Software Testing – Part A

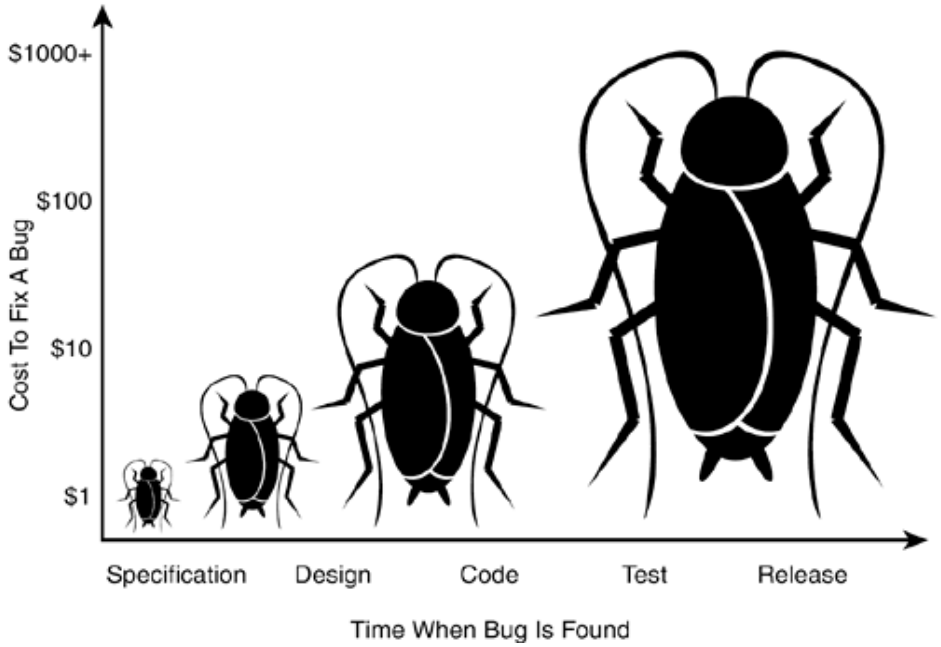
Dr Fani Deligianni,

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Fani.Deligianni@glasgow.ac.uk

Bugs

- Bugs are inevitable in any complex software system.
- Industry estimates: 10-50 bugs per 1000 lines of code.
- A bug can be visible or can hide in your code until much later.



Famous Quotes

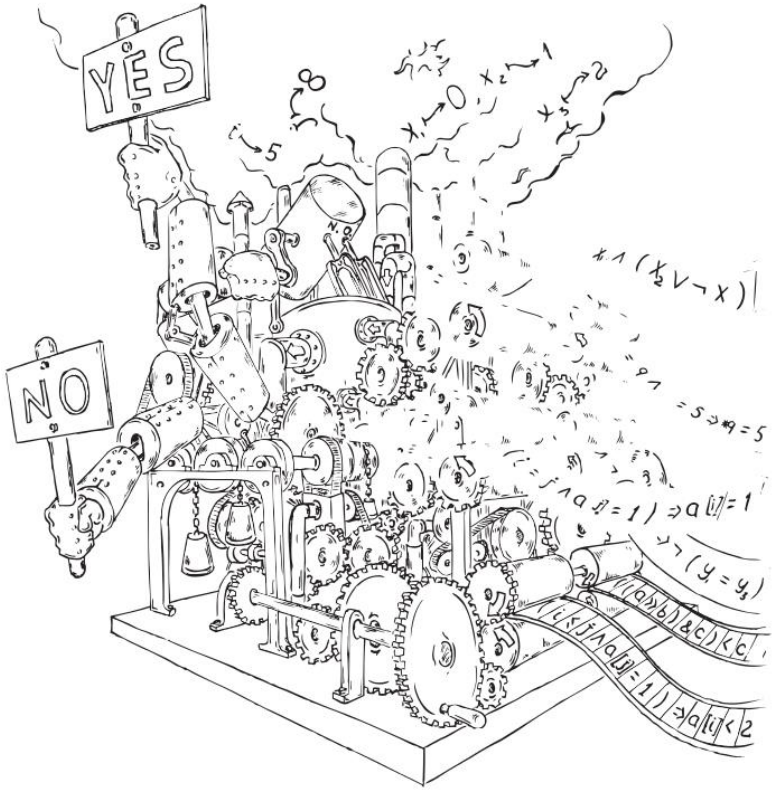
- “Testing is the process of comparing the invisible to the ambiguous, so as to avoid the unthinkable happening to the anonymous.”, James Bach

- "Testing is organised skepticism.“, James Bach

- **"Program testing can be used to show the presence of bugs, but never to show their absence!“, Edgar Dijkstra**

Software Reliability

- Probability that a software system will not cause failure under specified conditions.
- Measured by uptime, MTTF (mean time till failure), crash rate, etc.



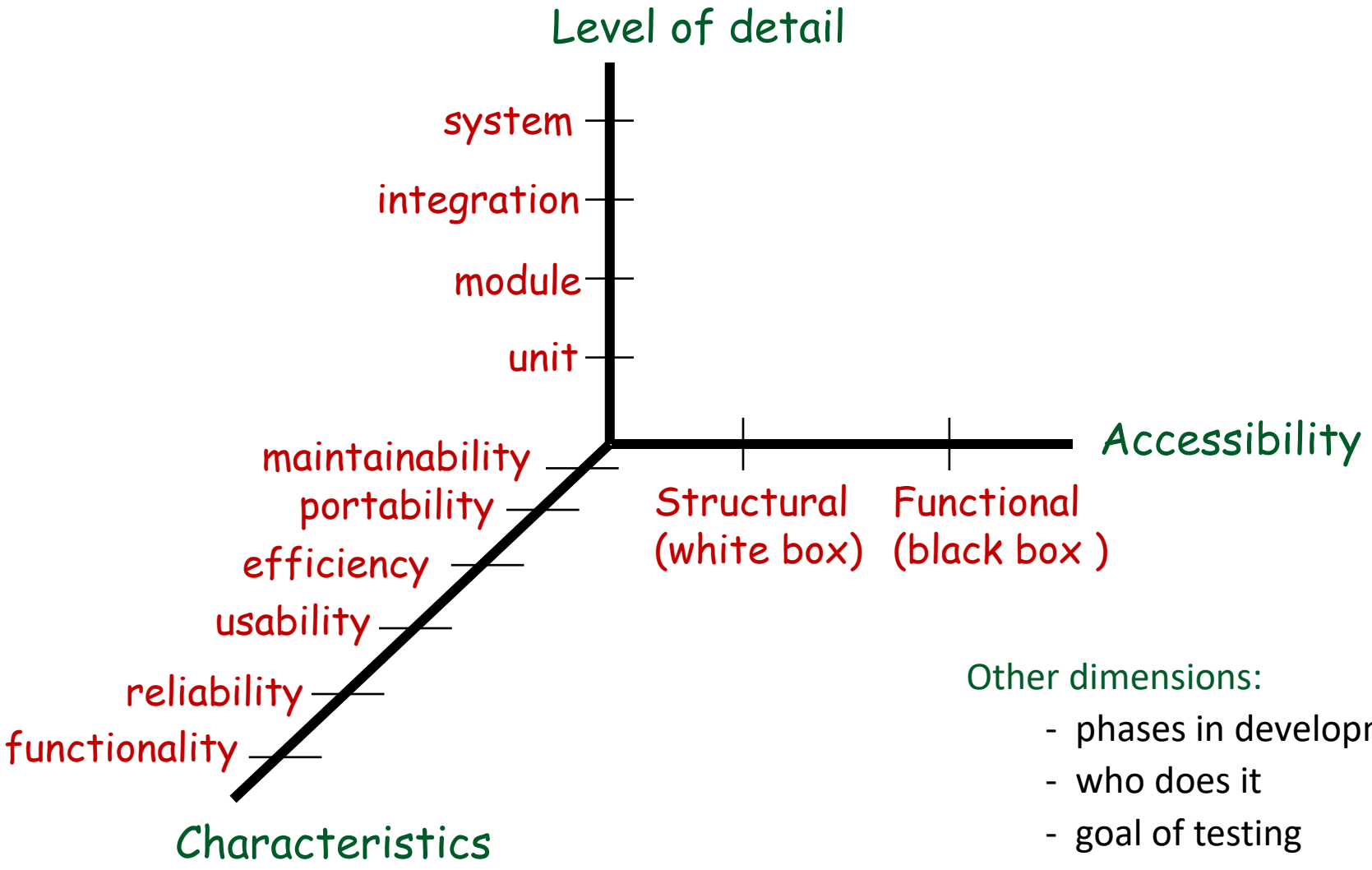
Testing Definitions

- “Testing is the process of establishing confidence that a program or system does what it is supposed to.”
- “Testing is the process of executing a program or system with the intent of finding errors.”
- “Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.”

Testing

- A systematic attempt to reveal errors.
- Failed test: an error was demonstrated.
- Passed test: no error was found (**for this particular situation**).

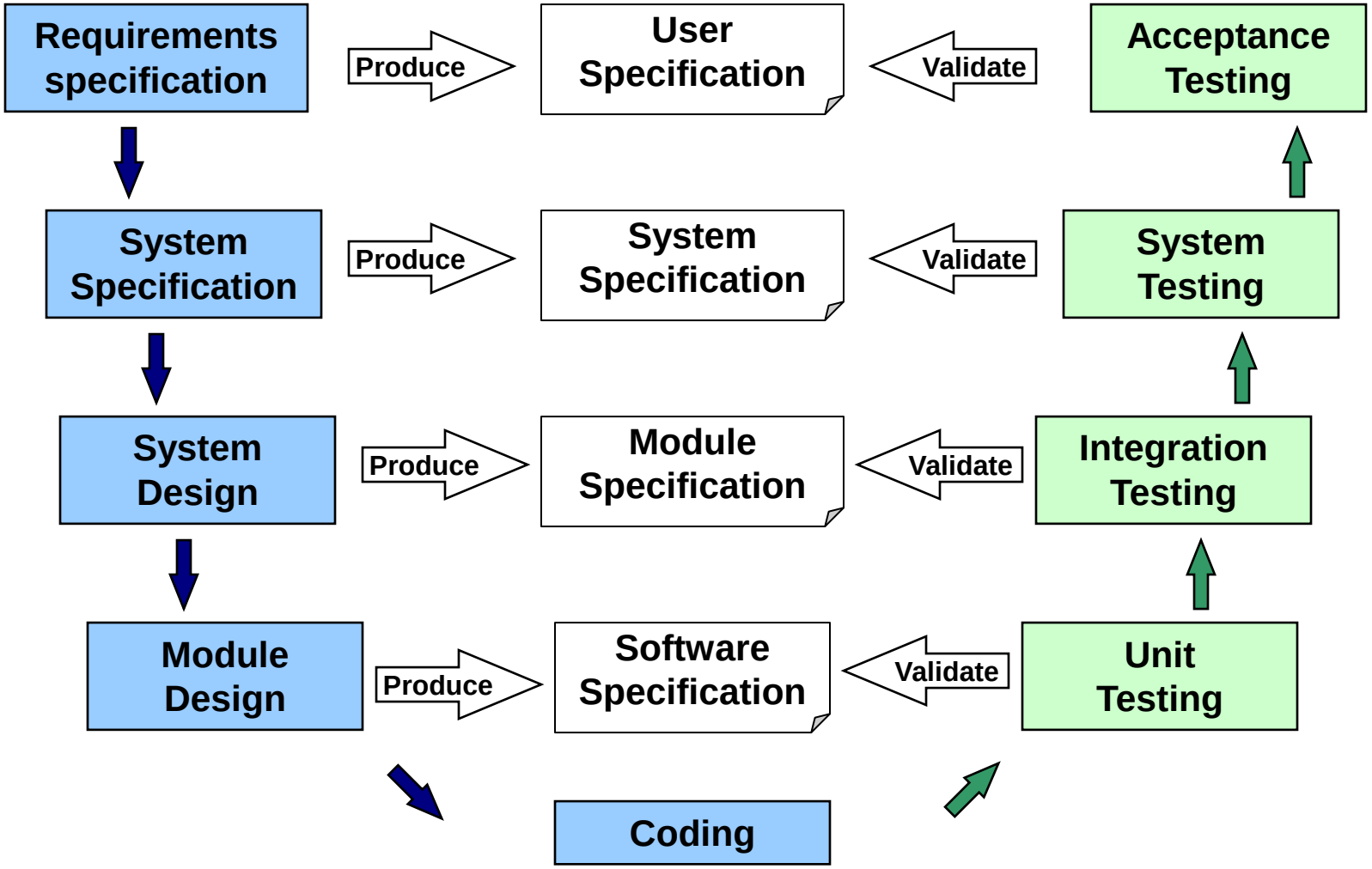
Dimensions of Testing



Other dimensions:

- phases in development
- who does it
- goal of testing
- . . . . .

Testing Organisation (V-model)



Software Testing - Automation

- **Testability:**
  - How easy is to establish test criteria
  - How test perform to meet those criteria
- **Software Observability**
  - How easy is to observer input/output
- **Software Controllability**
  - How easy is to provide with the needed inputs (values, operations, behaviours)

Summary

- Program testing cannot can only prove that a bug exist
- Automate testing is important
- Software testing is hard and it has several dimensions

Software Testing – Part B

Dr Fani Deligianni,

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

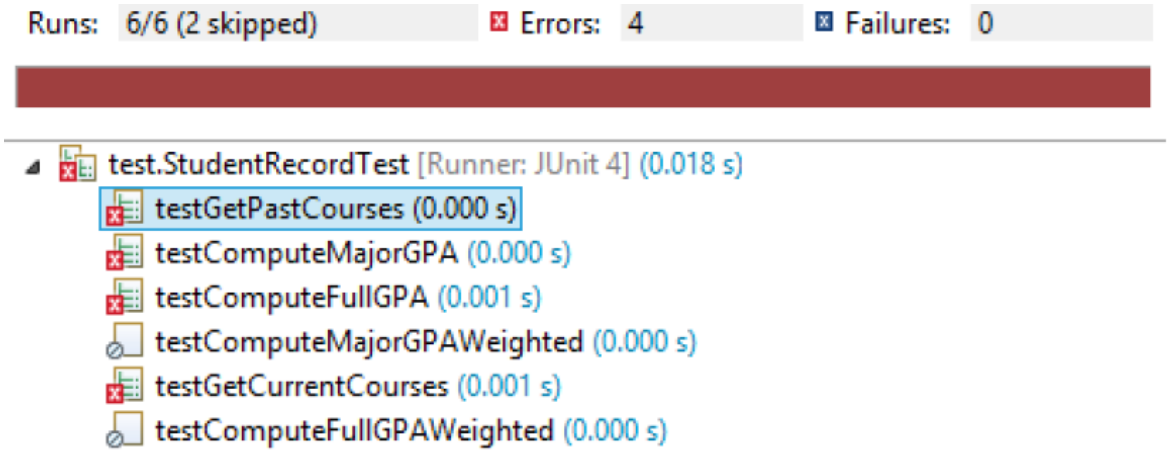
Fani.Deligianni@glasgow.ac.uk

Unit Testing

- Testing individual modules:
  - Methods or Functions.
  - Classes.
- It is based on the information about the structure of a code fragment.
- The objective is to test that the unit performs the function for which it is designed.
- Unit tests can be designed before coding begins, or just after the source code is generated.

Test-driven development a.k.a. 'Red-green refactoring'

- Write a new test
- Run all tests (newly added test should fail)
- Write the implementation code to make the test pass (KISS – Keep it Simple Stupid)
- Run all tests (tests should now pass)
- Refactor
- Repeat



https://technologyconversations.com/2013/12/24/test-driven-development-tdd-best-practices-using-java-examples-2/

Writing a unit testing

A good unit test is

- **Atomic:** it tests exactly one piece of code, does not depend on any other tests, and it can be run repeatedly and/or concurrently
- **Trustworthy:** it should run every time on every machine
- **Maintainable:** test code is code too avoid repetition, magic numbers, etc
- **Readable:** code should make sense; names should describe what they are testing

https://philippetruche.wordpress.com/2011/03/22/unit-testing-101-fundamentals-of-unit-testing/

What does a unit test actually do?

1. Set up a predictable context
2. Run a method and check that the result meets a specification
    - Return value correct?
    - Expected exception thrown?
    - Correct side effects detected?

https://philippetruche.wordpress.com/2011/03/22/unit-testing-101-fundamentals-of-unit-testing/

Overview of JUnit

- “A simple framework to write repeatable tests.” http://junit.org/junit4/index.html
- Widely used over 30% of Java projects on github.org make use of JUnit tests (http://blog.takipi.com/githubs 10000 most popular java projects here are the top-libraries they use/)

History:

- Started in 1998 as SUnit (for Smalltalk)
- Ported to Java, and eventually other OO languages (Ruby, etc.) collectively known as “ xUnit

https://philippetruche.wordpress.com/2011/03/22/unit-testing-101-fundamentals-of-unit-testing/

JUnit

- The basic idea:
  - For a given class Foo, create another class FooTest to test it, containing various "test case" methods to run.
  - Each method looks for particular results and passes / fails.
- Test Classes
  - A collection of test methods
  - Methods to set up the program state – test cases

Junit: Terminology

Test Fixture

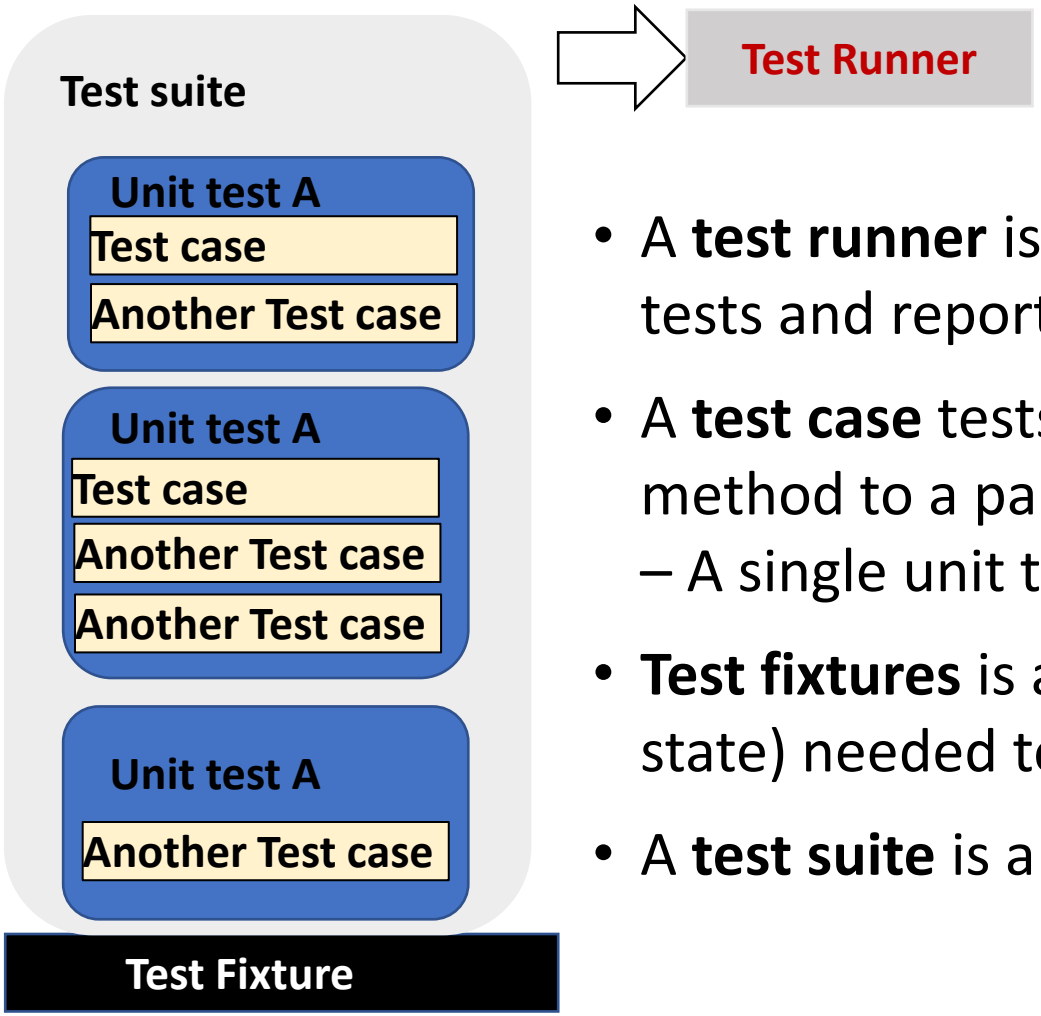
- A test fixture is a fixed state of **a set of objects used as a baseline for running tests**.
- Ensures that there is a well known and fixed environment in which tests are run so that results are **repeatable**.

Junit: Terminology

Test Fixture

- Example: When testing code that updates an employee record, you need an employee record to test it on.
- Fixtures can run **before or after every test**, or **one time fixtures** that run before and after only once for all test methods in a class.
  - Class-level fixtures: @BeforeClass and @AfterClass
  - Method-level fixtures:@Before and @After.

Junit: Architecture



- A **test runner** is a program that runs the tests and reports the results
- A **test case** tests the response of a single method to a particular set of inputs (@test) – A single unit test case
- **Test fixtures** is a set of preconditions (ie. state) needed to run a test
- A **test suite** is a collection of test cases

Summary - Benefits of unit testing

- Find problems early in the development cycle
- Support refactoring and other code changes
- Potentially simplify integration testing
- Provide “living documentation” of the system
- Unit tests can provide a design document for the class

Summary - Limitations of unit testing

- Cannot catch every error in the program - Cannot cover every possible execution path
- Cannot catch integration or system level errors
- Complexity of setting up realistic tests creating initial conditions
- Tests could be buggy themselves
- Cannot easily test problems that use randomness or multiple threads

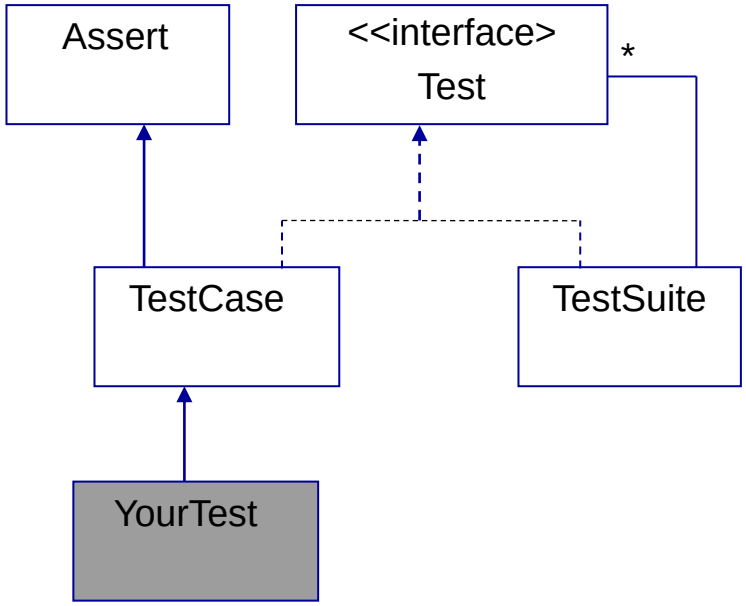
Software Testing – Part C

Dr Fani Deligianni,

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Fani.Deligianni@glasgow.ac.uk

Junit Usage



- Subclass **TestCase**
- Create Test methods
  - **public void test*XXX*()**
  - assertions per method
- Implement **main** to run from command-line, but not necessary
- Optionally add setUp/tearDown methods if any shared data is needed in tests

A sample Junit test

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class MyTests {
        @Test
        public void multiplicationOfZeroIntegersShouldReturnZero() {
                MyClass tester = new MyClass(); // MyClass is tested

                // assert statements
                assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
                assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
                assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));

        }
}
```

https://www.vogella.com/tutorials/JUnit/article.html

Naming conventions

- Widely used:
  - Add suffix “Test” to the name of the test classes
  - Create them in a new package “test”
- Name should explain what the test does
  - (If you use good names, you don’t need to read the code)
  - One possible convention: use “should” in the method name

    E.g., “addShouldThrowExceptionOnNull

Defining tests in Junit: use annotations

- **@Test:** identifies a method as a test method
  @Test (expected = Exception.class) - fails unless the given exception is thrown
  @Test (timeout = 100) fails if it takes more than 100msec
- **@BeforeEach**: executed before each test is run(sets up fixtures)
- **@AfterEach**: executed after each test is completed (deletes fixtures)
- **@BeforeAll** : executed once before all tests in the file are run (e.g., connect to database)
- **@AfterAll**: executed once after all tests in the file are run (e.g.,
- **@Ignore**: indicates that the given test should be ignored
  @Ignore("Reason") also gives an explanation - good practice to include

Writing the body of a test – use Assert

- Used to check the actual result against an expected result
- Optional (but strongly recommended0): specify the string to display if they do not match

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class MyTests {
        @Test
        public void multiplicationOfZeroIntegersShouldReturnZero() {
                MyClass tester = new MyClass(); // MyClass is tested

                // assert statements
                assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
                assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
                assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));

        }
}
```

https://www.vogella.com/tutorials/JUnit/article.html

Junit : assert*() methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. assertEquals("message", expected, actual)
- All in org.junit.Assert class

How many assertions per test?

- It depends …
  - Each test case should only potentially fail for **one reason**
  - But that reason could potentially involve multiple assertions

```
@Test
public void testResultShouldBeInRange() {
      int result = obj.getResult();
      assertTrue("Result too small", result > 0);
      assertTrue("Result too big", value < 100);
}
```

https://www.vogella.com/tutorials/JUnit/article.html

Advanced Junit – using matchers

- One more method in Assert: assertThat (message, Matcher<T>)
- Sample uses:

```
assertThat(actual, is(equalTo(expected)));
assertThat(actual, is(not(equalTo(expected))));
assertThat(actual, containsString(expected));
assertThat(123, is("abc"));     //does not compile
assertThat("test", anyOf(is("test2"), containsString("ca")));
```

More information and examples at
https://objectpartners.com/2013/09/18/the-benefits-of-using-assertthat-over-other-assert-methods-in-unit-tests/

Summary

- How to write a sample Junit test
- Use of annotations
- Use of appropriate naming conventions
- Use of assert

Software Testing – Part D

Dr Fani Deligianni,

https://www.gla.ac.uk/schools/computing/staff/fanideligianni/

Fani.Deligianni@glasgow.ac.uk

Example:

```
public class TestSameOrEquals {

 BigDecimal b1 = new BigDecimal("1.0");
 BigDecimal b2 = new BigDecimal("1.0");
 BigDecimal b3 = b1;

  int i1 = 5;
  int i2 = 5;

  @Test
  public void BigDecimaltest() throws Exception {
   // if(b1 == b2)
    assertSame(b1, b2);    // THIS TEST WILL FAIL

   // b1.equals(b2)
    assertEquals(b1, b2);  // should pass

   // (b1 == b3)
    assertSame(b1, b3);    // will pass

   //(b1.equals(b3))
    assertEquals(b1, b3);  // will pass
  }
```

```
  @Test
   public void intTest() throws Exception {
    // if(i1 == i2)
     assertSame(i1, i2);    // will pass

    // if(i1 == i2)
     assertEquals(i1, i2);  // will pass
   }
  }
```

Exercise 1:

Given a Calendar class with the following methods:

- Public GregorianCalendar(year, month, dayOfMonth)
- public void add(int field, int amount) //field= {year, month, dayOfMonth}
- public void get(int field)

Come up with unit tests to check the following:

1. That no Calendar object can ever get into an invalid state.
2. That the add method works properly. It should be efficient enough to add 1,000,000 days in a call.

What is wrong with this solution?

```
public class DateTest1 {
   @Test
   public void test1() {
      Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      cal.add(Calendar.DATE, 4);

      assertEquals(cal.get(Calendar.YEAR), 2050);
      assertEquals(cal.get(Calendar.MONTH), Calendar.FEBRUARY);
      assertEquals(cal.get(Calendar.DAY_OF_MONTH), 19);
   }

   @Test
   public void test2() {
      Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      cal.add(Calendar.DATE, 14);

      assertEquals(cal.get(Calendar.YEAR), 2050);
      assertEquals(cal.get(Calendar.MONTH), Calendar.MARCH);
      assertEquals(cal.get(Calendar.DAY_OF_MONTH), 1);
   }
}
```

Well-structured assertions!

```
public class DateTest2 {
   @Test
   public void test1() {
    Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

    cal.add(Calendar.DATE, 4);

    assertEquals(2050, cal.get(Calendar.YEAR));
    assertEquals(Calendar.FEBRUARY, cal.get(Calendar.MONTH));
    assertEquals(19, cal.get(Calendar.DAY_OF_MONTH));
   }

   @Test
   public void test2() {
    Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

    cal.add(Calendar.DATE, 14);

    assertEquals("year after +14 days", 2050, cal.get(Calendar.YEAR));
    assertEquals("month after +14 days", Calendar.MARCH, cal.get(Calendar.MONTH));
    assertEquals("day after +14 days", 1,  cal.get(Calendar.DAY_OF_MONTH));
  }
}
```

Well-structured assertions!

```
public class DateTest2 {
   @Test
   public void test1() {
    Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

    cal.add(Calendar.DATE, 4);

    assertEquals(2050, cal.get(Calendar.YEAR));            //expected value
    assertEquals(Calendar.FEBRUARY, cal.get(Calendar.MONTH));//should be
    assertEquals(19, cal.get(Calendar.DAY_OF_MONTH));      //at the LEFT
   }

   @Test
   public void test2() {
    Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

    cal.add(Calendar.DATE, 14);

    assertEquals("year after +14 days", 2050, cal.get(Calendar.YEAR));
    assertEquals("month after +14 days", Calendar.MARCH, cal.get(Calendar.MONTH));
    assertEquals("day after +14 days", 1,  cal.get(Calendar.DAY_OF_MONTH));

    // test cases should usually have messages explaining
    // what is being checked, for better failure output
   }
}
```

Use expected answer objects

```
public class DateTest3 {
   @Test
   public void test1() {
      Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      cal.add(Calendar.DATE, 4);

      Calendar expected = new GregorianCalendar(2050, Calendar.FEBRUARY, 19);
      assertEquals(expected, cal);
      //use an expected answer to object to minimise tests
      //Calendar must have a toString and equal method
   }

   @Test
   public void test2() {
      Calendar cal = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      cal.add(Calendar.DATE, 14);

      Calendar expected = new GregorianCalendar(2050, Calendar.MARCH, 1);
      assertEquals("date after +14 days",expected, cal);
   }
}
```

Use proper naming for test cases

```
public class DateTest4 {
   @Test
   // give test case methods useful descriptive names
   public void test_addDays_withinSameMonth_1() {
      Calendar actual = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      actual.add(Calendar.DATE, 4);

      Calendar expected = new GregorianCalendar(2050, Calendar.FEBRUARY, 19);
      assertEquals(expected, actual);

   }

   @Test
   public void test_addDays_wrapToNextMonth_2() {
      Calendar actual = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      actual.add(Calendar.DATE, 14);

      Calendar expected = new GregorianCalendar(2050, Calendar.MARCH, 1);
      assertEquals("date after +14 days",expected, actual);
      // give descriptive names to expected/actual values
   }
}
```

Tests with a timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...
@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

Pervasive timeouts

```
public class DateTest6 {
   // almost every test should have a timeout so it can't
   // lead to an infinite loop; good to set a default, too
   private static final int DEFAULT_TIMEOUT = 2000;

   @Test(timeout = DEFAULT_TIMEOUT)
   public void test_addDays_withinSameMonth_1() {
      Calendar actual = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      actual.add(Calendar.DATE, 4);

      Calendar expected = new GregorianCalendar(2050, Calendar.FEBRUARY, 19);
      assertEquals(expected, actual);
   }

   @Test(timeout = DEFAULT_TIMEOUT)
   public void test_addDays_wrapToNextMonth_2() {
      Calendar actual = new GregorianCalendar(2050, Calendar.FEBRUARY, 15);

      actual.add(Calendar.DATE, 14);

      Calendar expected = new GregorianCalendar(2050, Calendar.MARCH, 1);
      assertEquals("date after +14 days",expected, actual);
   }
}
```

What's wrong with this?

```
public class DateTest7 {
   private static final int[] DAYS_PER_MONTH = {
     31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
   }; // Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

   // test every day of the year
   @Test(timeout = 10000)
   public void tortureTest() {
      Calendar actual = new GregorianCalendar(2050, 1, 1);

      int month = 1; int day = 1;
      for (int i = 1; i < 365; i++) {
                  actual.add(Calendar.DATE, 1);

         if (day < DAYS_PER_MONTH[month]){
          day = day +1;
         }
         else{
          month = month+ 1; day=1;
         }
         Calendar expected = new GregorianCalendar(2050, month, day);
         assertEquals("date after "+i+ "day(s)",expected, actual);
      }
   }
}
```

Tips for testing

1. You cannot test every possible input, parameter value, etc.
   - So you must think of a limited set of tests likely to expose bugs.

2. Think about boundary cases
   - positive; zero; negative numbers
   - right at the edge of an array or collection's size

Tips for testing

3. Think about empty cases and error cases
   - 0, -1, null; an empty list or array

4. test behavior in combination
   - maybe `add` usually works, but fails after you call `remove`
   - make multiple calls; maybe `size` fails the second time only

Trustworthy tests

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.

- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
  - avoid `try/catch`
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it.

Test case "smells"

- Tests should be self-contained and not care about each other.

Test case "smells"

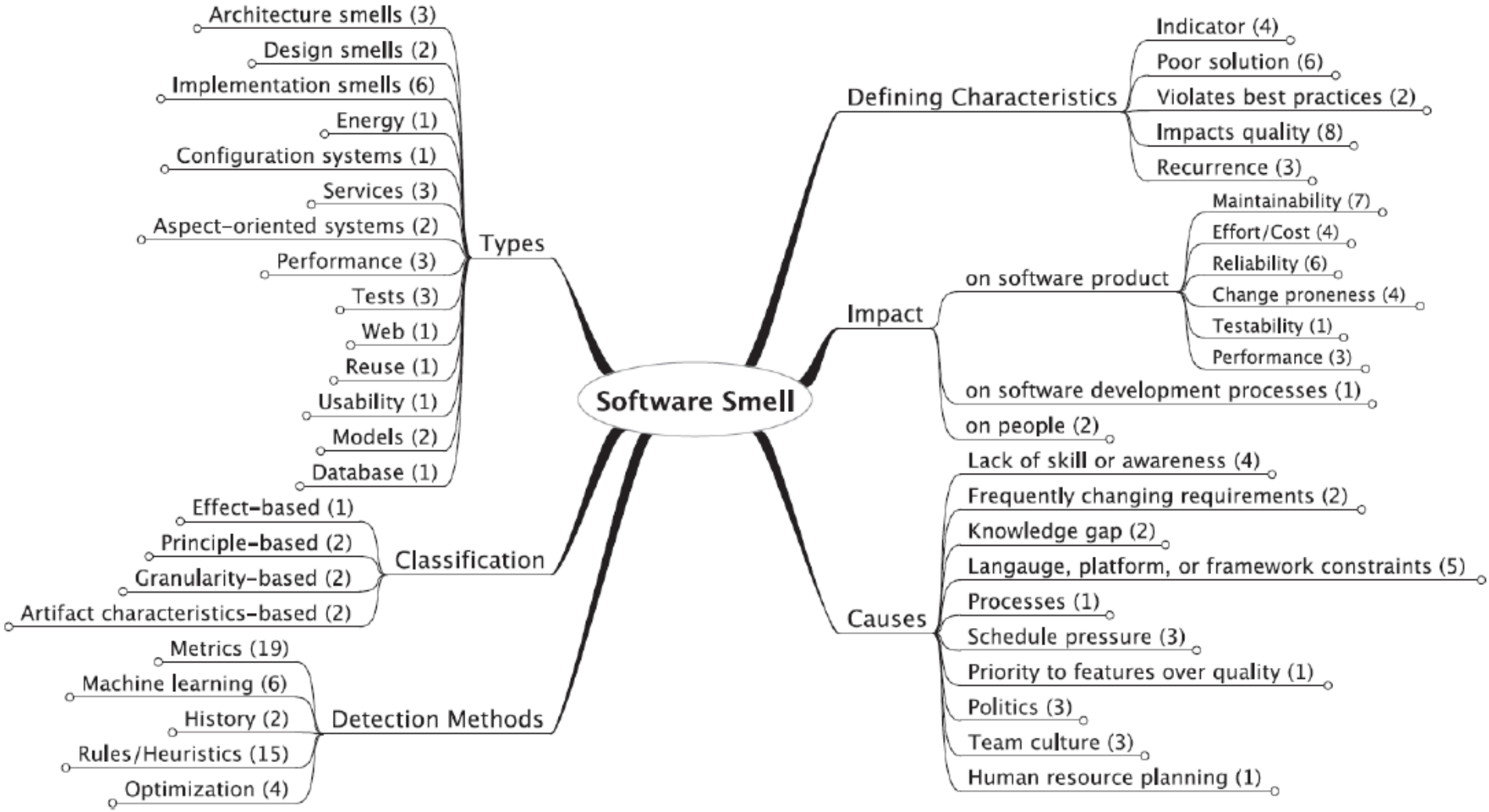
- "**Smells**" (bad things to avoid) in tests:
  - *Constrained test order*: Test A must run before Test B. (usually a misguided attempt to test order/flow)
  - *Tests call each other*: Test A calls Test B's method (calling a shared helper class is OK)
  - *Mutable shared state*: Tests A/B both use a shared object. (If A breaks it, what happens to B?)

Summary

- Tests **need *failure atomicity*** (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
    - Each test should have roughly just 1 assertion at its end.

Summary

- Always use a timeout parameter to every test.
- Test for expected errors / exceptions.
- Choose a descriptive assert method, not always assertTrue.
- Choose representative test cases from equivalent input classes.
- Avoid complex logic in test methods if possible.
- Use @Before to reduce redundancy between tests.



Sharma et al. 'A survey on Software smells', Journal of Systems and Software, 2018.