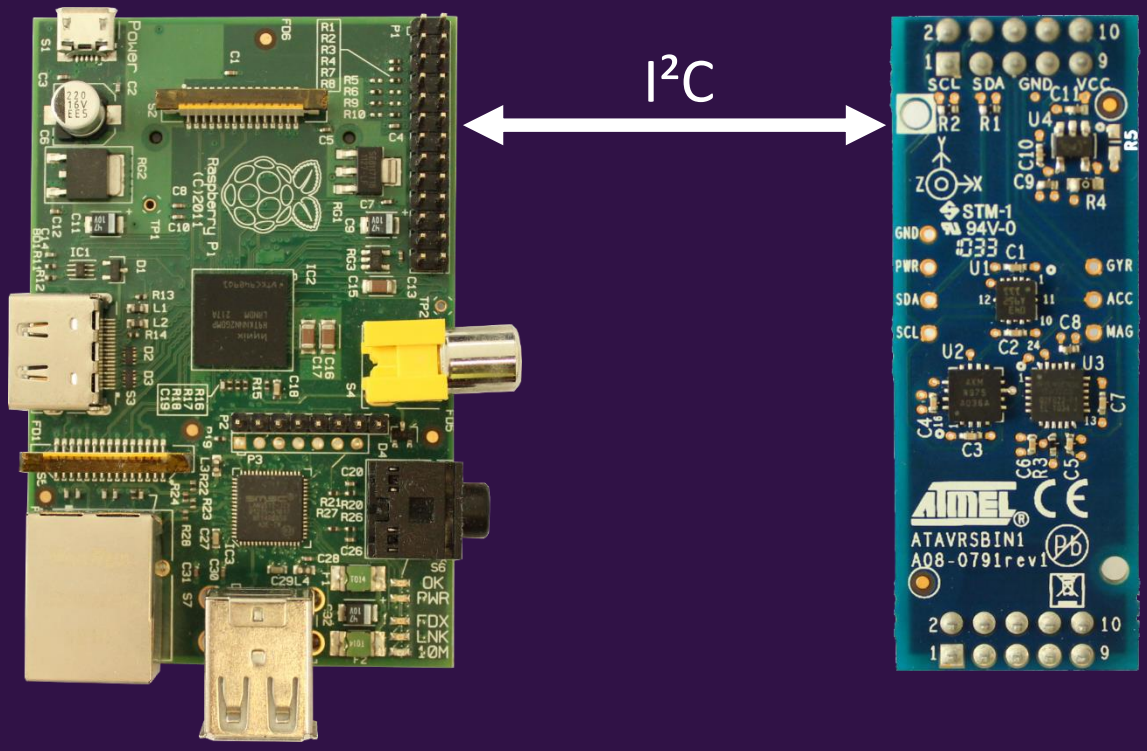
Fast
Isolated
Repeatable
Self Verifying
Timely

Compile time	Link time	Run time
• Macros (C/C++) • Templates (C++) • #includes (C/C++)	• Linking other object files (C/C++) • Weak linking functions (C)	• Interface (C++) • Inheritance (C++) • V-Table (C)

Bluefruit

Test Double Insertion

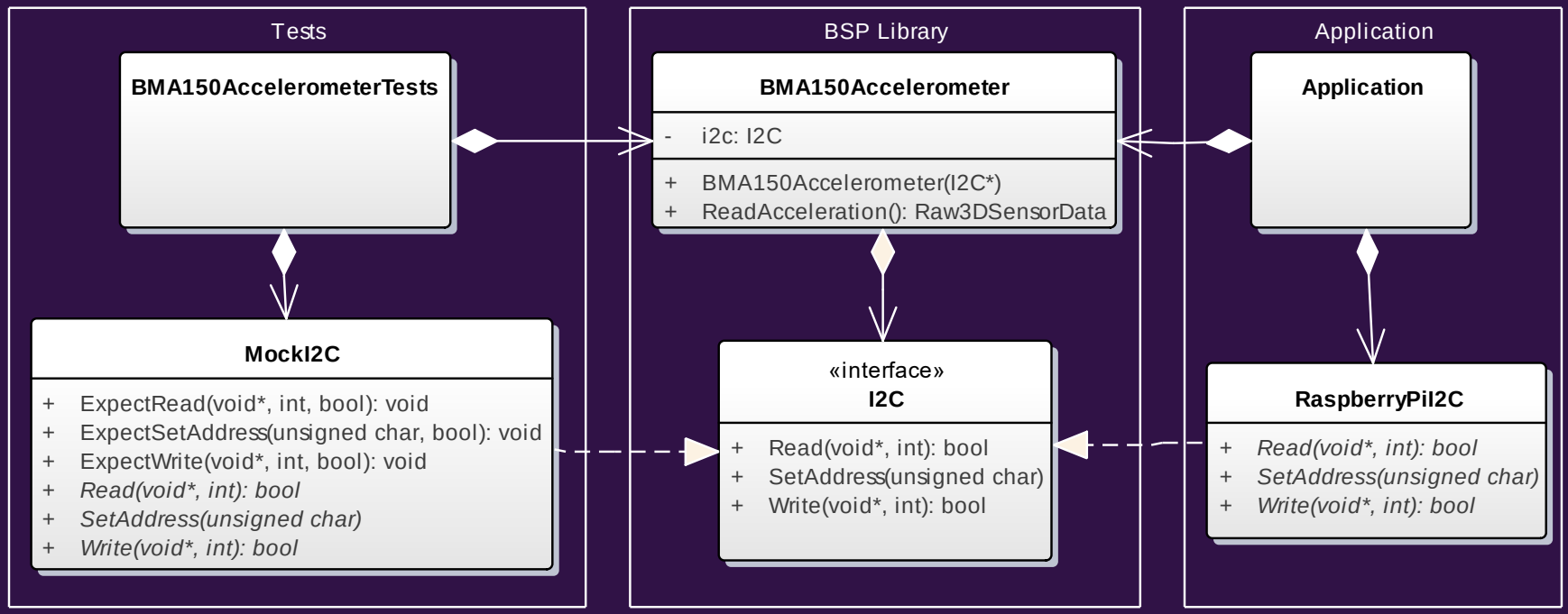
Fast
Isolated
Repeatable
Self Verifying
Timely



Bluefruit®

Test Doubles Insertion

Fast
Isolated
Repeatable
Self Verifying
Timely



Bluefruit

Fast
Isolated
Repeatable
Self Verifying
Timely

C++ Interfaces

We use this technique for everything

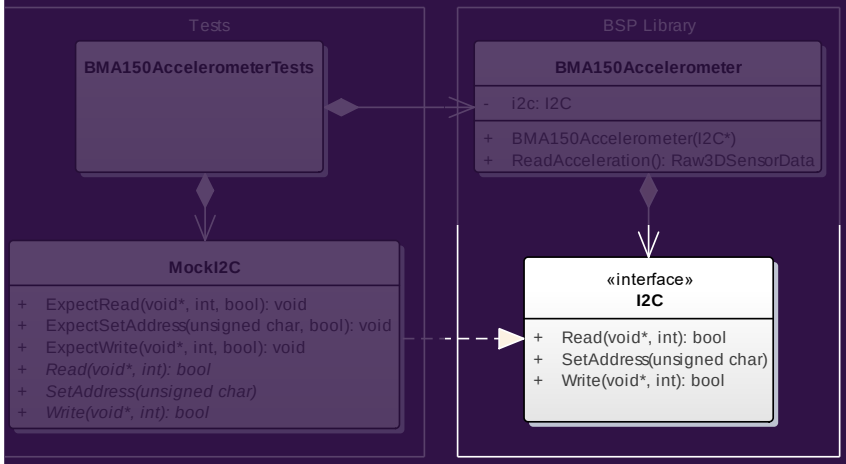
Bluefruit®

Dependency Interface

Test Doubles insertion using C++ Interfaces

Fast
Isolated
Repeatable
Self Verifying
Timely

```
class I2C
{
public:
 virtual ~I2C() { }
 virtual bool SetAddress(unsigned char
                         address) = 0;

 virtual bool Read(void * buffer,
                   int length) = 0;

 virtual bool Write(
                 const void * buffer,
                     int length) = 0;
};
```



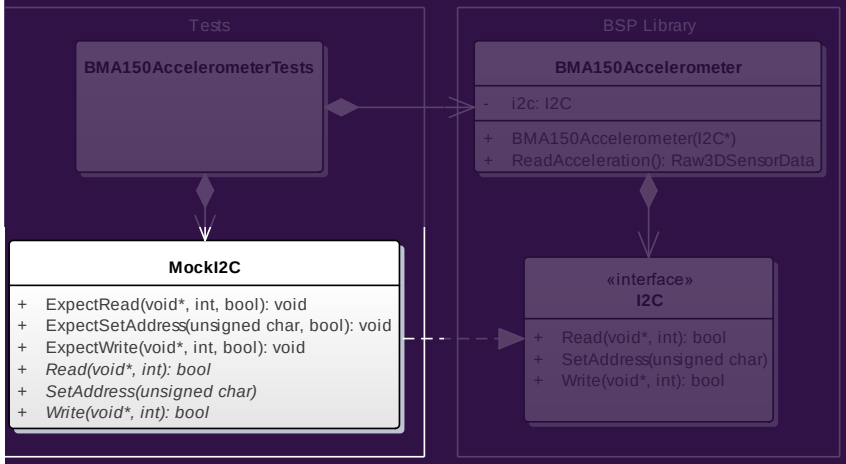
Bluefruit®

Dependency Mock

Test Doubles insertion using C++ Interfaces

Fast
Isolated
Repeatable
Self Verifying
Timely

```
class MockI2C : public I2C
{
public:
  virtual bool SetAddress(
                        unsigned char address);
  virtual bool Read(void * buffer, int length);
  virtual bool Write(const void * buffer,
                     int length);

  void ExpectSetAddress(unsigned char address,
                        bool returnValue);

  void ExpectRead(const void * buffer,
                  int length, bool returnValue);

  void ExpectWrite(const void * buffer,
                  int length, bool returnValue);

  void Verify();
  virtual ~MockI2C() { Verify(); }
};
```



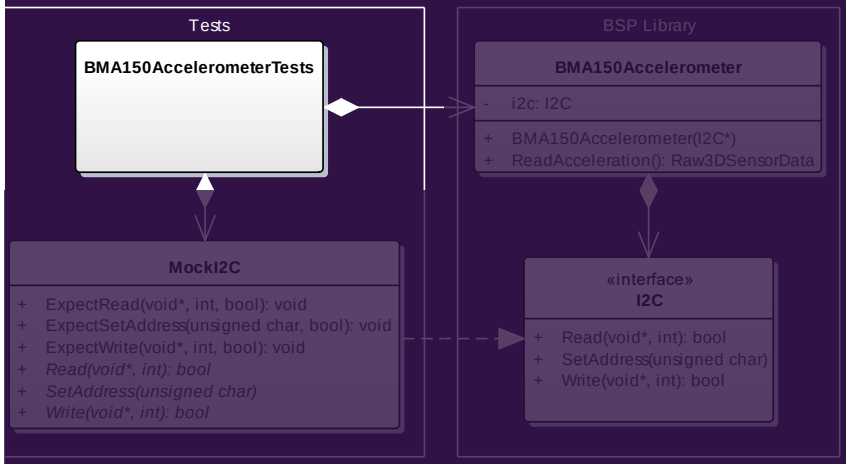
Bluefruit®

Test

Test Doubles insertion using C++ Interfaces

Fast
Isolated
Repeatable
Self Verifying
Timely

```
void testBMA150Accelerometer_Reading_an_acceleratio
n_of_0()
{
  // Given
  MockI2C i2c;

  const unsigned char readCommand[] = { 0x02 };
  const unsigned char readData[] =
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

  i2c.ExpectSetAddress(deviceAddress, true);
  i2c.ExpectWrite(readCommand, sizeof(readCommand),
                  true);
  i2c.ExpectRead(readData, sizeof(readData), true);

  // When
  BMA150Accelerometer target(&i2c);
  Raw3DSensorData result =
                       target.ReadAcceleration();

  // Then
  TEST_ASSERT_EQUAL(0, result.x);
  TEST_ASSERT_EQUAL(0, result.y);
  TEST_ASSERT_EQUAL(0, result.z);
}
```



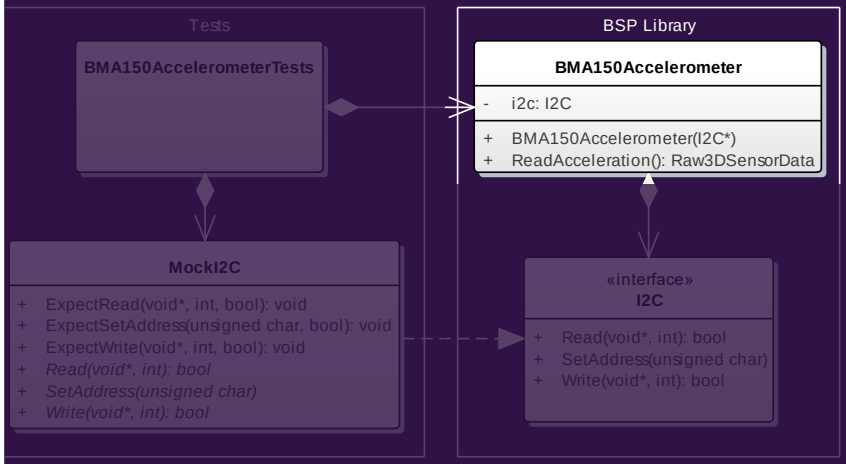
Bluefruit®

Code (System under test)

Test Doubles insertion using C++ Interfaces

Fast
Isolated
Repeatable
Self Verifying
Timely

```
class BMA150Accelerometer
{
private:
  I2C *i2c;
public:
  explicit BMA150Accelerometer(I2C *i2cPort)
    : i2c(i2cPort)
  { }

  Raw3DSensorData ReadAcceleration() const;
  {
    const unsigned char BMA150Address = 0x38;
    i2c->SetAddress(BMA150Address);

    const unsigned char registerAddress[] = { 0x02 };
    i2c->Write(registerAddress, sizeof(registerAddress));

    Raw3DSensorData rawAcceleration;
    i2c->Read(&rawAcceleration, sizeof(rawAcceleration));

    return rawAcceleration;
  }
};
```



Bluefruit®

Test Doubles insertion using C++ Interfaces

Fast
Isolated
Repeatable
Self Verifying
Timely

Advantages

- Easiest method of inserting Test Doubles

Disadvantages

- Virtual function calls are slower than directly calling a method
- The V Table will take up space (either RAM or ROM)

We use this technique for everything

Fast
Isolated
Repeatable
Self Verifying
Timely

Linking other object files

We use it as a last resort when virtual function calls are too expensive

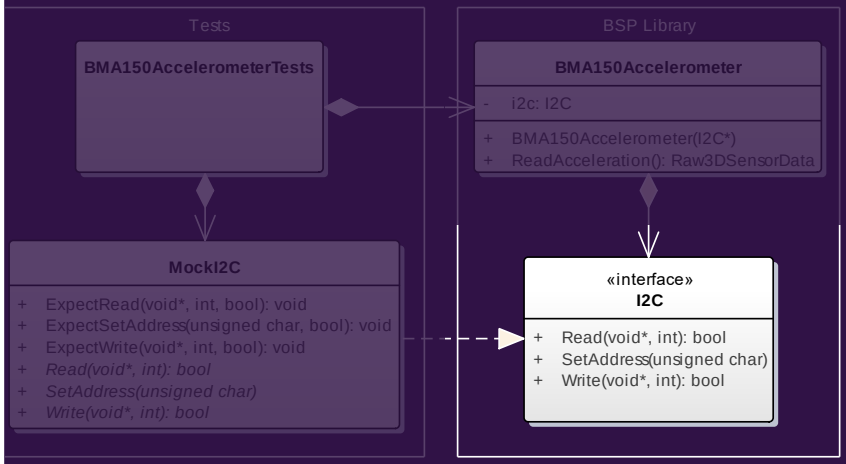
The example code is in C but this technique works in C++ as well

Dependency Interface

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

```
#ifndef I2C_H
#define I2C_H

#include <stdbool.h>

bool I2C_SetAddress(
                unsigned char address);

bool I2C_Read(void * buffer,
              int length);

bool I2C_Write(const void * buffer,
               int length);

#endif
```

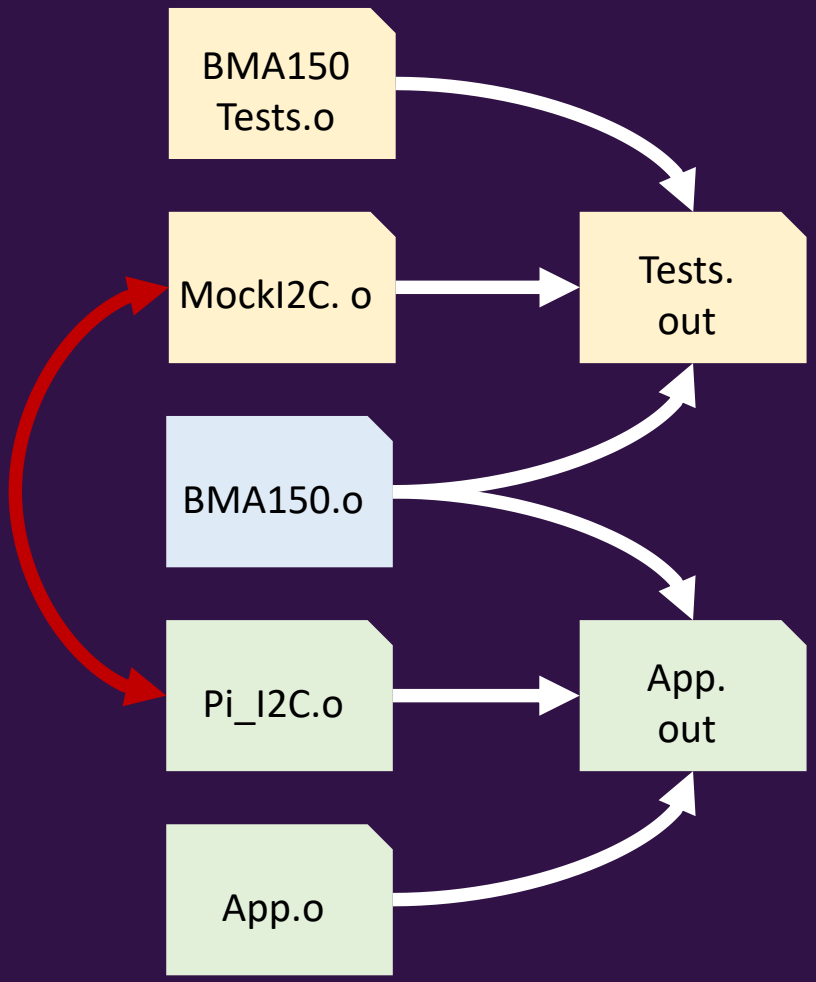


Bluefruit®

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

Contain different definitions of the same methods



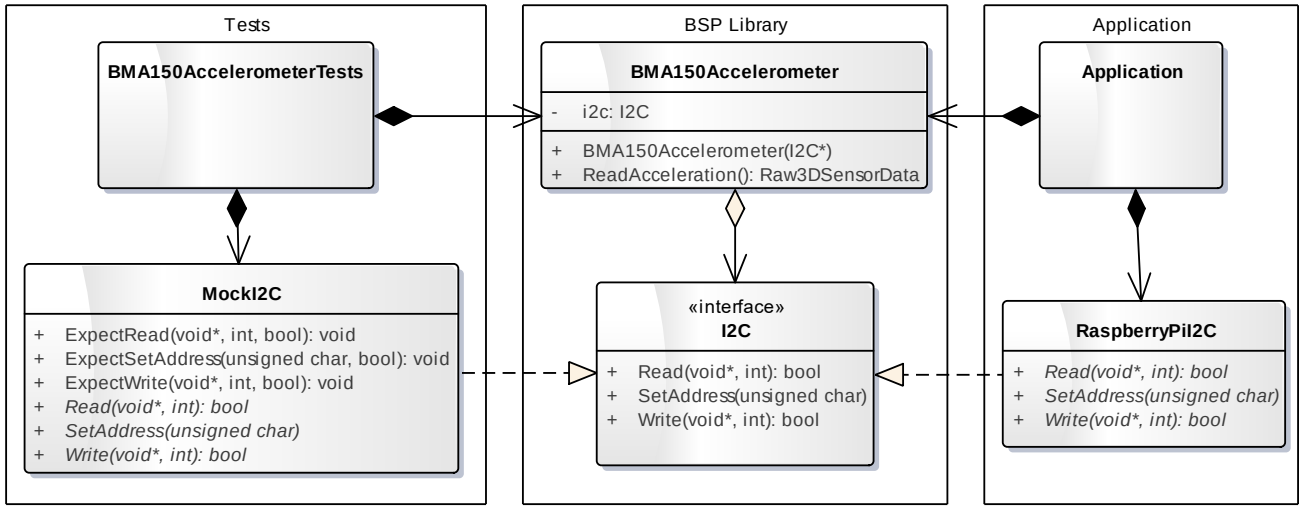
Bluefruit

Makefile

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

```
# tests
#------
tests : BMA150AccelerometerTests.o MockI2C.o BMA150Accelerometer.o
    $(CC) $(CFLAGS) $^ -o $@

# application
#------------
application : main.o RaspberryPiI2C.o BMA150Accelerometer.o
    $(CC) $(CFLAGS) $^ -o $@
```



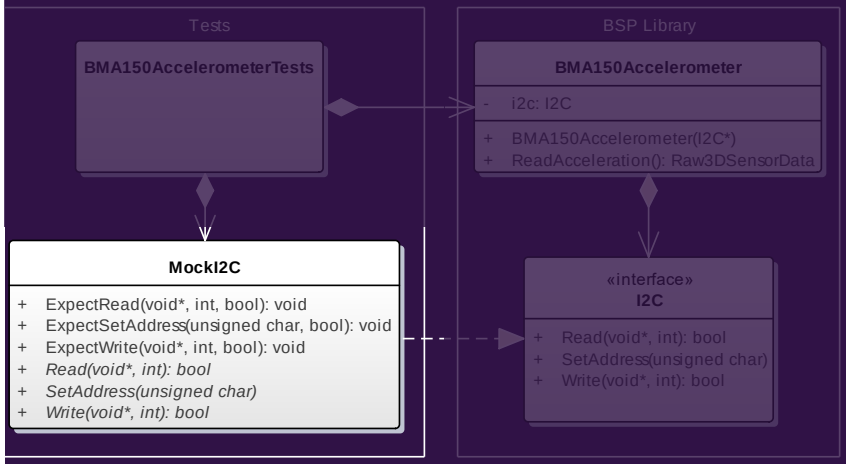
Bluefruit

Dependency Mock

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

```
bool I2C_SetAddress(
                unsigned char address)
{
  // ...
}

void MockI2C_ExpectSetAddress(
                unsigned char address,
                     bool returnValue)
{
  // ...
}

void MockI2C_Verify(void)
{
  // ...
}
```



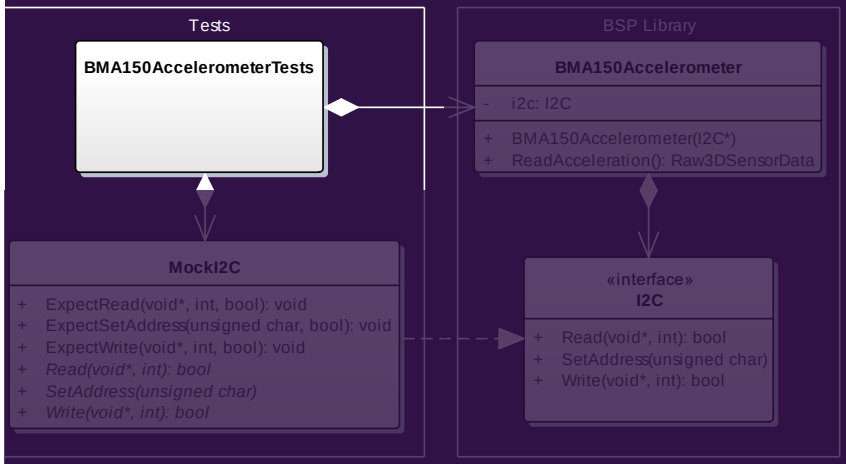
Bluefruit®

Test

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

```
void testBMA150Accelerometer_Reading_an_acceleratio
n_of_0(void)
{
  // Given
  const unsigned char readCommand[] = { 0x02 };
  const unsigned char readData[] =
            { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

  MockI2C_ExpectSetAddress(deviceAddress, true);
  MockI2C_ExpectWrite(readCommand,
                      sizeof(readCommand), true);
  MockI2C_ExpectRead(readData,
                     sizeof(readData), true);

  // When
  struct Raw3DSensorData result =
            BMA150Accelerometer_ReadAcceleration();

  // Then
  MockI2C_Verify();
  TEST_ASSERT_EQUAL(0, result.x);
  TEST_ASSERT_EQUAL(0, result.y);
  TEST_ASSERT_EQUAL(0, result.z);
}
```



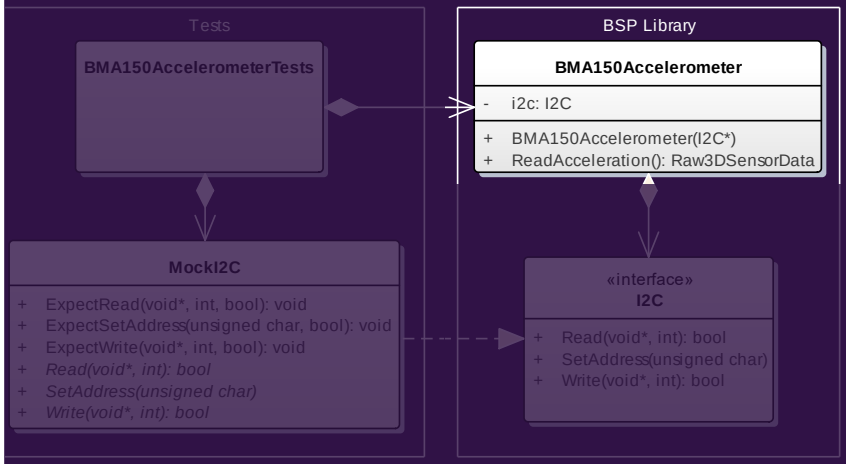
Bluefruit®

Code (System under test)

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

```
struct Raw3DSensorData
  BMA150Accelerometer_ReadAcceleration(void)
{
  const unsigned char BMA150Address = 0x38;
  I2C_SetAddress(BMA150Address);

  const unsigned char registerAddress[] =
                                  { 0x02 };
  I2C_Write(registerAddress,
            sizeof(registerAddress));

  struct Raw3DSensorData rawAcceleration;
  I2C_Read(&rawAcceleration,
           sizeof(rawAcceleration));

  return rawAcceleration;
}
```



Bluefruit

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

Advantages

- No virtual function calls

Disadvantages

- Adds complexity to the build system

We use this technique

- As a last resort when virtual function calls are too expensive. We profile the calls first to see what is causing the problem

Test Double Insertion Techniques
When we use them

Fast
Isolated
Repeatable
Self Verifying
Timely

C++ Interfaces – For everything

- Easiest method of inserting test doubles

Linking other object files – When virtual function calls are too expensive

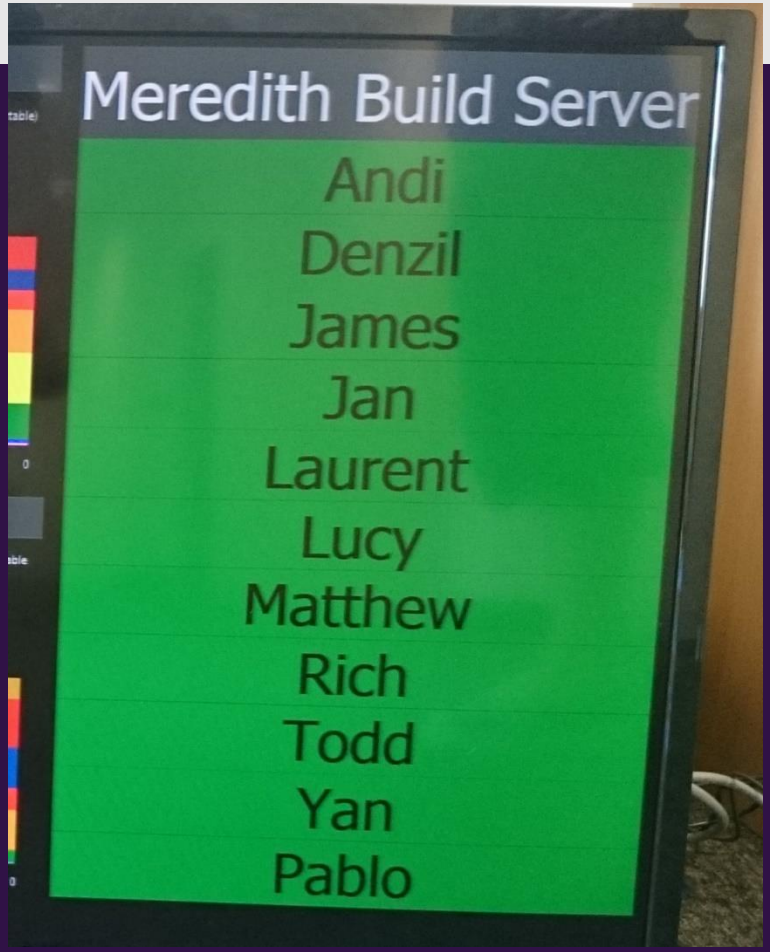
- Removes the performance hit from making virtual function calls

What else?

Bluefruit®

Other practices

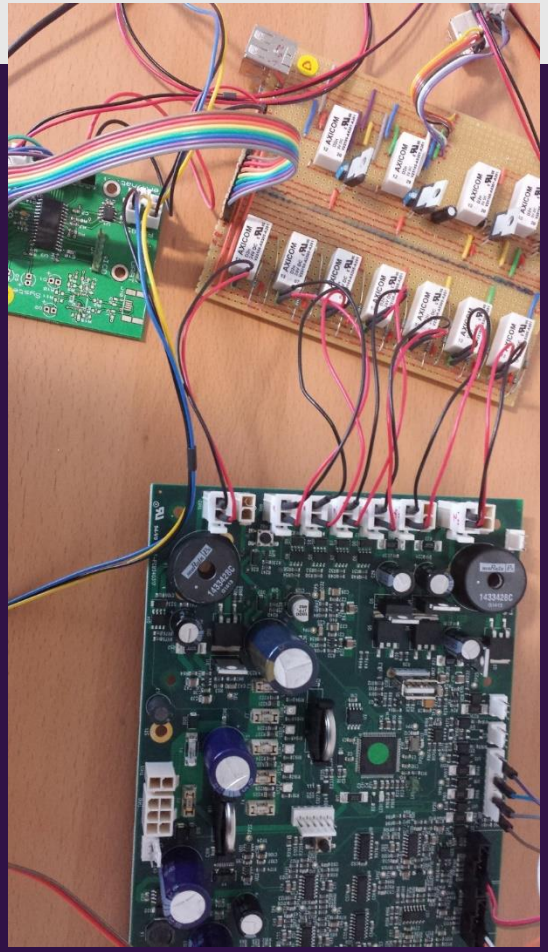
- When hardware is in short supply we use our Build server to run tests on the target platform



Bluefruit®

Other practices

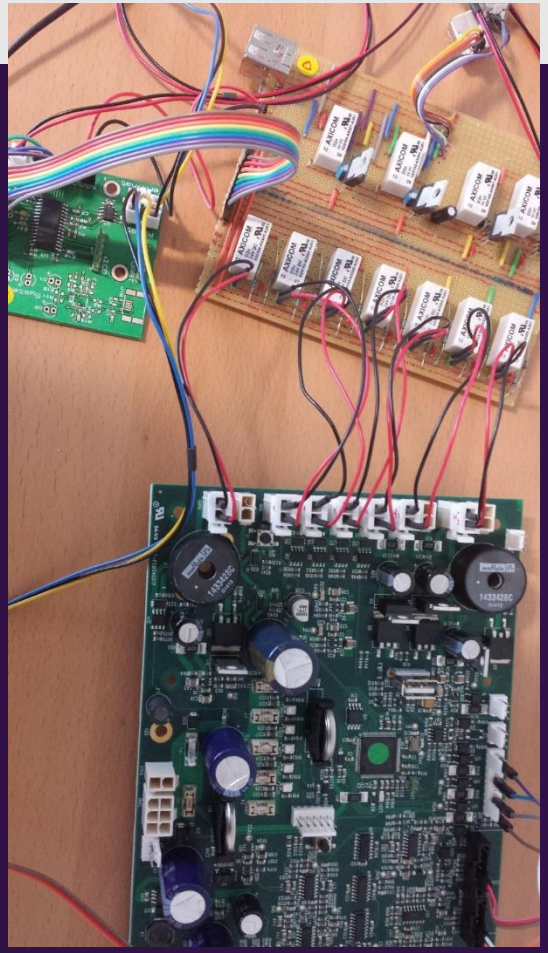
- When hardware is in short supply we use our Build server to run tests on the target platform
- Integration Tests that check hardware interaction



Bluefruit®

Other practices

- When hardware is in short supply we use our Build server to run tests on the target platform
- Integration Tests that check hardware interaction
- System Testing



Bluefruit®

Faster System Testing

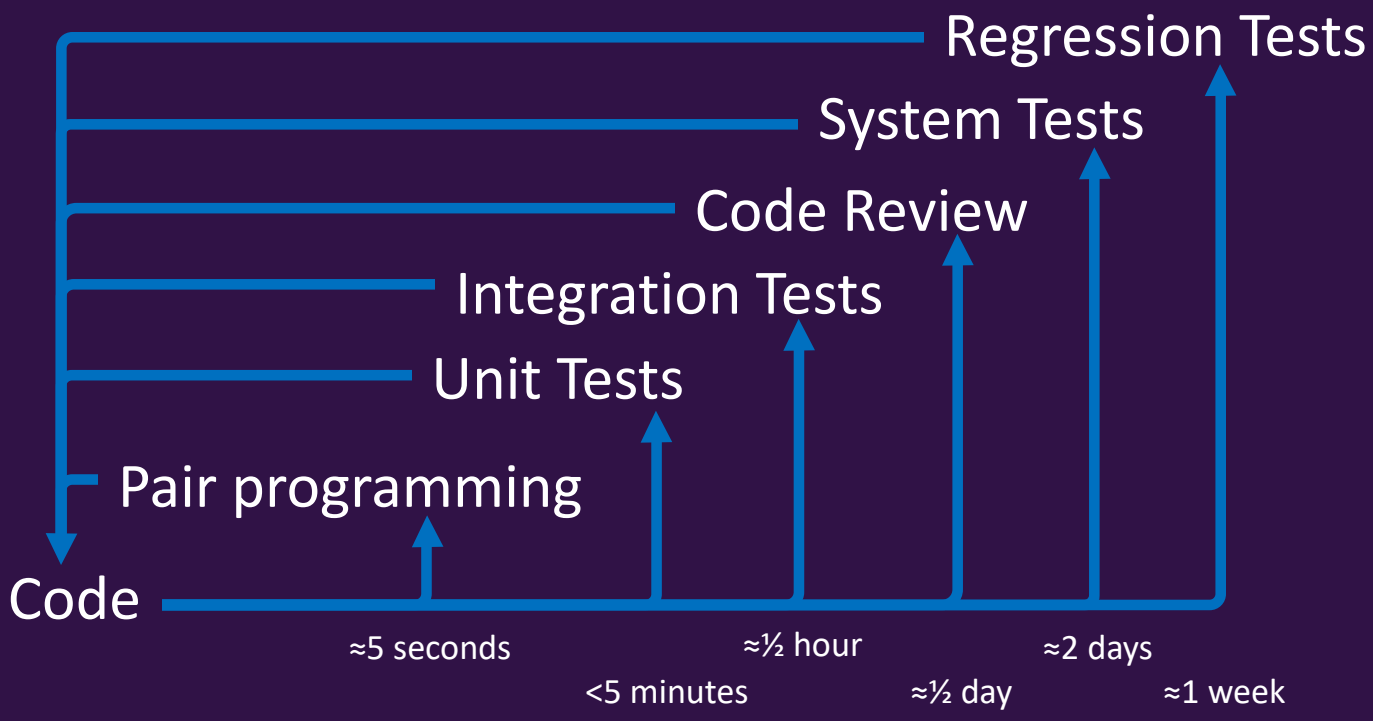
Problem:

Some system tests take a long time to run, in the order of hours per test, even when they are automated.

This slows down our outer feedback loops.

Feedback loops

Fast feedback from long System Tests



Bluefruit®

Faster System Test

Fast feedback from long System Tests

```
Scenario: The EDI stack turns on
          when water starts
          flowing

  Given there is no water flowing
  When the water flow rate changes
       to 2000ml/minute
  Then the EDI Stack is on
```

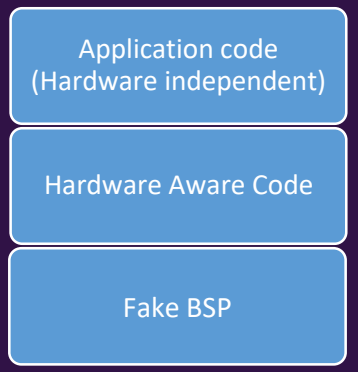
Bluefruit®

Faster System Test

Fast feedback from long System Tests

```
Scenario: The EDI stack turns on
          when water starts
          flowing

  Given there is no water flowing
  When the water flow rate changes
       to 2000ml/minute
  Then the EDI Stack is on
```

As we're not testing the entire system we only use this to determine if we've broken anything, not if the system is working



Bluefruit

Summary

- How we keep tests running fast by dual targeting
- How we use different TDD Style and how this effects how the verification of our tests
- Different Test Double insertion techniques to keep our tests isolated and repeatable
- Other practices we use in our testing process

Company : http://www.bluefruit.co.uk

Code : https://bitbucket.org/hiddeninplainsight

Blog : https://hiddeninplainsight.co.uk