Software Design And Development (CM2010)

Course Notes

Felipe Balbi

December 29, 2020

Contents

Week 1

Key Concepts

- Define the terms module and module complexity in terms of computer programs and systems.
- Identify the modules present in computer programs and systems.
- Analyse program code in terms of its complexity.

Module Introduction

The objectives of the module are:

1. Write programs using control flow, variables, and functions
2. Use defensive coding and exception handling techniques to prevent processing of invalid data and to handle unexpected events
3. Use Version Control tools to manage a codebase individually and collaboratively (`git`)
4. Define Test-Driven Development and write Unit Tests
5. Assign different categories of module coupling and cohesion to a given program
6. Describe how User Testing can be carried out and evaluated

We will use three different languages throughout the course. They are: C++, Python, and JavaScript.

1.0202 Reference points

Two main references will be used during this course: the *SWEBOK* and ISO/IEC/IEEE 24765:2010 - IEEE Systems And Software Engineering Vocabulary.

SWEBOK is the Software Engineering Body Of Knowledge. From this reference material, we focus on the topic of *Design*.

Design is concerned with the Design fundamentals, key issues of software, software structure and architecture, user interface design, software design quality analysis and evaluation, software design notations, software design strategies and methods, and software design tools.

ISO/IEC/IEEE 24765:2010 is a sort of *dictionary* defining common terms.

1.0203 SWEBOK guide and IEEE vocab

- ISO/IEC/IEEE International standard – Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) Dec 2010, pp.1–418.
- R.E.D. Fairley, P. Bourque and J. Keppler, The impact of SWEBOK Version 3 on software engineering education and training in 2014 IEEE 27th Conference on Software Engineering Education and Training (CSEE&T). (Klagenfurt, Austria: IEEE, 2014).

1.0204 What is a module?

"... a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time". Parnas, 1972.

Once software has been modularized, different parts can be replaced and/or used in different software. Moreover, modularity makes software easier to understand because each small piece can be studied and understood in isolation.

During this course, we define a module as:

- program unit that is discrete and identifiable with respect to **compiling**, **combining** with other units, and **loading**;
- logically separable part of a program;
- set of source code files under version control that can be manipulated as one;
- collection of both data and the routines that act on it.

1.0206 What is module complexity?

Complexity is defined as:

1. The degree to which a system's design or code is **difficult to understand** because of numerous components or relationships among components;
2. Pertaining to any of a set of structure-based **metrics** that measures the attributes in (1);
3. The degree to which a system or component has a design or implementation that is difficult to understand and **verify**.

Simplicity is defined as:

1. The degree to which a system or component has a design or implementation that is **straightforward** and **easy** to understand;
2. Software attributes that provide implementation of functions in the most understandable manner.

1.0208 Complexity references

Please read the following articles. The first is a paper about structural complexity and how it changes over time. The second is the classic McCabe paper on module complexity.

- R.S. Sangwan, P. Vercellone-Smith and P.A. Laplante 'Structural epochs in the complexity of software over time', IEEE Software 25(4) Jul-Aug 2008, pp.66–73.
- T.J. McCabe 'A complexity measure', IEEE Transactions on Software Engineering SE-2(4) Dec 1976, pp.308–320.

Week 2

Key Concepts

- Define module cohesion in terms of computer program architecture.
- Define types of module cohesion and identify them in computer programs.
- Use programming techniques to improve module cohesion.

2.0102 Week 2 reading

This document contains the definitions of various types of module cohesion that you will encounter in the videos. We recommend that you download this and keep it to hand while you watch the videos.

ISO/IEC/IEEE International standard – Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) Dec 2010, pp.1–418.

2.0201 What is module cohesion?

Module cohesion is a way to reason about the contents of a module.

We're concerned about a single module and its contents, not about interactions between modules.

From the ISO Standard Software Engineering Vocabulary, Module Cohesion is defined as:

- Manner and degree to which the tasks performed by a single software module are related to one another;
- In software design, a measure of the strength of association of the elements within a module.

What these tell us is that the *stuff* within a module should be strongly related, otherwise, perhaps, they shouldn't be part of a single module. In summary, a module should do a single thing and a single thing only.

There are several types of module cohesion, they are:

Communicational Type of cohesion in which the tasks performed by a software module use the same input data or contribute to producing the same output data

Functional Type of cohesion in which the tasks performed by a software module all contribute to the performance of a single function

Logical Type of cohesion in which the tasks performed by a software module perform logically similar functions

Procedural Type of cohesion in which the tasks performed by a software all contribute to a given program procedure such as an iteration or decision process

Sequential Type of cohesion in which the output of one task performed by a software module serves as input to another task performed by the module

Temporal Type of cohesion in which the tasks performed by a software module are all requried at a particular phase of program execution

Coincidental Type of cohesion in which the tasks performed by a software module have no functional relationship to one another

2.0204 Why are different types of module cohesion good or bad?

Communicational cohesion is **good** because it's about combining things in a single module that are working on similar data.

Functional cohesion is **good** too. That's because we put into a module functions that work together to achieve a certain goal.

Logical cohesion is generally considered **bad**. Just because software looks like it's doing similar things, doesn't necessarily mean they should be part of the same module.

Procedural cohesion is **bad**. It tends to result in large procedures that do many communicationally different things.

Sequential cohesion is **bad**. This is the idea that a program is doing a sequence of things and, as such, that sequence of things should be put together.

Temporal cohesion is **bad**. This is the idea that because things are happening at the same time, they should be put together.

Coincidental cohesion is **bad**. In fact, this is terrible. Things are put together due to mere coincidence. They just happen to be placed together.

Week 3

Key Concepts

- Give a high-level definition of module coupling and illustrate with an example.
- Analyse computer programs to identify different types of module coupling.
- Describe different types of module coupling and discuss which are desirable and which are not.

3.0102 Week 3 reading

ISO/IEC/IEEE International standard – Systems and software engineering – Vocabulary, ISO/IEC/IEEE 24765:2010(E) Dec 2010, pp.1–418.

3.0201 What is module coupling?

Module Coupling is defined as:

- Manner and degree of interdependence between software modules
- Strength of the relationships between modules
- Measure of how closely connected two routines or modules are
- In software design, a measure of the interdependence among modules in a computer program

3.0203 Different types of coupling: good or bad?

There are different types of module coupling:

Common Environment type of coupling in which two software modules access a common data area

Content type of coupling in which some or all of the contents of one software module are included in the contents of another module

Control type of coupling in which one software module communicates information to another module for the explicit purpose of influencing the latter module's execution

Data type of coupling in which output from one module serves as input to another module

Hybrid type of coupling in which different subsets of the range of values that a data item can assume are used for different and unrelated purposes in different software modules

Pathological type of coupling in which one software module affects or depends upon the internal implementation of another

3.0205 Common environment coupling: good or bad?

Common environment coupling is where two modules have a shared environment. In this environment we have two modules and a block data.

Both modules have access to the data and can change it and do what they like to it as it's shared.

While this sounds like a good idea, it can result in *Race Conditions* where one module changes the data without the other module knowing about it. A better approach would be to have smaller environments of modules each with their own data.

Because of this reason, Common Environment Coupling is not necessarily bad, but one must be careful when implementing it in order to limit its scope.

3.0207 Content coupling: good or bad?

Content coupling would be like having one module (*m1*) with another module (*m2*) contained inside it. It makes it so that nothing outside of *m1* can see *m2* or even know it exists.

It is a regular type of module coupling which is used, for example, is event listeners in JavaScript.

3.0209 Control coupling: good or bad?

Control coupling is when one module is modifying the operation of another one by sending a flag to change how it operates.

Because of that, module 1 tends to become rather complicated. It's easier to illustrate with some python-like code:

```
1 def compute(a, b, op):
2     if op == "add":
3         return a + b
4     else if op == "mul":
5         return a * b
6     else if op == "div":
7         return a / b
```

```
8       else if op == "sub":
9           return a - b
```

While the above is a very contrived example, it's already clear how cumbersome this is. It would have been better to split compute into `add`, `mul`, `div`, and `sub` primitives.

3.0211 Data coupling: good or bad?

Data or Input/Output coupling looks like a production line. The output of one module is fed as input into another. Basically, we can look at it as a function call. Module 1 calls module 2 to run some computation. Data coupling is good.

3.0213 Hybrid coupling: good or bad?

Hybrid coupling is when different modules treat the data source in different ways. In general, this can get confusing and things can go awry very easily.

An example may be that we use a single integer, e.g. 32-bits, and subdivide it into something of our own. Perhaps bit 31 is some Dirty/Clean flag, bits 28:30 hold a state of a state machine, bits 22:27 hold some size information to whatever other memory area and the remaining bits 21:0 hold the top-most 22 bits of the address to a memory area. It should be clear that this can get confusing.

There are cases where this sort of memory usage is the only option, but in general Hybrid Coupling is bad.

3.0215 Pathological coupling: good or bad?

"It's a bit like control coupling, but worse"[1]. Module 1 can either affect the internal workings module 2, or it has a direct dependency in its implementation. This means that module 2 can't be easily replaced, refactored, modified. It's bad.

[1] MYK has awesome comments :-)

Week 4

Key Concepts

- Explain the connection between common programming concepts and module concepts.
- Use language features to improve module coupling and cohesion.
- Use programming techniques to improve module coupling and cohesion.

4.012 Week 4 reading

We will be looking at some JavaScript programming concepts this week. Here are some relevant links:

- w3schools JavaScript scope (2020).
- w3schools JavaScript const (2020).

4.0201 Reasoning about scope

Given the JavaScript code below, let's consider *scope* from the point of view of Module Coupling and Cohesion.

```
1 var x, y, z;
2
3 function addition() {
4   z = x + y;
5 }
6
7 console.log(z);
8 addition();
9 console.log(z);
```

When we run this piece of code, we will see that all variables start with the value of `undefined`, which is expected. When we call `addition()`, `z` gets the value of `NaN`.

From our definition of *Module*, this piece of code can be called a module. The `addition()` function is a little module inside our program which could, potentially, be part of a bigger module.

In terms of coupling, the `addition()` function is coupled to the particular variables `x`, `y`, and `z`. Because all three variables are global, anything in the program can modify them. This is rather dangerous, because the data which `addition()` is coupled with, can be modified "behind its back". If some other part of the program changes the value of `x` to a string and `y` to a number, then `addition()` won't work as expected.

In terms of cohesion, `addition()` and the global variables are really strongly coupled but it's not disconnected from the rest. We want everything in `addition()` to be discrete and disconnected. The obvious fix is to pass `x` and `y` as parameters to `addition()` and have it evaluate a new value for `z`. This way, `addition()` won't depend on the global state.

Therefore, we modify the code to the new version below:

```
var x, y, z;

function addition(p1, p2) {
  return p1 + p2;
}

x = 10;
y = 11;
z = 12;

console.log(z);
z = addition(x, y);
console.log(z);
```

After this modification, we can see that `addition()` isn't depending on the rest of the program. As long as we pass sensible arguments, it contains everything it needs to produce a result that's independent of what's happening on the rest of the program.

4.0203 Reasoning about function parameters

Using the framework of module coupling and cohesion, let's look at function parameters and how to decide what parameters a function needs. The following code snippet is used to motivate the argument:

```
var game_state = {
  'lives': 3,
  'score': 125,
  'level': 4
};

function canHaveExtraLife(current_game_state) {
  if (current_game_state.lives < 2 &&
```

```
9         current_game_state.level > 4) {
10      return true;
11    }
12
13    return false;
14  }
```

Because we're passing the entire `game_state` object as argument to `canHaveExtraLife`, we've coupled that function to the structure of `game_state` object. What this means is that if we want to modify the internal representation of `game_state` we have to modify `canHaveExtraLife` as well.

Instead, we should decouple the function from the internal structure of `game_state` and the easiest way to do that is to simply pass the number of lives and the current level as arguments to the function, instead of the entire `game_state` object.

The modified code is shown below:

```
var game_state = {
  'lives': 3,
  'score': 125,
  'level': 4
};

function canHaveExtraLife(lives, level) {
  if (lives < 2 && level > 4) {
    return true;
  }

  return false;
}
```

Which can be further simplified to:

```
var game_state = {
  'lives': 3,
  'score': 125,
  'level': 4
};

function canHaveExtraLife(lives, level) {
  return lives < 2 && level > 4;
}
```

This function is now decoupled from the internal structure of `game_state`. It has now knowledge that it even exists. A caller only needs to know how to pass the number of lives and current level, regardless of where that data is held.

4.0205 Replacing functions dynamically and const

The snippet below is an example of pathological coupling in JavaScript:

```
function addition(p1, p2) {
  return p1 + p2;
}

function pathos() {
  addition = function(p1, p2) {
    return p1 * p2;
  }
}

var z;
z = addition(10, 12);
console.log(z);

pathos();
z = addition(10, 12);
console.log(z);
```

We can see that `pathos()` re-implements `addition()` and modifies its behavior. It's a clear example of pathological coupling because `pathos()` modifies the inner workings of `addition()`.

To prevent this sort of pathological coupling in JavaScript, we can make use of the `const` keyword to prevent a variable from being modified after its creation.

```
const addition = function(p1, p2) {
  return p1 + p2;
}

function pathos() {
  addition = function(p1, p2) {
    return p1 * p2;
  }
}

var z;
z = addition(10, 12);
console.log(z);

pathos();
z = addition(10, 12);
console.log(z);
```

When we try to run the modified code above, the JavaScript runtime will prevent `pathos()` from modifying `addition` and crash early on. This would force us to remove the pathological coupling.

Week 5

Key Concepts

- Define test-driven development
- Identify the processes involved in test-driven development
- Critically analyse examples of test-driven development in source code repositories

5.0002 Topic reading

- Martin, R.C. Professionalism and test-driven development, IEEE Software 24(3) 2007, pp.32–36.
- IEEE Software 24(3) 2007
- Segura, S. and Z.Q. Zhou, Metamorphic testing 20 years later: a hands-on introduction, 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion) 2018, pp.538–539
- Borle, N., M. Feghhi, E. Stroulia, R. Grenier and A. Hindle [Journal First] Analyzing the effects of test driven development in GitHub, 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE) 2018, pp.1062–1062.

5.0104 The Three Laws according to Uncle Bob

Test-Driven Development is governed by the following three laws:

1. You may not write production code unless you have first written a failing unit test
2. You may not write more of a unit test than is sufficient to fail
3. You may not write more production code than is sufficient to make the failing unit test pass

As an added challenge: we have to complete this process in 2 minutes. The idea is to have the tests guide the development process in short iterative bursts.

5.0113 What to test? An overview

Interface Testing Tests that the function receives the arguments it should receive and returns a value of the expected kind

Exercising Data Structures Verifies that the module under test utilizes the required data structures correctly

Boundary Conditions Testing boundary conditions of functions; i.e. what happens if we send an unexpected value?

Execution Paths In a module that has multiple paths of executions (conditionals), we want to make sure every path is exercised

Error Handling Make sure that errors are properly identified and handled to avoid crashes

Week 6

Key Concepts

- Write unit tests using the Python unittest package
- Describe the elements of a unit testing framework.
- Carry out the test-driven development workflow in Python

6.0102 Week 6 reading

- Python `unittest` Documentation

6.013 Super quick Python tutorial

Variables

Variables don't need to be declared and their type is assigned when initializing. Whenever we need a variable, just type a name and give it a value:

```
myNumber = 10
myString = "Hello"
```

Lists

Python lists are an implementation of a dynamic array. Values in a list are separated by commas and enclosed in square brackets. Individual items in a list can be index like in JavaScript or C.

```
myList = [0, 1, 2, 3, 4]
myOtherList = ["this", "is", "a", "list", "of", "strings"]

myList[2]         # this gives me the number 2
myOtherList[0]        # this gives me the string "this"
```

Python also has nice iterators for lists:

```
myList = [10, 11, 12, 13, 14, 15]

for n in myLists:
    print(n)
```

We can get the length of a list using `len()` and produce a range of numbers using `range()`:

```
myList = [10, 11, 12, 13, 14, 15]

for i in range(len(myList)):
    print(myList[i])
```

We can add more items to a list using `append()`:

```
myList = [10, 11, 12, 13, 14, 15]

myList.append(20)
```

To extend a list with the items from another list, we can use `extend()`:

```
myList = [10, 11, 12, 13, 14, 15]
anotherList = [20, 21, 22, 23, 24, 25]

myList.extend(anotherList)
```

Functions

We define functions in python with the `def` keyword:

```
def increment(n):
    """ Increment argument by 1 """
    return n + 1
```

If we need more than one argument, they should be comma separated:

```
def add(a, b):
    """ Return the addition of a and b """
    return a + b
```

Keep in mind that Python uses *pass-by-value*, not *pass-by-reference*.

6.0201 Introduction to unittest in Python

Python has built-in unit test module named `unittest`. Here's an example of how to use it:

```
import unittest

class TestSomething(unittest.TestCase):
```

```
4       def test_pass(self):
5           self.assertEqual(2 + 2, 4)
6
7       def test_fail(self):
8           self.assertEqual(2 + 2, 42)
9
10  unittest.main(argv = ['ignored', 'v'], exit = False)
```

6.0203 Assertion function in unittest

The `unittest` module from python has a host of assertion functions built-in. Instead of repeating them here, it's to access its documentation for reference to all functions.

6.0205 Assert functions in unittest

- `unittest` classes and functions

6.0301 Introduction to statistics libraries

A statistics library is a program which contains ready-made methods for computing common statistics on a set of data. Typically, these libraries contain methods for computing the mean, min, max, standard deviation, and variance. Some other common, but more advanced, methods would be calculating correlation coefficients, ANOVAS, or histograms.

Week 7

Key Concepts

- Write unit tests using the C++ cppunit package
- Describe the elements of a unit testing framework.
- Carry out the test-driven development workflow in C++

7.0102 Week 7 reading

- cplusplus.com Structure of a program (2020).

7.0201 C++ primer

Usually, a C++ program starts with including some libraries and a `main()` function. We can use the minimal skeleton below:

```
#include <iostream>

int main(void)
{
    return 0;
}
```

Variables

C++ is a strongly typed language. Variables must be explicitly declared, together with their types before they can be used.

```
#include <iostream>

int main(void)
{
    int x = 42;

    printf("x = %i\n", x);

    return 0;
}
```

In order to run this, we must first compile it to generate a runnable binary, we do that at the terminal with the following command (assuming the code above is saved to a file named main.cpp):

```
$ g++ main.cpp -o main
$ ./main
x = 42
```

C++ supports many different primitive data types, for example:

```
#include <iostream>

int main(void)
{
    int x = 42;
    float y = 3.14f;

    printf("x = %i\n", x);
    printf("y = %f\n", y);

    return 0;
}
```

And here we can find a longer description of many of the possible data types in C++.

Arrays

Arrays are declared using the skeleton below:

```
type arrayName[arraySize];
```

For example, for an array of integers, we can use:

```
int numbers[10];
```

Which would give us an array of 10 integers. Indices start at 0, i.e. the first element of the array is 0.

```
#include <iostream>

int main(void)
{
    int x[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };

    printf("x[4] = %i\n", x[4]);

    return 0;
}
```

Loops

C++ supports several kinds of loops. A `for` loop can help us, for example, iterate over the elements of an array and print one by one:

```
#include <iostream>

int main(void)
{
    int x[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 };
    int i;

    for (i = 0; i < 10; i++)
        printf("x[i] = %i\n", x[i]);

    return 0;
}
```

Functions

We can define our own functions in C++. The basic format is:

```
type functionName(type arg1, type arg2, ...)
{
    // function body
}
```

For example, if we want to create a function that takes one integer and produces its successor, we would implement it like so:

```
#include <iostream>

static int succ(int n)
{
    return n + 1;
}

int main(void)
{
    int x = 41;

    printf("successor of %i is %i\n", x, succ(x));

    return 0;
}
```

That `static` keywords tells the compiler that `succ()` is only supposed to be accessible from the current source file.

Classes

C++ is an object oriented language and, as such, has built-in support for classes. The basic syntax for declaring a class is:

```
class ClassName {
public:
    ClassName(type arg1, ...);
    type methodName1(type arg1, ...);
    type methodName2(type arg1, ...);
    ...

    type attribute1;
    type attribute2;
    ...

private:
    type privateAttribute1;
    type privateAttribute2;
    type privateAttribute3;

    type privateMethodName1(type arg1, ...);
    type privateMethodName2(type arg1, ...);
};
```

For example:

```
class Thing {
public:
    Thing(float x, float y)
    {
        this->x = x;
        this->y = y;
    }

    float getX(void)
    {
        return this->x;
    }

private:
    float x;
```

```
    float y;
};

int main(void)
{
    Thing thing {10.2f, 10.5f};

    printf("Thing's X is %f\n", thing.getX());

    return 0;
}
```

7.0206 Introduction to cppunit

cppunit is a C++ Unit Testing framework which we can use to automate tests on C++ software. Here's a simple example:

```
#include <cppunit/TestCase.h>

class BasicTest : public CppUnit::TestCase {
public:
    BasicTest(std::string name) : CppUnitTestCase(name) {}

    void runTest(void) override
    {
        CPPUNIT_ASSERT(2 + 2 == 4);
        CPPUNIT_ASSERT(2 + 2 == 3);
    }
};

int main(void)
{
    BasicTest test{"BasicTest"};
    test.runTest();

    return 0;
}
```

By inheriting from `CppUnit::TestCase`, we get a set of functionality and assertions from `CppUnit` to test the functionality of our code.

When compiling this code, we need to link it against the `cppunit` library. Like so:

```
$ g++ main.cpp -lcppunit -o test
$ ./test
```

```
terminate called after throwing an instance of 'CppUnit::Exception'
  what():  assertion failed
- Expression: 2 + 2 == 3

Aborted
```

7.0208 Adding better output and multiple tests

We can implement some more advanced testing setup with cppunit.

```
#include <cppunit/TestCase.h>
#include <cppunit/TestCaller.h>
#include <cppunit/ui/text/TestRunner.h>

class FixtureTests : public CppUnit::TestFixture {
public:
    void testAddition(void)
    {
        CPPUNIT_ASSERT(2 + 2 == 3);
        CPPUNIT_ASSERT(2 + 2 == 4);
    }
};

int main(void)
{
    CppUnit::TextUi::TestRunner runner {};

    runner.addTest(new CppUnit::TestCaller<FixtureTests> {
            "test the addition operator",
            &FixtureTests::testAddition
        });

    runner.run();

    return 0;
}
```

What this gives us, is an easier way to combine several tests together. Essentially, we can add as many tests as we want, for example:

```
#include <cppunit/TestCase.h>
#include <cppunit/TestCaller.h>
#include <cppunit/ui/text/TestRunner.h>

```

```
5  class FixtureTests : public CppUnit::TestFixture {
6  public:
7      void testAddition(void)
8      {
9          CPPUNIT_ASSERT(2 + 2 == 3);
10         CPPUNIT_ASSERT(2 + 2 == 4);
11     }
12
13     void testLogic(void)
14     {
15         CPPUNIT_ASSERT(true == true);
16         CPPUNIT_ASSERT(true == false);
17     }
18 };
19
20 int main(void)
21 {
22     CppUnit::TextUi::TestRunner runner {};
23
24     runner.addTest(new CppUnit::TestCaller<FixtureTests> {
25             "test the addition operator",
26             &FixtureTests::testAddition
27         });
28
29     runner.addTest(new CppUnit::TestCaller<FixtureTests> {
30             "test the logic",
31             &FixtureTests::testLogic
32         });
33
34     runner.run();
35
36     return 0;
37 }
```

This framework also has support for `setUp()` and `tearDown()` methods which can be used to allocate and free resources before and after each test. Like so:

```
1  #include <cppunit/TestCase.h>
2  #include <cppunit/TestCaller.h>
3  #include <cppunit/ui/text/TestRunner.h>
4
5  class FixtureTests : public CppUnit::TestFixture {
6  public:
7      void setUp(void) override
```

```
8       {
9           printf("Setup called\n");
10      }
11
12      void tearDown(void) overrid
13      {
14          printf("Teardown called\n");
15      }
16
17      void testAddition(void)
18      {
19          CPPUNIT_ASSERT(2 + 2 == 3);
20          CPPUNIT_ASSERT(2 + 2 == 4);
21      }
22
23      void testLogic(void)
24      {
25          CPPUNIT_ASSERT(true == true);
26          CPPUNIT_ASSERT(true == false);
27      }
28  };
29
30  int main(void)
31  {
32      CppUnit::TextUi::TestRunner runner {};
33
34      runner.addTest(new CppUnit::TestCaller<FixtureTests> {
35              "test the addition operator",
36              &FixtureTests::testAddition
37          });
38
39      runner.addTest(new CppUnit::TestCaller<FixtureTests> {
40              "test the logic",
41              &FixtureTests::testLogic
42          });
43
44      runner.run();
45
46      return 0;
47  }
```

7.021 Assert functions in cppunit

- cppunit: Making assertions

Week 8

Key Concepts

- Carry out the test-driven development workflow in JavaScript.
- Write unit tests using the JavaScript Mocha package.
- Describe the elements of a unit testing framework.

8.0102 Week 8 reading

- Node.js (version 12LTS recommended)
- Mocha
- Chai

8.0204 Introduction to Mocha

In an empty directory, we run `npm init` and follow the prompts. At the end, we will have a `package.json` file created for us. It should look similar to this:

```
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
  },
  "author": "",
  "license": "ISC"
}
```

When we run the command `npm test`, the line labeled `test` inside `scripts` will run. Whichever command is placed there, will run. That's how we will run our `mocha` tests.

Before we use `mocha` and `chain`, we have to install them with `npm add mocha chai`. After this command completes, we will be able to use `mocha` and `chai`, so let's update our `package.json` accordingly:

```
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "mocha"
  },
  "author": "",
  "license": "ISC"
}
```

With that setup, we can create a new folder `test` to hold all our tests. When `mocha` runs, it will look for a folder named `test` and will execute the files placed there. Let's create our first test in the file `test.js`.

```
var assert = require('assert')

describe("A feature", function () {
  describe("One test", function () {
    // Passing test
    it("should have two elements", function () {
      var s = "Hello mocha"
      var parts = s.split(" ")

      assert.equal(parts.length, 2)
    })

    // Failing test
    it("should have three elements", function () {
      var s = "Hello mocha"
      var parts = s.split(" ")

      assert.equal(parts.length, 3)
    })
  })
})
```

8.0206 Mocha with es6 syntax

Here's the previous code in ES6 syntax:

```
const assert = require('assert')

```

```
describe("A feature", () => {
  describe("One test", () => {
    // Passing test
    it("should have two elements", () => {
      const s = "Hello mocha"
      const parts = s.split(" ")

      assert.equal(parts.length, 2)
    })

    // Failing test
    it("should have three elements", () => {
      const s = "Hello mocha"
      const parts = s.split(" ")

      assert.equal(parts.length, 3)
    })
  })
})
```

8.0303 Considerations when testing a web API

When we need to run HTTP requests during testing, we can use `chai-http` package. We can use `npm add mocha chai chai-http` to make all necessary packages are installed.

After creating the necessary `test` directory and a `test.js` file, we can fill it up with tests:

```
const chai = require('chai')
const chaiHttp = require('chai-http')
const assert = require('assert')

chai.use(chaiHttp)

describe("Top level / route", () => {
  it("should have 200 status code", (done) => {
    chai.request("http://localhost:3000")
      .get("/")
      .end((err, res) => {
        assert.equal(res.status, 200)
        done()
      })
  })
})
```

Week 9

Key Concepts

- Define what an assertion is in a computer program.
- Explain the difference between assertions and logical control flow.
- Write assertion code and reason about how and when to enable and disable it.

9.0202 Week 9 reading

Opinion 1 Always use assertions

Holzmann, G.J. 'Assertive testing [reliable code]', IEEE Software 32(3) 2015, pp.12–15.

Opinion 2 In industry, assertions are often removed in release builds

Clarke, L.A. and D.S. Rosenblum 'A historical perspective on runtime assertion checking in software development', SIGSOFT Software Engineering Notes 31(3) 2006, pp.25–37.

Other interesting sources:

- What are assertions?
- Classic paper: Hoare, C.A.R 'Assertions: a personal perspective', IEEE Annals of the History of Computing 25(2) 2003, pp.14–25.
- Ariane Rocket explosion: Jazequel, J.-M. and B. Meyer, 'Design by contract: the lessons of Ariane', Computer 30(1) 1997, pp.129–30.

9.0301 Introduction to assertions

*An **Assertion** is a check in your code that evaluates a boolean expression.* Focussing, for now, in *Runtime Assertions* what we're trying to understand is if the program in a desirable state.

There are different types of assertions:

1. Runtime Assertions
2. Unit Tests

3. Compile-time Assertions

An example of a runtime assertion may look like:

```
int computeGameScore(GamePlayer& player)
{
    int gameScore;

    /* ... */

    assert(gameScore >= 0);

    return gameScore;
}
```

When an assertion fails, there are a few options. For each of them we look at their Pros and Cons in the following subsections

Terminating The Program

- **Pros**

  Prevents program from executing anything harmful

- **Cons**

  Prevents legitimate users from using the program

Printing An Error

- **Pros**

  User knows something wrong has happened

- **Cons**

  Program doesn't know the context of the situation

Throwing An Exception

- **Pros**

  Forces caller to deal with the error

- **Cons**

  Caller may not have a good exception resolution method for that particular exception

Carrying On Regardless

- **Pros**

  None

- **Cons**

  Program is very likely to produce bogus outputs

9.0303 Assertions and the software development lifecycle

When should we run assertions? Should we run them in Release Builds, Debug Builds, or both?

We have to remember that a check, small as it may be, consumes CPU cycle and increases object size. According to Clarke and Rosenblum, *"assertion checking is frequently suppressed in production versions of software"* in order to save that space and execution time taken by assertions.

Conversely, Holzmann states that one shouldn't /"disable those carefully crafted assertions when you ship a product to your customer"/. He supports his claim with examples from Microsoft and the JPL where neither disable assertions on production builds.

Week 10

Key Concepts

- Explain the wider context of secure programming techniques.
- Describe and use basic secure programming techniques.
- Give real-world industry examples of secure programming workflows.
- Course level: Define test driven development and write unit tests
- Course level: Use defensive coding and exception handling techniques to prevent processing of invalid data and to handle unexpected events
- Course level: Write programs using variables, control flow and functions
- Course level: Assign different categories of module coupling and cohesion to a given program

10.0102 Week 10 reading

- Classic paper from 1975: Saltzer, J.H. and M. D. Schroeder 'The protection of information in computer systems', Proceedings of the IEEE 63(9) 1975, pp.1278–308
- Wheeler, D.A. Secure programming for Linux and Unix HOWTO (1999).
- Microsoft SDL practices
- Software Assurance Maturity Model
- Building Security in Maturity Model
- Comprehensive, Lightweight Application Security Process (CLASP)/Open Source Foundation for Application Security (OWASP): Introduction to the CLASP process
- OWASP: Top 10 issues for web application security

10.0201 Secure programming overview

The goals of security:

Confidentiality Who can see?

Integrity Who can modify and how?

Availability Can they access it?

Secure programming hit list:

Validate all input regardless of where data comes from, it should be validated before being acted upon

Restrict operations to buffer bounds always verify that we never overflow a buffer

Follow good security design principles follow the list of design principles from Saltzer and Schroeder, 1975.

Carefully call out to other resources when accessing external resources, we must be careful

Send information back judiciously being careful about what's sent back to the user

10.0203 Secure programming and the software development lifecycle

One example software development lifecycle is the Test Driven Development methodology. It starts with writing tests, then writing the minimal amount of code to get the test to pass and moving on to writing more tests, where the cycle restarts.

It can be defined as *a structure for various software development activities to be performed within a project.*

Microsoft has its own Security Development Lifecycle which is defined to be *a set practices that support security assurance and compliance requirements.* I other words, it helps developers build more secure software.

The list of practices from Microsoft SDL is:

1. Provide Training
2. Define Security Requirements
3. Define Metrics and Compliance Reporting
4. Perform Threat Modelling
5. Establish Design Requirements

6. Define and Use Cryptography Standards
7. Manage the Security Risk of Using Third-Party Components
8. Use Approved Tools
9. Perform Static Analysis Security Testing
10. Perform Dynamic Analysis Security Testing
11. Perform Penetration Testing
12. Establish a Standard Incident Response Process

Week 11

Key Concepts

- Define the terms throw, try and catch.
- Differentiate between exceptions, assertions and control flow.
- Write exception handling code that can throw and catch exceptions.

11.0201 Different types of errors

Syntax error error in the syntax of the program

Compile error the compiler can't compile the program

Link error the program fails to link. Usually due to missing definition for some declaration

Non-errors Program is syntactically correct, it builds and links fine, but the implementation is non-sensical; e.g. calculating area of negative width/height.

Runtime error errors that only happen during program execution

11.0203 Introduction to exceptions

Definition of an exception:

1. event that causes suspension of normal program execution
2. indication that an operation request was not performed successfully

"The fundamental idea is to separate the detection of an error (which should be done in a called function) from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored." – Bjarne Stroustrup

The point is that the caller has context while the callee does not. In other words, the function that gets called, doesn't know who called it and for what purpose, therefore it can't possibly know what the correct resolution for the error may be.

11.0205 The try and catch pattern

Assuming we call a function to e.g. verify user credentials, two things can happen:

1. Everything behaves normally

   In this case, we call the function, it checks user credentials and returns whatever it needs to return.

2. An exception is thrown

   In this case, the program will be taken to an alternative execution path, the exception path.

So there are two execution paths: the *everything-is-a-okay* path and the *exception* path. The *Try and catch* pattern says that we will *try* to call the function, assuming it will work and if any exception happens, we will *catch* it for proper handling.

11.0207 Try and catch in javascript

```
try {
  verifyUser();
} catch (ex) {
  console.log(ex.name, ex.message);
}
```

11.0209 Throw in javascript

```
function verifyUser(username, password)
{
  throw {
    name: "DatabaseError",
    message: "Unable to connect to database",
  };
}

try {
  verifyUser("foo", "bar");
} catch (ex) {
  console.log(ex.name, ex.message);
}
```

Week 12

Key Concepts

- Use conditional breakpoints and watchpoints to automatically trigger debugger breaks.
- Explain the key operations that can be carried out with a debugger.
- Use debug operations such as stepping and stack tracing to explore a running program.

12.0201 Introduction to debuggers

Debugging is the process of finding and correcting errors in a program. A *Debugger* is a tool to inspect a running program without modifications to the source code, thereby limiting impact of observing the program behavior.

When we rely on `printf()` or `console.log()` or similar console printing routines, we are producing a crude debugger that has a high impact to the execution pattern of the program.

Debuggers help with dynamic analysis, while compilers and linters help with static analysis.

12.0208 Advanced reading about debugging

- Wong, W.E., R. Gao, Y. Li, R. Abreu and F. Wotawa 'A survey on software fault localization', IEEE Transactions on Software Engineering 42(8) 2016, pp. 707-740.

Week 13

Key Concepts

- Use the 'Easy Approach to Requirements Syntax'(EARS) to write requirements
- Name several methods that people use to write requirements.
- Explain why requirements are necessary and who the stakeholders are.

13.0102 From the topic to the week

- Gregory, S. 'The unplanned journey of a requirements engineer in industry: an introduction', IEEE Software 34(5) 2017, pp.16-19.

13.0202 Introduction to requirements

When designing new Software Systems, we need a way to ensure that the resulting Software does what it's supposed to do. That's where requirements come into the picture.

Requirements are a way of describing what the Software is supposed to do.

When it comes to the *SWEBOK 3.0*, requirements are a top-level area with many sub-areas, including requirements fundamentals requirements process, requirements elicitation, requirements analysis, requirements specification, requirements validation, practical considerations, and software requirements tools.

13.0204 Whirlwind tour of requirements techniques

Looking at the pros and cons of different approaches to Requirement techniques:

	Pros	**Cons**
Natural Language	Easy to understand	Ambiguous
Simplified Technical Language	Less ambiguous	Limited
Unified Modelling Language	Graphical	Can be hard to understand
Z Formal Language	Unambiguous	Hard to understand

13.0206 Back to basics: EARS

The *Easy Approach to Requirements Syntax* (or EARS) is a method developed and tested through a case study converting specification of engines into requirements.

Requirements are written in a standard format containing a pre-condition, a trigger, a system action, and a response:

Pre The user is logged in

Trigger The user clicks on the profile button

System The user profile drop down

Response Appears on the top left

Ubiquitous requirements can further simplify the task of writing requirements with a simple statement of the form *"The <**system name**> shall <**system reponse**>"*.

Event-driven requirements have a format of their own: /"WHEN <**optional preconditions**> <**trigger**> the <**system name**> shall <**system response**>"/

Unwanted behaviors can use the following format: /"IF <**optional preconditions**> <**trigger**>, THEN the <**system name**> shall <**system response**>"/

State-drive requirements look like so: *"WHILE <**in state**> the <**system name**> shall <**system response**>"*

13.0208 EARS reference

- Mavin, A., P. Wilkinson, A. Harwood and M. Novak 'Easy Approach to Requirements Syntax (EARS)', 2009 17th IEEE International Requirements Engineering Conference (Atlanta, GA: IEEE, 2009), pp.317-322.

Week 14

Key Concepts

- Explain how automated blackbox testing can be achieved in video games.
- Differentiate between black box and glass/white box testing and provide examples of each.
- Write out step-by-step and matrix style test procedures.

14.0201 Black box and white box testing

Black box are defined as[1]:

1. System or component whose inputs, outputs, and general function are known but whose contents or implementation are **unknown or irrelevant**.
2. Pertaining to an approach that treats a system or component whose inputs, outputs, and general function are known but whose contents or implementation are **unknown or irrelevant**.

Glass or whitebox is defined as[2]:

1. System or component whose internal contents or implementation are **kwown**.
2. Pertaining to an approach that treats a system or component are in (1).

Unit Testing is an example of white box testing, because unit testing exercises each method in isolation. Requirements can be specified for both black box and white box testing.

14.0203 Testing and the body of knowledge

- *Testing* is an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component[3]

[1]ISO/IEEE Systems and Software Engineering Vocabulary 2017 p47, 3.390
[2]ISO/IEEE Systems and Software Engineering Vocabulary 2017 p199, 3.1744
[3]ISO/IEEE Systems and Software Engineering Vocabulary 2017 p473, 3.4272

- *Test Case Specification* is a document specifying inputs, predicted results, and a set of execution conditions for a test item[4].
- *Test Procedure Specification* is a document specifying a sequence of actions for the execution of a test[5].

There is an ISO standard for softare testing documentation: ISO/IEC/IEEE International Standard Software And Systems Engineering, Software Testing, Part 3: Test Documentation. In ISO/IEC/IEEE 29119-3:2013(E), doi: 10.1109/IEEESTD.2013.6588540.

14.0205 Two short articles about testing taxonomies

- R. L. Glass 'An ancient (but still valid?) look at the classification of testing', IEEE Software 2(6) Nov-Dec 2008, pp.112-112.
- R. L. Glass 'A classification system for testing, part 2', IEEE Software 26(1) Jan-Feb 2009, pp.104-104.

14.0206Test procedure specification: step by step and matrix

Step-by-step Test Procedure

The procedure is specified step-by-step in a standard format. It contains a test identifier, test objective and priority, and an estimated duration.

Any pre-conditions must be defined. This follows with a test log and the list of steps proper.

Matrix Test Procedure

It's similar to step-by-step and contains the same information. The difference lies in how the data is laid out.

14.0210 Factors affecting test effectiveness

- Chernak, Y. 'Validating and improving test-case effectiveness', IEEE Software 18(1) Jan-Feb 2001, pp.81-86.

[4]ISO/IEEE Systems and Software Engineering Vocabulary 2017 p466, 3.4212
[5]ISO/IEEE Systems and Software Engineering Vocabulary 2017 p471, 3.4254

14.0301 Automated black box testing

Video games are a very complex piece of software without a large amount of possibilities and combinations, because of that it becomes very hard to test the game mechanics to an adequate level.

A paper by EA Digital Platform discusses a method used during development of the game *The Sims Mobile* which consists on employing artificial agents to playtest video games. Zhao et al argue that relying exclusively on humans can be costly and inneficient, while artificial agents could perform much longer play sessions and explore much more of the game space in shorter time.

In the paper, Zhao et al talk about the difference between (play) style and skill. The goal in the paper was to ensure that each career within the game has a balanced player experience.

For the automated testing system, they ran a simulation in two parts:

Gameplay environment manages game state in response to agent actions

Agent environment observes game state and generates actions

The game was, then, modeled as state transition probabilities, which allowed them to apply A* Search algorithm to find paths through the state transition graph. In the end, the team could compare career paths based on the amount of work or resources required to climb the levels of that career path. They found that the Barista career path could be exploited and completed in a lower number of actions than the others.

14.0303 Automated testing in video games

- Zhao, Y., I. Borovikov, A. Beirami, J. Harder, J. Kolen, J. Pestrak, J. Pinto, R. Pourabolghasem, H. Chaput, M. Sardari et al. 'Winning isn't everything: Training agents to playtest modern games', AAAI Workshop on Reinforcement Learning in Games 2019.
- Albaghajati, A.M. and M.A.K. Ahmed 'Video game automated testing approaches: an assessment framework', IEEE Transactions on Games.