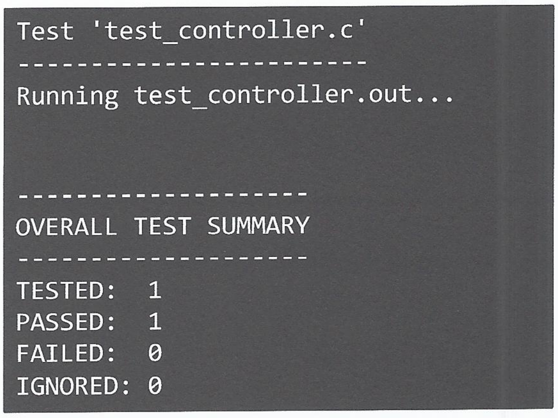
Controller test code

dependent modules

```
#include "unity.h"
#include "controller.h"

#include "ds1820.h"
#include "display.h"

void setUp(void){}
void tearDown(void){}

void test_controller_temperature_warning(void)
{
  float result = update();
  TEST_ASSERT_FLOAT_WITHIN (0.5f, 25.0625f, result);
}
```



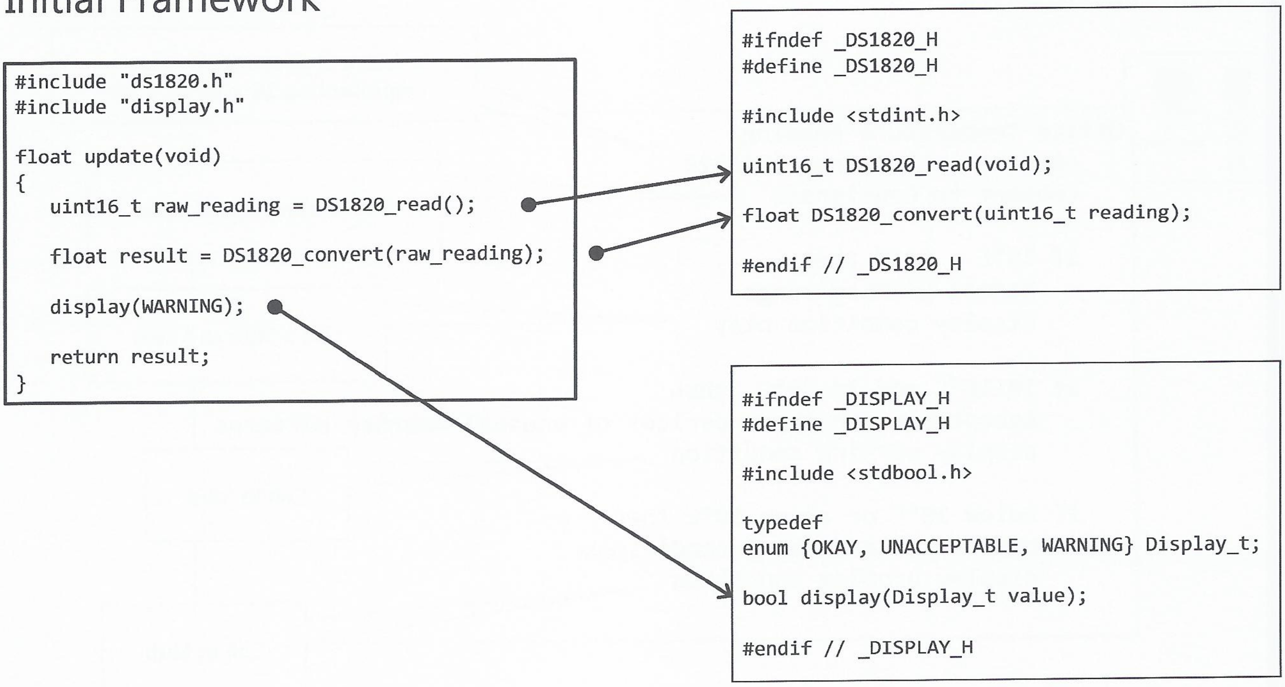
```
Test 'test_controller.c'
-------------------------
Running test_controller.out...


--------------------
OVERALL TEST SUMMARY
--------------------
TESTED:  1
PASSED:  1
FAILED:  0
IGNORED: 0
```

The stub code allows us to compile and link the UUT within the test harness.

When the UUT is actioned within the test harness, the stubs are called.

With Unity, when one module depends on others (e.g. controller depends on ds1820 and display), then the headers for the dependent modules must be included in the test file.

Initial Framework



```
#include "ds1820.h"
#include "display.h"

float update(void)
{
   uint16_t raw_reading = DS1820_read();

   float result = DS1820_convert(raw_reading);

   display(WARNING);

   return result;
}
```

```
#ifndef _DS1820_H
#define _DS1820_H

#include <stdint.h>

uint16_t DS1820_read(void);

float DS1820_convert(uint16_t reading);

#endif // _DS1820_H
```

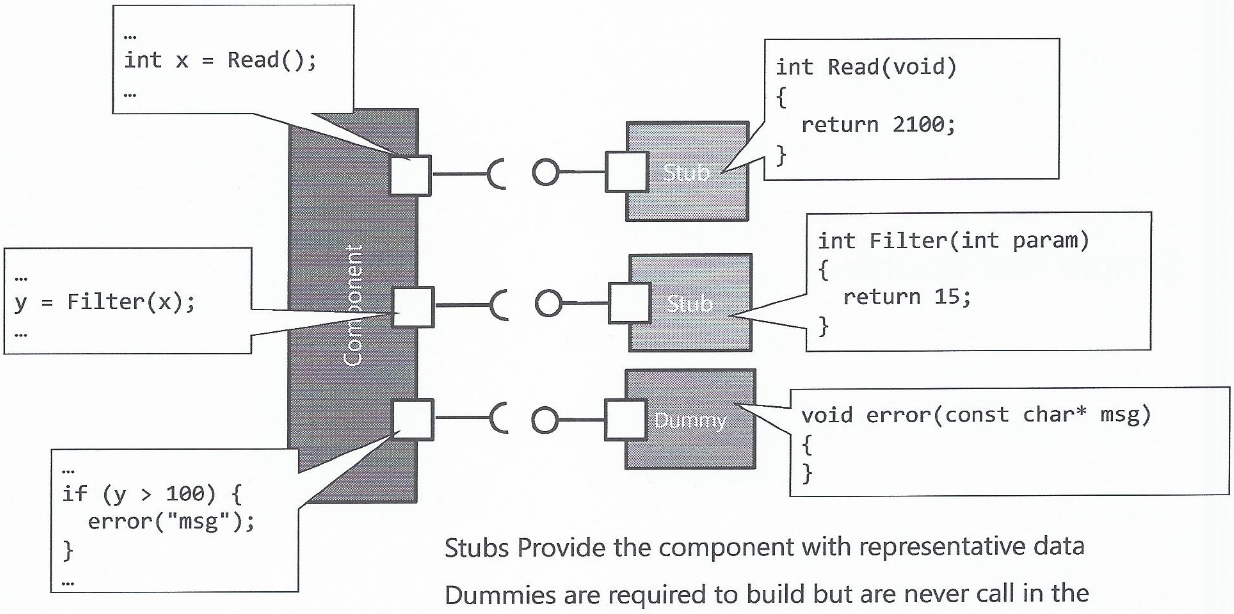
```
#ifndef _DISPLAY_H
#define _DISPLAY_H

#include <stdbool.h>

typedef
enum {OKAY, UNACCEPTABLE, WARNING} Display_t;

bool display(Display_t value);

#endif // _DISPLAY_H
```

The UUT (update function) has dependencies on three lower level functions:

1. DS1820_read
2. DS1820_convert
3. display

The DS1820_convert has already been tested, but to be able to test the update function the other two depended functions require stubbing.

Stubs and Dummies replace actual system code



```
…
int x = Read();
…
```

```
…
y = Filter(x);
…
```

```
…
if (y > 100) {
  error("msg");
}
…
```

```
int Read(void)
{
  return 2100;
}
```

```
int Filter(int param)
{
  return 15;
}
```

```
void error(const char* msg)
{
}
```

Stubs Provide the component with representative data

Dummies are required to build but are never call in the (current) tests

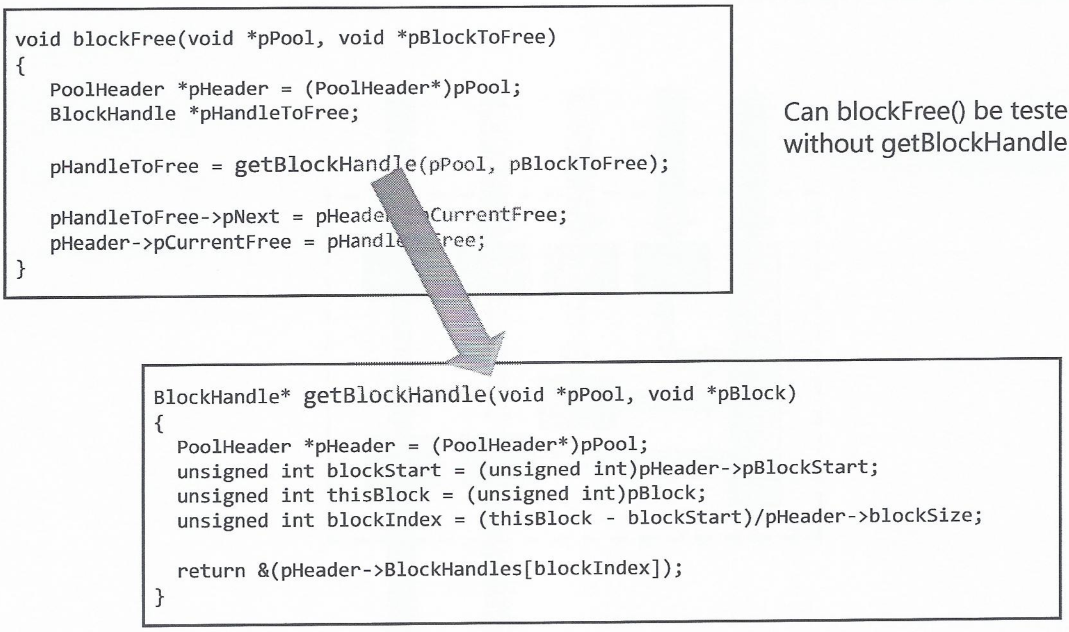
Stubs and Dummies replace the actual system code.

Both have the same interface as the actual system code.

The Stubs provides a simulation of its behaviour. Typically, a stub produces a fixed response, or a narrow range of responses; as long as the responses are adequate to test the behaviour of the unit under test.

The use of stubs and dummies to support higher-level code leads to them sometimes being referred to as scaffold code.

Stubs can be replaced by working code as development continues. Dummies are often replaced by Stubs first, and then working code.



```
void blockFree(void *pPool, void *pBlockToFree)
{
   PoolHeader *pHeader = (PoolHeader*)pPool;
   BlockHandle *pHandleToFree;

   pHandleToFree = getBlockHandle(pPool, pBlockToFree);

   pHandleToFree->pNext = pHeader->pCurrentFree;
   pHeader->pCurrentFree = pHandleToFree;
}
```

```
BlockHandle* getBlockHandle(void *pPool, void *pBlock)
{
  PoolHeader *pHeader = (PoolHeader*)pPool;
  unsigned int blockStart = (unsigned int)pHeader->pBlockStart;
  unsigned int thisBlock = (unsigned int)pBlock;
  unsigned int blockIndex = (thisBlock - blockStart)/pHeader->blockSize;

  return &(pHeader->BlockHandles[blockIndex]);
}
```

Can blockFree() be tested without getBlockHandle()?

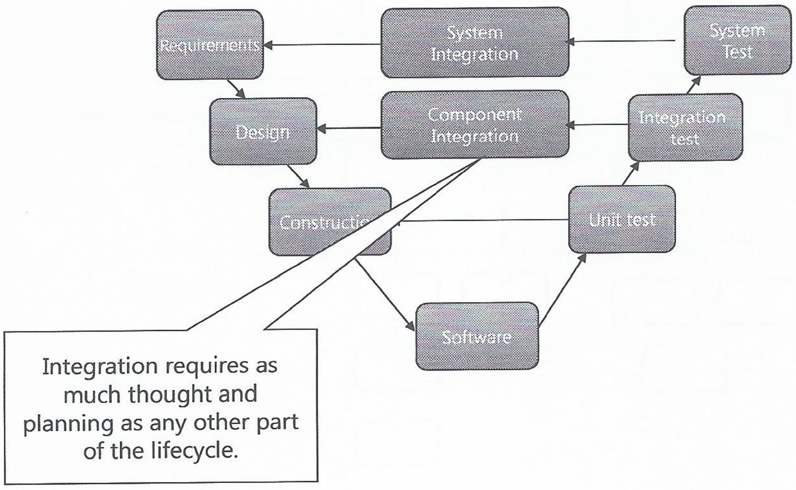
Unit isolation

Slide 8

A simple hierarchy of modules (Top-down integration).

In this example the code is part of a fixed-block memory allocator. When attempting to free a block it is necessary to identify where in the memory pool the block is, and hence locate its management structure (its BlockHandle). This operation is subordinated to the getBlockHandle function.

The function getBlockHandle(...) would have been tested in isolation; but can we test blockfree() in isolation.

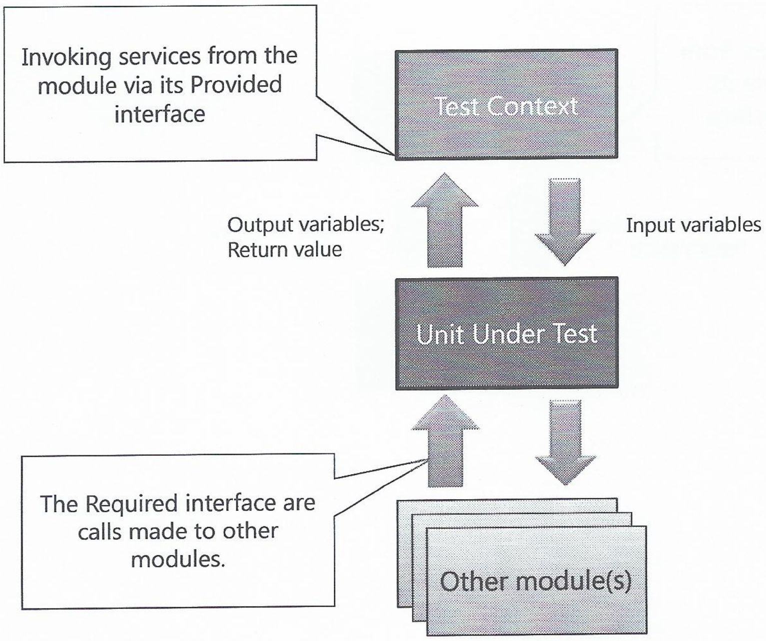


Integration requires as much thought and planning as any other part of the lifecycle.

Integration's place in the V-model.

Integration is considered a testing activity in many organisations; many have a 'Test and Integration' team. Other organisations see it as a Development activity. In some cases Integration gets lost in a 'no-man's land' between development and testing.

Integration must be a planned activity, with its own set of artefacts.



Invoking services from the module via its Provided interface

Test Context

Output variables; Return value

Input variables

Unit Under Test

The Required interface are calls made to other modules.

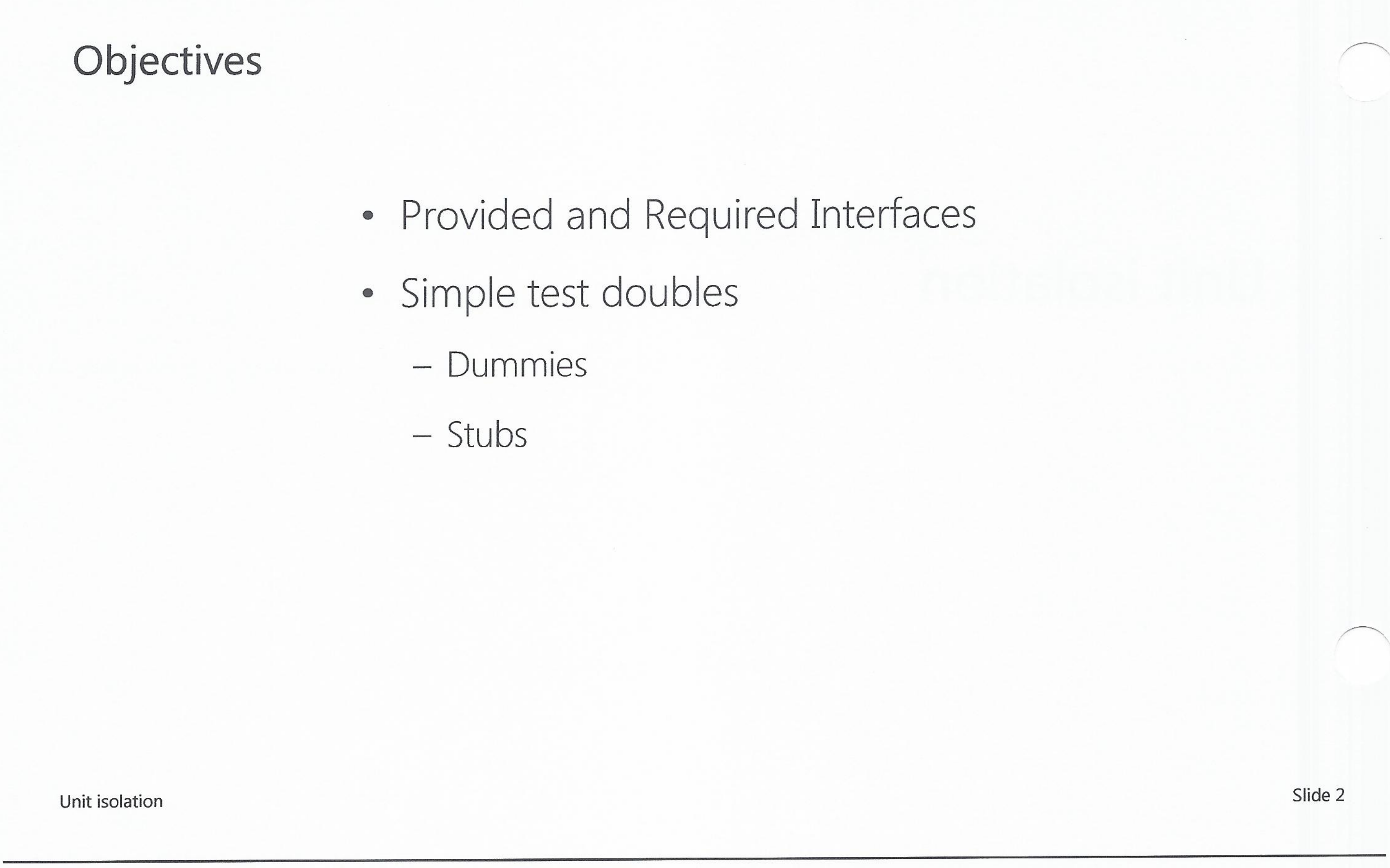
Other module(s)

Unit isolation

Slide 4

In practice, modules typically communicate with other modules. The public interface of a module is therefore the combination of:

- The Provided Interface. This is the services the module provides to its clients.
- The Required Interface. This is the set of calls the module will make to its peers or subordinates in order to fulfil its function.



Objectives

- Provided and Required Interfaces
- Simple test doubles
  - Dummies
  - Stubs

Unit isolation

Slide 2