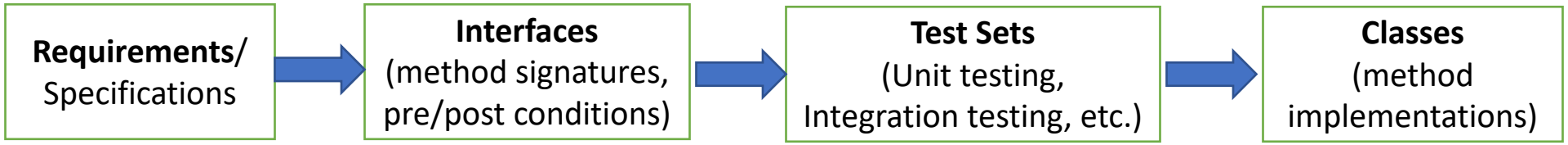
COMP2511
Test Design

"Testing shows the presence, not the absence of bugs."

—*Edsger W. Dijkstra*

Software Testing: Test-Driven Development (TDD)

- Every iteration in the software development process must be preceded with a plan to properly verify (test) that the developed software meets the requirements (i.e., post conditions).
- A software developer must not create a software artifact and later think of how to test it!
- Testing is an essential integral part of developing a software solution. It must not be considered as an afterthought !
- Incremental development must be tested against test suites, during every iteration. Every code modification and/or refactoring must be followed by a proper testing, using the predefined test suites.
- Testing must be setup, based on the requirement specifications, **before** you start implementing your solution.



Software Testing: Input Space Coverage

- Testing must **not** be conducted haphazardly by trial-and-error.
- Testing must be conducted systematically, with a well thought out **testing plan**.
- The aim should be to consider a possible *input space* and cover it as much as possible.
- Often this is achieved by dividing the *input space* into "*equivalence groups*" and selecting a representative input from each equivalence group. Here, the assumption is: from the same equivalence group, a program is expected to behave similarly on each input.
- For example, for the method `boolean isSorted (list);`
  possible input cases to consider,
  - Input list: `34, 12, 15, 21, 5, 21.` [random all positive]
  - Input list: `-13, –12, –77, –60, –55.` [random all negative]
  - Input list: `10, –11, 17, 31, 50, 42.` [random mix positive/negative]
  - Input list: `22, 22, 22, 22, 22, 22.` [all same]
  - Input list: `3, 7, 34, 41, 53, 99.` [increasing order]
  - Input list: `99, 45, 0, –10, –34, –89.` [decreasing order]
  - Etc.
  - Etc.

Software Testing: Input Space Coverage

- Consider borderline cases, often called boundary testing.
- For example, for the method `String getGrade( marks );` possible input cases to consider,
  - 0
  - 50
  - 65
  - 75
  - 85
  - 100

- For multiple input values, consider possible input combinations, prioritise them and consider as many as possible, given the available time and resources. Again, divide possible combinations into *homogenous* subsets and select representative combinations.

Software Testing: Code Coverage

- Code coverage is a useful metric that can help you assess the quality of your test suite.
- Code coverage measures the degree to which a software is verified by a test suite, by determining the number of lines of code that is successfully validated by the test suite.

- The common metrics in most coverage reports include:
  - Function coverage: how many of the functions defined have been called.
  - Statement coverage: how many of the statements in the program have been executed.
  - Branches coverage: how many of the branches of the control structures (if statements for instance) have been executed.
  - Condition coverage: how many of the boolean sub-expressions have been tested for a true and a false value.
  - Line coverage: how many of lines of source code have been tested.

- For more, see https://www.atlassian.com/continuous-delivery/software-testing/code-coverage

Randomness in Software Testing and Simulation

Randomness is also useful!

- Software **Testing**:
  - random data is often seen as *unbiased data*
    - gives average performance (e.g. in sorting algorithms)
  - *stress test* components by bombarding them with random data

- Software **Simulation**:
  - generating random behaviours/movements.
    For example, may want players/enemies to move in a random pattern.
    Possible approach: randomly generate a number between 0 to 3,
    - **0** means **front** movement, **1** means **left** movement,
      2 means **back** movement, **3** means **right** movement.
  - the layout of a dungeon may be randomly generated
  - may want to introduce unpredictability

Random Numbers

- How can a computer pick a number at random?
  - it cannot !

- Software can only produce *pseudo random numbers*.
  - a pseudo random number is one that is predictable!
    - (although it may appear unpredictable)

- Implementation may deviate from expected theoretical behaviour.

Generating Random Numbers in Java

Using random class,

- Need to import the class `java.util.Random`
- ***Option-1:*** Creates a new random number generator.
  - `Random rand = new Random();`

- ***Option-2:*** Creates a new random number generator using a single long seed.
  - *Important:* Every time you run a program with the **same seed**, you get exactly the **same sequence** of *'random'* numbers.

    `Random rand = new Random(long seed);`
- To vary the output, we can give the random seeder a starting point that varies with time. For example, a starting point (seed) is the current time.

- Go to the API for more information at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html

Basic Test Template

1) Set up Precondition (i.e. @BeforeEach, etc.)
2) Act (call the method)
3) Verify Post condition (i.e. @AfterEach, Asserts, etc.)

- Normally, each test should run independently, order of execution should not be important.
- However, if required, you can order execution of the tests using @TestMethodOrder

Avoid Repetition in Test Suites: Parameterized Tests

- JUnit offers *Parameterized Tests.*
- *Parameterized test* executes the same test over and over again using different input values and tests output against the corresponding expected results.
- A data source can be used to retrieve data for input values and expected results.
- The @Before annotation can be used if you want to execute some statement such as preconditions before each test case.
- The @After annotation can be used if you want to execute some statements after each Test Case for e.g. resetting variables, deleting temporary files, variables, etc.
- For more information, see JUnit 5 tutorial .

Parameterized Test: An Example

```
5   public class UsingParameterizedTest {
6
7       public static int[][] data() {
8           return new int[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
9       }
10
11      @ParameterizedTest
12      @MethodSource(value =  "data")
13      void testWithStringParameter(int[] data) {
14          MyClass tester = new MyClass();
15          int m1 = data[0];
16          int m2 = data[1];
17          int expected = data[2];
18          assertEquals(expected, tester.multiply(m1, m2));
19      }
20
21      // class to be tested
22      class MyClass {
23          public int multiply(int i, int j) {
24              return i * j;
25          }
26      }
27  }
```

For more information, see JUnit 5 tutorial at https://www.vogella.com/tutorials/JUnit/article.html.

Avoid Repetition in Test Suites: Dynamic Tests

- JUnit also offer Dynamic Tests.
- For more information, see JUnit 5 tutorial .

```java
class DynamicTestCreationTest {

    @TestFactory
    Stream<DynamicTest> testDifferentMultiplyOperations() {
        MyClass tester = new MyClass();
        int[][] data = new int[][] { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.stream(data).map(entry -> {
            int m1 = entry[0];
            int m2 = entry[1];
            int expected = entry[2];
            return dynamicTest(m1 + " * " + m2 + " = " + expected, () -> {
                assertEquals(expected, tester.multiply(m1, m2));
            });
        });
    }

    // class to be tested
    class MyClass {
        public int multiply(int i, int j) {
            return i * j;
        }
    }
}
```

For more information, see JUnit 5 tutorial at https://www.vogella.com/tutorials/JUnit/article.html.

Types of tests

Spectrum of Tests: **Unit** Tests, **Integration** Tests, **Systems** Tests, **Usability** tests / **acceptance** tests

Unit test

- Test a single piece of functionality.
- Ideally, should test in isolation – scientific method, keep all other variables controlled.
- Minimise dependencies on other functionality.
- Therefore, difficult to write black-box unit tests
  - We can make our tests unit-like by reducing the number of dependencies as much as possible.
  - Mock testing and mock objects can be used, though this requires knowledge of what functionality the method relies on.
  - Not easy to say whether changes took place testing a single method without calling another method!

Types of tests

Integration test

- Test a web of dependencies (coupling) that catches any bugs “lurking in the cracks” that the unit tests didn’t pick up.
- Every failing integration test should be able to be written as a failing unit test.
- Tests interactions (couplings) between software components.

System test

- Perform a black-box test on the entire system as a whole.
- This can be done at different levels of abstraction, for COMP2511 we system test at the controller level.

Types of tests

Usability tests / acceptance tests

- “Test that it works on the frontend”
- Does the functionality achieve the intended goal?
- Is it usable?

Property-based tests

- Test individual properties of code rather than testing the output directly

Creating a Testing Plan

Need to properly devise a way for testing that ensure:

- high coverage.
- there is a mix of different types of tests.

Beware!

- Writing too many tests is **bad** – if you have unit, integration and system tests all for the same thing then the test suite becomes **tightly coupled** and hard to maintain.
- Need to strike a **balance** – one way is to test everything with unit tests, but only test the main flows / use cases of the program with integration/system tests – for the project this will be a team decision documented in your testing plan.

Principles of writing test code

- Everything applies to test code as it does to normal code, DRY, KISS.
- You can use design patterns in test code.
- However, in writing test code there are some other things to consider:
  - You want to make your test code **as simple as possible**, otherwise you end up having something that is more complex (and as a result bug-prone) than the software you are testing in the first place!
  - Conditionals, loops, any control flow should be kept to a minimum to reduce test complexity.
- Factory pattern is often very useful in test design since you can write a factory to produce dummy objects for testing.

Software Testing: Summary

- Always follow Test Driven Development.
- Software Testing is hard!
- Not possible to completely test a nontrivial software system, given the limited available resources.
- We assume that a selected *representative* test cases capture system behavior of test cases not considered.

End