CM2010: Software Design and Development

Summary

Arjun Muralidharan

10th March 2021

Contents

List of Figures

List of Listings

1 Modules and module complexity

Learning Outcomes

- ✓ Assign different categories of module coupling and cohesion to a given program
- ✓ Write programs using variables, control flow and functions

Important reference points for software design and development are the Software Engineering Body of Knowledge (SWEBOK) and IEEE vocabulary.

1.1 Module Complexity

Modularity A mechanism for improving the **flexibility** and **comprehensibility** of a system while allowing the **shortening** of its development time.

Module A program unit that is discrete and identifiable with respect to **compiling**, **combining** and **loading**. It is a **logically separable** part of a program. It can be represented by a **set of source code files** under version control that can be manipulated together as one. It is a collection of both **data and routines** that act on it (such as a class).

Complexity The degree to which a system's design or code is **difficult to understand** because of **numerous components** or relationships among components, any of a set of structure-based metrics that measure these attributes, or the degree to which a system or component has a design or implementation that is difficult to understand and verify

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

Classical metrics Software complexity can be measured using the **cyclomatic complexity**, that is the number of execution paths through the code, **coupling**, that is how much modules interact with each other, and **cohesion**, that is the amount of functionality in a single module..

Fat and Tangle The terms *fat and tangle* refer to the amount of cohesion and content of a module (*fat*) and the amount of interaction found between modules (*tangle*). Breaking up a program into modules reduces *fat* but increases **tangle**. Issues arise if the tangle is cyclical, that is modules interact with other modules in a circular dependency.

1.1.1 Measures of Complexity

Scientific literature describes multiple approaches to measuring complexity.

McCabe [1] describes **cyclomatic complexity** as a measure based on graph theory. This approach measures the number of unique paths taken through a program calculated as the cyclomatic complexity $v(G)$:

$$v(G) = e - n + 2p$$

for a graph G with e edges, n vertices and p connected components (subgraphs where any two nodes are connected by a path). McCabe then assigned a cyclomatic number to various common control structures such as sequences, conditional statements, and loops as shown in Figure 1. Properties of the cyclomatic number are:

1. $v(G) \geq 1$
2. $v(G)$ is the maximum number of linearly independent paths in G.
3. Inserting or deleting functional statement to G does not affect $v(G)$.
4. G has only one path if and only if $v(G) = 1$.
5. Inserting a new edge in G increases $v(G)$ by 1.
6. $v(G)$ depends only on the decision structure of G.

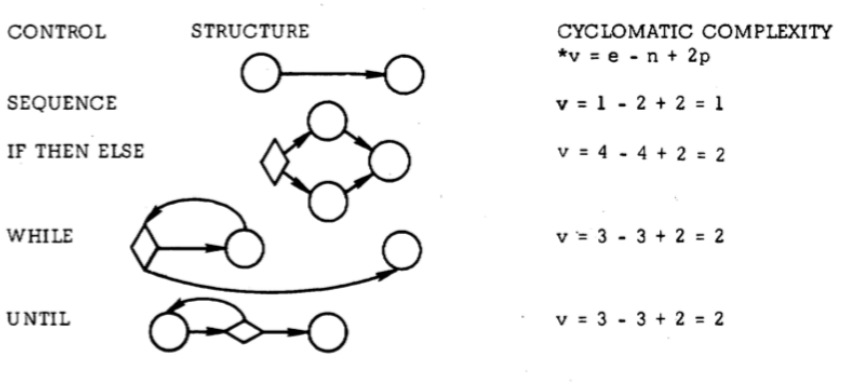

Figure 1. *Cyclomatic complexity using a graph*

Sangwan [2] measured **excessive complexity** based on Structure 101 over time as "structural epochs". This method measures complexity in *fat* and *tangles*, with fat representing the difficulty in understanding a specific module, and tangle representing the number of cyclical dependencies between modules. The method removes as many edges as possible to achieve as close to a *directed acyclic graph* as possible.

Sangwan found in a review of open source projects that excessive complexity shifts upward from lower level modules to higher level modules, and often grows over time until a refactoring occurs, reducing the excessive complexity to an idealized growth line, shown in Figure 2.

Other approaches define **scenario-based** metrics that measure how well a system support specific business scenarios (such as adapting the system to a business change or support for legacy systems).

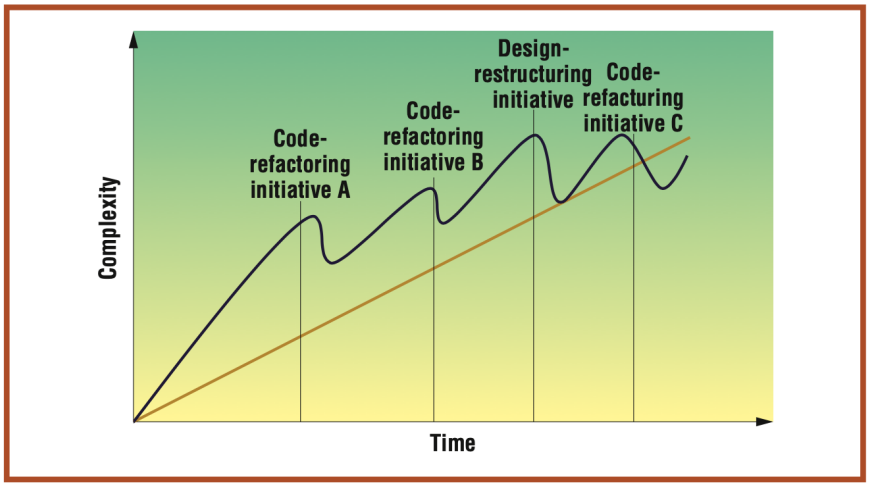

Figure 1. Idealized evolutionary pattern in which complexity grows until code refactoring, design restructuring, or architectural renovation is necessary. (The gold line represents idealized growth of source lines of code over time.)

Figure 2. *Shifts in excessive complexity*

1.2 Module Cohesion

Cohesion is the manner and degree to which the tasks performed by a **single software module** are **related to one another** within a single module. It is the **measure of the strength of association** of the elements in a module.

There are 7 types of cohesion.

Communicational Cohesion The tasks performed by a module use the **same input data** or contribute to producing the **same output data**. This kind of cohesion is **always** acceptable.

Functional Cohesion The tasks performed by a module all **contribute to the performance of a single function**. This kind of cohesion is **always** acceptable. It is also the foundation of object-oriented programming.

Logical Cohesion The tasks performed by a module perform **logically similar functions**. This kind of cohesion can **sometimes** be acceptable but generally considered bad as software that might *look* similar but do completely different things.

Procedural Cohesion The tasks performed by a module all contribute to a **given program procedure** such as an iteration or decision process. This kind of cohesion is **rarely** considered good, as a module works on different data in each step of a procedure.

Sequential Cohesion The **output of one task** performed by a module serves as the **input to another task** performed by the module. This kind of cohesion is **never** acceptable as it combines parts of the program that happen in sequence, but might do completely different things with completely different data.

Temporal Cohesion The tasks performed by a module are all **required at a particular phase of program execution**. This kind of cohesion is **never** acceptable, as it combines program parts that happen at the same time but otherwise might do completely different things, making it hard to understand the individual parts.

Coincidental Cohesion The tasks performed by a module have **no functional relationship** to each other. This kind of cohesion is **never** acceptable as it's completely random.

1.3 Module Coupling

Coupling is the manner and degree of interdependence between modules, the strength of the relationship between modules or how closely related two routines or modules are. There are types of coupling.

Common-environment coupling Type of coupling in which two software modules access a common data area. This is **acceptable** as long as modules are not operating on global data, which might result in unexpected behaviour. Modules should be limited in their access to a specific data area, possibly in sub-environments.

Content Coupling Type of coupling in which some or all of the contents of one software module are included in the contents of another module. An example of this are **Lambda functions** in C++ or Python, or event listeners. This is **acceptable** as long as the contained module really remains contained and not accessed from outside the parent module directly by building bridges into the submodule.

Control coupling Type of coupling in which one software module communicates information to another module for the explicit purpose of influencing the latter module's execution. This is considered **bad** because the latter module can become quite complicated. It is better to move the logic of decision making into the first module and call separate, distinct, clearly understood subroutines from it.

Data Coupling Type of coupling where output from one module serves as input to another module. This is the most common use in the form of functions that have a single purpose and receive some data from another part of the program. This is **acceptable** as long as the receiving function does the same thing with any input data (and does not morph into control coupling).

Hybrid coupling 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 a different software module. For example, two modules might use the same data in completely different ways. Module 1 might use some integer value to store a timestamp, while another uses it to calculate the color of a pixel. This was common when memory was limited, and considered **bad**.

Pathological Coupling Type of coupling in which one module affects or depends upon the internal implementation of another. This is **bad** as it can result in very unpredictable behaviour, which is not desirable outside of perhaps performative programming.

2 Test-driven development

Learning Outcomes

- ✓ Define test driven development and write unit tests
- ✓ Write programs using variables, control flow and functions

2.1 Definition of TDD

Test-driven development is a method of developing software which operates in a repeated 'test and develop' cycle. The first step is to write a test which the software will fail. Next, the code is written to pass the test, then the test is run again.

It is the discipline of writing tests first and working code to pass those tests. Uncle Bob (Robert C. Martin) outlines the three laws of TDD [3]:

1. You may not write production code unless you've 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.

This loop is supposed to occur within 2 minutes, making the development process very stable, resulting in your code being working most of the time.

Unit tests can be of different types:

Interface Testing Programmatic interfaces (not graphical interfaces) can be tested by evaluating that a module receives certain inputs and provides certain outputs. This might including the types of inputs, including how default arguments of a function are handled, and the outputs.

Exercising data structures Verifying data structure and their correct usage. This might include the number of items in a data structure, ensuring a data structure retains integrity and what might

happen at the limits of the data structure (e.g. if it is too large).

Boundary testing We can evaluate what happens when we pass certain inputs at the boundaries of acceptable inputs? This kind of test makes sure that the code works correctly when handling data at boundaries, e.g. have we looked at the beginning and end of an array properly?

Execution Paths Tests that follow all deliberate paths of the module execution ensure that the code works in all possible conditions by sending it different kinds of arguments to explore different pathways. This kind of test would also indicate if **control coupling** is too high in the given module. This might either require some refactoring, or is beyond the scope of unit testing and requires **integration testing** instead.

Error Handling Tests can evaluate if an error is reported properly, and if the reported error matches the encountered error. It also tests if the error is being handled and if the program reaches that error handling.

2.2 Unit Testing in Python

Python provides the `unittest` package that provides a framework for running tests and producing reports from those tests. Tests work by expressing **assertions** about certain expressions in the code. For example, we can assert that two variables need to be equal, and the test will pass if that is true and fail otherwise.

Listing 1 Unit Testing in Python

```
import unittest


class TestSetForOneModule(unittest.TestCase):


        ## Tests need to be prefixed with 'test'
        def test_a_test(self):
                self.assertEqual(10, 10.0)
                self.assertNotEqual(12, 10)
                self.assertTrue(2 == 2)
                self.assertFalse(2 == 4)


                a = 10
                b = a
                c = 10.0
                # assertIs() tests for object identity, not just logical
                ↪  equivalence
                self.assertIs(a,b)
                self.assertIsNot(a,c)


unittest.main(argv=[''], verbosity=2, exit=False)
```

2.3 Unit Testing in C++

One testing framework available in C++ is provided by the `cppunit` library, which has a simple `CPPUNIT_ASSERT` macro function to evaluate expressions. It can be installed on macOS using `brew install cppunit`.

Listing 2 Unit Testing in C++

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

class BasicTest : public CppUnit::TestCase {
public:
        BasicTest(std::string name) : CppUnit::TestCase(name) {};
        void runTest() override
        {
        CPPUNIT_ASSERT(2+2 == 5);
 }
};

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

A basic approach is shown in Listing 2. However, compiling and running this just crashes out of the program if the test fails. In order to improve the handling and output of the tests, we can add a test fixture. This approach allows us to add multiple tests easily with individual function calls, as shown in Listing 3.

Listing 3 Using a test fixture in C++

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

class FixtureTests : public CppUnit::TestFixture {
public:
        void setUp() override {
                printf("Setup is called.\n");
        }

        void tearDown() override {
                printf("Teardown is called.\n");
        }

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

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

int main()
{
        CppUnit::TextUi::TestRunner runner {};
        runner.addTest(new CppUnit::TestCaller<FixtureTests> {
                "Test the addition operator", &FixtureTests::testAddition });

        runner.addTest(new CppUnit::TestCaller<FixtureTests> {
                "Test the logic operator", &FixtureTests::testLogic });

        runner.run();
        return 0;
};
```

Finally, there are two special functions called `setup()` and `tearDown()` that allow us to define code that should run before and after each test is run. Once the test is started, a period is printed, and for each test, the result is printed as `F` or `T`.

2.4 Unit Testing RESTful APIs

A JavaScript framework for testing is provided by Mocha. The framework allows the structural definition of tests in a Node.js application. In addition, an *assertion library* provides the logical evaluations that can be used for the actual tests. One possible library is the ChaiJS assertion library, which provides a range of possible assertions. An extension useful for testing **RESTful APIs** is the `chai-http` plugin, which allows executing HTTP requests and handling the responses in tests. All of these can be installed using `npm`.

The Mocha framework provides a structure for describing tests as shown in Listing 4. Note the `done` callback argument given in the `it` function, as this allows the test to know when the test contents have run fully and returned, since they run asynchronously.

Listing 4 Setting up tests with Mocha

```
describe("Get everything at /spells", () => {
        it("should return successfully", (done) => {
                // Tests and assertions go here
        });
});
```

These tests can be nested, by embedding `describe` clauses within each other. The use of ChaiJS is shown in Listing 5 and this block is placed in the callback function body of the second argument in the `it` function. Note that HTTP operations can be chained, and the response can be asserted on in a further callback function in the `end` method. The assertion is ended with a call to `done()` in order to inform the test framework that the assertions have completed and the tests can continue.

Listing 5 Using Chai to assert against an HTTP response

```
chai.request(app)
        .get("/spells")
        .end((err, res) => {
                assert.equal(res.status, 200);
                done();
        });
```

3 Robust and secure programming

Learning Outcomes

- ✓ Use defensive coding and exception handling techniques to prevent processing of invalid data and to handle unexpected events
- ✓ Write programs using variables, control flow and functions

3.1 Assertions

3.1.1 Definition and Treatment of Assertions

An assertion is a boolean formula that expresses whether a program is in a desirable state. When assertions fail (i.e., they evaluate to false), there are four potential courses of action.

- **Terminate the program.** A program can stop running completely, or stop running part of the the program. It is important the program can maintain state if it does so. For example, websites may crash out and need to be reloaded, but succeed to maintain the state they were in prior to the crash. This might be useful when a certain assertion is critical to pass or the rest of the program might cause damage. For example, the airplane doors need to lock successfully or the plane should not take off.
- **Printing an error.** This might be useful but you might not always know how to communicate with the user. The user might not have a console to print out to, or you will print errors to users who can't do anything about it. If the context is more known, and the users are known, you can print errors.
- **Throw an exception.** Generally a good way to deal with an assertion failure, but you still need to decide what to do about it. This is usually the best way to handle assertion failures.
- **Carry on regardless.** This might be useful if you know that the program can continue without any issues.

3.1.2 Assertions in the SDLC

Assertions might take up additional CPU cycles, such as executing an assertion on every execution of a loop. For example, if you check each time your program divides a number to ensure it doesn't not divide by zero, this can be expensive. Therefore, you might:

- Use assertions in the **debug build**
- Remove all assertions in the **release build**

This approach allows programs to run more efficiently in production where assertions are not strictly needed for the program to run. However, opinions exist that this is bad practice and can make sure

Listing 6 Assertions in C++

```cpp
#include <cassert>

int main()
{
        double sensorReading = 65700;
        // The maximum value in a short is 65535
        unsigned short storedValue = sensorReading;

        // This assertion will fail
        assert(storedValue == sensorReading);

        return 0;
}
```

your production version matches your debug version's execution, and you should improve performance of your program if assertions are slowing it down.

Case Study: Ariane 5 The Ariane 5 rocket blew up on launch in 1996. This happened because:

- A computation that should normally run for a short time was allowed to run for longer to avoid a lengthy system restart.
- The computation involved a conversion from a 64 bit float to a 16 bit integer.
- The result of the computation threw an exception which was not caught.

Using an assertion or a language with built-in assertions and design-by-contract would have prevented this literal crash.

3.1.3 Assertions in C++

Assertions in C++ can be done using the `<cassert>` library, as shown in Listing 6. The program terminates if an assertion error is encountered, and an error is thrown.

3.1.4 Assertions in Python

Python provides built-in assertions via the `assert()` function, as shown in Listing 7. The program terminates if an assertion error is encountered, and an error is thrown.

Listing 7 Assertions in Python

```
sensorReading = 65700
storedValue = sensorReading
assert(storedValue == sensorReading)
```

Listing 8 Assertions in Node.js

```
var assert = require('assert');
var sensorReading = 65700;
var storedValue = sensorReading;
assert(storedValue == sensorReading);
```

3.1.5 Assertions in Node.js

Node.js provides assertions via the `assert` node module, as shown in Listing 8. The program terminates if an assertion error is encountered, and an error is thrown.

3.2 Secure Programming

David Wheeler [4] outlines general principles on secure programming.

Security Goals

- **Confidentiality:** Who can see?
- **Integrity:** Who can modify and how?
- **Availability:** Can they access it?

Security Hitlist

- Validate all input
- Restrict operations to buffer bounds
- Follow good design principles [5]
  - Least privilege
  - Economy of mechanism / simplicity
  - Open design
  - Complete mediation: check every access
  - Failsafe defaults

  - Separation of privilege
  - Least common mechanism
  - Psychological acceptability / easy to use
- Carefully call out to other resources
  - Library routines
  - Databases
  - External programs
  - Files
- Send information back judiciously (e.g. passwords or debug print-outs)

3.2.1 Secure Programming and the SDLC

Microsoft has developed a detailed security development lifecycle [6]. We will look at three of those practices in particular.

Manage the security risk of using third-party components

- Keep an inventory of any third party components used
- Perform security analysis of those components
- Keep third party components up to date

Use approved tools Based on similar practices as third-party components, it's good practice to ensure that tools used have some form of approval from an authority.

Perform static analysis security testing Static analysis involves analysing source code (as opposed to dynamic analysis, which looks at running software) and identify security issues.

Other frameworks are SAMM [7], BSIMM [8] and OWASP [9]. A real-world tool to perform static analysis is `bandit` [10], which is a Python package that can analyze Python files for common security problems.

3.3 Exception Handling

3.3.1 Types of Errors

We can classify errors into different types listed below.

Syntax Errors An error that occurs when we don't follow the rules of the language. For example, when we name a variable `@#!-` which are invalid characters for a variable in most languages.

Compile/Interpret Errors An error when the program has a structural mistake, such as calling a function with the wrong number of arguments.

Link errors In compiled languages such as C++, this error occurs if an implementation file refers to code that relies on a header file, which in turn is either not present or does not contain the definitions of your functions. For example, calling a function that wasn't defined in the class declaration in a header file.

Non-errors An error that occurs when you use a function in a non-sensical way or an error in business logic. This will not surface as an actual, technical error during development unless you test for it specifically.

Runtime error An error that occurs only at runtime, e.g. when a variable gets an unexpected value assigned, or a divide-by-zero situation occurs.

3.4 Exceptions

An **exception** is an event that causes suspension of normal program execution, or an indication that an operation request was not performed successfully.

A fundamental principle when dealing with exceptions is to **separate detection from handling**. Detection should happen in a **called function**, while handling should happen in the **calling function**, because the **caller has context** that the callee does not.

Assertions are used in conjunction with exceptions, where an assertion might escalate to an exception if something has gone wrong.

An assertion is (1) a logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution, or (2) a function or macro that complains loudly if a design assumption on which the code is based is not true.

Exceptions can behave like **control flow** blocks, but they should not be used to implement actual control flow in a program.

Try-Catch Pattern In programming, assertions and exceptions are often represented using the `try-catch` pattern, which "try" a specific assertion, and escalate to an exception if one is thrown. The called function inside the try-block has no context and only throws an exception. The catch-block then handles the exception, which is done at the caller level.

Try-Catch in Javascript As shown in Listing 9, we can use a try-catch block to call a function and handle an exception. Additionally, we can specify the exception more precisely by using a `throw`

Listing 9 Try-Catch in Javascript

```
function verifyUser(username, password) {
        throw {name: "DatabaseError:", message:"Could not connect
        ↪  to the database."}
}

try {
        verifyUser();
        console.log("After verifyUser");
} catch (error) {
        console.log("Exception caught");
        console.log("Name:" + ex.name);
        console.log("Message:" + ex.message);
}
console.log("I'm still running.");
```

statement that defines a dictionary for the exception to use. If this is not done, JavaScript uses its own default library of errors [11].

3.5 Debugging

Debugging is simply the process of removing errors from your code. A **debugger** is a tool that you can use to inspect your program as it is running. This allows **dynamic analysis** as opposed to static analysis.

Debugging in C++ We can use `gdb` to debug C++ programs. This is done by invoking the compile command with certain flags to produce a debuggable executable.

```
g++ -g debug.cpp -o debugme
```

This executable can then be debugged by launching the debugger with the following two commands.

```
gdb debugme
run
```

This will just run the program in debug mode, but in order to investigate the program, we need to set breakpoints.

```
gdb debugme
run
```

At the `gdb` prompt, you can set a breakpoint at a line, e.g. line 9, with `break 9`. Executing `run` then runs the program until the breakpoint is reached. We can now inspect the program with the following commands.

1. `print i` to print a variable i
2. `info locals` to display all local variables
3. `info variables` to display all variables the program can see from that point
4. `info args` to display the current arguments of a function
5. `info stack` to see the entire stack trace from the current point upwards
6. `next` to continue one line in the program
7. `step` to continue in the program and step into function calls

Conditional Breakpoints We can set breakpoints that only break if a condition is met, e.g. `break 9 if i > 2`.

Watch Points We can ask the program to break when the value of a certain watched variable changes, with `watch i`.

4 User Testing

Learning Outcomes

✓ Describe how user testing can be carried out and evaluated

4.1 Requirements

How to ensure that software does what it is supposed to do?

We describe what a system is supposed to do as **requirements**. These are addressed to stakeholders making the software and buying the software, as well as programmers, but not necessarily end users.

4.1.1 Techniques

Natural Language Suitable for most stakeholders and able to express most requirements but leaves room for interpretation.

Simplified technical English Uses short sentences, short paragraphs and limited grammar. It's written in an active voice and useful to address non-native English speakers.

Unified Modeling Language (UML) Allows semi-formal definition of requirements using diagrams specifying classes, entities and relationships, in addition to flowcharts.

Z A formal language that can be used for research settings, where mathematical precision is required. This is currently trending back due to the popularity of machine learning. Allows expressing how a system works through a logical argument and processes information.

4.1.2 EARS

The **Easy Approach to Requirements Syntax** for requirements engineering allows defining requirements using natural language in a constrained way.

Generic Requirements The structure for a generic requirement is as follows.

```
<optional preconditions> <optional trigger> the <system name> shall <system
↪  response>
```

Ubiquitous requirements A *ubiquitous* requirement has no preconditions or trigger. It is not invoked by an event detected at the system boundary or in response to a defined system state, but is always active.

```
The <system name> shall <system response>
```

Event-driven requirements An event-driven requirement is initiated only when a triggering event is detected at the system boundary. The keyword When is used for event-driven requirements. The general form of an event-driven requirement is:

```
WHEN <optional preconditions> <trigger> the <system name> shall <system response>
```

Unwanted behaviours Requirements to handle unwanted behaviour are defined using a syntax derived from event-driven requirements.

```
IF <optional preconditions> <trigger>, THEN the <system name> shall <system
↪  response>
```

State-driven requirements A state-driven requirement is active while the system is in a defined state.

```
WHILE <in a specific state> the <system name> shall <system response>
```

4.2 White & Black Box Testing

Black Box Inputs, outputs, and general function are known but the contents and implementation are unknown.

White Box Internal contents or implementation are known.

Unit testing is a kind of white box testing, since the internal contents and implementation of the system are known when writing unit tests.

Requirements can be formulated for both approaches.

4.2.1 Testing & SWEBOK

SWEBOK outlines a few distinct concepts related to testing.

Test Case Specifications Very similar to requirements, these specify predicted results, and a set of execution conditions for a test item.

Test procedure specification These specify a sequence of actions for the execution of a test. A testing procedure specification may include:

1. Test ID
2. Objective and Priority
3. Estimated Time
4. Preconditions / Startup Conditions
5. Test Log (documenting the pass/fail status)

Some test specifications might additionally use a **matrix test** that document the expected outcomes for various user roles or system states (for example, documenting which user roles have which permissions).

4.2.2 Automated Black Box Testing

Some systems might not be feasible to test with humans, such as testing all possible outcomes in an open world video game.

Automated gameplay agents are used to test these systems, and they employ various **skills** and **styles**, as shown in figure 3 and described in [12].

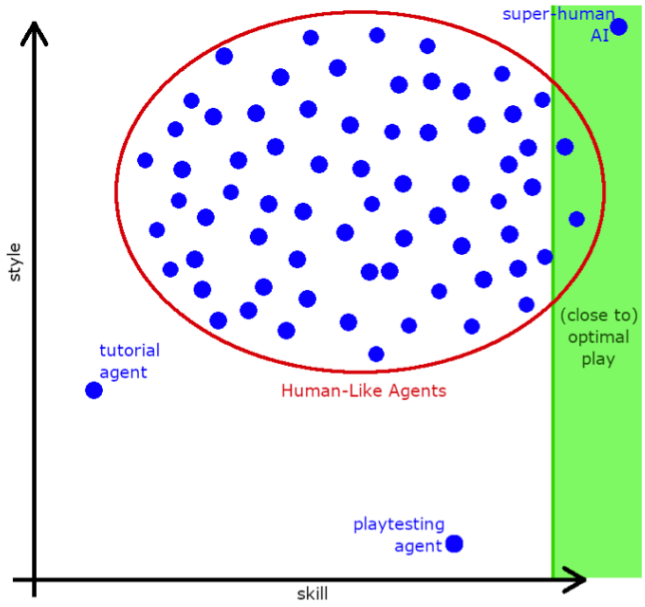

Figure 3. *Playtesting Agents*

4.3 Usability

4.3.1 Measuring Usability

Usability is the extent to which a system, product or service can be used by specified users to achieve specified goals with **effectiveness** , **efficiency** and **satisfaction** in a specified context of use.

There are various metrics you can use for measuring usability.

System Usability Scale (SUS) A set of 10 questions that users are asked about and can rate on a scale of 1 to 5.

Usability metric for user experience (UMUX) This metric asks the user a set of 4 questions, rated on a 7-point scale. This metric correlates highly with SUS, so it's a faster way to get the same result with slightly less nuance.

Creativity Support Index Explores dimensions of *collaboration*, *enjoyment*, *exploration*, *expressiveness*, *immersion* and *results worth the effort*. This method is a lot less popular but useful for software in creative fields.

4.3.2 Usability Principles

Nielsen's 10 principles of usability are:

1. **Visibility of system status.** Do we know what state the system is in at a given time?

2. **Match between system and real world.** Does the system leverage existing, real-world metaphors (e.g. files and folders). This principle is a bit dated today.
3. **User control and freedom.** Does the user ever get stuck in a specific mode or state?
4. **Consistency and standards.** Does the system behave the same across its various components?
5. **Flexibility and efficiency of use.** Can I do the same thing in different ways?
6. **Aesthetic and minimalist design.** Do I have any unnecessary elements on screen?
7. **Help users recognize, diagnose and recover from errors.** Does the system show good error messages when things go wrong?

4.3.3 Accessibility

Accessibility is the extent to which products, systems, services, environments and facilities can be used by **people from a population with the widest range of characteristics and capabilities** to achieve a specified goal in a specified context of use.

Accessibility requirement is legally defined in the UK as the requirement to make a website or mobile application accessible by making it **perceivable, operable, understandable, and robust**.

A **disability** is defined as a long-standing illness, disability or impairment which causes substantial difficulty with day-to-day activities. Statistics in the UK track disabilities such as *Mobility, Stamina, Dexterity, Mental Health, Memory, Hearing, Vision, Learning, and Social/Behavioural.*

5 Version control

Learning Outcomes

✓ Use version control tools to manage a codebase individually and collaboratively

5.1 Version Control

When working on some code, some subset of this code is usually used in a **release**.

There are various scenarios that illustrate the need for version control.

Scenario 1: One developer latest release only A single developer is working on code, and there is only one release out in the wild. An update will overwrite the previous release. We don't need to care about previous code or previous releases.

Scenario 2: Multiple releases Code is growing over time, and a new release lives in the wild alongside older releases. We need to maintain codebases from the past relating to older releases. Fixes

to older releases might need to be propagated to newer releases.

Scenario 3: Multiple developers All working on the code, usually working on separate files, but they need to manage conflicts when working on the same code base.

Some version control systems include:

1. Source Code Control System, 1975
2. Concurrent Versions System, 1986
3. Git, 2005

Design goals of **Git** are:

1. Distributed
2. Good performance
3. What comes out is what came in (file corruption checking)

Git allows for distributed work on a code base.

5.2 Git

A Git repository contains all the information about the working tree and history, stored in the `.git` folder.

1. `git init` - Initializes an empty Git repository
2. `git status` - shows the current state of the repository
3. `git commit` - store the file into the repository
4. `git log` - show the history of the repository
5.
6. `git branch` - list the available branches
7. `git checkout main` - switch to the main branch
8. `git checkout -b feature` - create a new branch names “feature”.
9. `git merge feature` - merge the feature branch onto my current branch
10. `git clone` - clone a branch from a remote repository locally.
11. `git pull` - get the latest version of this repository from a remote location.
12. `git push` - upload my local repository to the remote location.
13. `git log --pretty=reference` - show a compact history of the repository.

14. `git log --pretty=fuller` - show a more verbose history.
15. `git log -3` - show the last 3 commits.
16. `git log --graph --oneline --decorate --all` - show a visualisation of the git history with branches

Merge conflicts need to be resolved manually by examining the conflicting file and looking for the `HEAD` keyword and choosing the right hunk to merge.

A merge will remain open and await closure as long as the conflict is unresolved and we instruct Git to do abort or re-attempt the merge.

We can use `ungit` to explore repositories visually.

References

[1] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976. [Online]. Available: https://ieeexplore.ieee.org/document/1702388

[2] R. S. Sangwan, P. Vercellone-Smith, and P. A. Laplante, "Structural epochs in the complexity of software over time," *IEEE Software*, vol. 25, no. 4, pp. 66–73, 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4548410

[3] R. C. Martin, "Professionalism and test-driven development," *IEEE Software*, vol. 24, no. 3, pp. 32–36, 2007. [Online]. Available: https://ieeexplore.ieee.org/document/4163026

[4] D. Wheeler, "Secure programming howto," 2015. [Online]. Available: https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/

[5] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[6] Microsoft, "Microsoft security development lifecycle practices." [Online]. Available: https://www.microsoft.com/en-us/securityengineering/sdl/practices

[7] P. Chandra, "Software assurance maturity model." [Online]. Available: https://www.opensamm.org

[8] BSIMM, "Building security in maturity model." [Online]. Available: https://www.bsimm.com

[9] OWASP, "Owasp top ten." [Online]. Available: https://owasp.org/www-project-top-ten/

[10] PyPI, "Bandit." [Online]. Available: https://pypi.org/project/bandit/

[11] M. W. Docs, "Javascript error reference." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors

[12] Y. Zhao, I. Borovikov, F. de Mesentier Silva, A. Beirami, J. Rupert, C. Somers, J. Harder, J. Kolen, J. Pinto, R. Pourabolghasem, J. Pestrak, H. Chaput, M. Sardari, L. Lin, S. Narravula, N. Aghdaie, and K. Zaman, "Winning isn't everything: Enhancing game development with intelligent agents," 2020.