Agile Software Development Practices

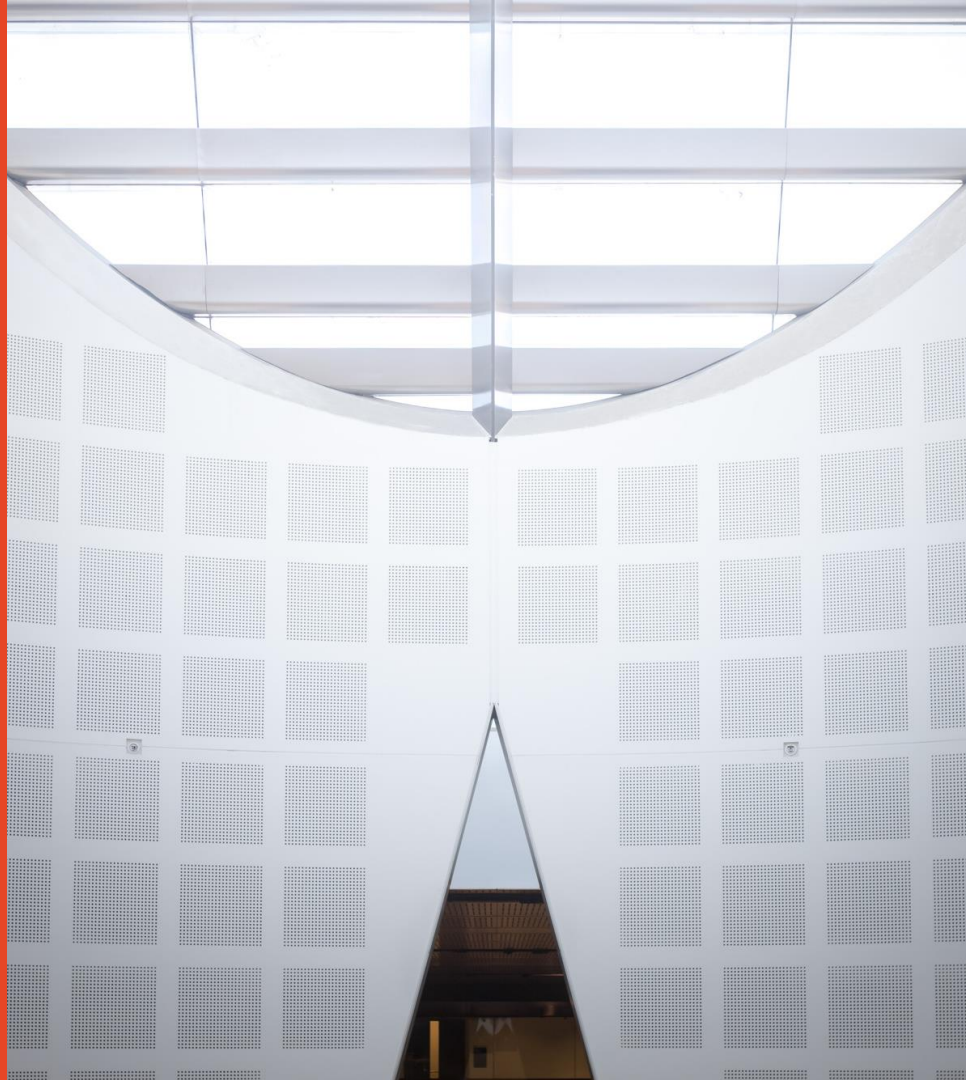
SOF2412 / COMP9412

Software Testing

Dr. Basem Suleiman

School of Information Technologies

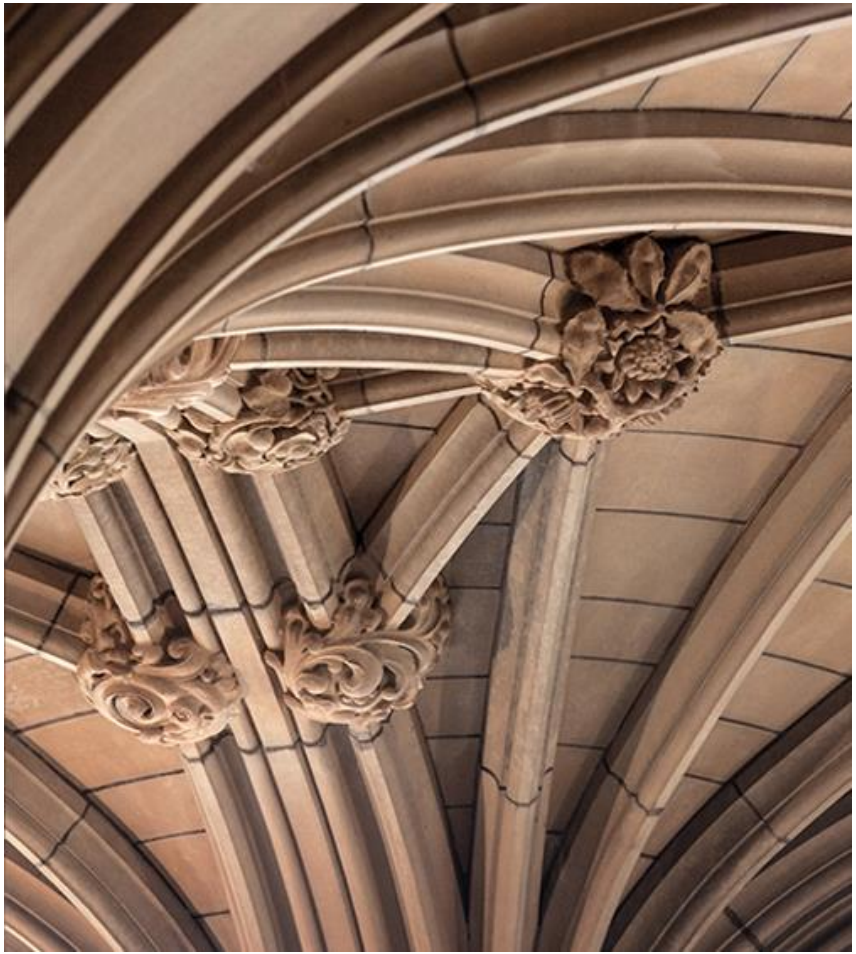
THE UNIVERSITY OF SYDNEY



Agenda

- Software Quality Assurance
- Software Testing
  - Why, what and how?
  - Testing levels and techniques
  - Test cases design
  - Code (Test) Coverage
- Unit Testing Framework
  - JUnit
- Code Coverage Tools

Software Quality Assurance



THE UNIVERSITY OF SYDNEY

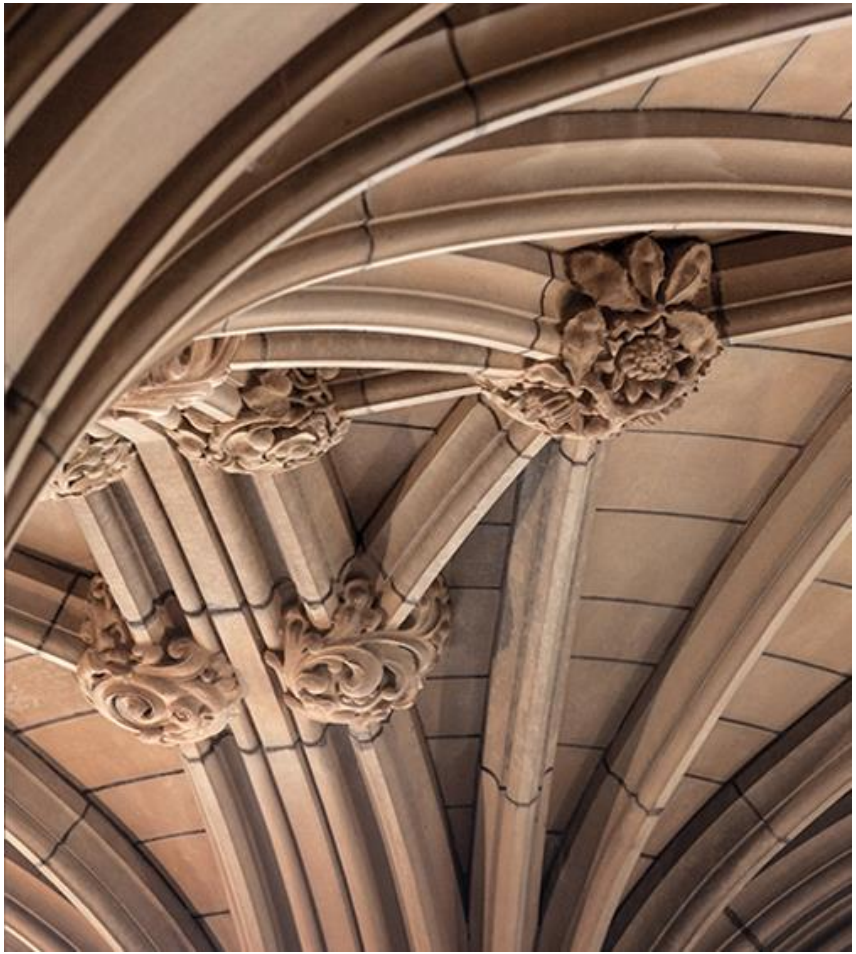
Software Quality Assurance

- Software quality
  - Satisfying end use's needs; correct behaviour, easy to use, does not crash, etc.
  - Easy to the developers to debug and enhance

- Software Quality Assurance
  - Ensuring software under development have high quality and creating processes and standards in organization that lead to high quality software
  - Software quality is often determined through Testing

Juran and Gryna 1998

Why Software Testing?



THE UNIVERSITY OF SYDNEY

Nissan Recall - Airbag Defect*

- What happened?
  - Front passenger airbag may not deploy in an accident
  - ~ 3.53 million vehicles recall of various models 2013-2017
- Why did it happen?
  - Software that activates airbags deployment improperly classify occupied passenger seat as empty in case of accident
  - Software sensitivity calibration due to combination of factors (high engine vibration and changing seat status)

http://www.reuters.com/article/us-autos-nissan-recall/nissan-to-recall-3-53-million-vehicles-air-bags-may-not-deploy-idUSKCN0XQ2A8
https://www.nytimes.com/2014/03/27/automobiles/nissan-recalls-990000-cars-and-trucks-for-air-bag-malfunction.html

Therac-25 Overdose*

- What happened?
  - Therac-25 radiation therapy machine
  - Patients exposed to overdose of radiation (100 times more than intended) - 3 lives!!

- Why did it happen?
  - Nonstandard sequence of keystrokes was entered within 8 seconds
  - Operator override a warning message with error code ("MALFUNCTION" followed by a number from 1 to 64) which is not explained in the user manual
  - Absence of independent software code review
  - 'Big Bang Testing': software and hardware integration has never been tested until assembled in the hospital

*https://en.wikipedia.org/wiki/Therac-25#Problem_description

Software Failure - Ariane 5 Disaster[5]

What happened?

- European large rocket - 10 years development, ~$7 billion
- Unmanaged software exception resulted from a data conversion from 64-bit floating point to a 16-bit signed integer
- Backup processor failed straight after using the same software
- Exploded 37 seconds after lift-off

Why did it happen?

- Design error, incorrect analysis of changing requirements, *inadequate validation and verification, testing and reviews*, ineffective development processes and management

[5] http://iansommerville.com/software-engineering-book/files/2014/07/Bashar-Ariane5.pdf

Examples of Software Failures

Project	Duration	Cost	Failure/Status
e-borders (UK Advanced passenger Information System Programme)	2007 - 2014	Over £ 412m (expected), £742m (actual)	Permanent failure - cancelled after a series of delays
Pust Siebel - Swedish Police case management (Swedish Police)	2011 - 2014	$53m (actual)	Permanent failure – scraped due to poor functioning, inefficient in work environments
US Federal Government Health Care Exchange Web application	2013 – ongoing	$93.7m (expected), $1.5bn (actual)	Ongoing problems - too slow, poor performance, people get stuck in the application process (frustrated users)
Australian Taxation Office's Standard Business Reporting	2010 - ongoing	~$1bn (to-date), ongoing	Significant spending on contracting fees (IBM & Fjitsu), significant scope creep and confused objectives

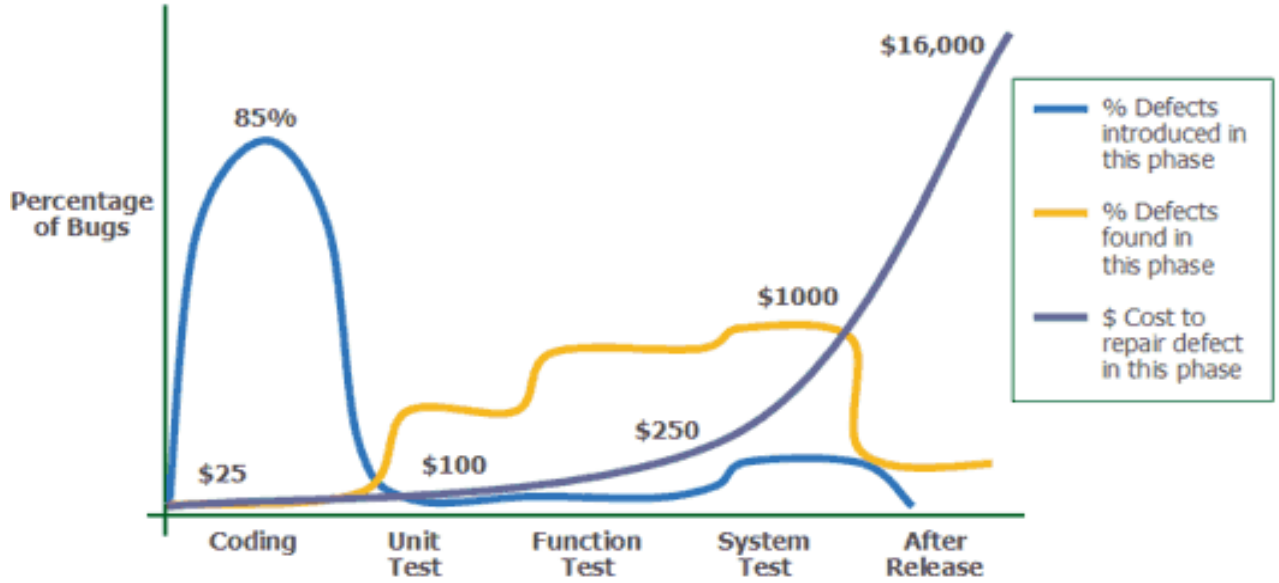
https://en.wikipedia.org/wiki/List_of_failed_and_overbudget_custom_software_projects

Software Testing – Costs

- Software development and maintenance costs
- Total costs of inadequate software testing on the US economy is $59.5bn
  - NIST study 2002*
  - One-third of the cost could be eliminated by *improved software testing*
- Need to develop functional, robust and reliable software systems
  - Human/social factor - society dependency on software in every aspect of their lives
    - Critical software systems - medical devices, flight control, traffic control
  - Meet user needs and solve their problems
  - Small software errors could lead to disasters

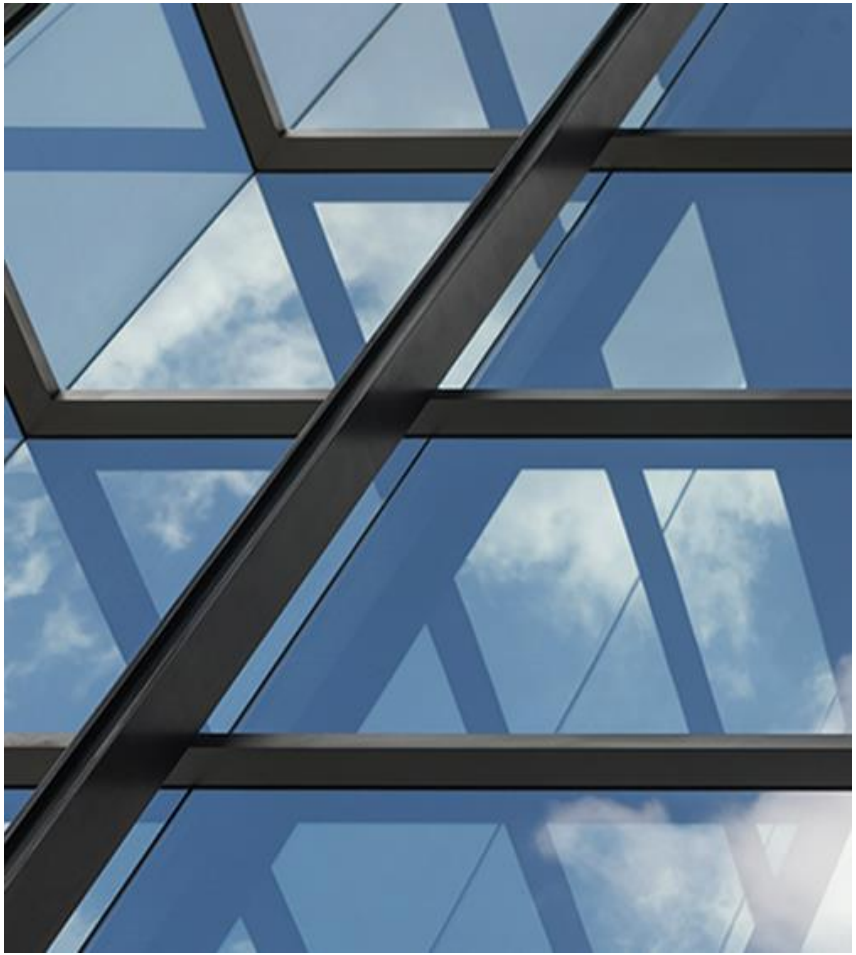
* https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf

Software Testing - Costs



Capers Jones, Applied software measurement (2nd ed.): assuring productivity and quality, (1997), McGraw-Hill

What is Software Testing?



THE UNIVERSITY OF SYDNEY

Software Testing

- Software process to
  - Demonstrate that software meets its requirements
  - Find incorrect or undesired behavior caused by defects/bugs
    - E.g., System crashes, incorrect computations, unnecessary interactions and data corruptions

- Different system properties
  - Functional: performs all expected functions properly
  - Non-functional: secure, performance, usability

Testing Objectives

"Program testing can be used to show the presence of bugs, but never to show their absence" – **Edsger W. Dijkstra**

Testing Objectives

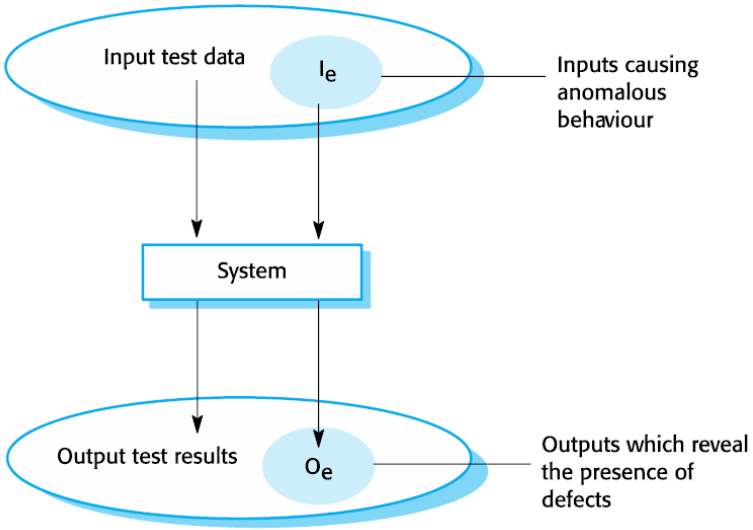
- Objectives should be stated precisely and quantitatively to measure and control the test process

- Testing completeness is never been feasible
  - So many test cases possible - exhaustive testing is so expensive!
  - Risk-driven or risk management strategy to increase our confidence

- How much testing is enough?
  - Select test cases sufficient for a specific purpose (test adequacy criteria)
  - Coverage criteria and graph theories used to analyse test effectiveness

Tests Modeling

- Testing modelled as input test data and output test results
- Tests that cause defects/problems (*defective testing*)
- Tests that lead to expected correct behavior (*validation testing*)



Who Does Testing?

- Developers test their own code
- Developers in a team test one another's code
- Many methodologies also have specialist role of tester
  - Can help by reducing ego
  - Testers often have different personality type from coders
- Real users, doing real work

Testing takes creativity

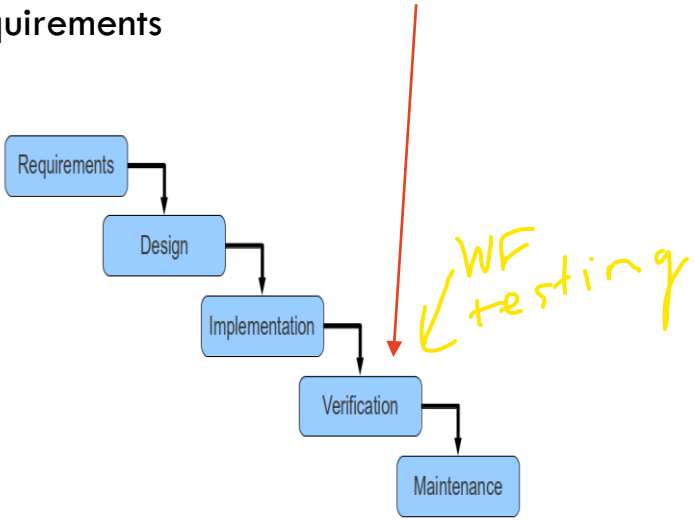
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques

- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should be in a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else

When is Testing happening?

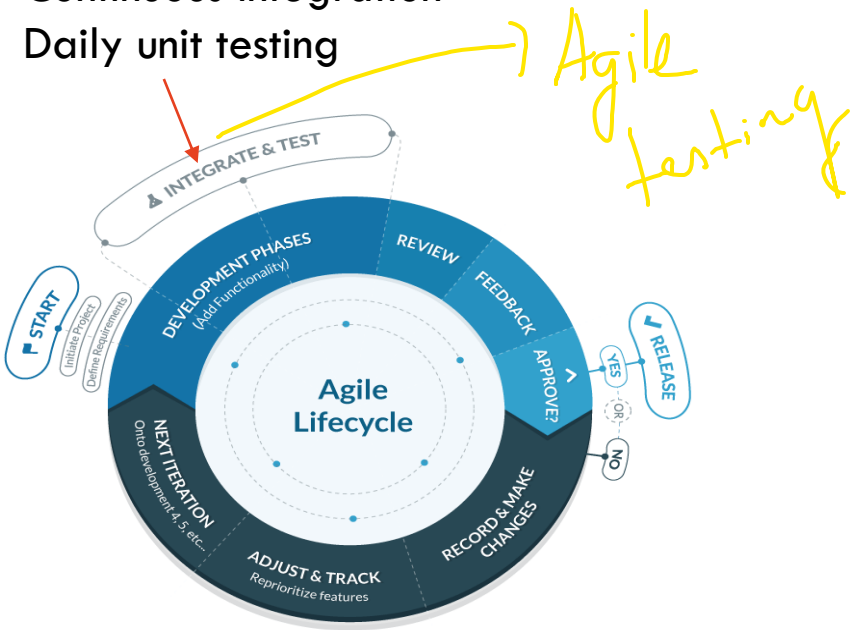
Waterfall Software Development

- Test whether system works according to requirements



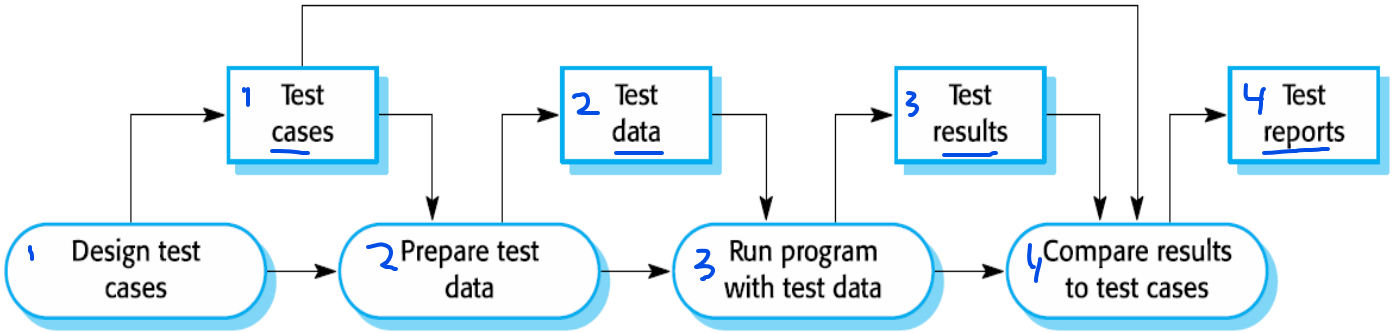
Agile Software Development

- Testing is at the heart of agile practices
- Continuous integration
- Daily unit testing



https://www.spritecloud.com/wp-content/uploads/2011/06/waterfall.png
https://blog.capterra.com/wp-content/uploads/2016/01/agile-methodology-720x617.png

Software Testing Process



Ian Sommerville. 2016. *Software Engineering* (10th ed.). Addison-Wesley, USA.

Software Testing Process

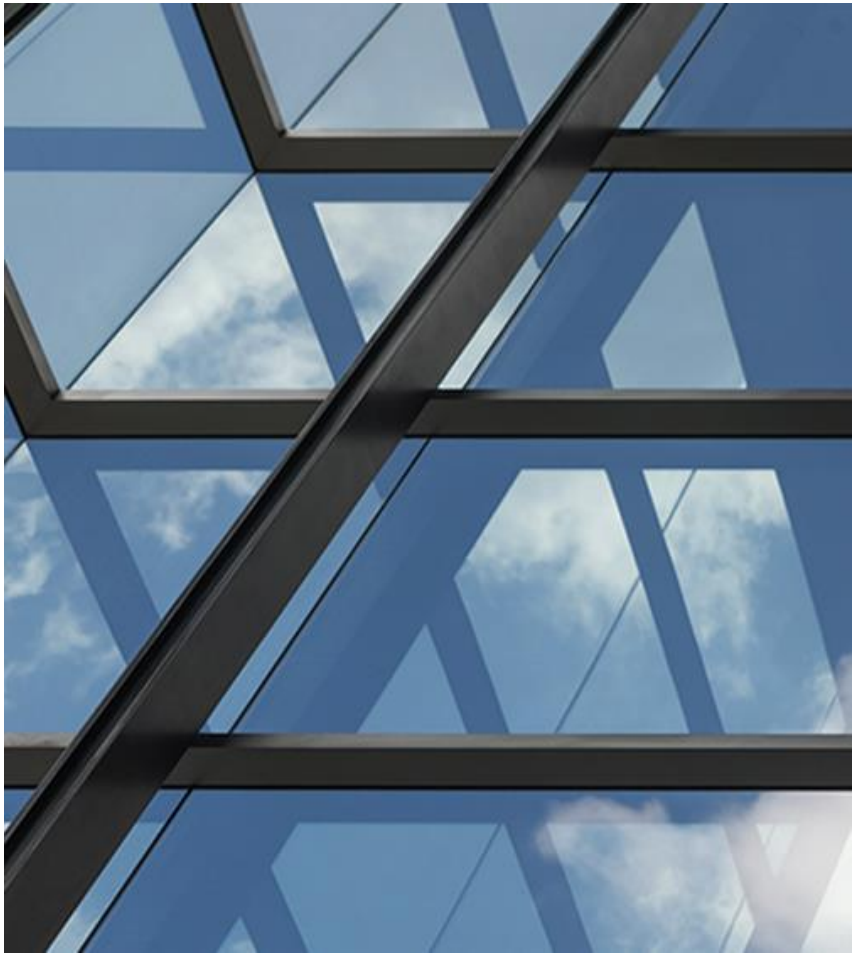
- Design, execute and manage test plans and activities
  - Select and prepare suitable test cases (selection criteria)
  - Selection of suitable test techniques
  - Test plans execution and analysis (study and observe test output)
  - Root cause analysis and problem-solving
  - Trade-off analysis (schedule, resources, test coverage or adequacy)

- Test effectiveness and efficiency
  - Available resources, schedule, knowledge and skills of involved people
  - Software design and development practices ("Software testability")
    - Defensive programming: writing programs in such a way it facilitates validation and debugging using assertions

Types of Defects in Software

- Syntax error
  - Picked up by IDE or at latest in build process
  - Not by testing
- Runtime error
  - Crash during execution
- Logic error
  - Does not crash, but output is not what the spec asks it to be
- Timing Error
  - Does not deliver computational result on time

Software Testing Levels



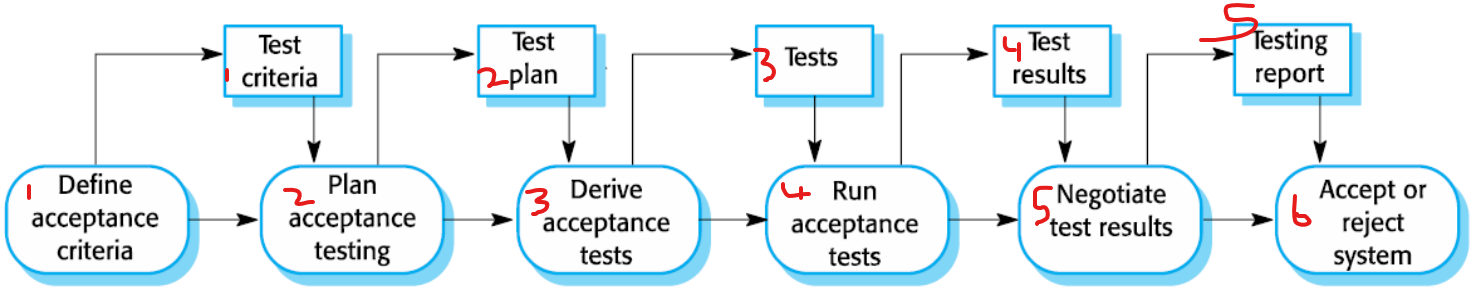
Testing Levels

Testing level	Description
Unit / Functional Testing	The process of verifying functionality of software components (functional units, subprograms) independently from the whole system
Integration Testing	The process of verifying interactions/communications among software components. Incremental integration testing vs. "Big Bang" testing
System Testing	The process of verifying the functionality and behaviour of the entire software system including security, performance, reliability, and external interfaces to other applications
Acceptance Testing	The process of verifying desired acceptance criteria are met in the system (functional and non-functional) from the user point of view

Integration Testing

- The process of verifying interactions/communications among software components behave according to its specifications
- Incremental integration testing vs. "Big Bang" testing
- Independently developed (and tested) units may not behave correctly when they interact with each other
- Activate corresponding components and run high-level tests

Acceptance Testing Process



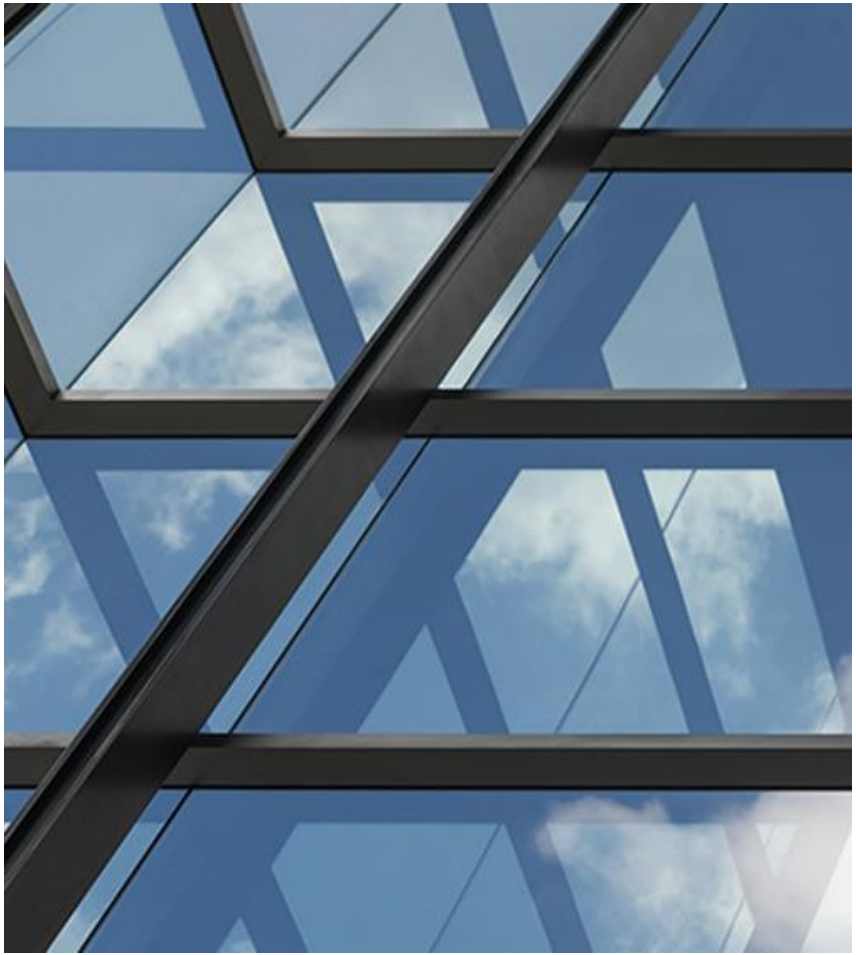
choose based on reports

Ian Sommerville. 2016. *Software Engineering* (10th ed.). Addison-Wesley, USA.

Regression Testing

- Verifies that a software behaviour has not changed by incremental changes to the software
- Modern software development processes are iterative/incremental
- Changes may be introduced which may affect the validity of previous tests
- Regression testing is to verify
  - Pre-tested functionality still working as expected
  - No new bugs are introduced

Software Testing Techniques

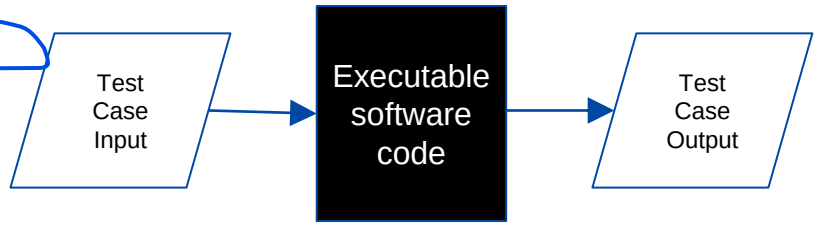


THE UNIVERSITY OF SYDNEY

Principle Testing Techniques

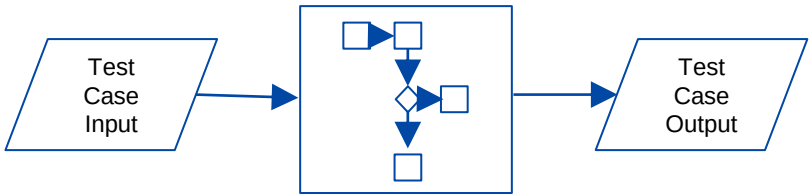
Black-box Testing

- No programming and software knowledge
- Carried by software testers
- Acceptance and system testing (higher levels)



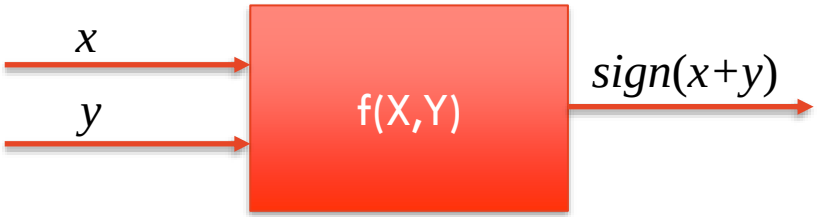
White-box Testing

- Software code understanding
- Carried by software developers
- Unit and integration testing (lower level)



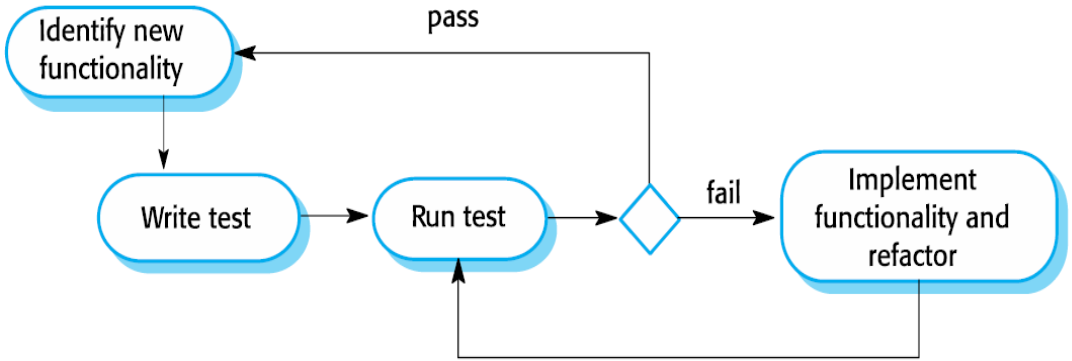
Black Box Testing – Example

- Test planned without knowledge of the code
- Based only on specification or design
- E.g., given a function that computes $sign\ (x+y)$



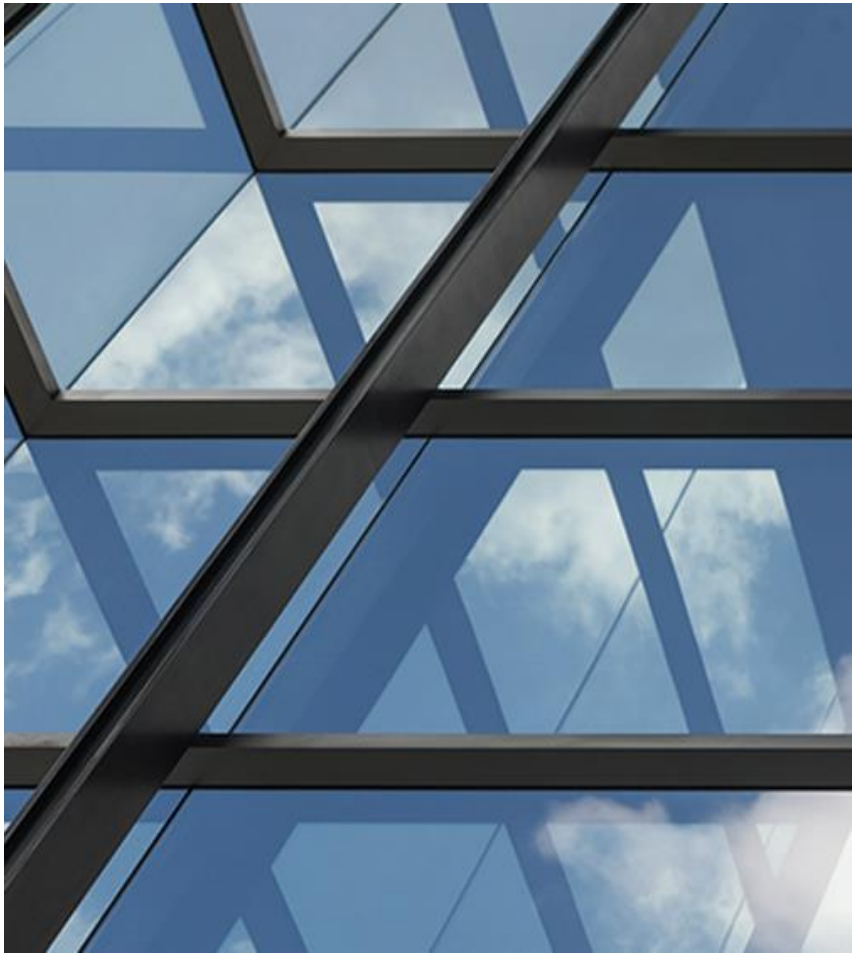
Test-Driven Development (TDD)

- A particular aspect of many (not all) agile methodologies
- Write tests before writing code
- And indeed, only write code when needed in order to pass tests!



Ian Sommerville. 2016. *Software Engineering* (10th ed.). Addison-Wesley, USA.

Test Cases Design



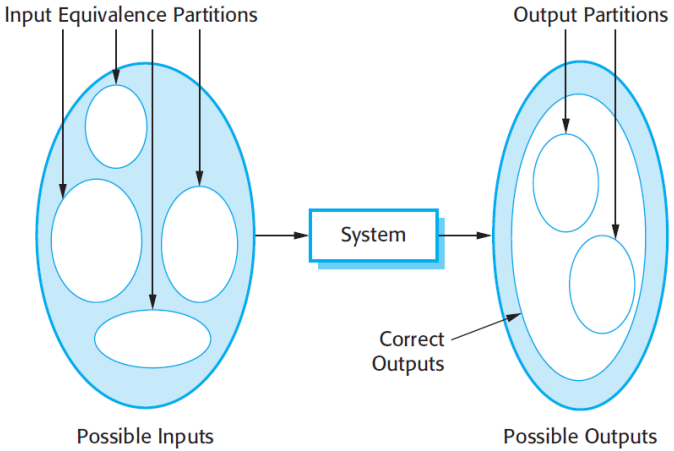
THE UNIVERSITY OF SYDNEY

Choosing Test Cases – Techniques

1. Partition testing (equivalence partitioning)
   - Identify groups of inputs with common characteristics
   - For each partition, choose tests on the boundaries and close to the midpoint
2. Guideline-based testing
   - Use testing guidelines based on previous experience of the kinds of errors often made
   - Understanding developers thinking

Equivalence Partitioning

- Different groups with common characteristics
  - E.g., positive numbers, negative numbers
- Program behave in a comparable way for all members of a group
- Choose test cases from each of the partitions
- Boundary cases
  - Select elements from the edges of the equivalence class



Choosing Test Cases – Exercise

- For the following class method, apply equivalence partitioning to define appropriate test cases.

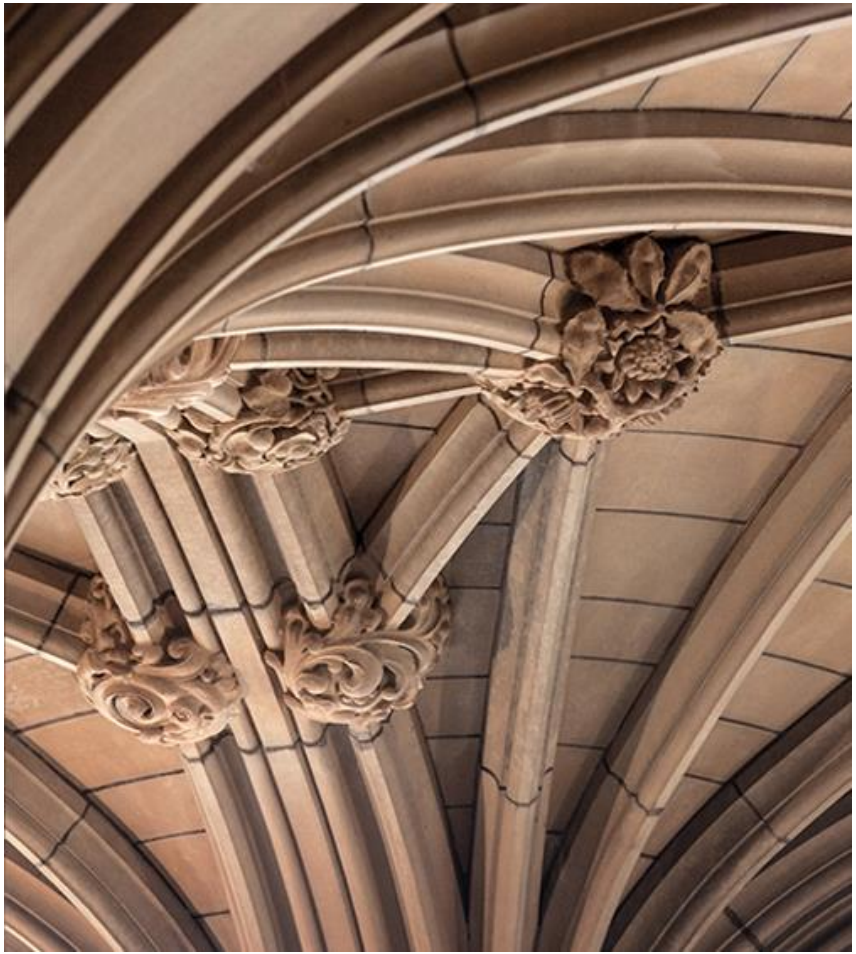
```
1 class MyGregorianCalendar {
2     ....
3     public static int getNumDaysInMonth(int month, int year){
4         ....
5     }
6 }
```

Choosing Test Cases – Solution Sample

Equivalence Class	Value for month	Value for year
Months with 31 days, non-leap years	7 (July)	1901
Months with 31 days, leap years	7 (July)	1904
Months with 30 days, non-leap years	6 (June)	1901
Months with 30 days, leap year	6 (June)	1904
Months with 28 or 29 days, non-leap year	2 February	1901
Months with 28 or 29 days, leap year	2 February	1904

Equivalence Class	Value for month	Value for year
Leap years divisible by 400	2 (February)	2000
Non-leap years divisible by 100	2 (February)	1900
Non-positive invalid month	0	1291
Positive invalid months	13	1315

Code (Test) Coverage



THE UNIVERSITY OF SYDNEY

Code (Test) Coverage

- The extent to which a source code has been executed by a set of tests
- Usually measured as percentage, e.g., 70% coverage
- Different criteria to measure coverage
  - E.g., method, statement, loop

Coverage Criteria

Coverage Criteria	Description
Method	How many of the methods are called, during the tests
Statement	How many statements are exercised, during the tests
Branch	How many of the branches have been exercised during the tests
Condition	Has each separate condition within each branch been evaluated to both true and false
Condition/decision coverage	Requires both decision and condition coverage be satisfied
Loop	Each loop executed zero times, once and more than once

Coverage Criteria – Statement

- Examine the code snippet. Compute the statement coverage after executing each of the following test cases:
  - year = 1901
  - year = 1904

- Is this good/enough? coverage? Why/why not?

```
1  //....
2  class MyGregorianCalendar {
3      public static boolean isLeapYear(int year) {
4          boolean leap;
5          if ((year%4) == 0){
6              leap = true;
7          } else {
8              leap = false;
9          }
10         return leap;
11     }
12     //....
```

Coverage Target

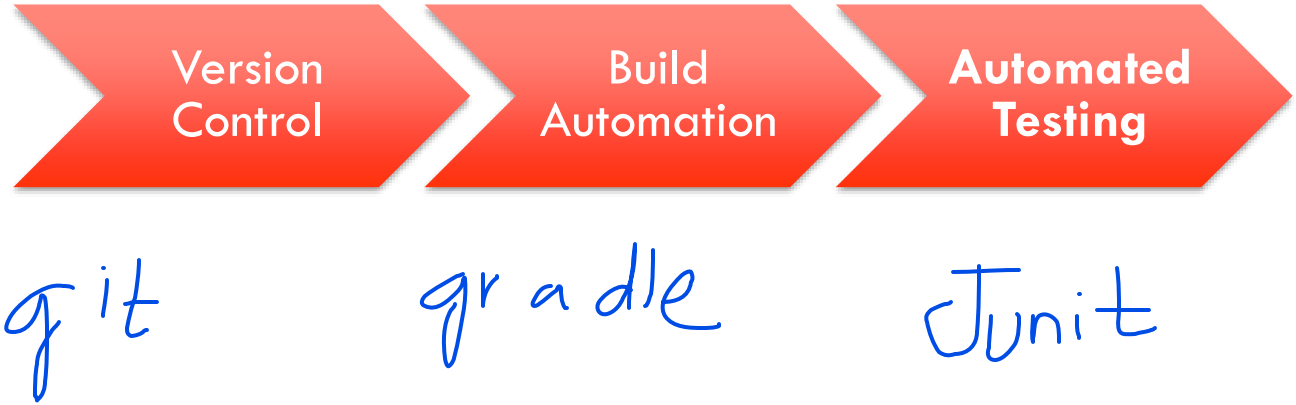
- What coverage should one aims for?

- Software criticality determines coverage level
  -
- Extremely high coverage for safety-critical (dependable) software
  - Government/standardization organizations

e.g.

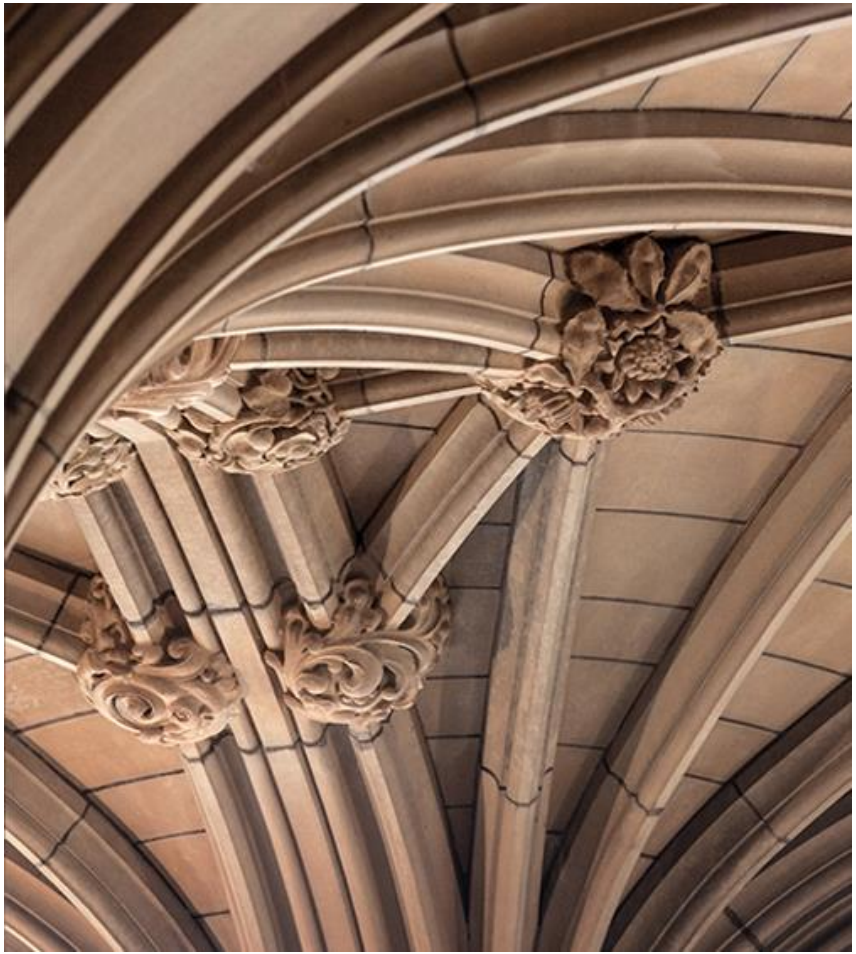
  - E.g., European corporation for space standardization (ESS-E-ST-40C)
    100% statement and decision coverage for 2 out of 4 criticality levels

Tools for Agile Development



Unit Testing

Junit



THE UNIVERSITY OF SYDNEY

Unit Testing – Terminology

- Code under test
- Unit test
  - Code written by a developer that executes a specific functionality in the code under test and asserts a certain behavior/state
  - E.g., method or class
  - External dependencies are removed (mocks can be used)
- Test Fixture
  - The context for testing
    - Usually shared set of testing data
    - Methods for setup those data

Test Frameworks

- Software that allows test cases to be described in standard form and run automatically
- Tests are managed and run automatically, and frequently
- Easy to understand reports
  - Big Green or Red signal

Unit Testing Frameworks for Java

- Junit
- TestNG
- Jtest (commercial)
- Many others …

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java

Unit Testing Frameworks – Junit

- An open source framework for writing and running tests for Java
- Uses annotations to identify methods that specify a test
- Can be integrated with Eclipse, and build automation tools (e.g., Ant, Maven, Gradle)

https://github.com/junit-team/junit4

JUnit – Constructs

- **JUnit test (Test class)**
  - A method contained in a class which is only used for testing (called *Test class*)
- **Test suite**
  - Contains several test classes which will be executed all in the specified order
- **Test Annotations**
  - To define/denote test methods (e.g., @Test, @Before)
  - Such methods execute the code under test
- **Assertion methods (assert)**
  - To check an expected result vs. actual result
  - Variety of methods
  - Provide meaningful messages in assert statements

JUnit – Annotations

JUnit 4*	Description
import org.junit.*	Import statement for using the following annotations
@Test	Identifies a method as a test method
@Before	Executed before each test to prepare the test environment (e.g., read input data, initialize the class)
@After	Executed after each test to cleanup the test environment (e.g., delete temporary data, restore defaults) and save memory
@BeforeClass	Executed once, before the start of all tests to perform time intensive activities, e.g., to connect to a database
@AfterClass	Executed once, after all tests have been finished to perform clean-up activities, e.g., to disconnect from a database

*See Junit 5 annotations and compare them https://junit.org/junit5/docs/current/user-guide/#writing-tests-annotations

Junit Test – Example

```
1
2 //Class to be tested
3 public class MyClass{
4     public int multiply (int m, int n){
5         return m * n;
6     }
7
8 }
9
```

MyClass' multiply(int, int) method

```
1  import static org.junit.Assert.assertEquals;
2
3  import org.junit.Test;
4
5  public class MyClassTests {
6
7      @Test
8      public void multiplicationOfZeroIntegersShouldReturnZero() {
9          MyClass tester = new MyClass(); // MyClass is tested
10
11         // assert statements
12         assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
13         assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");
14         assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");
15     }
16 }
```

MyClassTests class for testing the method multiply(int, int)

Junit – Assertions

- **Assert class** provides static methods to test for certain conditions

- Assertion method compares the actual value returned by a test to the expected value
  - Allows you to specify the expected and actual results and the error message
  - Throws an *AssertionException* if the comparison fails

Junit – Methods to Assert Test Results*

Method / statement	Description
assertTrue(Boolean condition [,message])	Checks that the Boolean condition is true.
assertFalse(Boolean condition [,message])	Checks that the Boolean condition is false.
assertEquals(expected, actual [,message])	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals(expected, actual, delta [,message])	Test that float values are equal within a given delta.
assertNull(object [,message])	Checks that the object is null.
assertNotNull(object, [,message])	Checks that the object is not null.

*More assertions in Junit 5 – https://junit.org/junit5/docs/current/user-guide/#writing-tests-assertions

Junit – Static Import

needed if this statement not used

```
// without static imports you have to write the following statement
Assert.assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));


// alternatively define assertEquals as static import
import static org.junit.Assert.assertEquals;

// more code

// use assertEquals directly because of the static import
assertEquals("10 x 5 must be 50", 50, tester.multiply(10, 5));
```

Junit – Executing Tests

- From the command line
  - *runClass()*: *org.junit.runner.JUnitCore* class allows to run one or several test classes
  - *org.junit.runner.**Result*** object maintain test results

- Test automation
  - Build tools (e.g., Maven or Gradle) along with a Continuous Integration Server (e.g., Jekins) can be configured to automate test execution
  - Essential for regular daily tests (agile development)

Junit – Executing Tests from Command line

```java
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class MyTestRunner {
  public static void main(String[] args) {
    Result result = JUnitCore.runClasses(MyClassTest.class);
    for (Failure failure : result.getFailures()) {
      System.out.println(failure.toString());
    }
  }
}
```

print each result/ error msg

Junit – Test Suites

```
1  import org.junit.runner.RunWith;
2  import org.junit.runners.Suite;
3  import org.junit.runners.Suite.SuiteClasses;
4
5  @RunWith(Suite.class)
6  @SuiteClasses({
7        1 MyClassTest.class,
8        2 MySecondClassTest.class })
9
10 public class AllTests {
11
12 }
```

selections for suite

Junit – Test Execution Order

- Junit assumes that all test methods can be executed in an arbitrary order
- Good test code should not depend on other tests and should be well defined
- You can control it but it will lead into problems (poor test practices)
- By default, Junit 4.11uses a deterministic order *(MethodSorters.DEFAULT)*
- @FixMethodOrder to change test execution order (not recommended practice)
  - *@FixMethod Order(MethodSorters.JVM)*
  - *@FixMethod Order(MethodSorters.NAME ASCENDING)*

https://junit.org/junit4/javadoc/4.12/org/junit/FixMethodOrder.html

Junit – Parameterized Test Example

```java
package testing;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import java.util.Arrays;
import java.util.Collection;
import static org.junit.Assert.assertEquals;
import static org.junit.runners.Parameterized.*;

@RunWith(Parameterized.class)
public class ParameterizedTestFields {

    // fields used together with @Parameter must be public
    @Parameter(0)
    public int m1;
    @Parameter(1)
    public int m2;
    @Parameter(2)
    public int result;

    // creates the test data
    @Parameters
    public static Collection<Object[]> data() {
        Object[][] data = new Object[][] { { 1 , 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
        return Arrays.asList(data);
    }

    @Test
    public void testMultiplyException() {
        MyClass tester = new MyClass();
        assertEquals("Result", result, tester.multiply(m1, m2));
    }


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

}
```

annotation

0 1 2 0 1 2 0 1 2 index

static test method

Junit – Parameterized Test

- A class that contains a test method and that test method is executed with different parameters provided
- Marked with *@RunWith(Parameterized.class)* annotation
- The test class must contain a static method annotated with @Parameters
  - This method generates and returns a collection of arrays. Each item in this collection is used a s a parameter for the test method

Junit – Verifying Exceptions

- Verifying that code behaves as expected in exceptional situations (exceptions) is important
- The *@Test* annotation has an optional parameter "expected" that takes as values subclasses of *Throwable*

```
1
2 new ArrayList<Object>().get(0);
3
```

Verify that ArrayList throws IndexOutOfBoundException

```
1
2 @Test(expected = IndexOutOfBoundsException.class)
3 public void empty() {
4       new ArrayList<Object>().get(0);
5 }
```

exceptionName.class

@Test(expected =

Junit – Verify Tests Timeout Behaviour (1)

- To automatically fail tests that 'runaway' or take too long:

- Timeout parameter on @Test
  - Cause test method to fail if the test runs longer than the specified timeout
  - *timeout in milliseconds in @Test*

```
1 @Test(timeout=1000)
2 public void testWithTimeout() {
3     ...
4 }
```

Junit – Rules

- A way to add or redefine the behaviour of each test method in a test class
  - E.g., specify the exception message you expect during the execution of test code
- Annotate fields with the *@Rule*
- Junit already implements some useful base rules

Junit – Rules

Rule	Description
TemporaryFolder	Creates files and folders that are deleted when the test finishes
ErrorCollector	Let execution of test to continue after first problem is found
ExpectedException	Allows in-test specification of expected exception types and messages
TimeOut	Applies the same timeout to all test methods in a class
ExternalResources	Base class for rules that setup an external resource before a test (a file, socket, database connection)
RuleChain	Allows ordering of TestRules

See full list and code examples of Junit rules https://github.com/junit-team/junit4/wiki/Rules

Junit – ErrorCollector Rule Example

- Allows execution of a test to continue after the first problem is found

```
1  public static class UsesErrorCollectorTwice {
2    @Rule
3    public final ErrorCollector collector = new ErrorCollector();
4
5    @Test
6    public void example() {
7      collector.addError(new Throwable("first thing went wrong"));
8      collector.addError(new Throwable("second thing went wrong"));
9    }
10 }
```

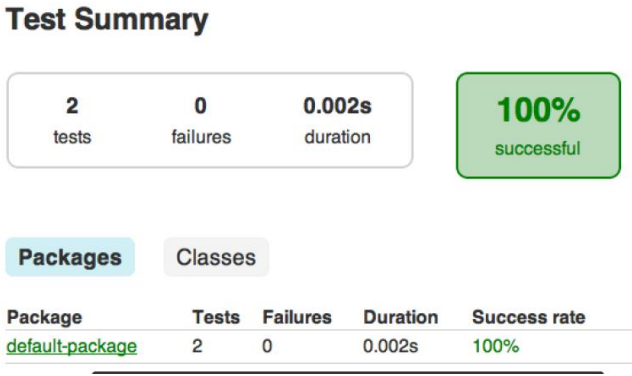
Junit – Eclipse Support

- Create Junit tests via wizards or write them manually
- Eclipse IDE also supports executing tests interactively
  - Run-as Junit Test will starts JUnit and execute all test methods in the selected class
- Extracting the failed tests and stack traces
- Create test suites

Tests Automation – Junit with Gradle

- To use Junit in your Gradle build, add a testCompile dependency to your build file
- Gradle adds the test task to the build and needs only appropriate Junit JAR to be added to the classpath to fully activate the test execution

```
1 ...
2 apply plugin: 'java'
3
4 repositories {
5     mavenCentral()
6 }
7 dependencies {
8     testCompile 'junit:junit:4.8.2'
9 }
```



Junit with Gradle – Parallel Tests

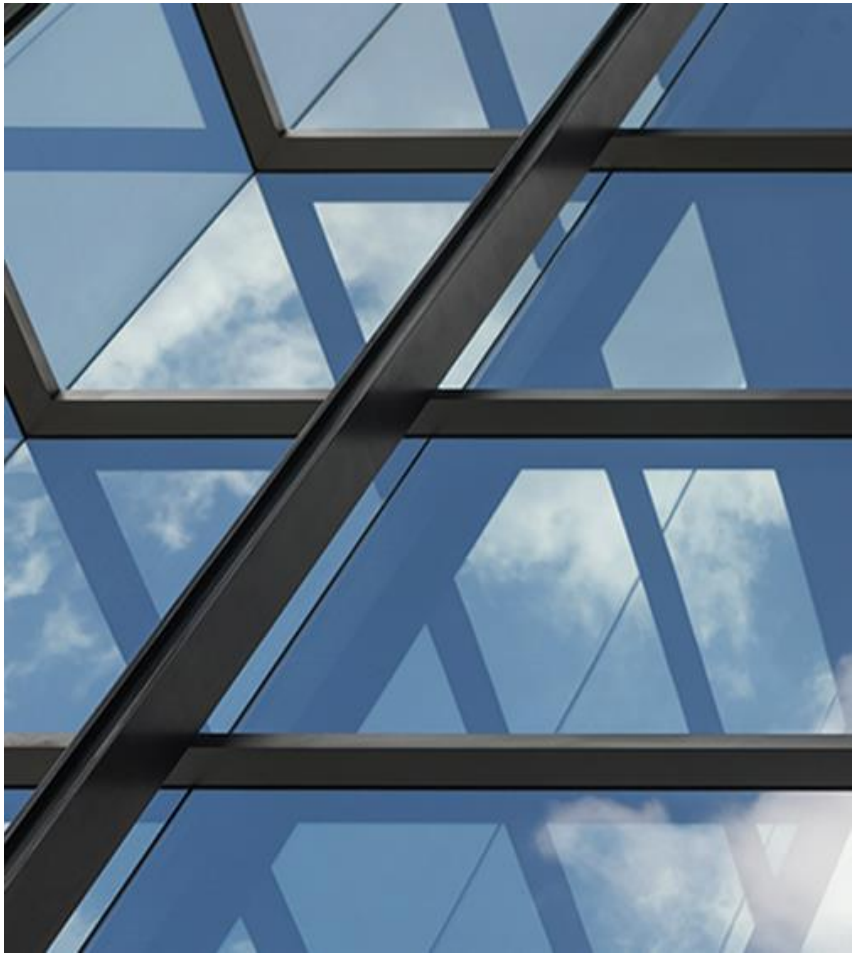
```
1  ...
2  apply plugin: 'java'
3
4  repositories {
5      mavenCentral()
6  }
7  dependencies {
8      testCompile 'junit:junit:4.8.2'
9  }
10 test {
11     maxParallelForks = 5
12     forkEvery = 50
13 }
```

maximum simultaneous JVMs spawned

causes a test-running JVM to close and be replaced by a brand new one after the specified number of tests have run under an instance

in this case 50

Code Coverage Tools



THE UNIVERSITY OF SYDNEY

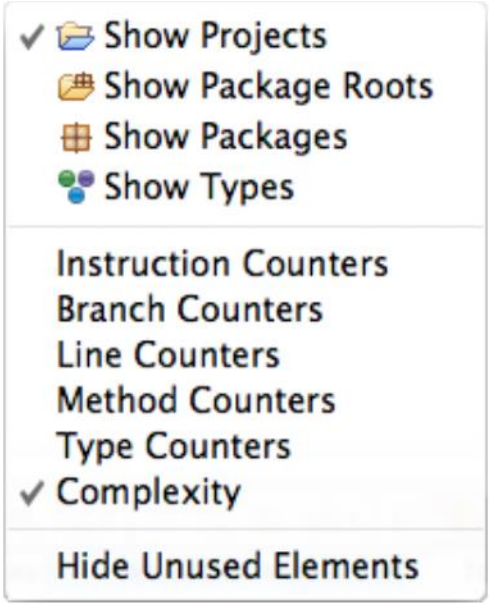
Tools for Code Coverage in Java

- There are many tools/plug-ins for code coverage in Java
- Example: EclEmma*
- EclEmma is a code coverage plug-in for Eclipse
  - It provides rich features for code coverage analysis in Eclipse IDE
  - EclEmma is based on the *JaCoCo* code coverage library
    - *JaCoCo* is a free code coverage library for Java, which has been created by the EclEmma team

https://www.eclemma.org/

EclEmma – Counters

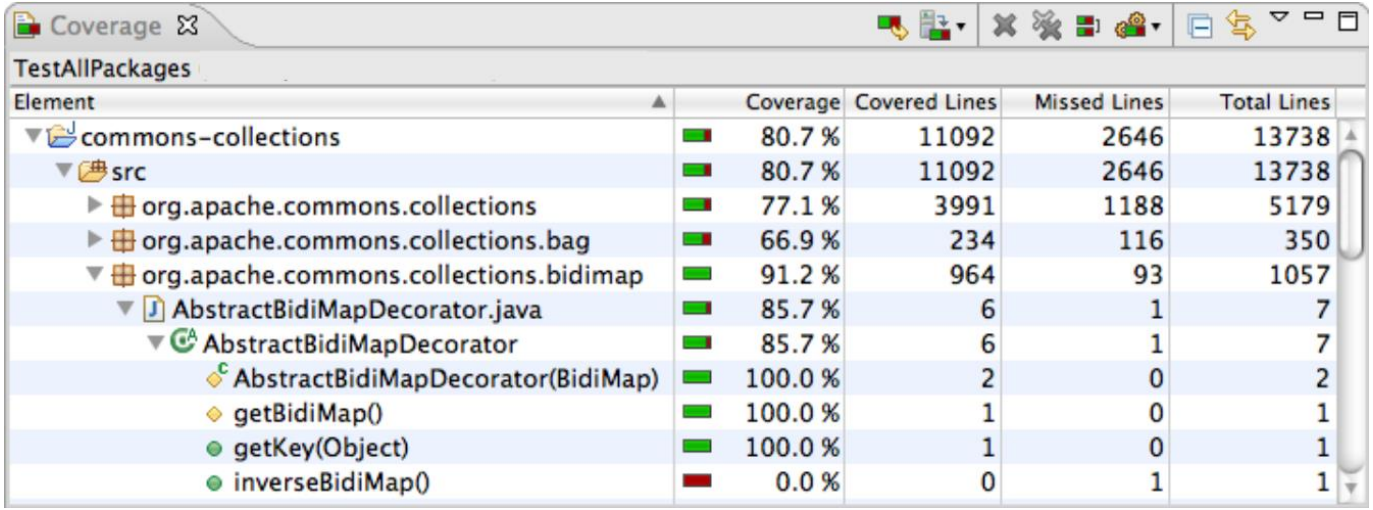
- EclEmma supports different types of counters to be summarized in code coverage overview
  - bytecode instructions, branches, lines, methods, types and cyclomatic complexity
  - Should understand each counter and how it is measured
  - Counters are based on JaCoCon - see JaCoCo documentation for detailed counter definitions



https://www.eclemma.org/

EclEmma Coverage View

The **Coverage view** shows all analyzed Java elements within the common Java hierarchy. Individual columns contain the numbers for the active session, always summarizing the child elements of the respective Java element



Element	Coverage	Covered Lines	Missed Lines	Total Lines
commons-collections	80.7 %	11092	2646	13738
src	80.7 %	11092	2646	13738
org.apache.commons.collections	77.1 %	3991	1188	5179
org.apache.commons.collections.bag	66.9 %	234	116	350
org.apache.commons.collections.bidimap	91.2 %	964	93	1057
AbstractBidiMapDecorator.java	85.7 %	6	1	7
AbstractBidiMapDecorator	85.7 %	6	1	7
AbstractBidiMapDecorator(BidiMap)	100.0 %	2	0	2
getBidiMap()	100.0 %	1	0	1
getKey(Object)	100.0 %	1	0	1
inverseBidiMap()	0.0 %	0	1	1

https://www.eclemma.org/

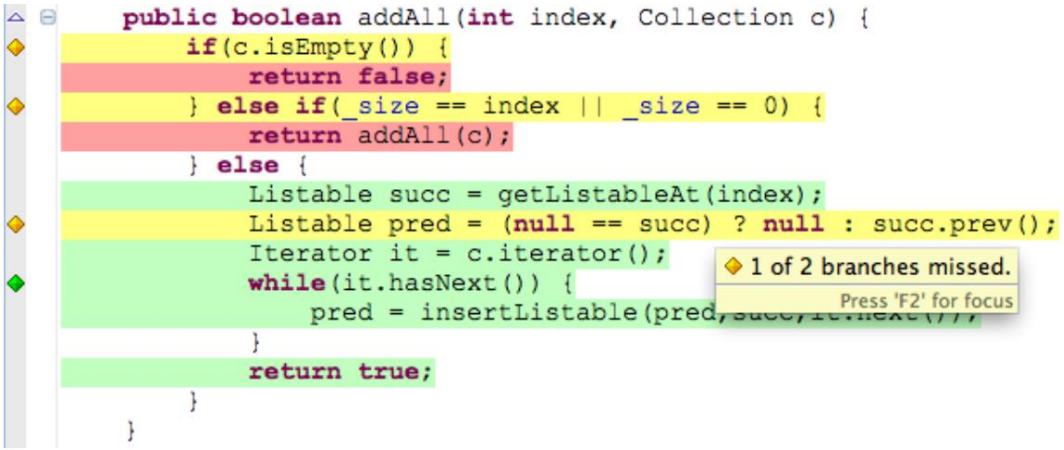
EclEmma Coverage – Source Code Annotations

Source lines color code:

- **green** for fully covered lines,
- **yellow** for partly covered lines (some instructions or branches missed)
- **red** for lines that have not been executed at all

Diamonds color code

- **green** for fully covered branches,
- **yellow** for partly covered branches
- **red** when no branches in the particular line have been executed.



https://www.eclemma.org/

References

- Armando Fox and David Patterson 2015. Engineering Software as a Service: An Agile Approach Using Cloud Computing (1st Edition). Strawberry Canyon LLC
- Ian Sommerville 2016. Software Engineering: Global Edition (3rd edition). Pearson, Englad
- Tim Berglund and Matthew McCullough. 2011. Building and Testing with Gradle (1st ed.). O'Reilly Media, Inc.
- Vogella GmbH, JUnt Testing with Junit – Tutorial (Version 4.3,21.06.2016) http://www.vogella.com/tutorials/JUnit/article.html
- Junit 4, Project Documentation, https://junit.org/junit4/

Tutorial: Testing with Junit
Next week Lecturer: Continuous Integration (CI)

THE UNIVERSITY OF SYDNEY