Patterns and Practices for Embedded TDD in C and C++

How we introduced TDD into our company

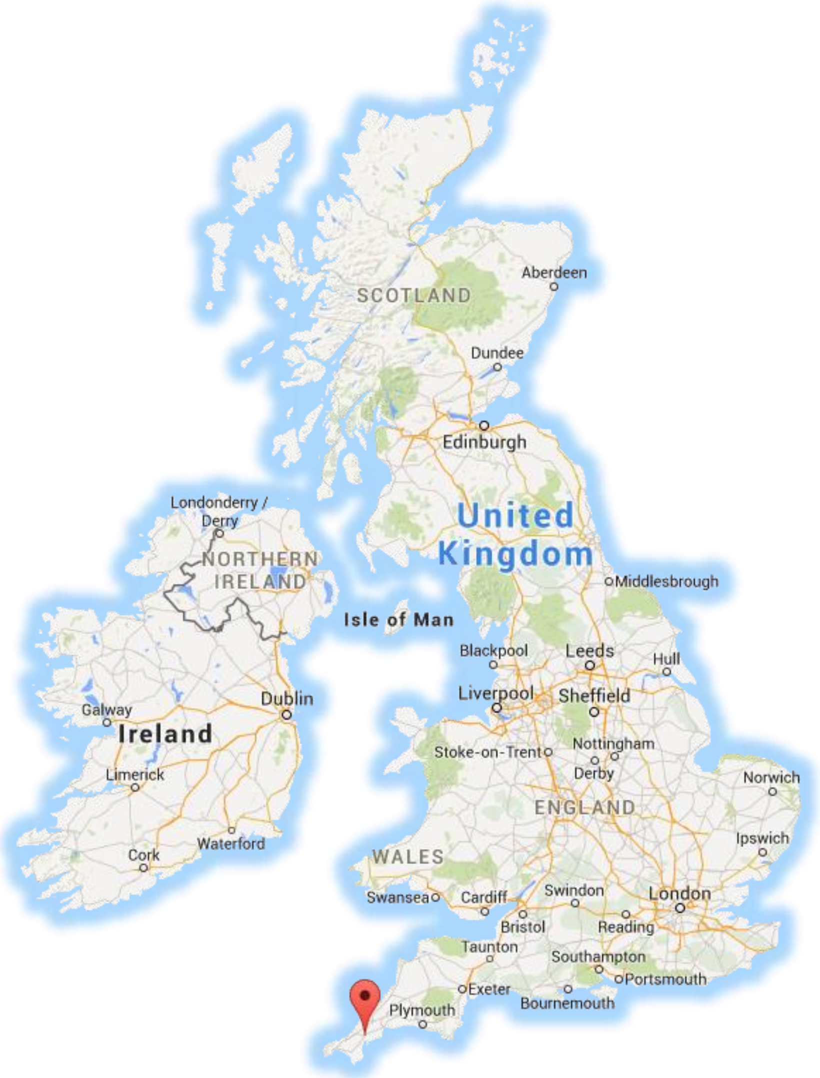
Bluefruit®

Work for Bluefruit based in Cornwall, England.

Provide an embedded software development service.

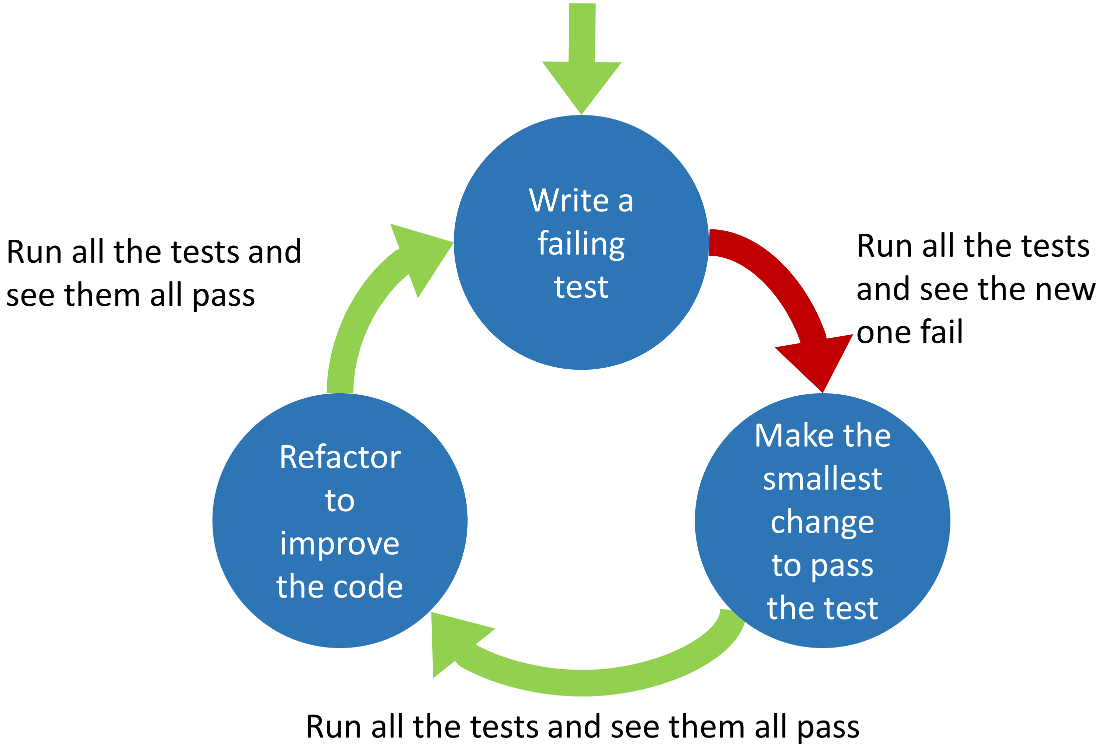
Introduced Lean/Agile practices in 2009 and have delivered approximately 30 projects since then.

Practices and Patterns we use.



Map data ©Google 2015

Standard TDD Cycle



Fast

Isolated

Repeatable

Self Verifying

Timely

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

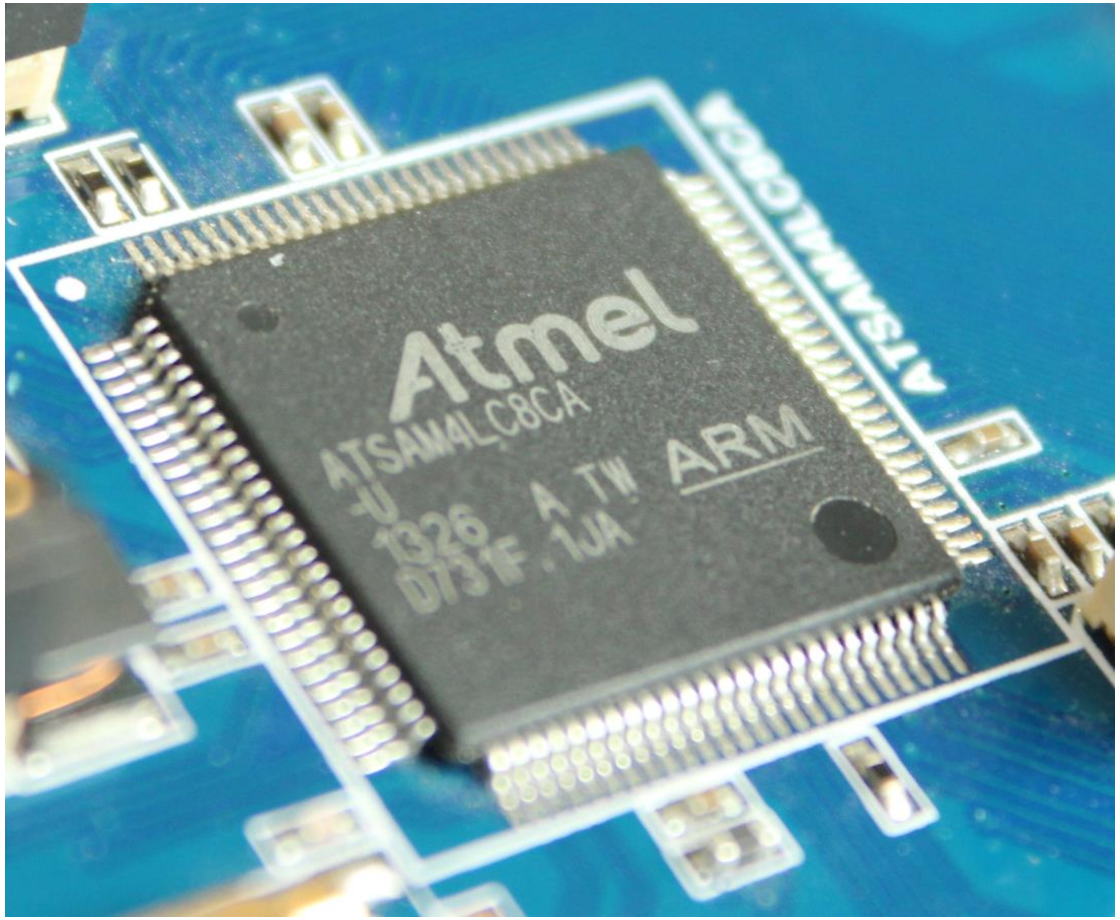
Fast
Isolated
Repeatable
Self Verifying
Timely

Where to run the tests?

Bluefruit®

Test on Target

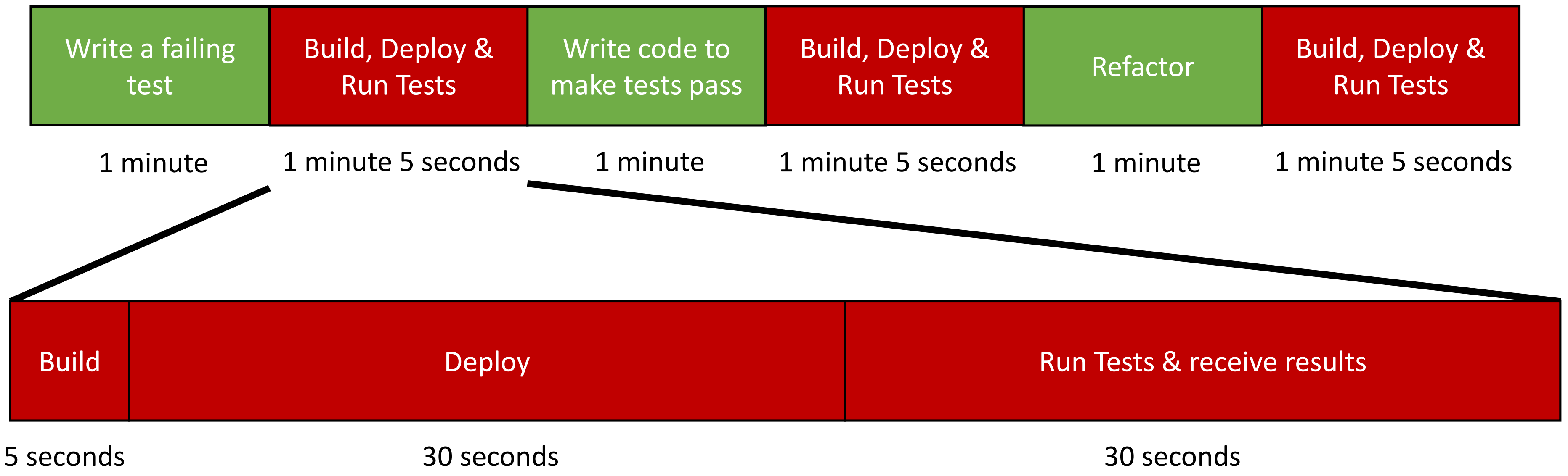
Fast
Isolated
Repeatable
Self Verifying
Timely



Bluefruit®

Analysis of TDD Cycle with Test on Target

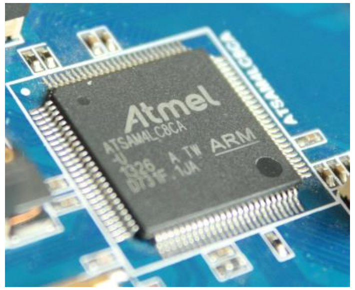
Fast
Isolated
Repeatable
Self Verifying
Timely



4th Generation Core i7
8GB RAM
SSD

Microchip C32 Compiler
PIC32MX575F512H
MPLAB 8
RealICE

Test on Target



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

We no longer exclusively run tests on the target

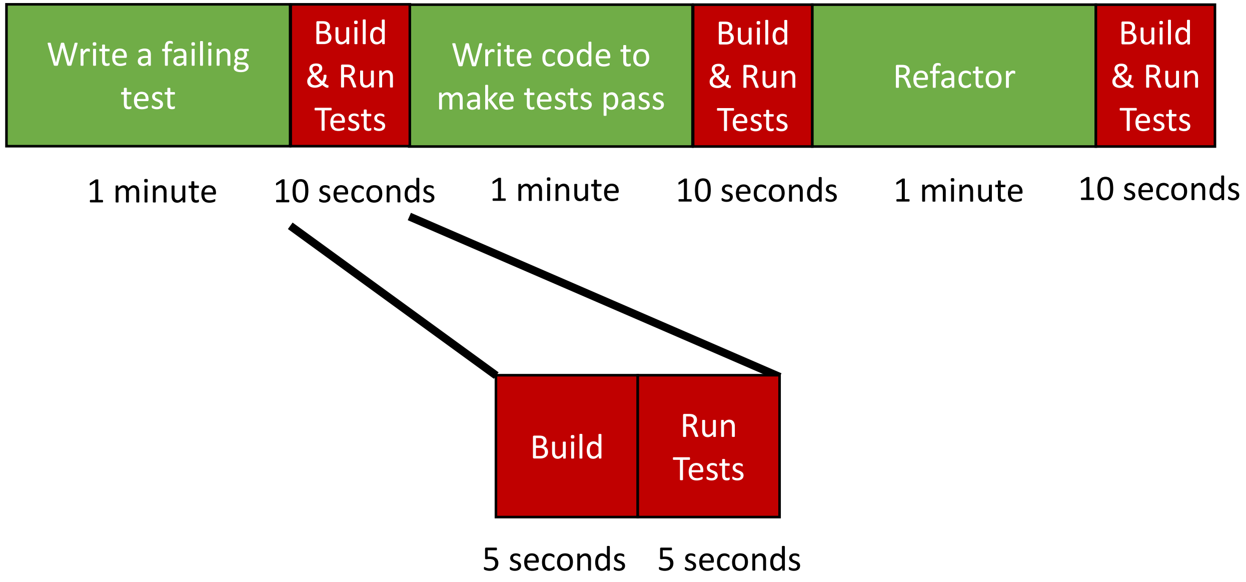
Test on Development Platform

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

Fast
Isolated
Repeatable
Self Verifying
Timely

Test on Development Platform

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

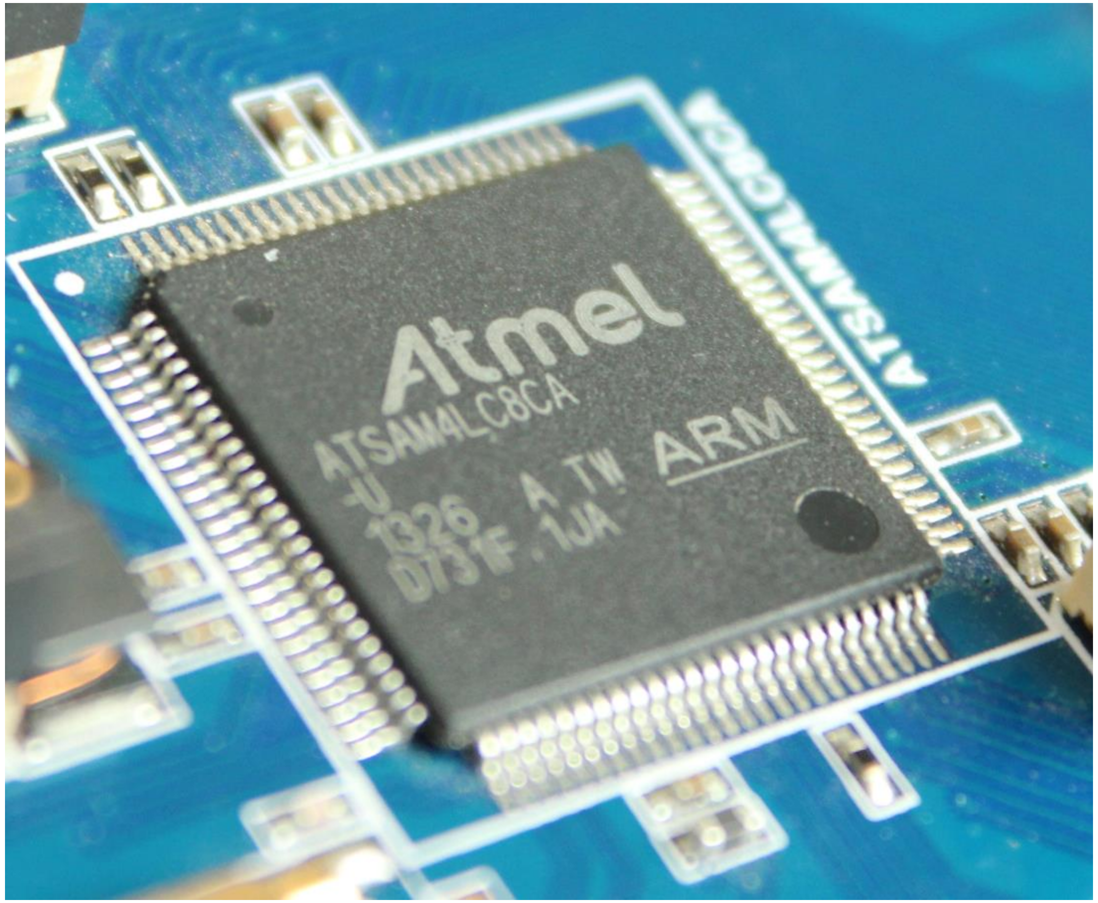
Disadvantages

- 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

Bluefruit®

Dual Targeting Tests

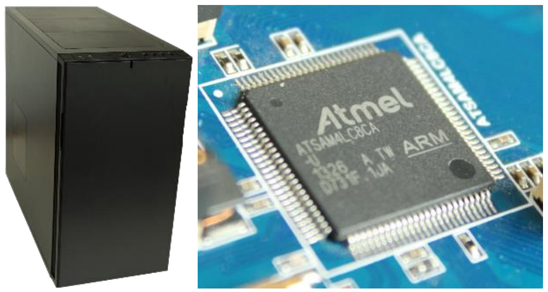
Fast
Isolated
Repeatable
Self Verifying
Timely

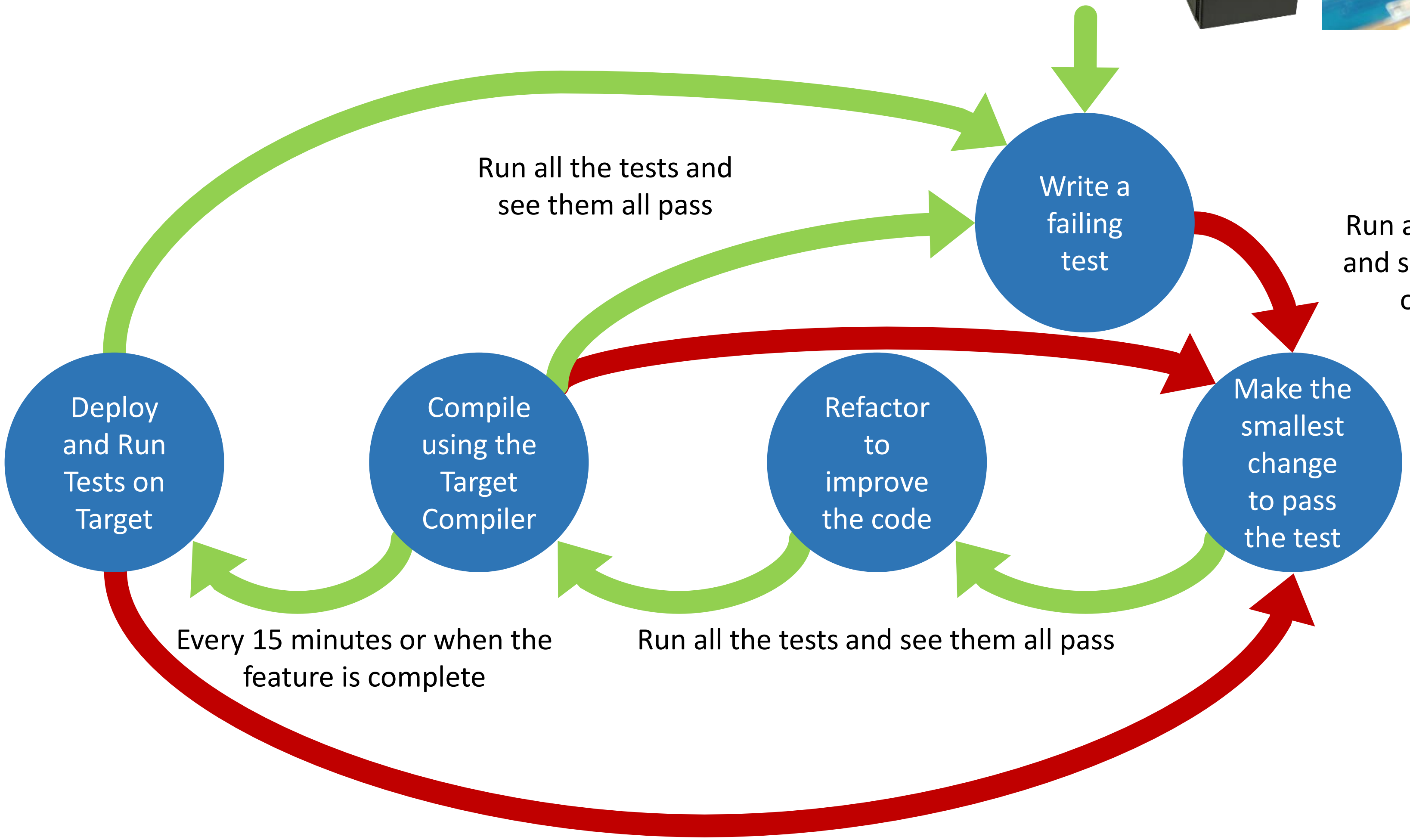


Bluefruit

Dual Targeting TDD Cycle

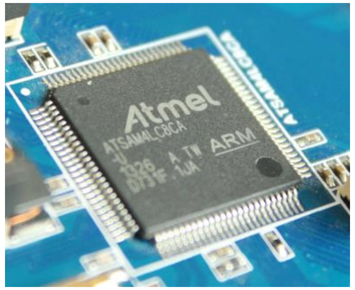
Fast
Isolated
Repeatable
Self Verifying
Timely





Bluefruit

Dual Targeting



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

Fast
Isolated
Repeatable
Self Verifying
Timely

Splitting and testing the solution

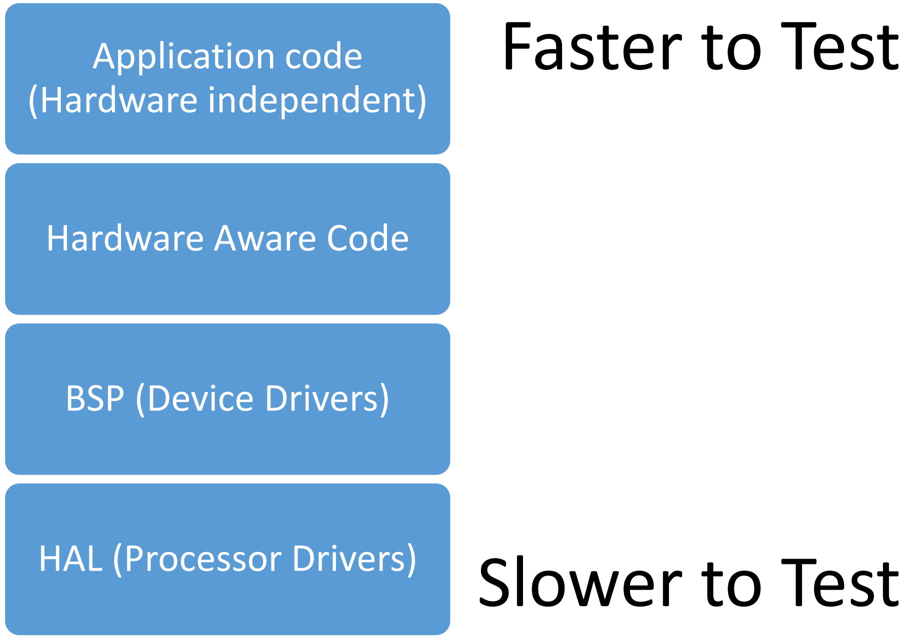
Bluefruit®

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



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

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

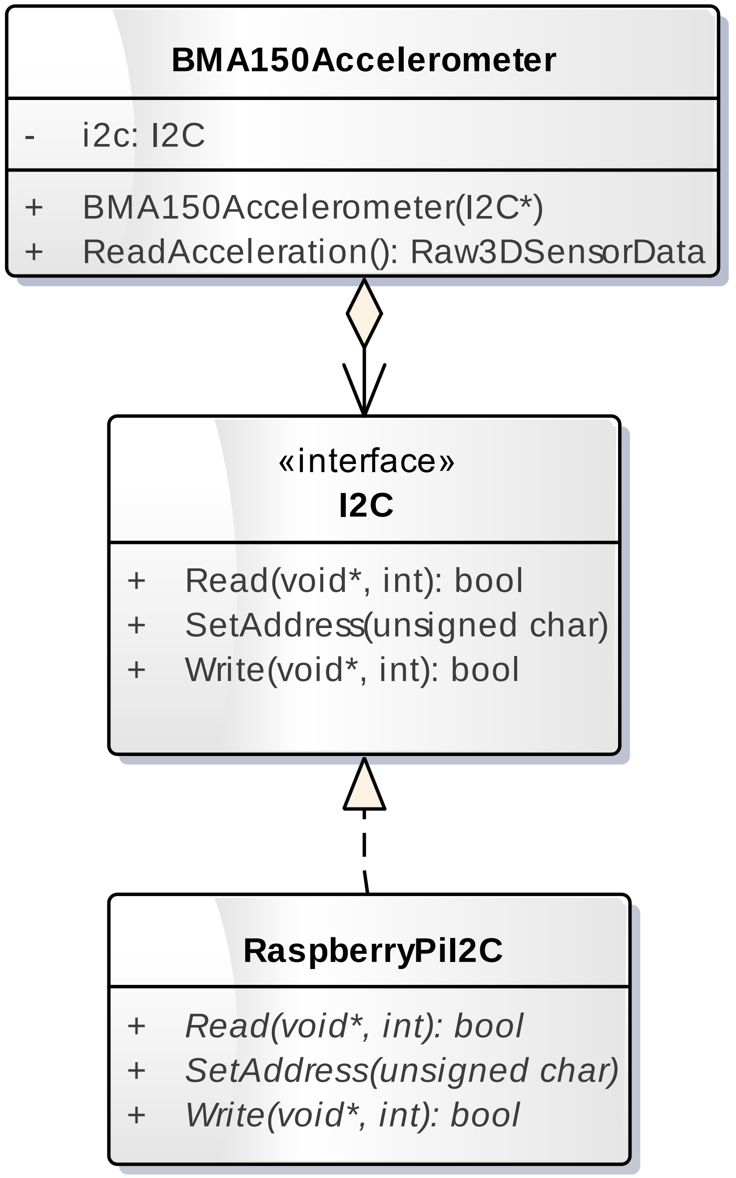
Running your tests in isolation

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 – 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.

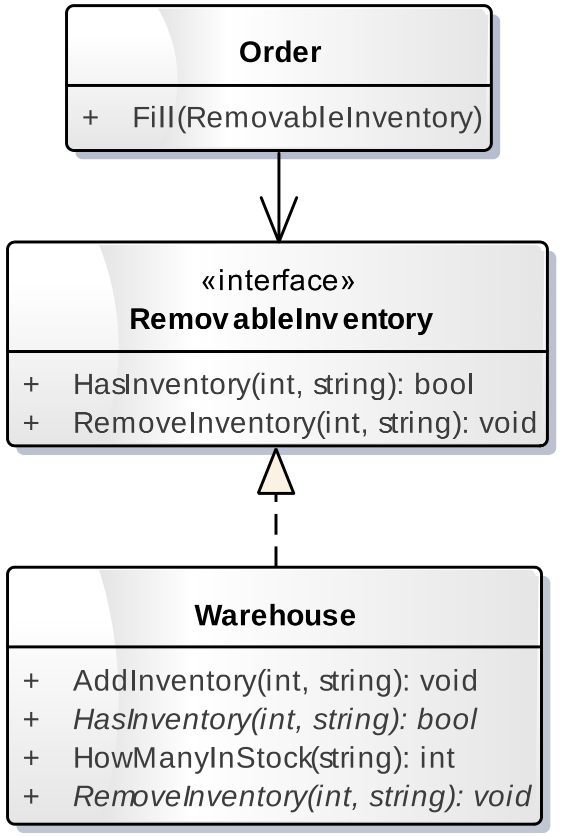
Mockist TDD

Bluefruit

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

TDD Style

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



Example Scenario

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);
}
```

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"));
}
```


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

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

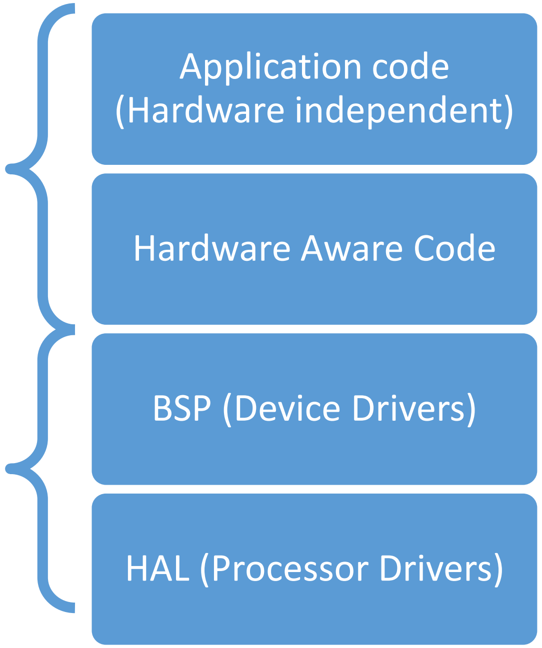
- Tests are coupled to implementation making refactoring harder

How I vary my TDD Style

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.

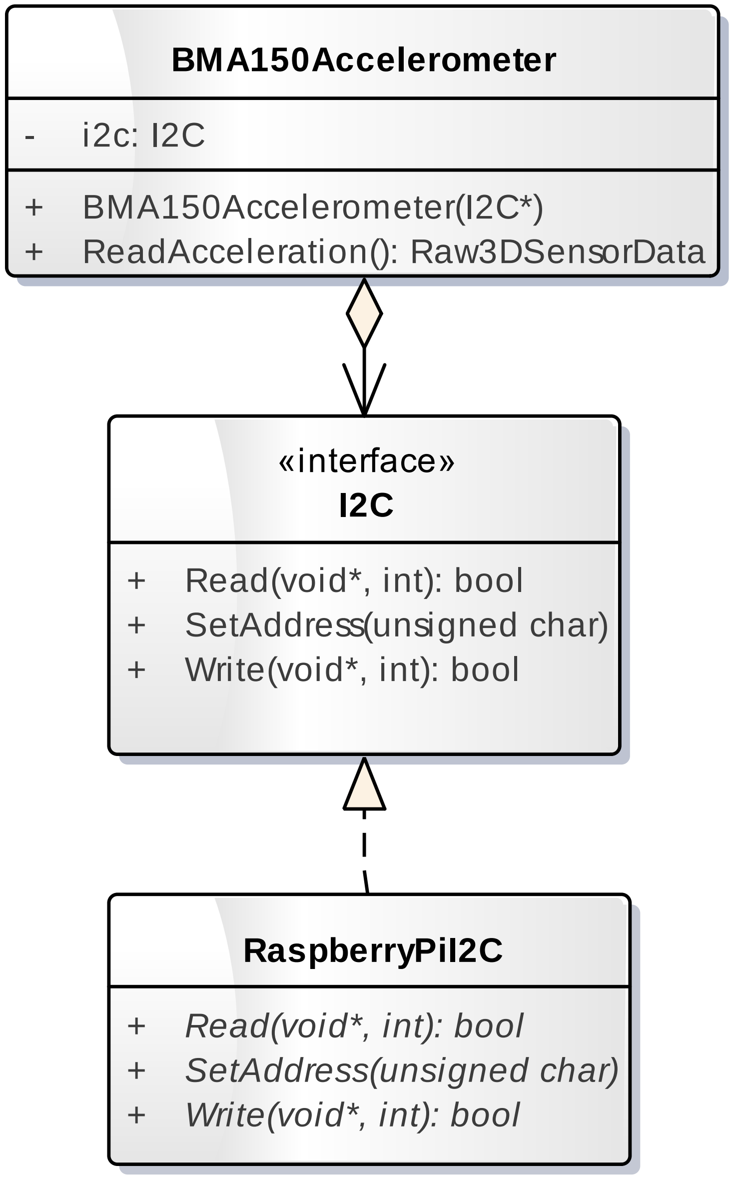
Running your tests in isolation

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

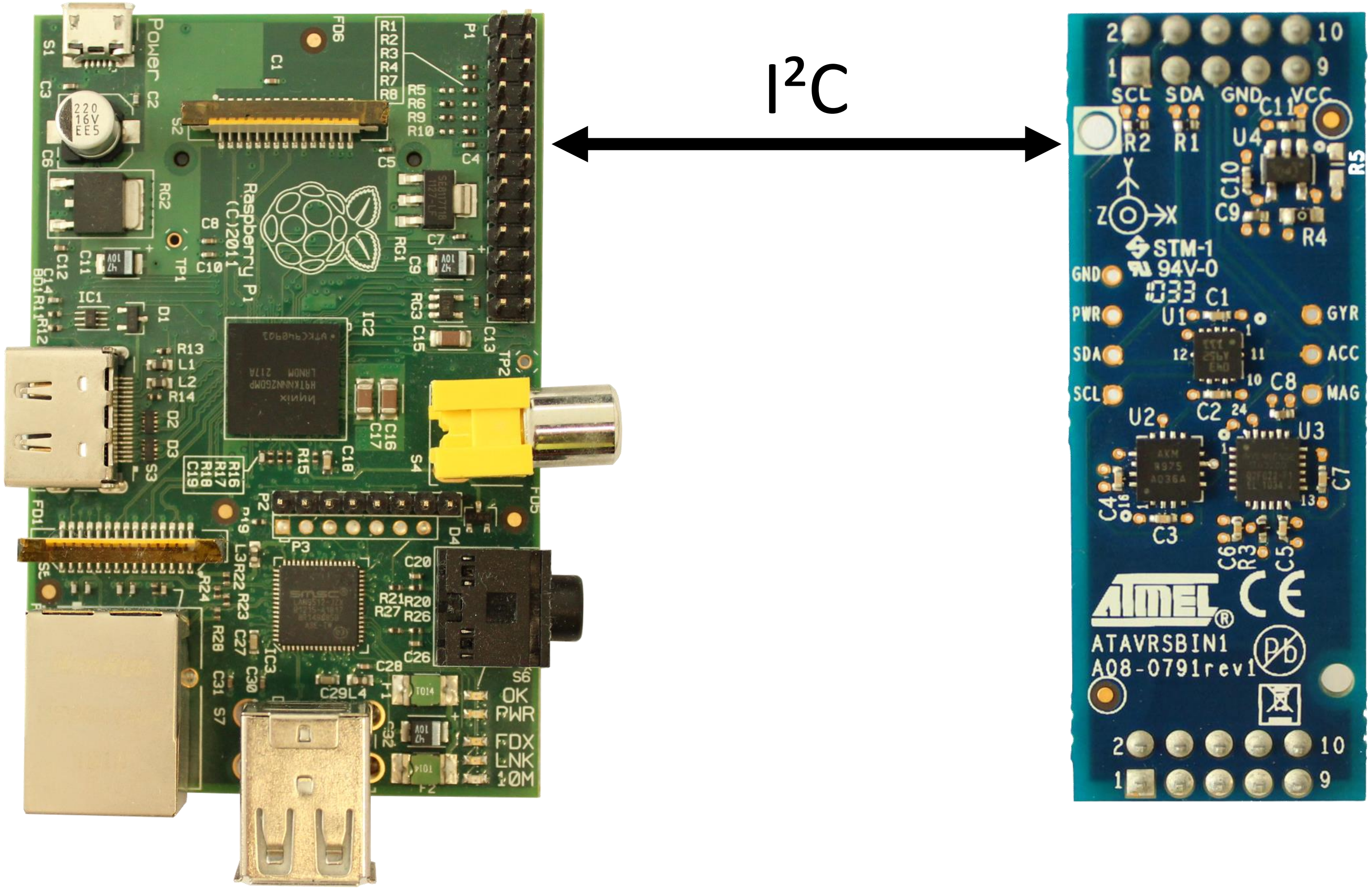
Fast
Isolated
Repeatable
Self Verifying
Timely

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

These are the three most common insertion techniques we use

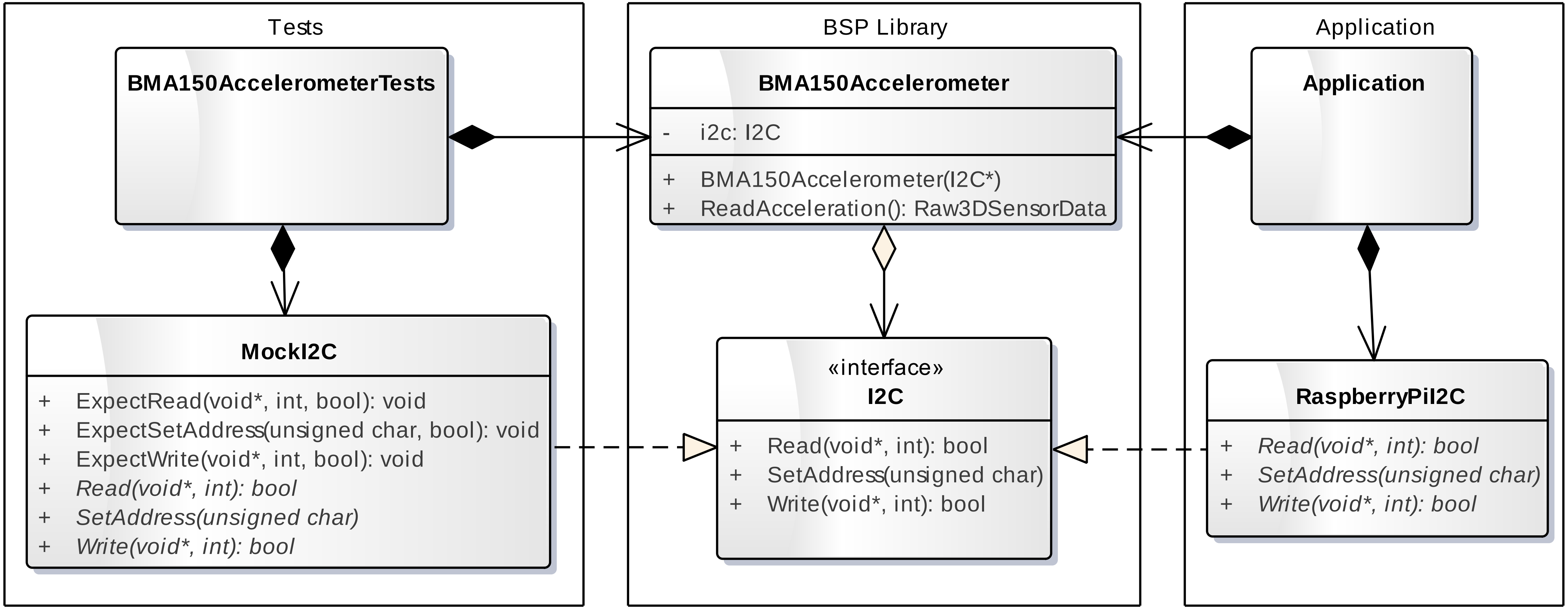
Test Double Insertion

Fast
Isolated
Repeatable
Self Verifying
Timely



Test Doubles Insertion

Fast
Isolated
Repeatable
Self Verifying
Timely



Fast
Isolated
Repeatable
Self Verifying
Timely

C++ Interfaces

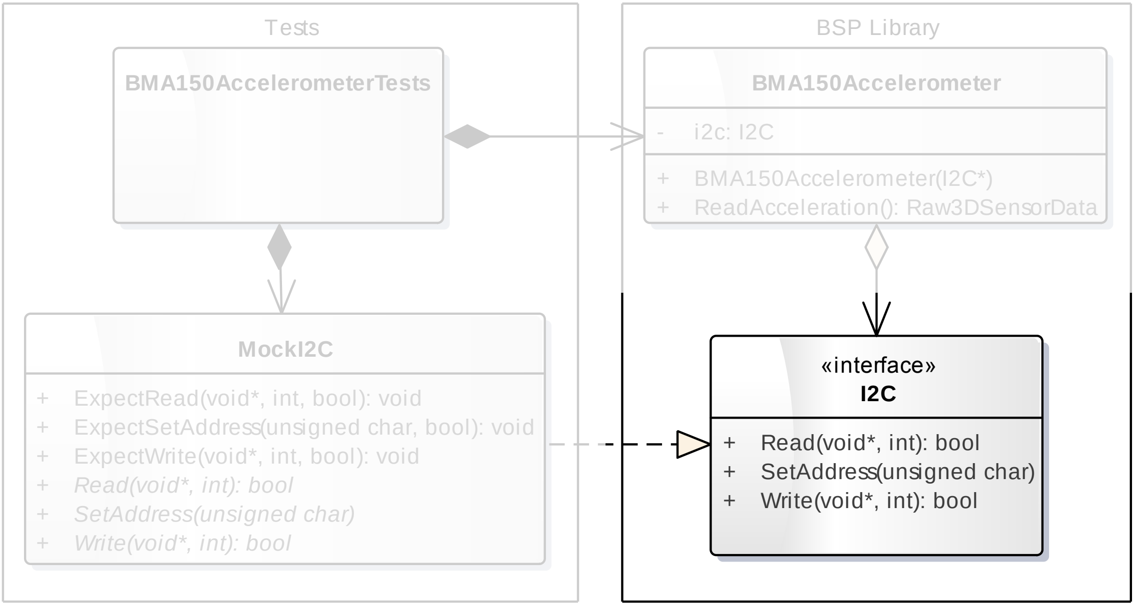
We use this technique for everything

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;
};
```



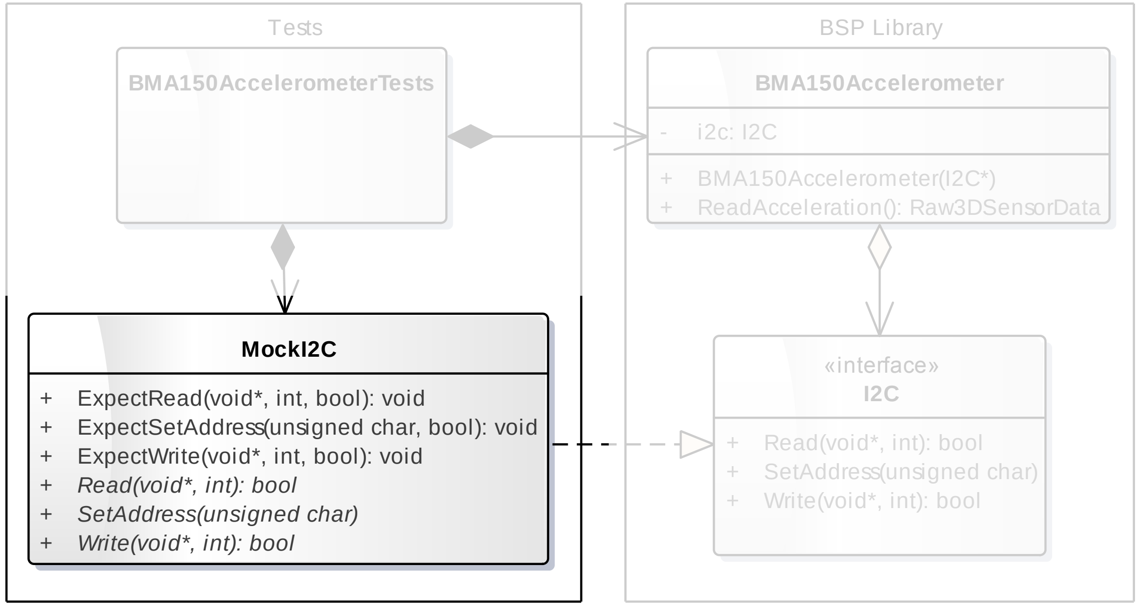
Bluefruit

Fast
Isolated
Repeatable
Self Verifying
Timely

Dependency Mock

Test Doubles insertion using C++ Interfaces

```
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(); }
};
```



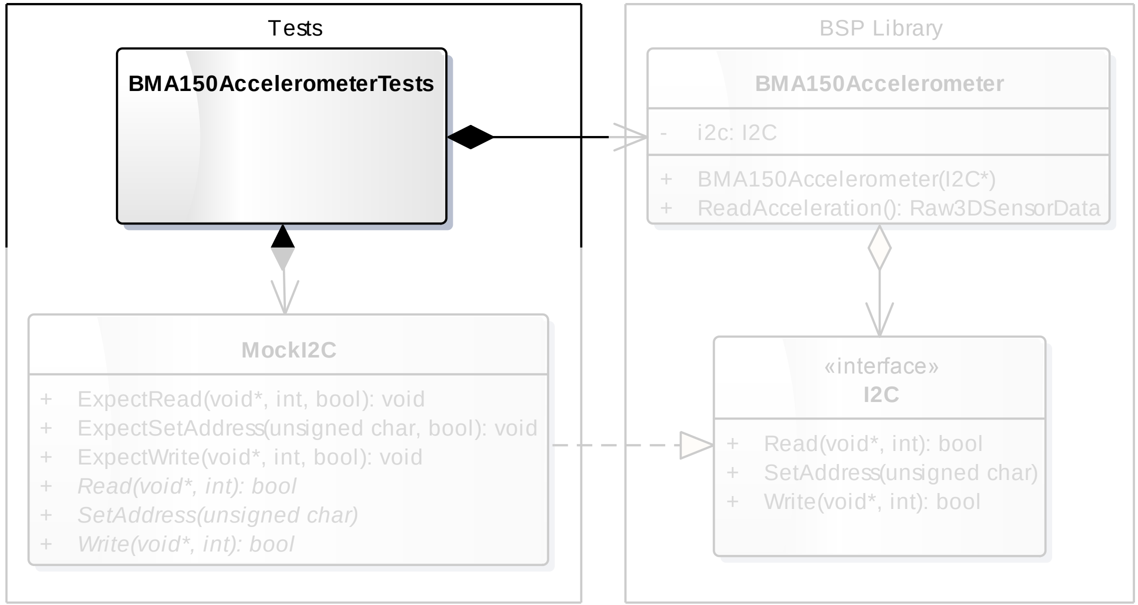
Bluefruit

Fast
Isolated
Repeatable
Self Verifying
Timely

Test

Test Doubles insertion using C++ Interfaces

```
void testBMA150Accelerometer_Reading_an_acceleration_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);
}
```



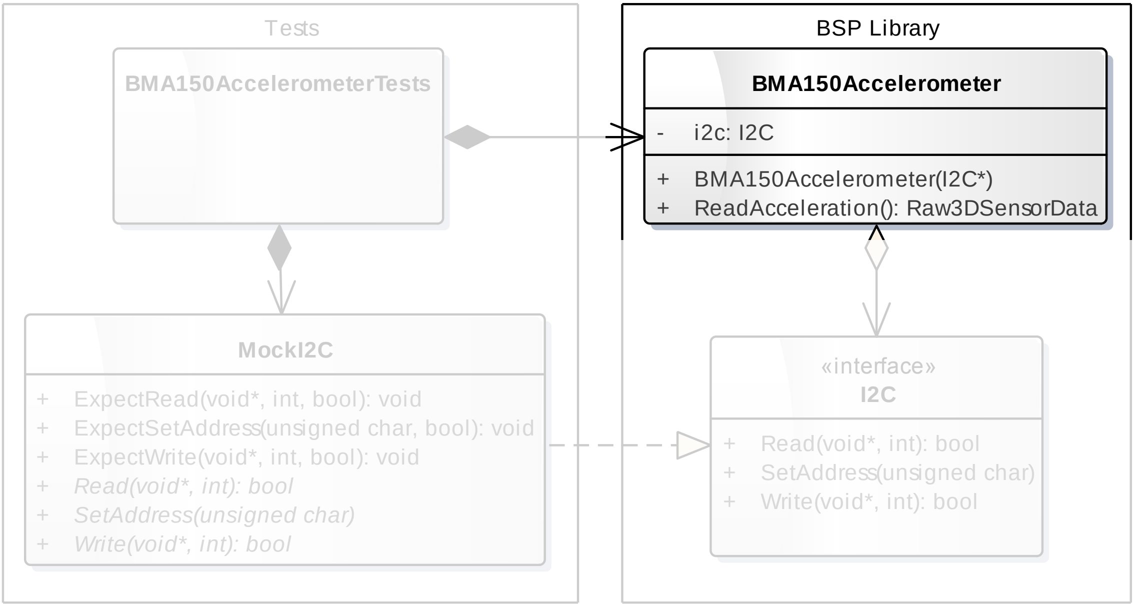
Bluefruit

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

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

C V-Tables (structs containing function pointers)

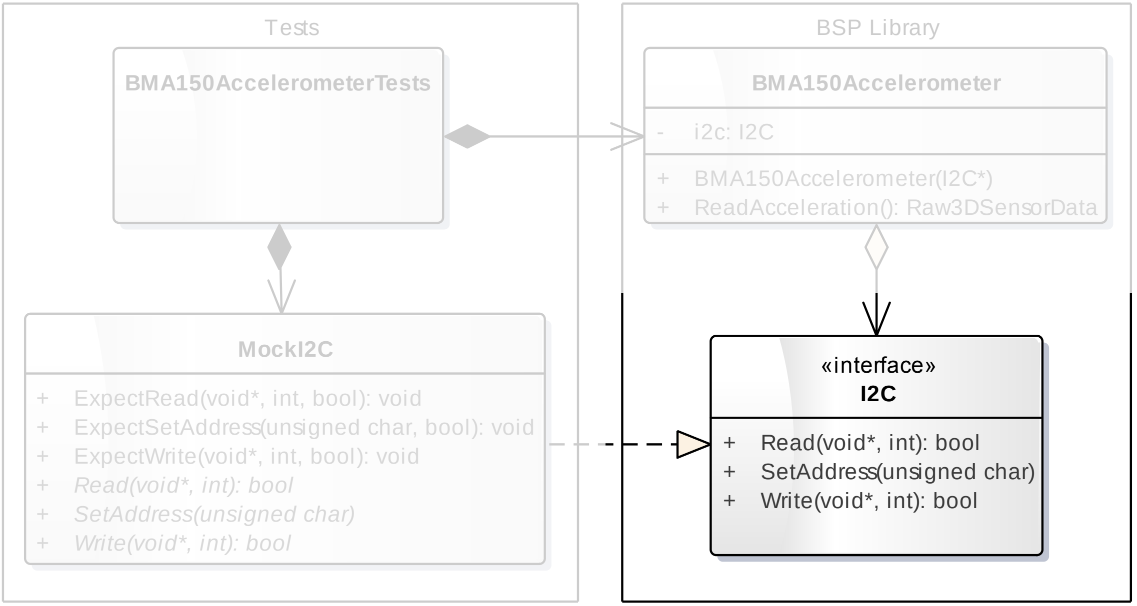
We use this technique when we can't use C++

Dependency Interface

Test Doubles insertion using C V-Tables (structs containing function pointers)

Fast
Isolated
Repeatable
Self Verifying
Timely

```c
#ifndef I2C_H
#define I2C_H

#include <stdbool.h>

struct I2C
{
  bool (*SetAddress)(unsigned char address);
  bool (*Read)(void * buffer, int length);
  bool (*Write)(const void * buffer, int length);
};

#endif
```

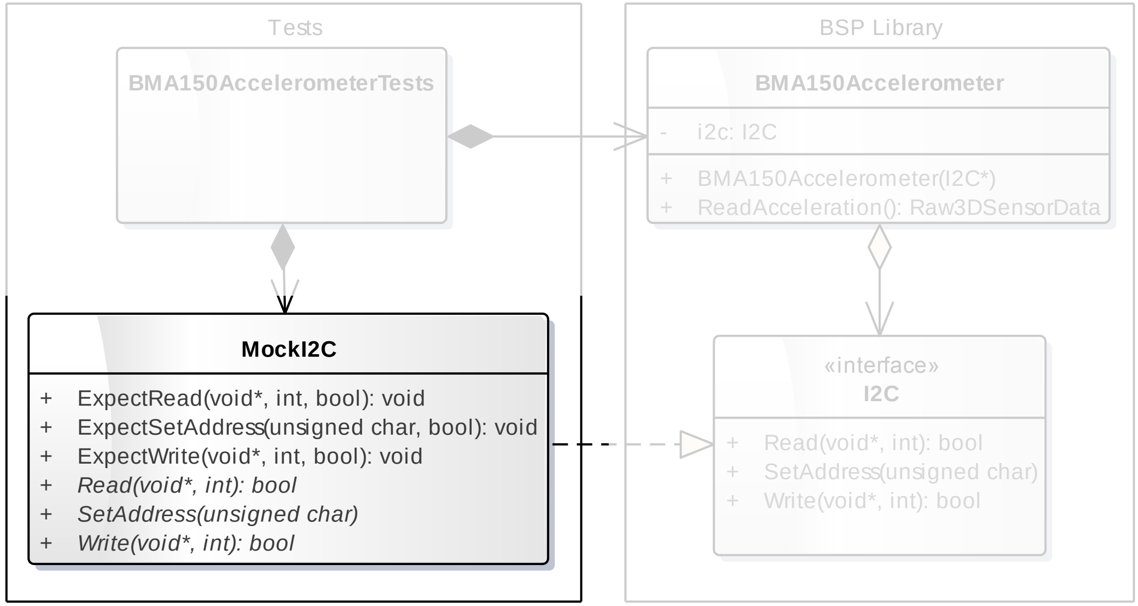


Dependency Mock

Test Doubles insertion using C V-Tables (structs containing function pointers)

Fast
Isolated
Repeatable
Self Verifying
Timely

```c
static bool MockI2C_SetAddress(
                unsigned char address)
{
  // ...
}

const struct I2C MockI2C =
{
  .SetAddress = MockI2C_SetAddress,
  .Read = MockI2C_Read,
  .Write = MockI2C_Write
};

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

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



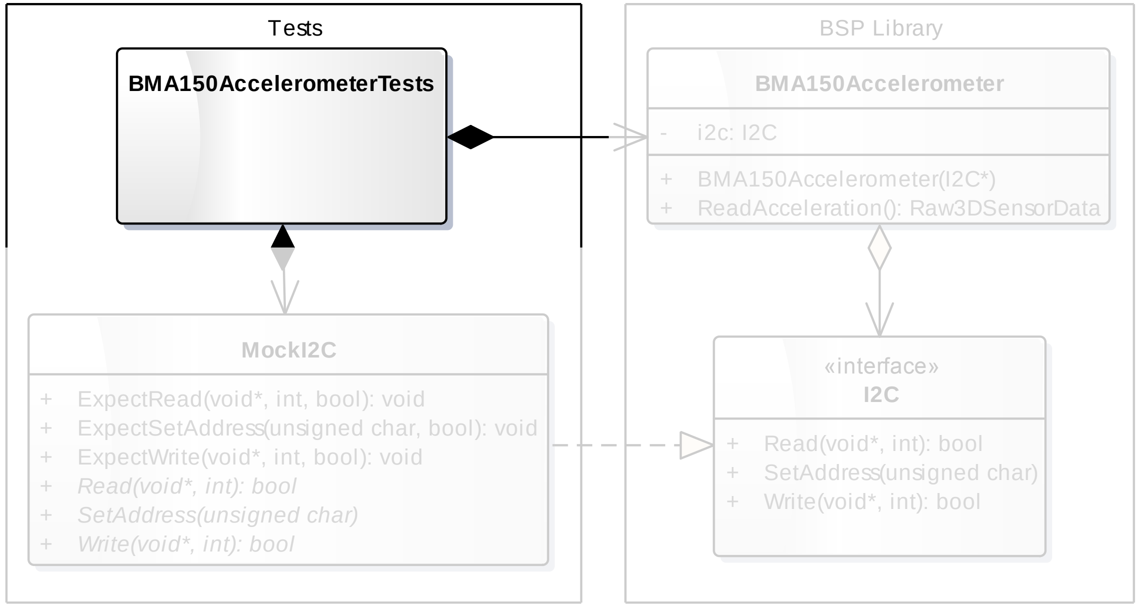
Bluefruit®

Test

Test Doubles insertion using C V-Tables (structs containing function pointers)

Fast
Isolated
Repeatable
Self Verifying
Timely

```
void testBMA150Accelerometer_Reading_an_acceleration_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
  BMA150Accelerometer_Initialise(&MockI2C);
  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);
}
```

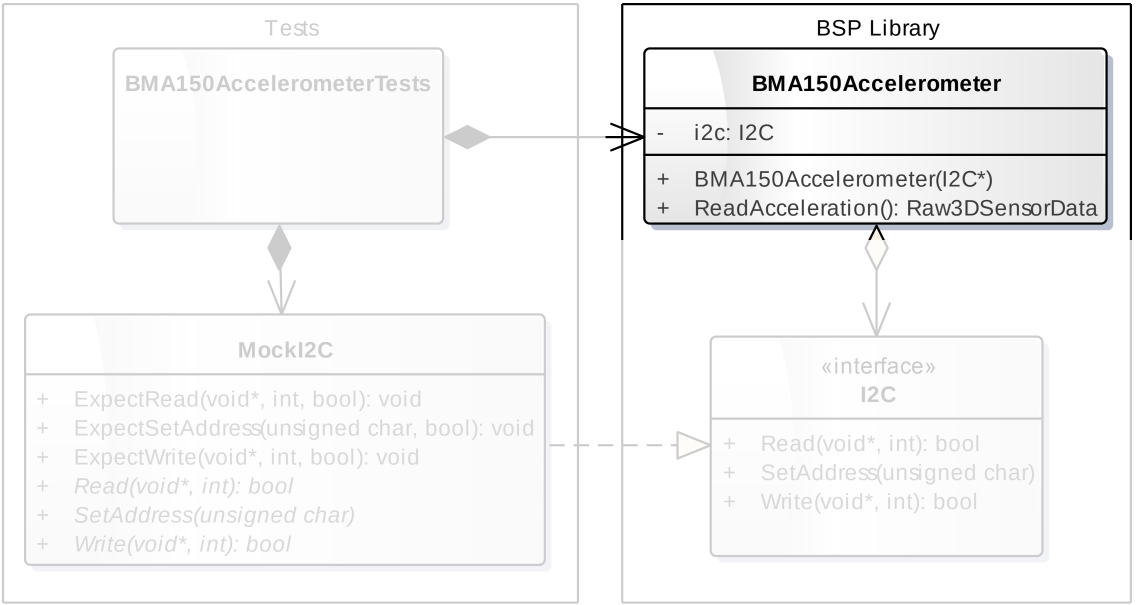


Code (System under test)

Test Doubles insertion using C V-Tables (structs containing function pointers)

Fast
Isolated
Repeatable
Self Verifying
Timely

```c
static const struct I2C *i2c;
void BMA150Accelerometer_Initialise(const struct I2C *i2cPort)
{
  i2c = i2cPort;
}

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 using C V-Tables (structs containing function pointers)

Fast
Isolated
Repeatable
Self Verifying
Timely

Advantages

- Allows runtime substitution in C

Disadvantages

- It is easy to make mistakes when creating the V-Tables
- Allowing multiple instances requires a lot of boilerplate code than was not shown in the previous slides

We use this technique when we can’t use C++

Linking other object files

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

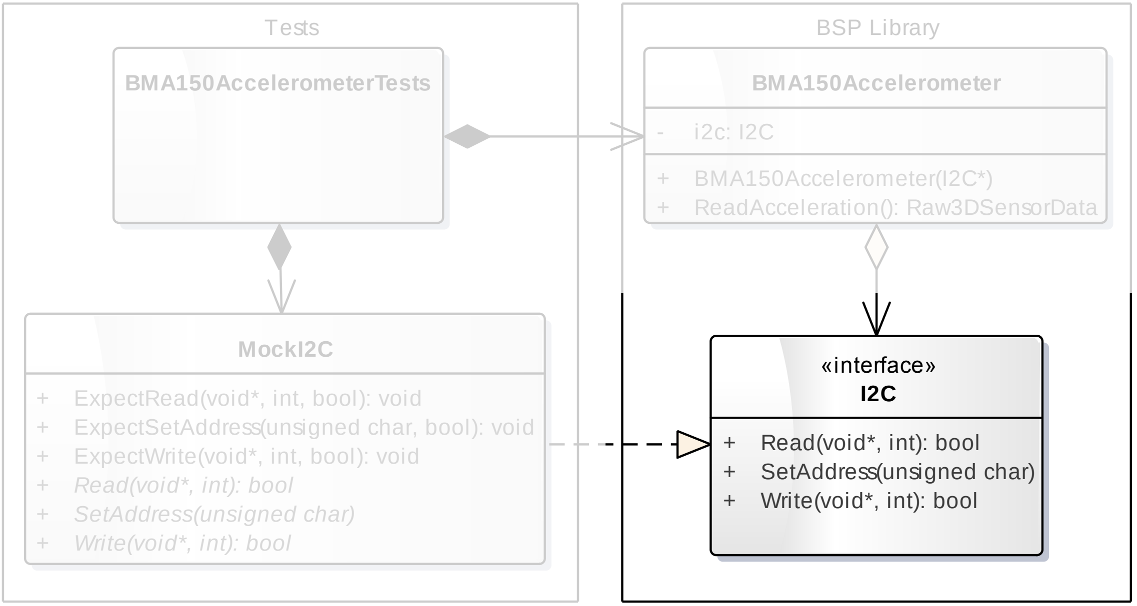
The example code is in C but this technique works in C++ as well

Fast
Isolated
Repeatable
Self Verifying
Timely

Dependency Interface

Test Doubles insertion by linking other object files

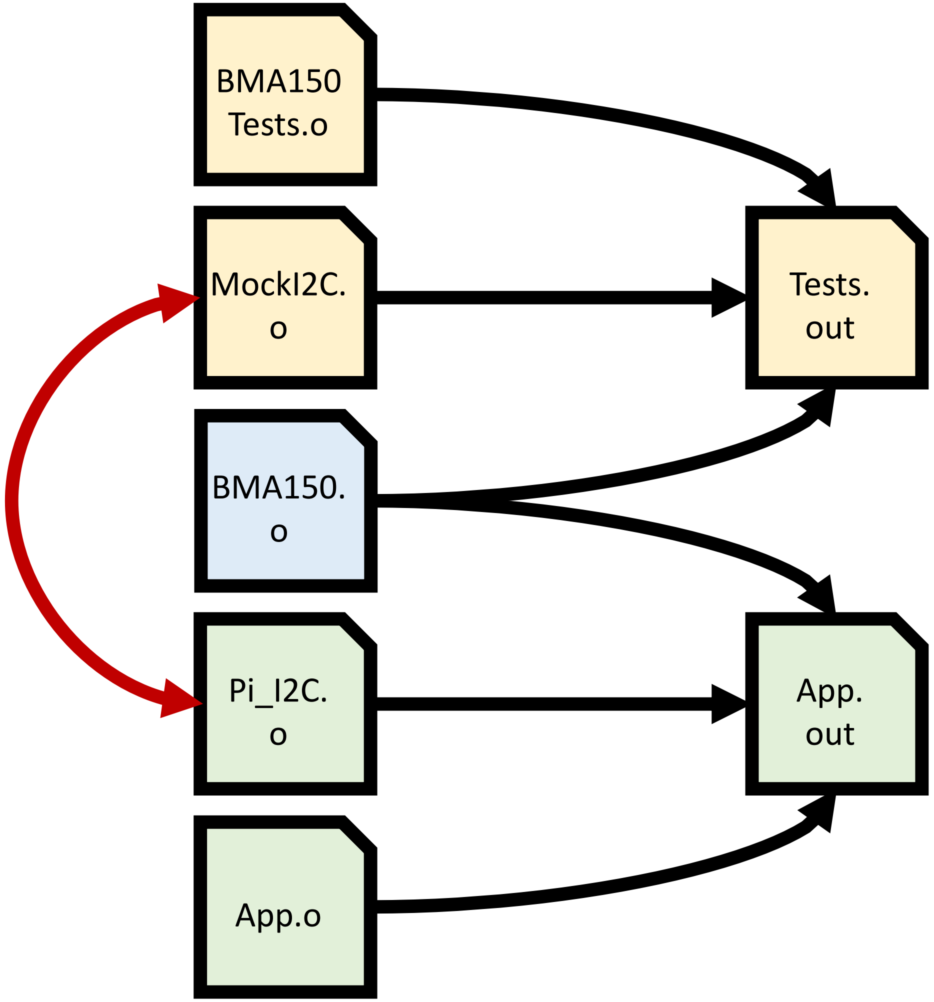
```c
#ifndef I2C_H
#define I2C_H

#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
```



Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

Contain different definitions of the same methods



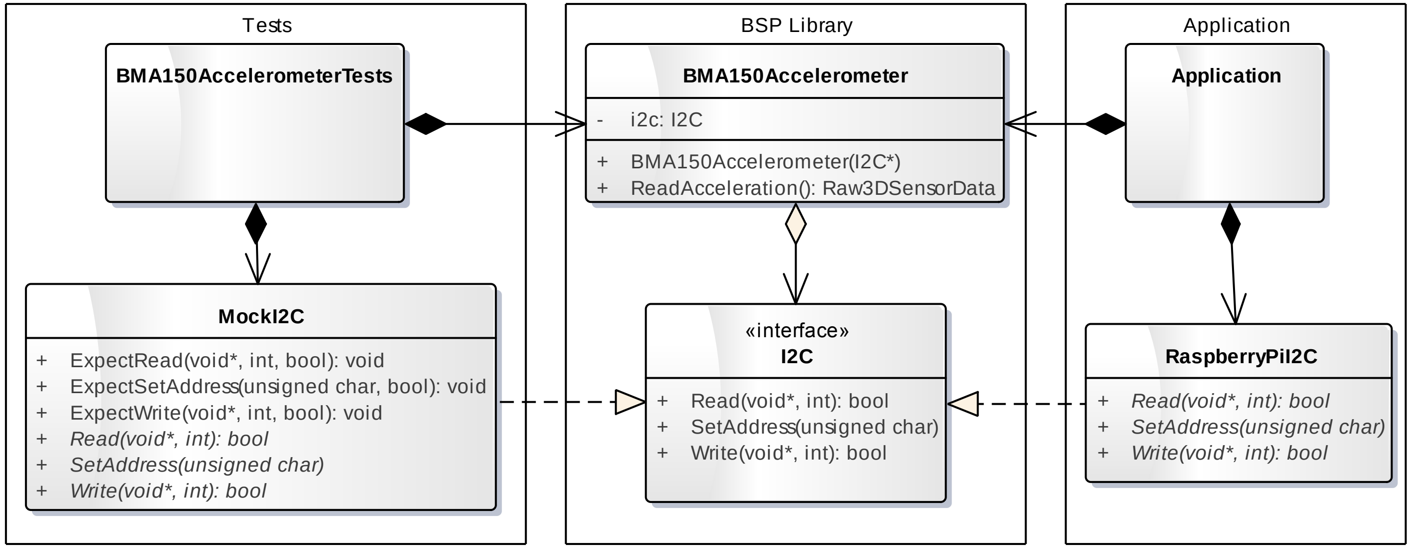
Bluefruit

Fast
Isolated
Repeatable
Self Verifying
Timely

Makefile

Test Doubles insertion by linking other object files

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

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

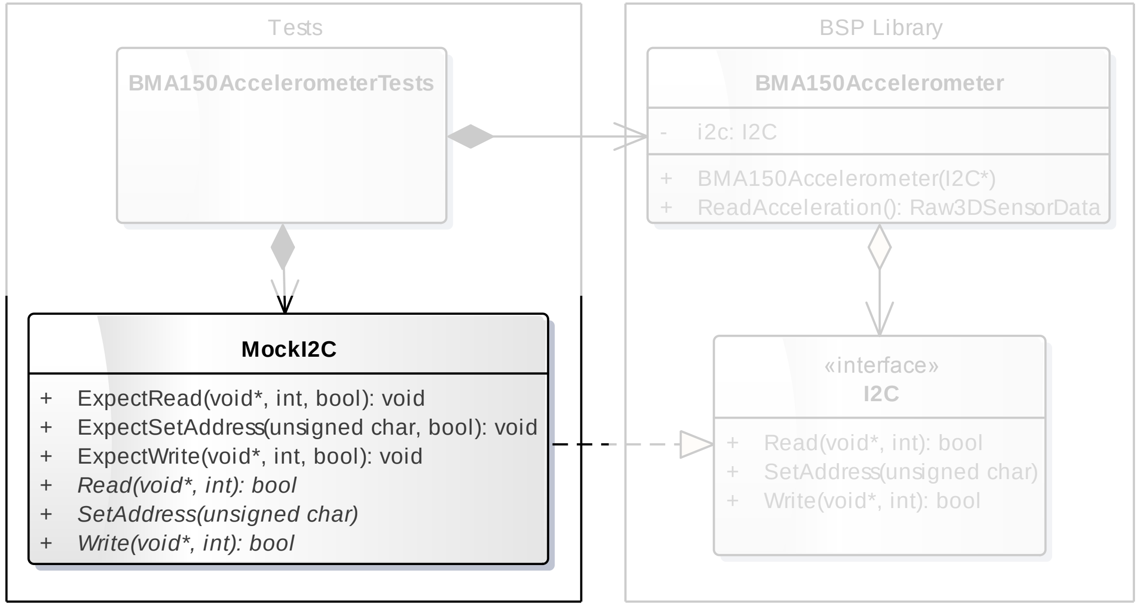


Bluefruit

Dependency Mock

Test Doubles insertion by linking other object files

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

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

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

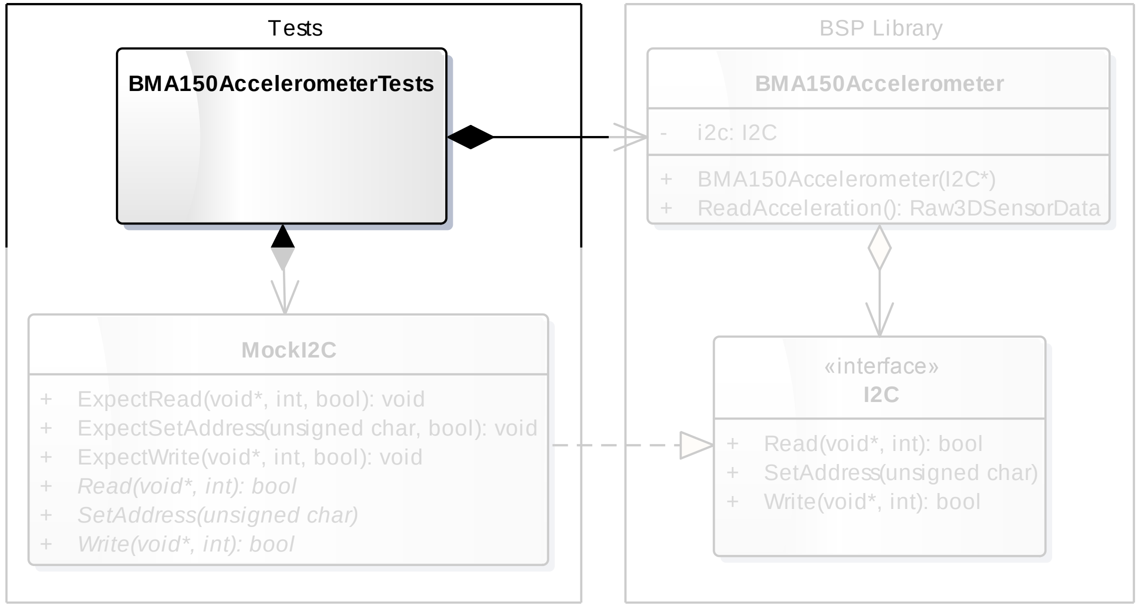


Test

Test Doubles insertion by linking other object files

Fast
Isolated
Repeatable
Self Verifying
Timely

```c
void testBMA150Accelerometer_Reading_an_acceleration_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);
}
```



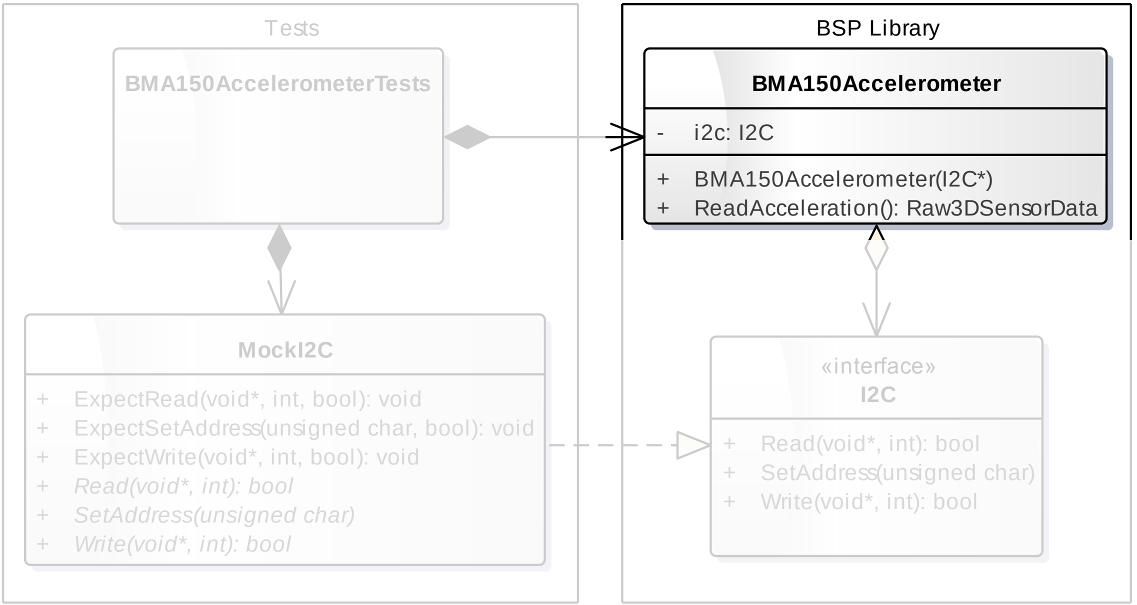
Bluefruit

Fast
Isolated
Repeatable
Self Verifying
Timely

Code (System under test)

Test Doubles insertion by linking other object files

```
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;
}
```



Test Doubles insertion by linking other object files

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

C++ Interfaces – For everything

- Easiest method of inserting test doubles

C V-Tables (structs of function pointers) – When we can not use C++

- Run time substitution in C

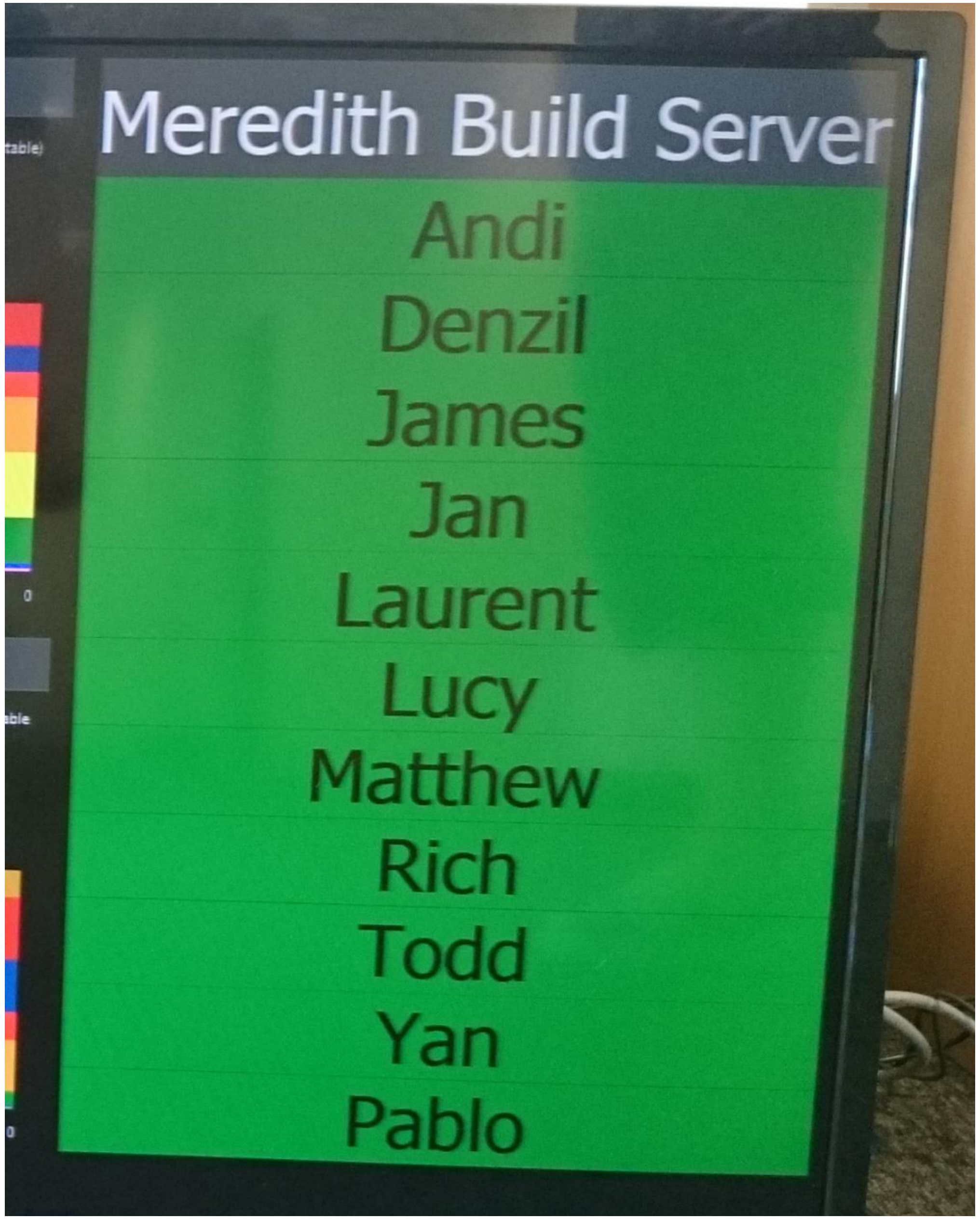
Linking other object files – When virtual function calls are too expensive

- Removes the performance hit from making virtual function calls

What else?

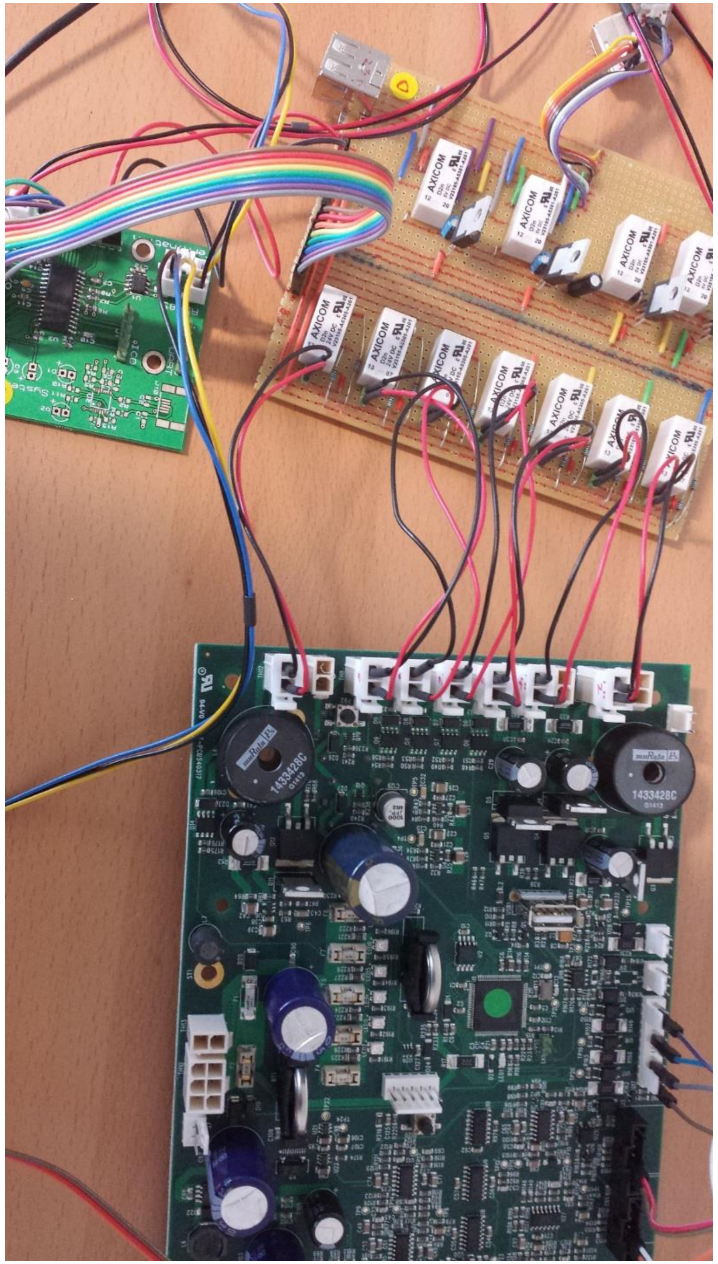
Other practices

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



Other practices

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



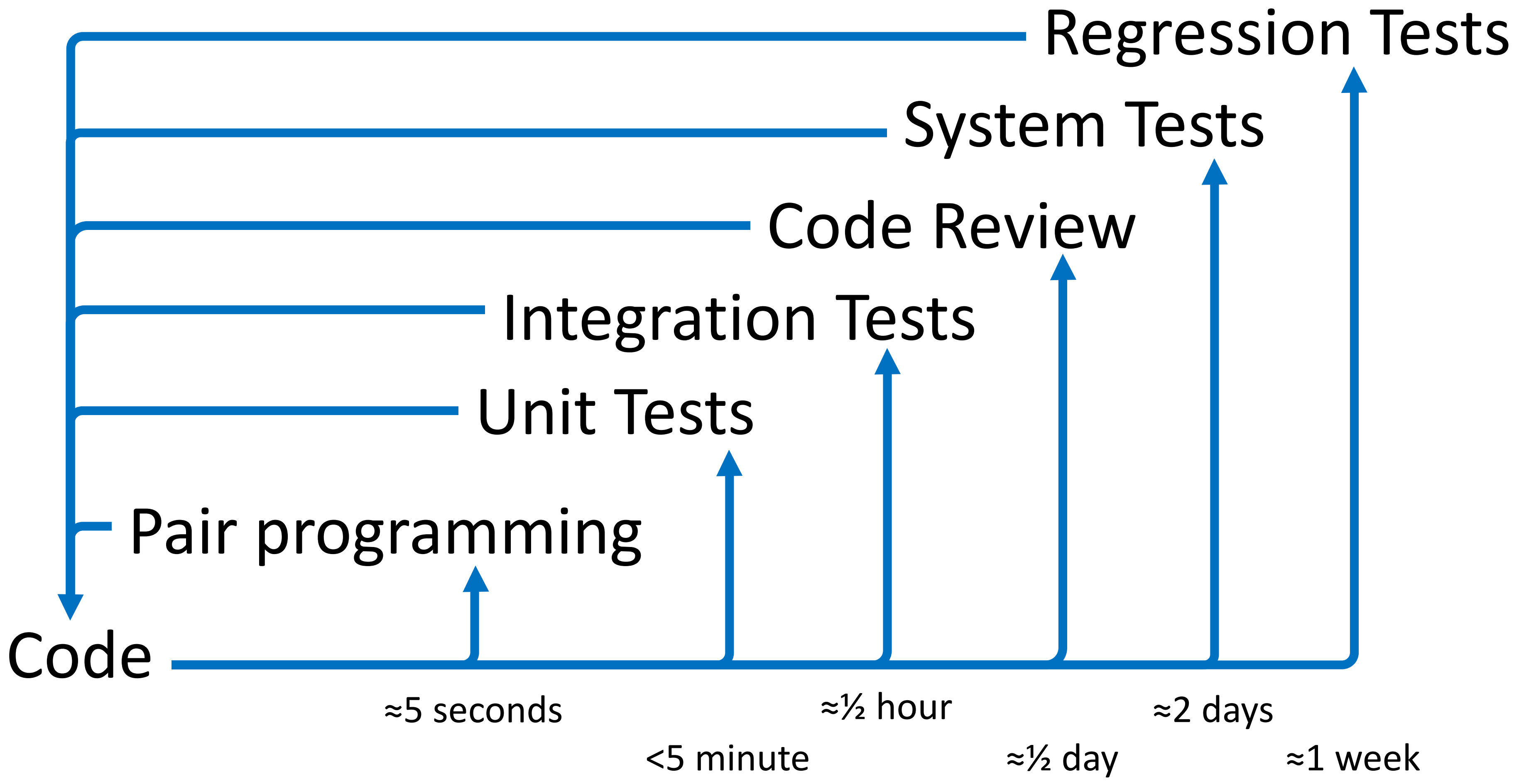
Polymorphic System Testing

Problem:

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

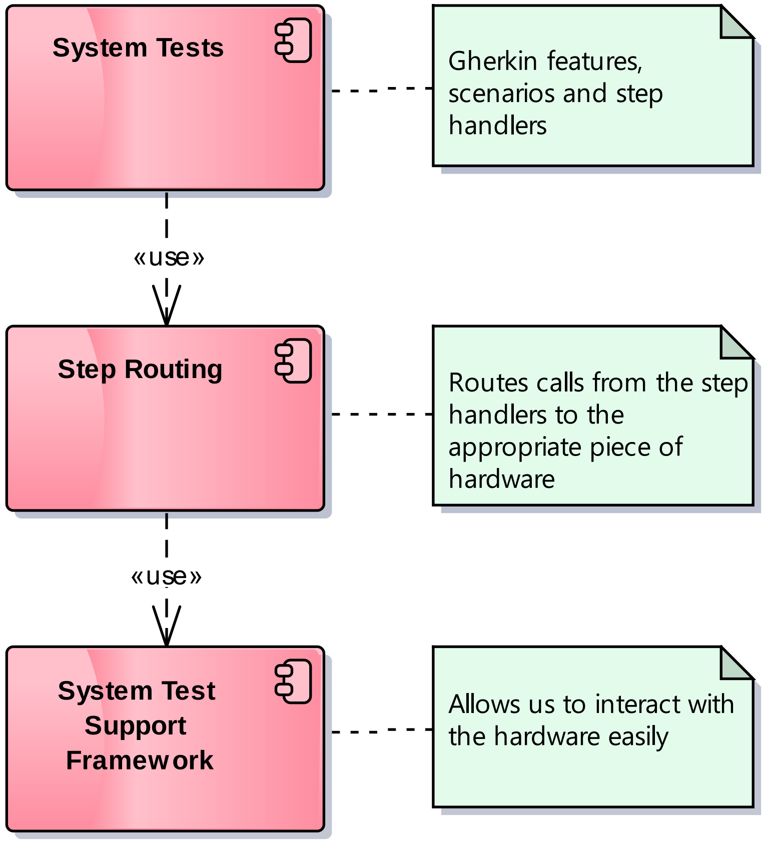
This slows down our outer feedback loops.

Feedback loops



Bluefruit

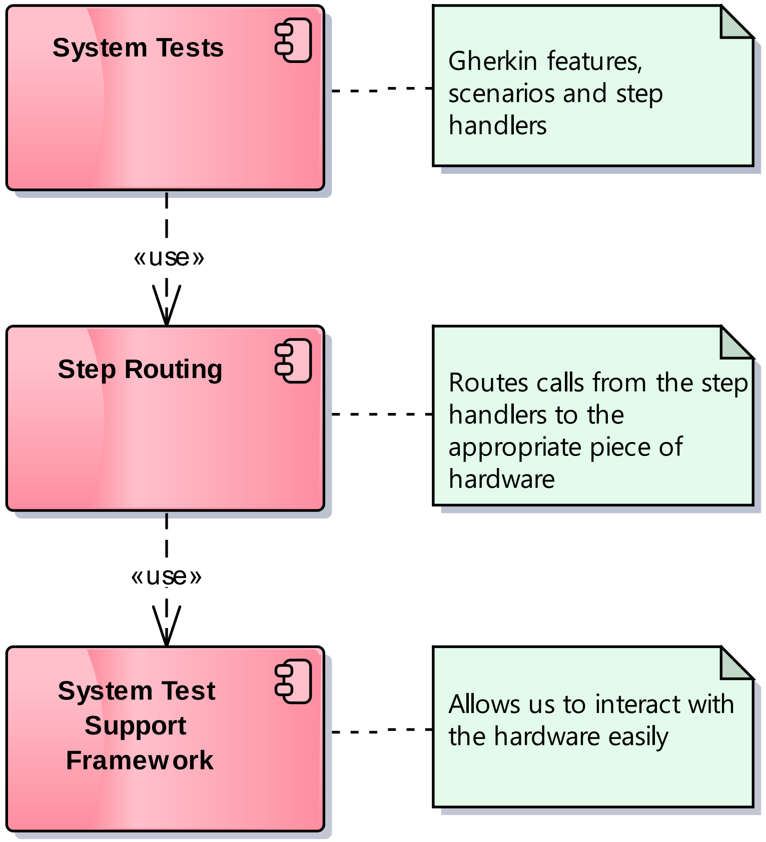
Automated System Test Overview




```
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
```

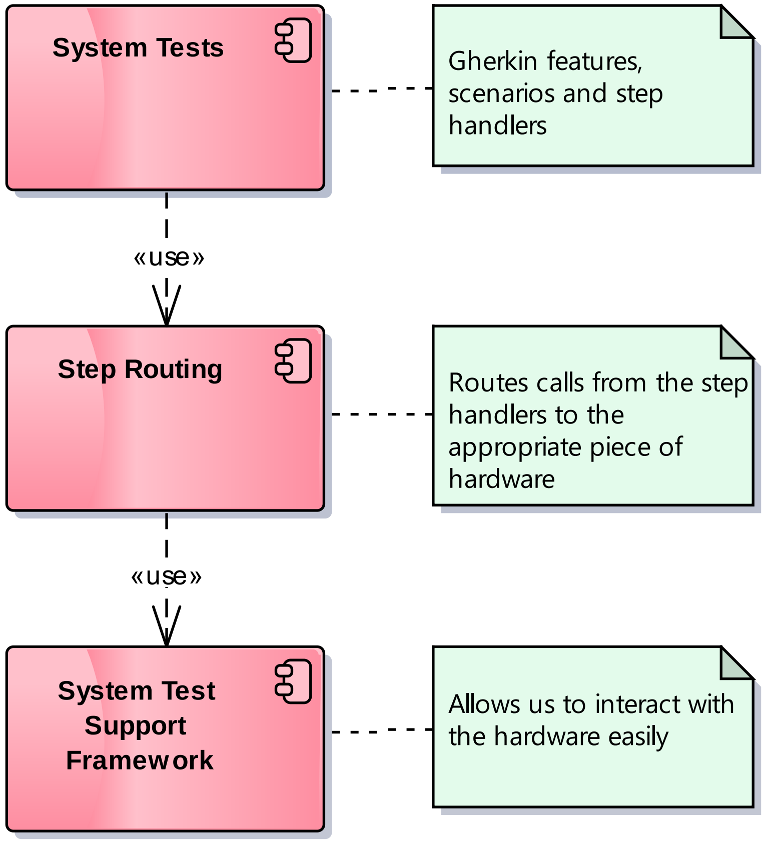
```
[Given(@"there is no water flowing")]
public void ThereIsNoWaterFlowing()
{
   Target.Instance.FlowRate = 0;
   Target.Instance.ElapseTimeMs(500);
}
```

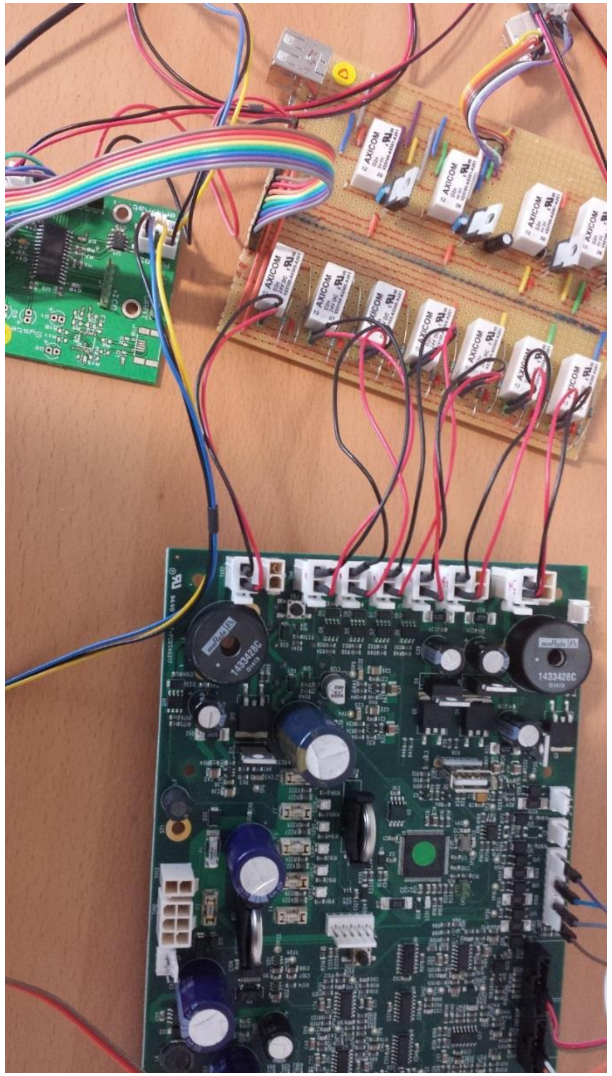
Automated System Test Overview




```
public partial class Target
{
    private PWMCommand _pwm;
    private static TargetDevice _instance = null;

    public static TargetDevice Instance {
        get {
            if (_instance == null)
                _instance = new Target();
            return _instance;
        }
    }
    public int FlowRate {
        set {
            if(!_pwm.SetFrequencyInHz(value)) {
                Assert.Fail("Hardware Error: " +
                  "Unable to set flow rate");
            }
        }
    }
}
```

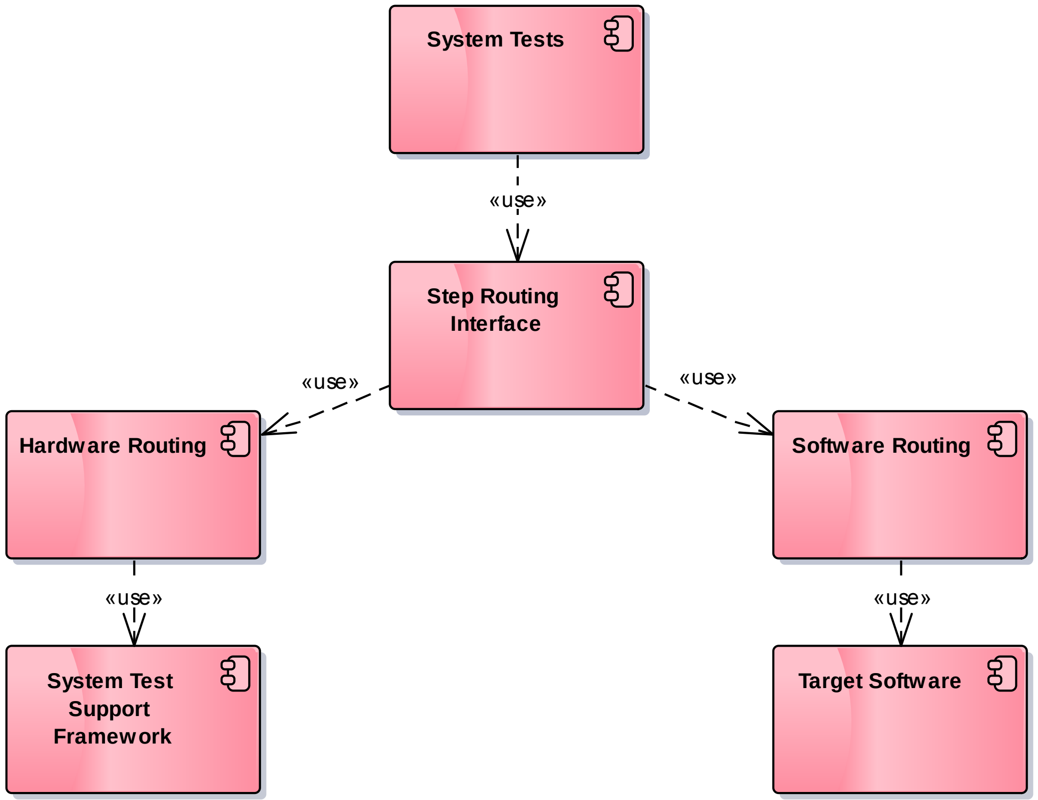
Automated System Test Overview





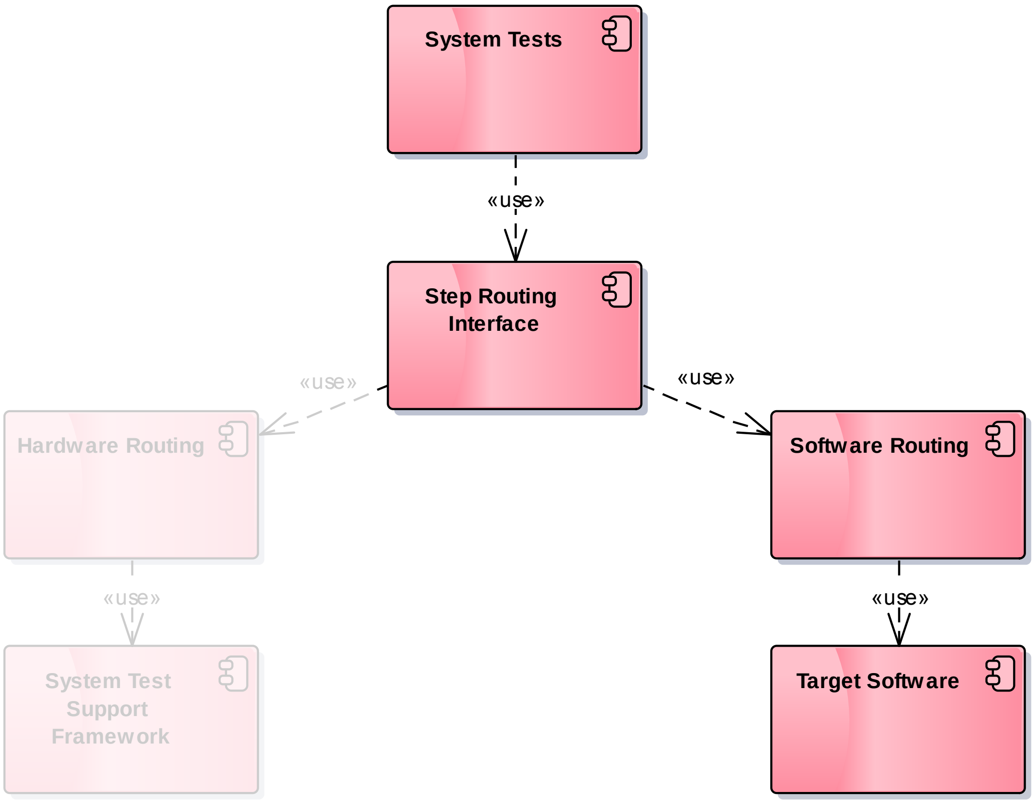
Bluefruit

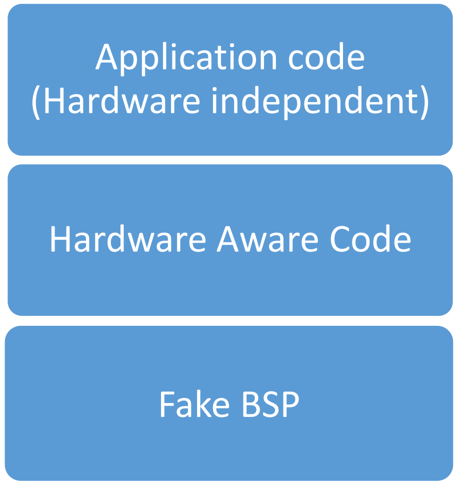
Polymorphic System Test Overview




```
class Target
{
  private static TargetDevice _instance = null;

  public static TargetDevice Instance
  {
    get {
      if (_instance == null) {
        #if CODETARGET
          _instance = new CodeTargetDevice();
        #else
          _instance = new HardwareTargetDevice();
        #endif
      }
      return _instance;
    }
  }
}
```

Polymorphic System Test – Software Routing





Polymorphic System Test
Reduced feedback time

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

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

Bluefruit®