Agile Software Development Practices

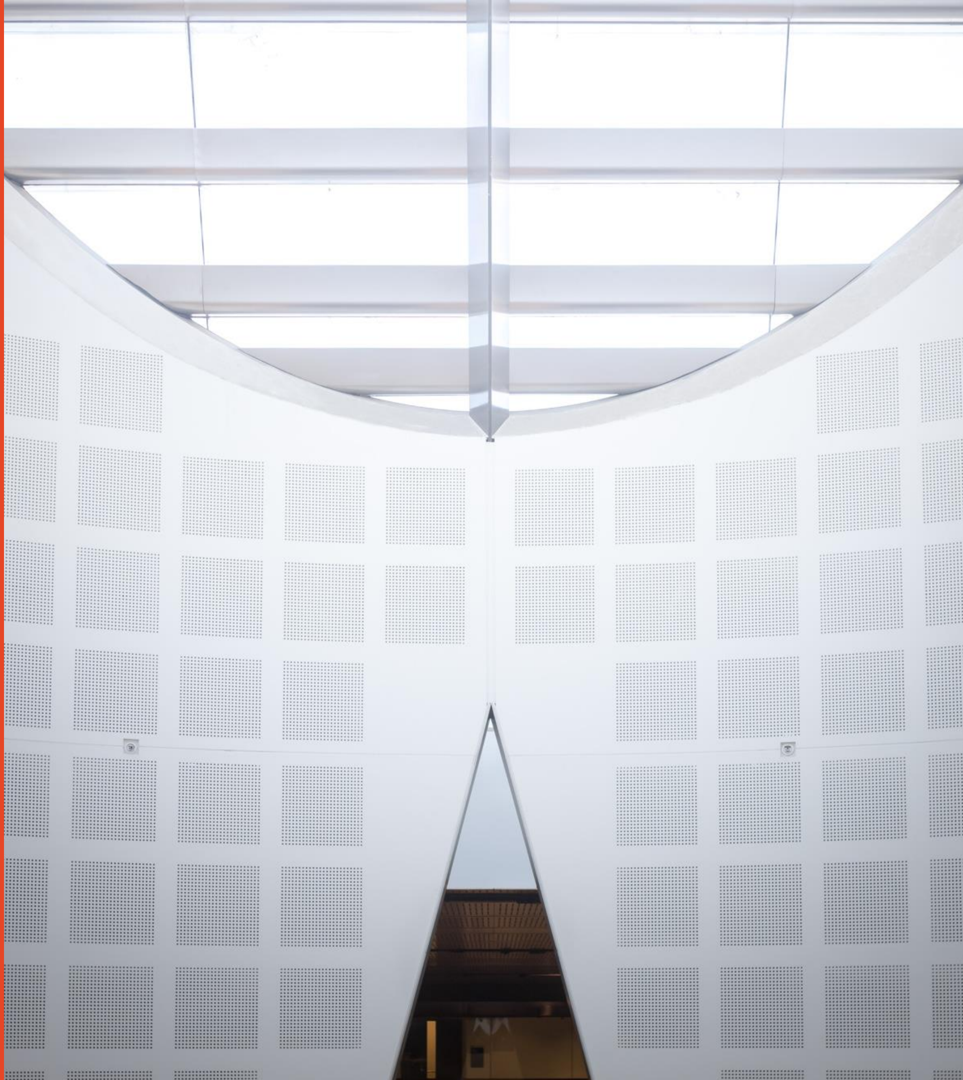
SOF2412 / COMP9412

Quality Assurance – Software Testing

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY

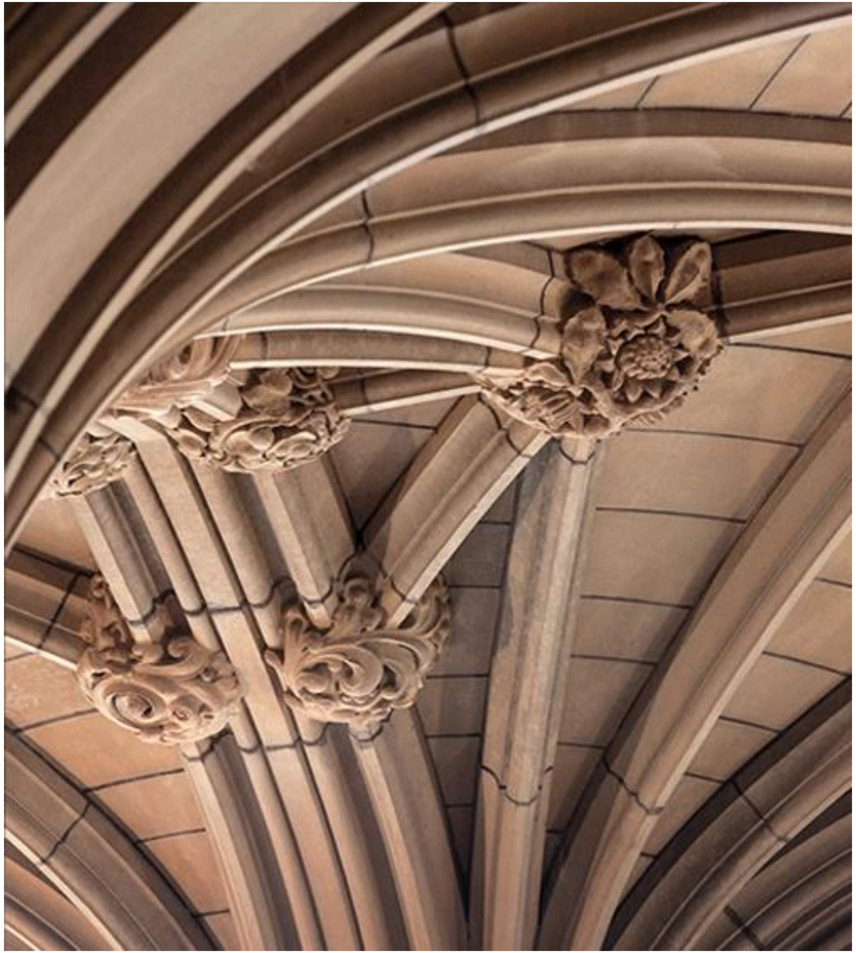


Copyright warning

Agenda

- Software Quality Assurance
- Software Testing
- Tools for Unit Testing
  - JUnit
  - Junit with Gradle
- Code Coverage

Software Quality Assurance



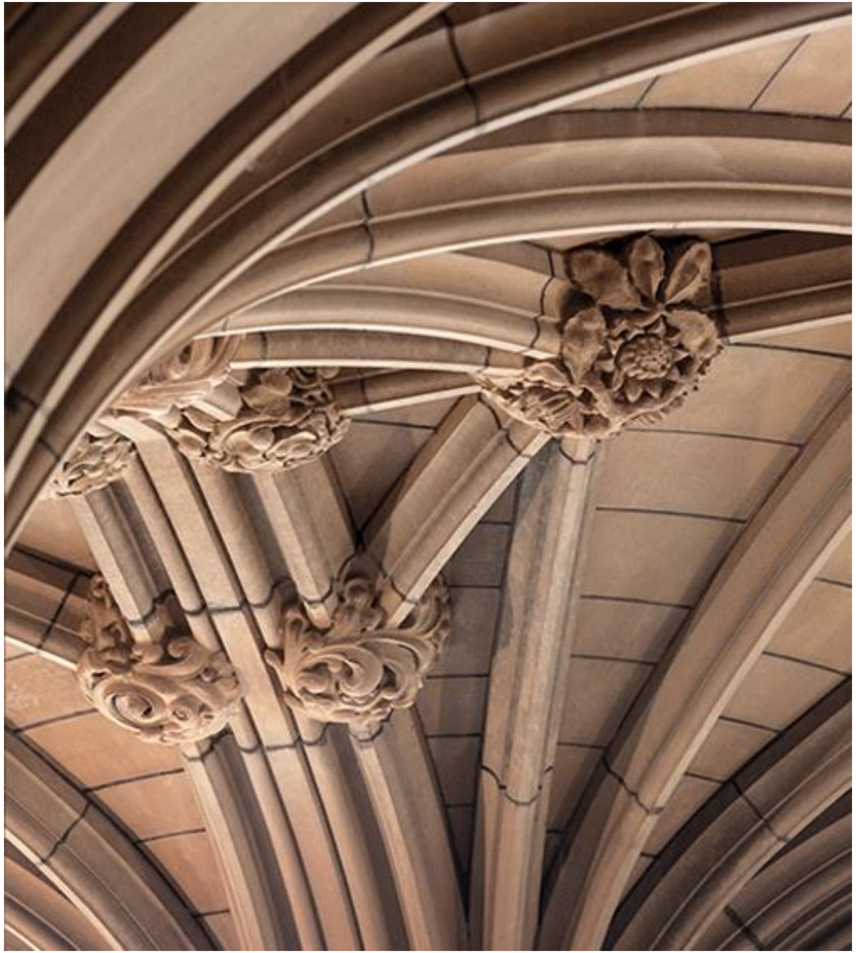
THE UNIVERSITY OF SYDNEY

Software Quality Assurance

- Quality (products)
  - "Fitness for use" which should provide value for customers and manufacturer
- Software quality
  - Satisfying end use's needs; correct behaviour, easy to use, does not crash, etc
  - Easy to the developers to debug and enhance
- Quality Assurance (QA)
  - Processes and standards that lead to manufacturing high quality products
- 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?



Why Software Testing?

- Software development and maintenance costs
  - Big financial burden

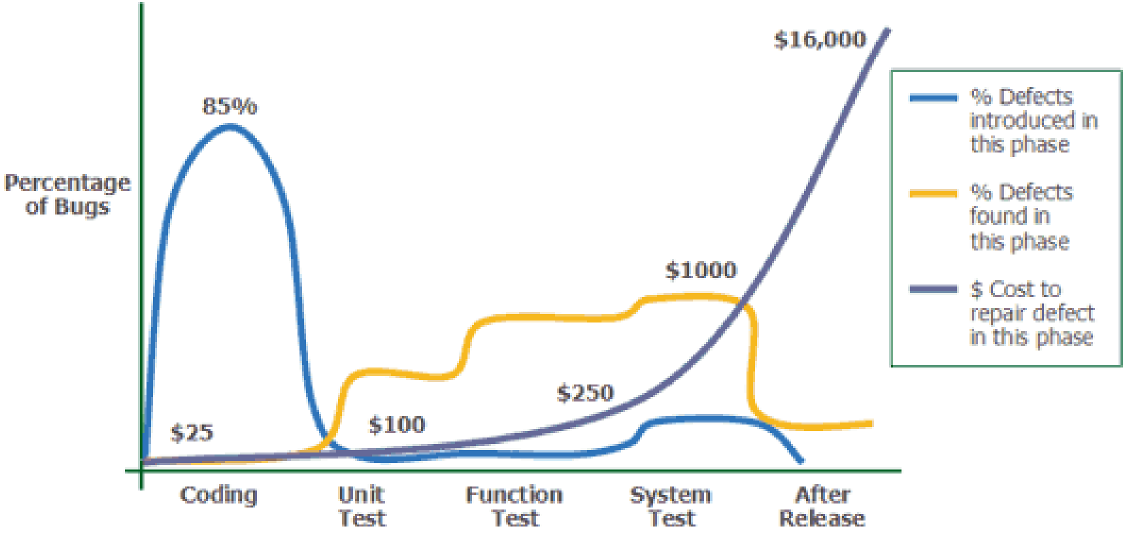
- Total costs of inadequate software testing on the US economy is $59.5bn
  - NIST study 2002*
  - One-third of the cost could be eliminated by *improved software testing*

- Risk management across the Software Development Lifecycle (SDLC)

- 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 as per cases in next slides

* 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

Nissan Recall - Airbag Defect*

- What happened?
  - ~ 3.53 million vehicles recall of various models 2013-2017
  - Front passenger airbag may not deploy in an accident

- Why Did happen?
  - Software that activates airbags deployment improperly classify occupied passenger seat as empty in case of accident
  - Software defect that could lead to improper airbag function (failure)
  - No warning that the airbag may not function properly
  - Software sensitivity calibration due to combination of factors (high engine vibration and changing seat status)

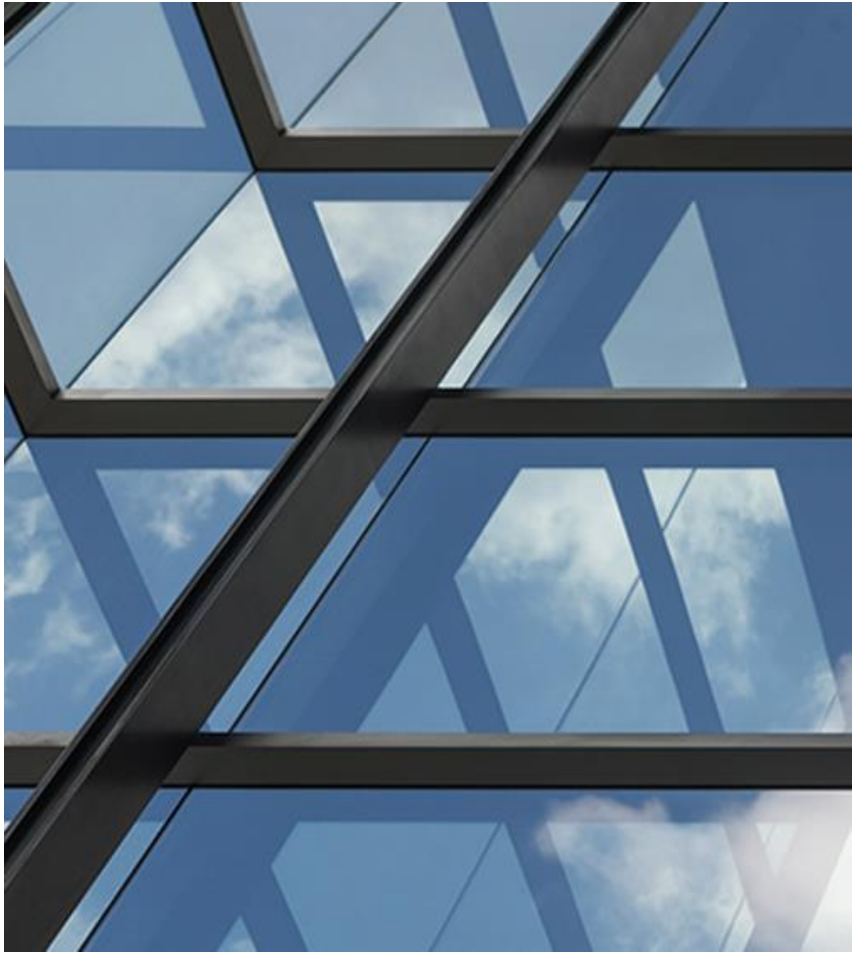
http://www.reuters.com/article/us-autos-nissan-recall/nissan-to-recall-3-53-million-vehicles-air-bags-may-not-deploy-idUSKCN0XQ2A8
https://www.nytimes.com/2014/03/27/automobiles/nissan-recalls-990000-cars-and-trucks-for-air-bag-malfunction.html

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 happen?
  - Particular nonstandard sequence of keystrokes was entered within 8 seconds (improbable event that rarely triggers and went unnoticed for long time)
  - 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
  - Software checks safety replacing hardware interlocks in previous Therac versions
  - Absence of independent software code review
  - Desired and possible results were not considered during software design
  - ‘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

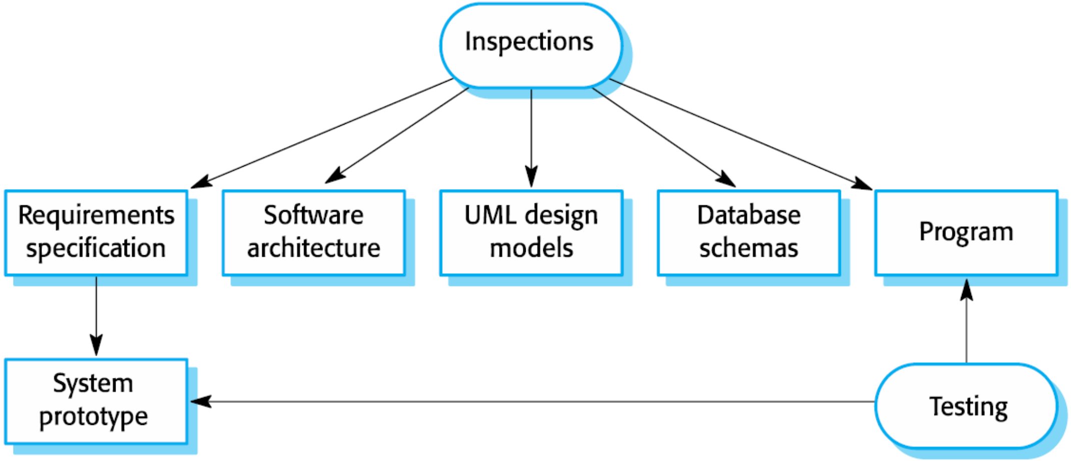
What is Software Testing?



Software Testing

- Software inspection (aka. static verification)
  - Static system analysis to discover problems, e.g., code review
  - May be applied to requirements, design/models, configuration and test data
- Software testing (aka. validation)
  - Functional: performs all expected functions properly
  - Non-functional: secure, performance, usability

Software Inspection (Static Verification)



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

What is testing?

- **Yes/No** answer for requirements
  - For all possible program inputs: **Verification**
  - Only for some inputs: **Validation**

- A test must be precise
  - For given input what is the expected output
  - The test must be able to decide if the actual output is acceptable or not

- What is the trouble?
  - How to specify requirements in a testing language
    - Natural language is not precise
  - How to find tests that check all requirements
    - Often the stakeholder hasn't thought carefully about some complicated situations

Test Cases Can Disambiguate the Requirements

- A requirement expressed in English may not capture all the details

- But we can write test cases for the various situations

  - the expected output is a way to make precise what the stakeholder wants

  - E.g. write a test case with empty input, and say what output is expected

Plans

- In traditional process methodologies, these are complicated and took a lot of effort to create
  - And they were signed-off, so have status of a contract obligation

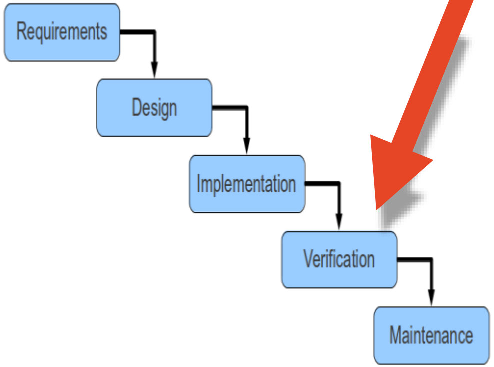
- In agile, they may be as simple as a prioritized list of tasks

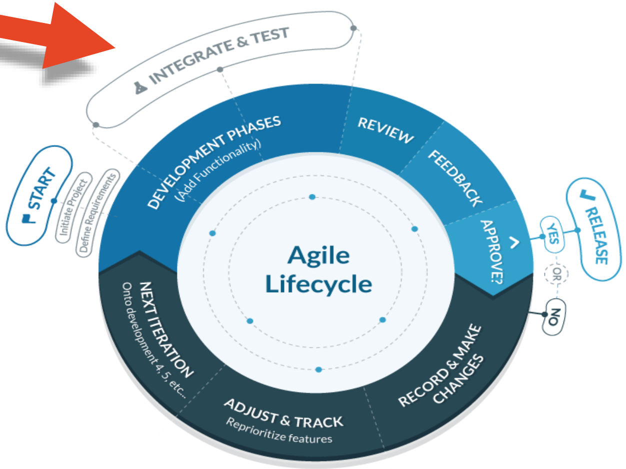
When is Testing happening?

- SW Development Process
  - Functional and Non-Functional Requirements
  - Design System
  - Implement system or sub-system
  - **Test whether system works according to requirements**

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

Testing Levels

Level	Description
Acceptance Testing	The process of verifying desired acceptance criteria (user requirements) are met in the system (functional and non-functional) from the user point of view
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 (top-down and bottom-up strategies). On-going activity Incremental integration testing vs. “Big Bang” testing
System Testing	The process of verifying the functionality and behaviour of the entire software system. Particularly, security, performance, reliability, and external interfaces to other applications

Testing Objectives

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

- Defect discovery
- Dealing with unknowns
- Incorrect or undesired behaviour, missing requirement, system property
- Verifying different system properties
  - Functional specification correctly implemented
  - Non-functional properties
    - security, performance, reliability, interoperability, and usability

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

Testing Terminology

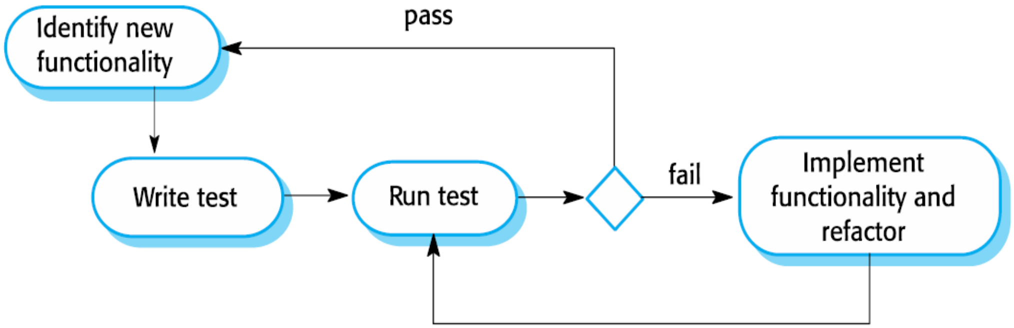
- **Fault:** cause of a malfunction
- **Failure:** undesired effect in the system's function or behaviour
- **Bug:** result of coding error incurred by a programmer
- **Debugging:** investigating/resolving software failure
- **Defect:** deviation from its requirements/specifications

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

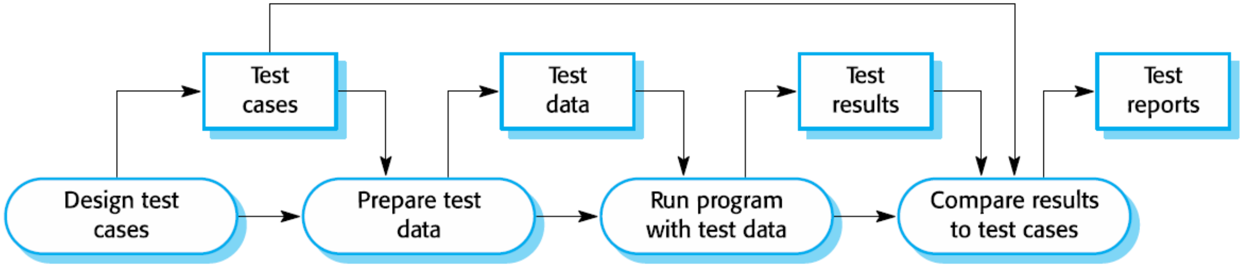
Test-driven Development

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


Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

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

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

Who Does Testing?

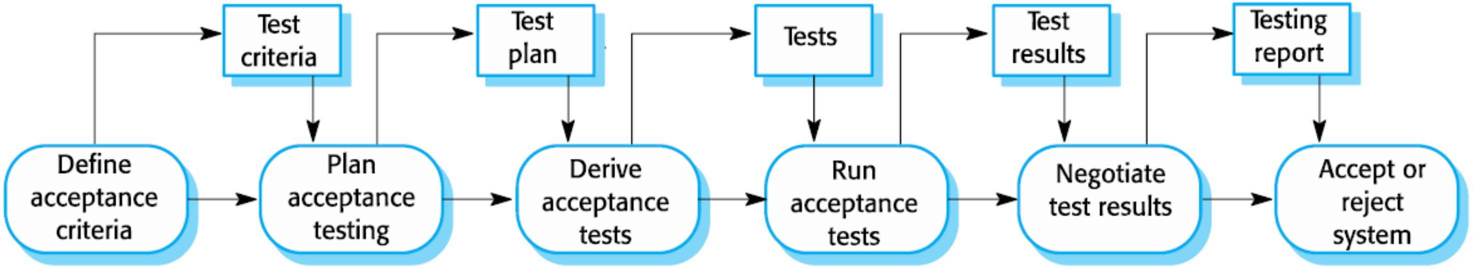
- Developers test their own code

- 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

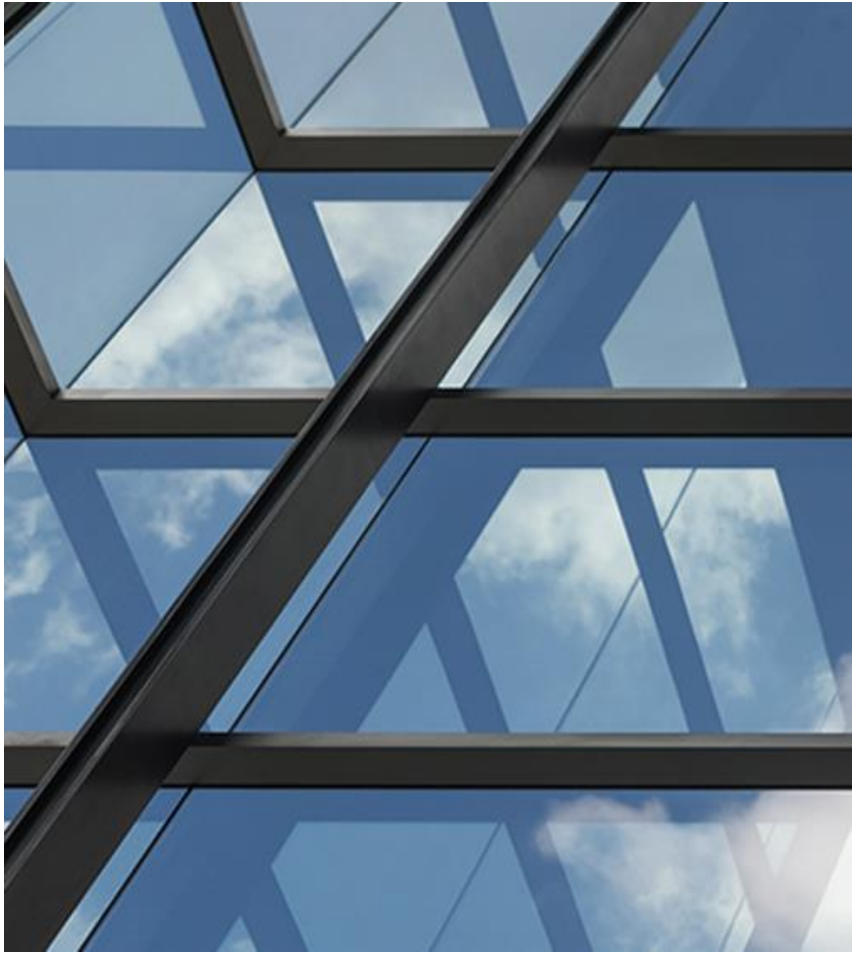
- Real users, doing real work

Acceptance Testing Process



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

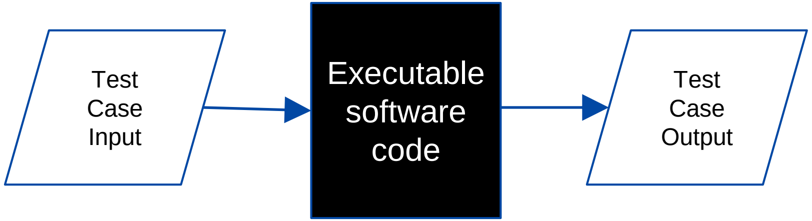
Software Testing Techniques



Principle Testing Techniques

Black-box Testing

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



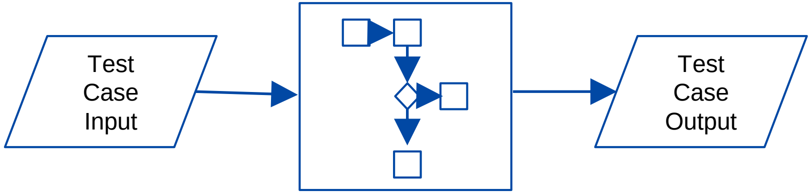
Test Case Input

Executable software code

Test Case Output

White-box Testing

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

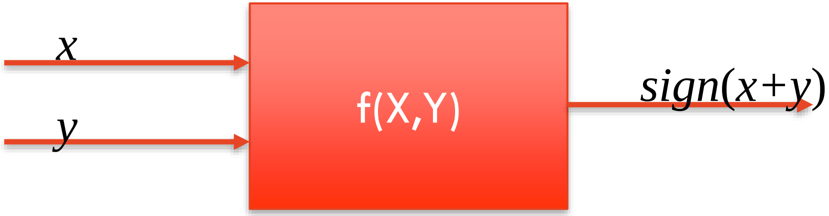


Test Case Input

Test Case Output

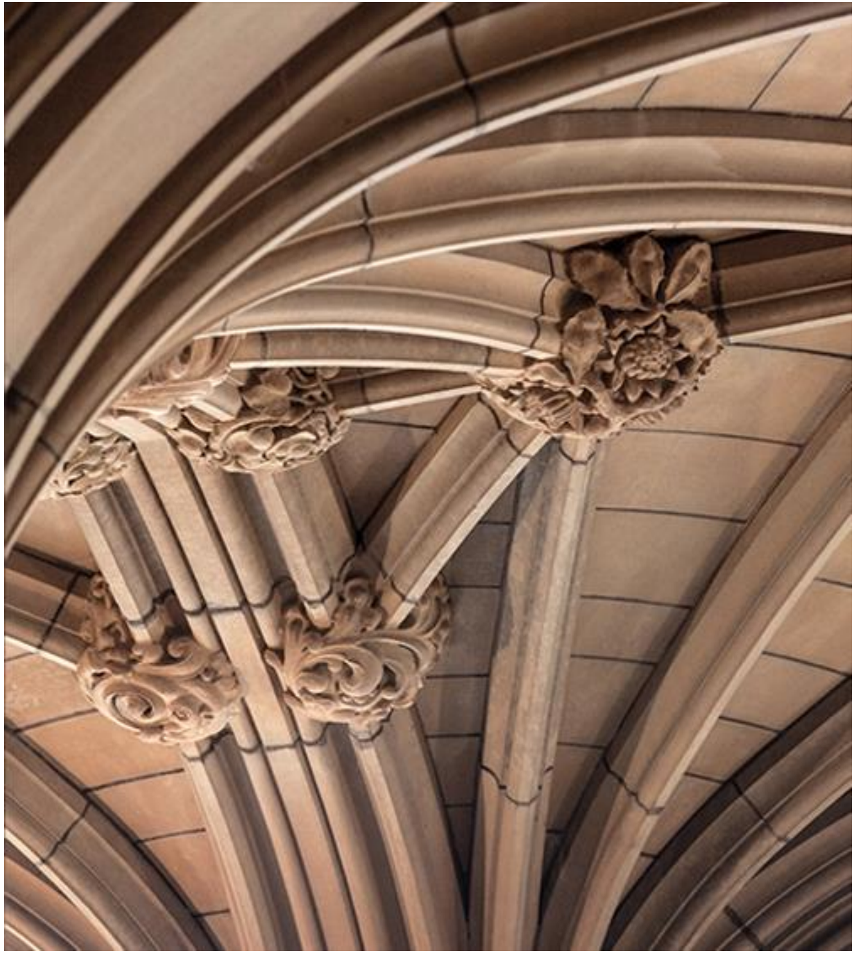
Black Box Testing – Example

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



Unit Testing

Junit



Automated Test Frameworks

- Key sign of good practice is that tests are kept as an asset of the process
  - and can be run automatically, and frequently

- Also, reports should be easy to understand
  - Big Green or Red Signal

- A framework is some software that allows test cases to be described in standard form and run automatically

Unit Testing Terminology

- **Application Under Test (AUT)**
- **Test Fixture:** a fixed state in code which is tested used as input for a test (test precondition)
  - E.g., a fixed string (test fixture), which is used as input for a method. The test would validate if the method behaves correctly with this input
- **Unit test:** a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state
  - Small unit of code e.g., a method or class
  - External dependencies are removed (replaced with test implementation or a mock object created by a test framework
  - Not suitable for complex user interface or component interaction
- **Test coverage:** percentage of code which is tested by unit tests

Junit

- Unit testing framework for Java programming language

- One of the unit testing frameworks collectively known as xUnit

- Junit composed of different modules from different sub-projects; Junit Platform, Junit Jupiter and Junit Vintage

- Uses annotations to identify methods that specify a test

- Can be integrated with Eclipse, Ant, Maven an Gradle

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

Junit Test

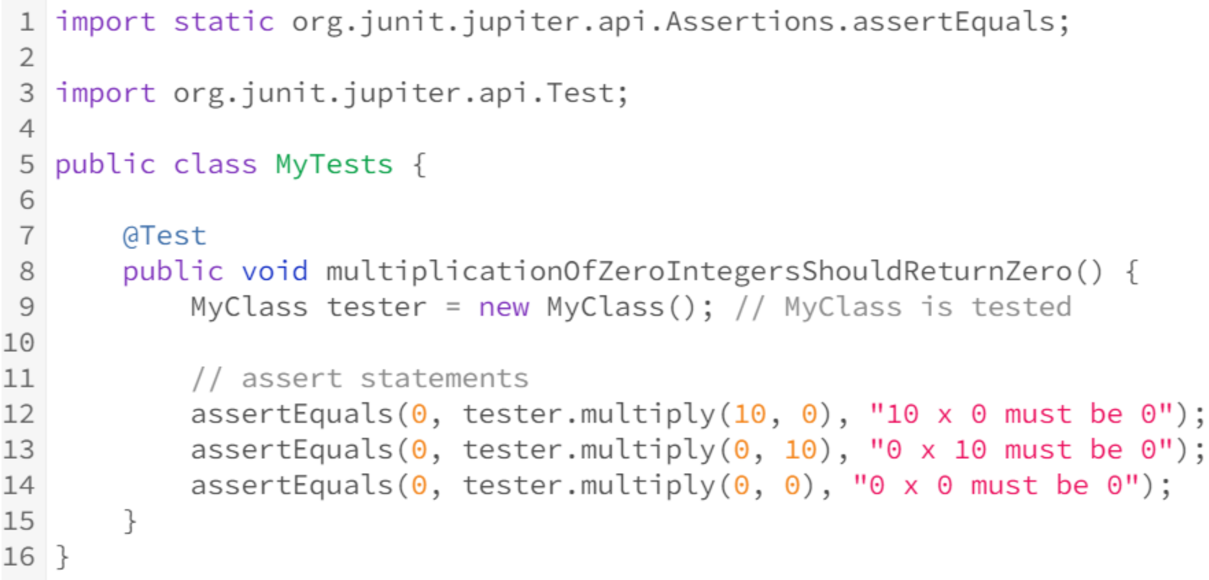
- **Junit test** is a method contained in a class which is only used for testing (called *Test class*)

- **Test Annotations (@Test)** used to define/denote test methods
  - Such methods execute the code under test

- **Assertions methods (asserts)** are used to check an expected result versus the actual results
  - Provide meaningful messages in assert statements

Junit Test – Example

MyTests for a MyClass which has got multiply (int, int) method



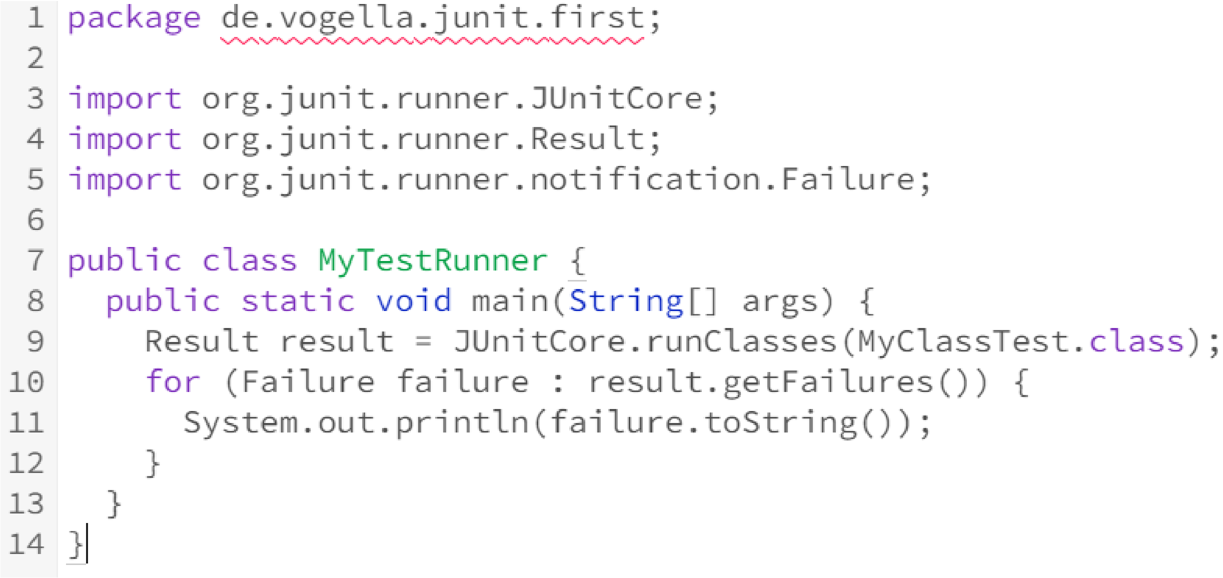
```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
        MyClass tester = new MyClass(); // MyClass is tested

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

Junit – Executing Tests

- Build tools such as Maven or Gradle along with a Continuous Integration Server (e.g., Jekins) can be configured to run tests automatically on a regular basis

- Tests can be executed from the command line as well
  - *runClass()* of the *org.junit.runner.JUnitCore* class allows developers to run one or several test classes
  - Information about tests can be retrieved using the org.junit.runner.Result object

Junit – Executing Tests



```java
package de.vogella.junit.first;

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());
    }
  }
}
```

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
- Use annotations to specify test method execution order
  - E.g., to run test methods sorted by method name use @FIXMETHODORDER(MethodSorters.NAME ASCENDING)

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. It is used to prepare the test environment (e.g., read input data, initialize the class).
@After	Executed after each test. It is used to cleanup the test environment (e.g., delete temporary data, restore defaults). It can also save memory by cleaning up expensive memory structures.
@BeforeClass	Executed once, before the start of all tests. It is used to perform time intensive activities, for example, to connect to a database. Methods marked with this annotation need to be defined as static to work with JUnit.
@AfterClass	Executed once, after all tests have been finished. It is used to perform clean-up activities, e.g., to disconnect from a database. Methods annotated with this annotation need to be defined as static to work with JUnit.

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

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
  - It throws an *AssertionException* if the comparison fails

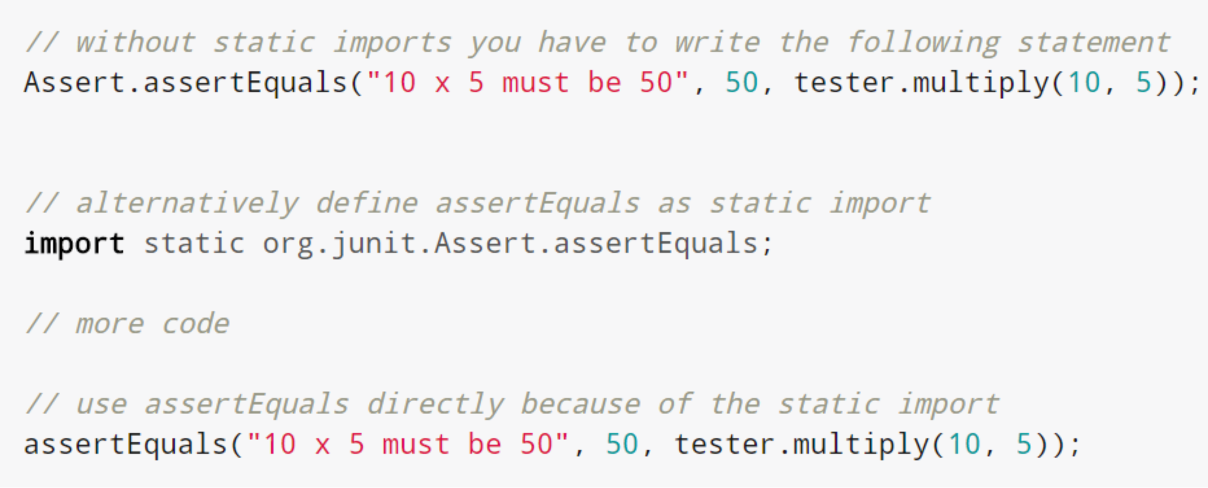
Junit – Methods to Assert Test Results*

Statement	Description
fail([message])	Let the method fail. Might be used to check that a certain part of the code is not reached or to have a failing test before the test code is implemented. Optional message parameter
assertTrue([message,] boolean condition)	Checks that the Boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is checked not the content of the arrays.
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number of decimals which must be the same.
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	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

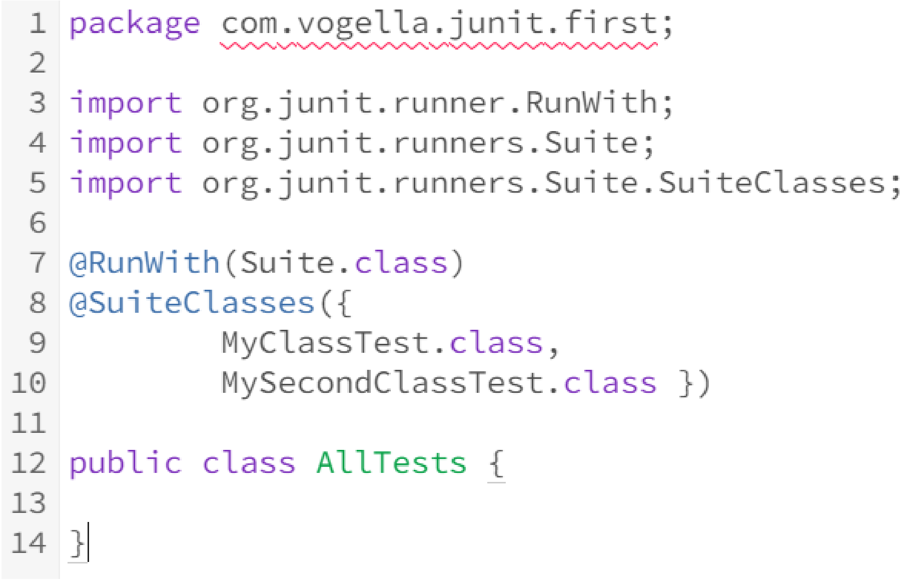
- Static import is a feature that allows fields and methods in a class as public static to be used without specifying the class in which the field s defined



```
// 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 – Test Suites

- **Test suit** contains several test classes which will be executed all in the specified order

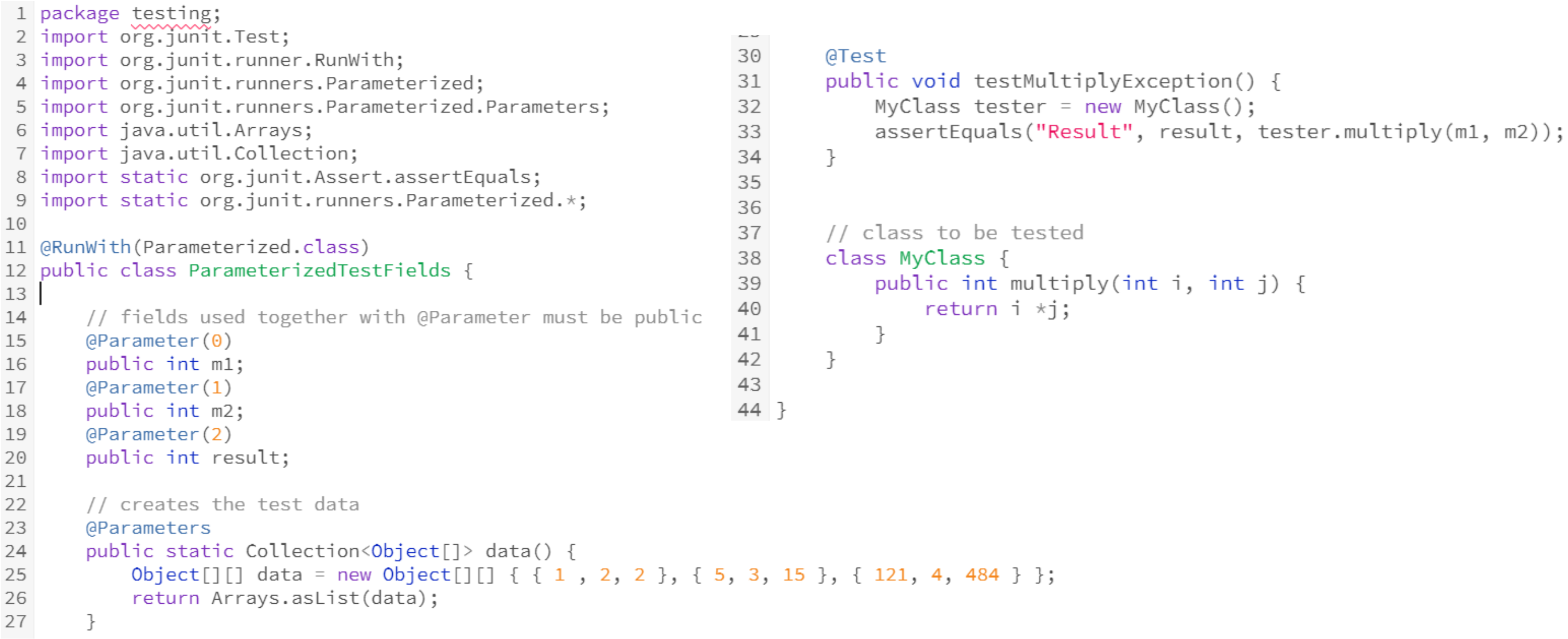


```java
package com.vogella.junit.first;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
        MyClassTest.class,
        MySecondClassTest.class })

public class AllTests {

}
```

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 – Parameterized Test



```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);
    }
```

```java
    @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;
        }
    }

}
```

Junit – Eclipse Support

- Create Junit tests via wizards or write them manually

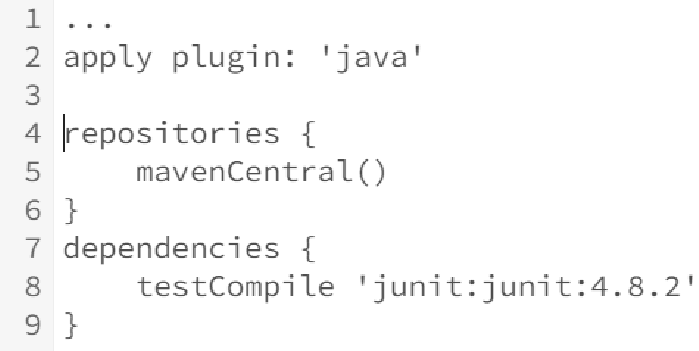
- Eclipse IDE also supports executing tests interactively
  - Run-as Junit Test will starts JUnit and execute all test methods in the selected class

- Extracting the failed tests and stack traces

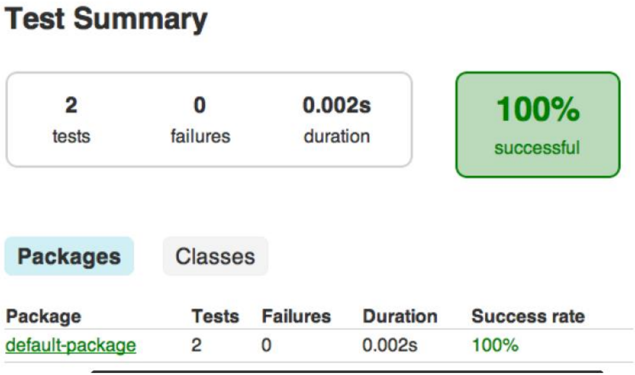
- Create test suites

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 graph and needs only appropriate Junit JAR to be added to the classpath to fully activate the test execution



```
...
apply plugin: 'java'

repositories {
    mavenCentral()
}
dependencies {
    testCompile 'junit:junit:4.8.2'
}
```



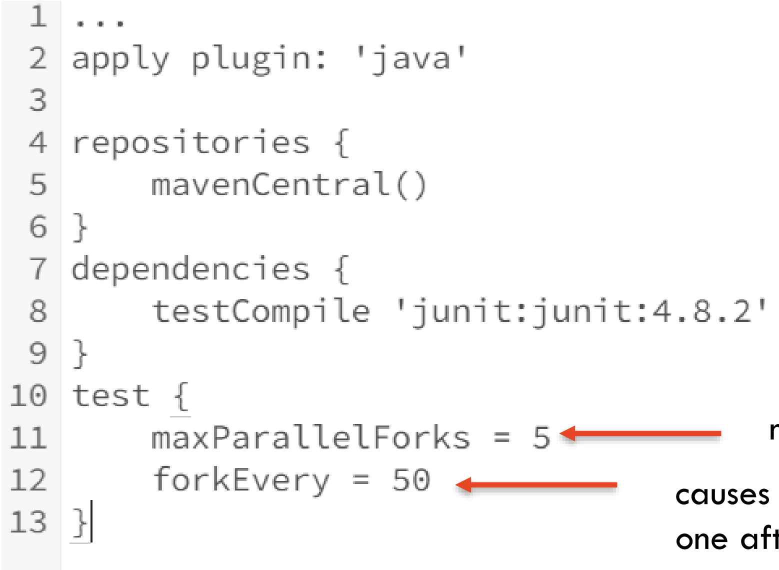
Test Summary

2 tests | 0 failures | 0.002s duration | 100% successful

Packages | Classes

Package	Tests	Failures	Duration	Success rate
default-package	2	0	0.002s	100%

Junit with Gradle – Parallel Tests



```
...
apply plugin: 'java'

repositories {
    mavenCentral()
}
dependencies {
    testCompile 'junit:junit:4.8.2'
}
test {
    maxParallelForks = 5
    forkEvery = 50
}
```

maximum simultaneous JVMs spawned

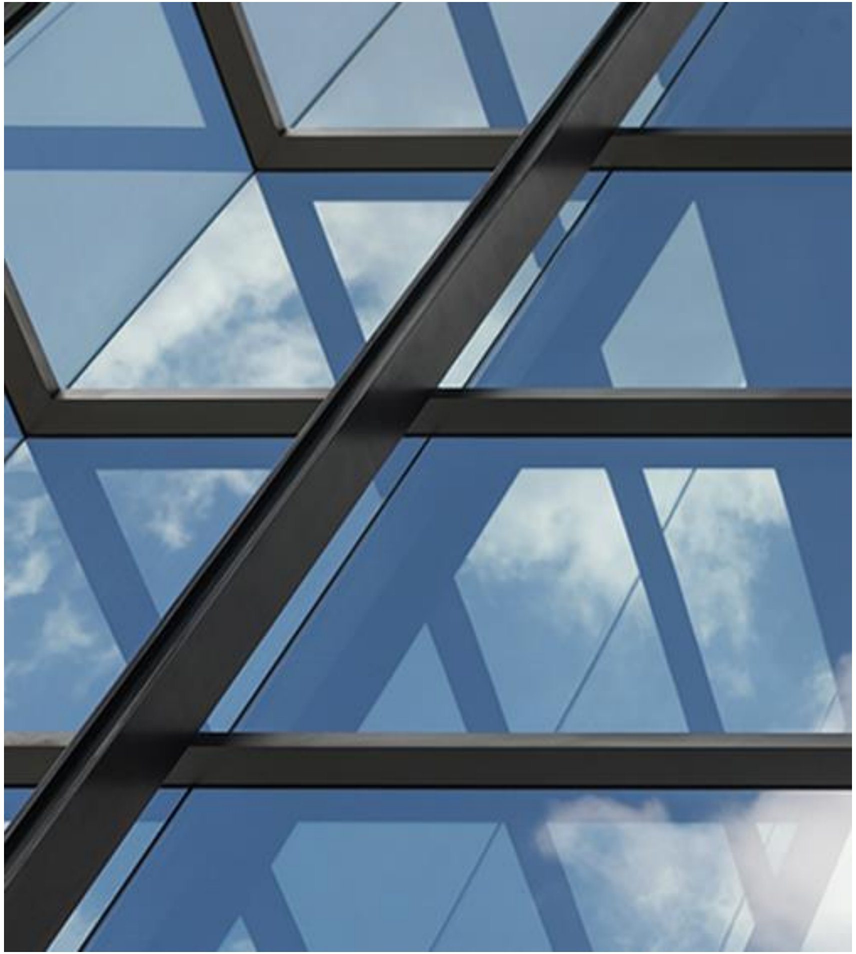
causes a test-running JVM to close and be replaced by a brand new one after the specified number of tests have run under an instance

Junit 5 Sub-Projects

- **Junit Platform:** foundation for launching testing frameworks on the JVM
  - It defines the TestEngine API for developing a testing framework that runs on the platform
  - Console Launcher to launch the platform from the command line and build plugins for Gradle and Maven for running any TestingEngine on the platform
- **Junit Jupiter:** combination of the new program model and extension model for writing tests and extensions in Junit 5
  - It provides a TestEngine for running Jupiter based tests on the platform
- **Junit Vintage:** Provides a testEngine for running Junit 3 and 4 based tests on the platform

https://junit.org/junit5/docs/current/user-guide/#writing-tests

Code Coverage



THE UNIVERSITY OF
SYDNEY

Code Coverage Tests

- Metric for testing quality

- Measure how thoroughly a set of tests deals with the code
  - Note: not a measure for a single test

- Quantify statements in source code executed by test
  - Assumption: high code coverage
    - lower chance of having undetected software bugs

- Safety-critical applications require 100% coverage

- Note programs may have infeasible paths
  - 100% coverage normally not achievable

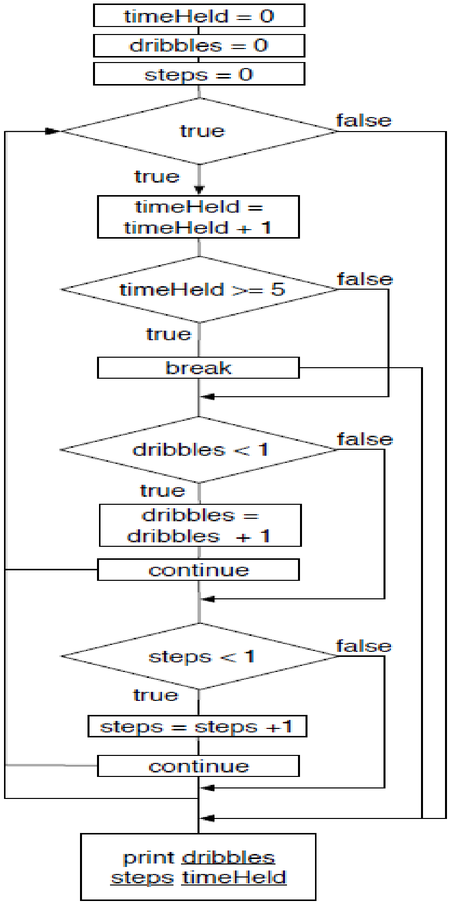
Code Coverage - Example

1. Study the “Basketball Rules” workflow

2. Identify possible code coverage levels

3. How easy to test this workflow?

4. What is good coverage for this workflow?

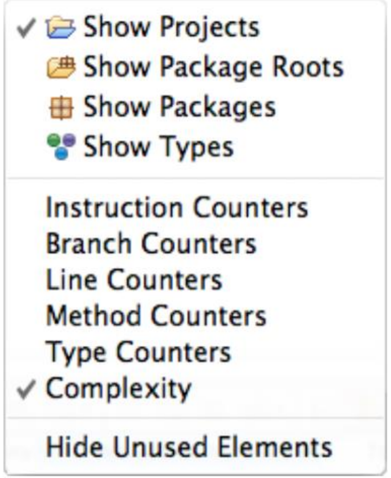


Code Coverage Types / Levels

Coverage Type / Level	Criteria
Method coverage	How many of the methods are called, during the tests
Call coverage	How many of the potential method calls are executed, during the tests
Statement coverage	How many lines are exercised, during the tests
Branch coverage	How many of the possible outcomes of the branches have occurred during the tests
Condition (Decision, Predicate) coverage	Has each separate condition within each branch been both true and false
Loop coverage	Each loop executed zero times, once and more than once
Data flow coverage	Each variable definition and its usage executed

Tools for Code Coverage in Java

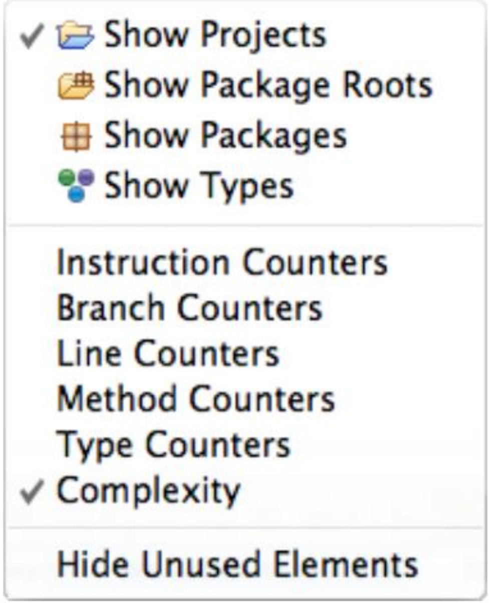
- There are many tools/plug-ins for code coverage in Java
- Example: EclEmma*
- EclEmma is a code coverage plug-in for Eclipse
  - It provides rich features for code coverage analysis in Eclipse IDE
  - EclEmma is based on the *JaCoCo* code coverage library
    - *JaCoCo* is a free code coverage library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



https://www.eclemma.org/

EclEmma – Counters

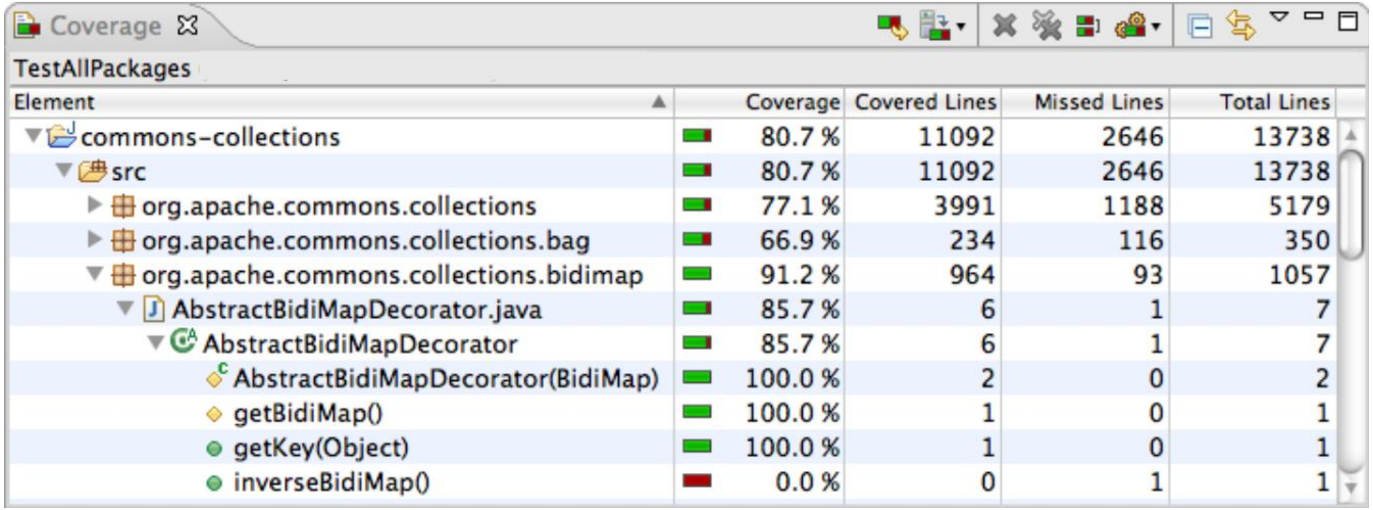
- EclEmma supports different types of counters to be summarized in code coverage overview
  - bytecode instructions, branches, lines, methods, types and cyclomatic complexity
  - Should understand each counter and how it is measured
  - Counters are based on JaCoCon - see [JaCoCo documentation]() for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



Coverage

TestAllPackages

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/

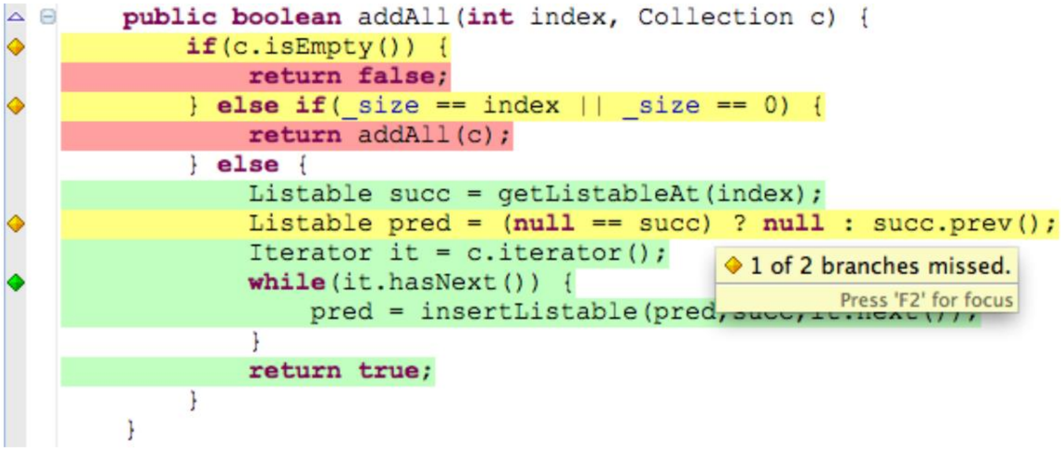
EclEmma Coverage – Source Code Annotations

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.



```
public boolean addAll(int index, Collection c) {
    if(c.isEmpty()) {
        return false;
    } else if(_size == index || _size == 0) {
        return addAll(c);
    } else {
        Listable succ = getListableAt(index);
        Listable pred = (null == succ) ? null : succ.prev();
        Iterator it = c.iterator();
        while(it.hasNext()) {
            pred = insertListable(pred,succ,it.next());
        }
        return true;
    }
}
```

◆ 1 of 2 branches missed.

Press 'F2' for focus

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