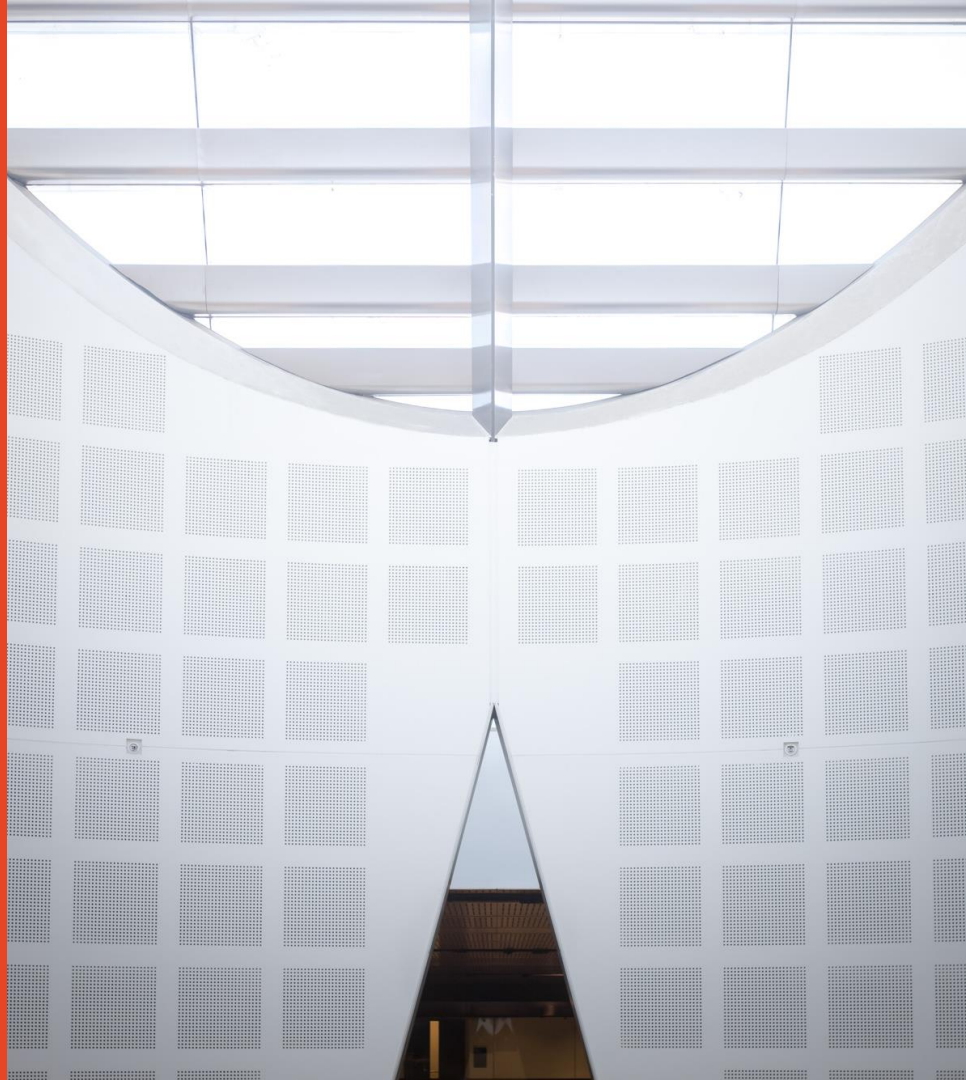
Software Design and Construction 2
SOFT3202 / COMP9202
Software Testing Theory, Design of Test

Dr. Basem Suleiman

School of Information Technologies

THE UNIVERSITY OF SYDNEY



Copyright Warning

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**).

The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

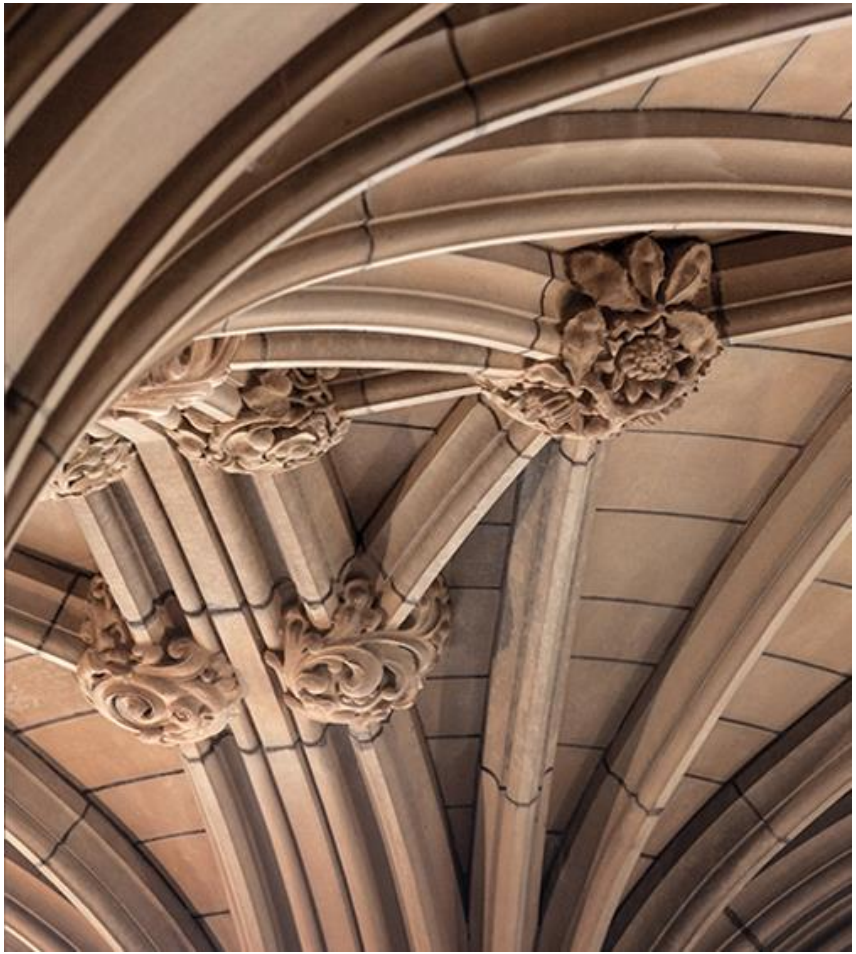
Do not remove this notice.

Agenda

- Theory of Testing
- Design of Tests
- Unit Testing
- Testable Code

Software Testing

Revisit – Theory behind



THE UNIVERSITY OF SYDNEY

Software Testing

- Software process to
  - Demonstrate that software meets its requirements (*validation testing*)
  - Find incorrect or undesired behavior caused by defects/bugs (defect testing)
    - E.g., System crashes, incorrect computations, unnecessary interactions and data corruptions
- Part of software verification and validation (V&V) process

Testing Objectives – Discuss

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

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

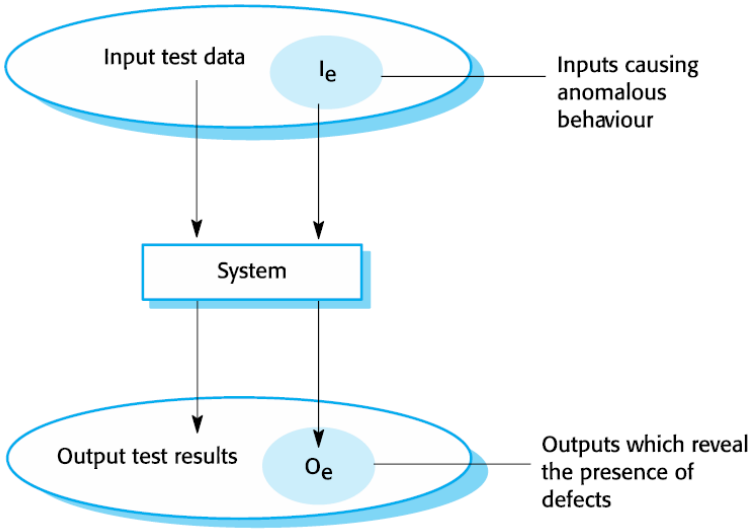
- Objectives should be stated precisely and quantitatively to measure and control the test process

- 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

Validation Testing vs. Defect Testing

- Testing modelled as input test data and output test results
- Defect testing: find I_e that cause anomalous behavior (defects/problems)
- Validation testing: find inputs that lead to expected correct outcomes



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

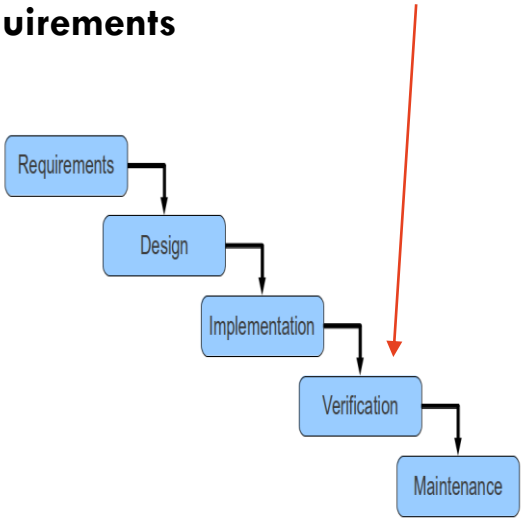
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques

- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should 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?

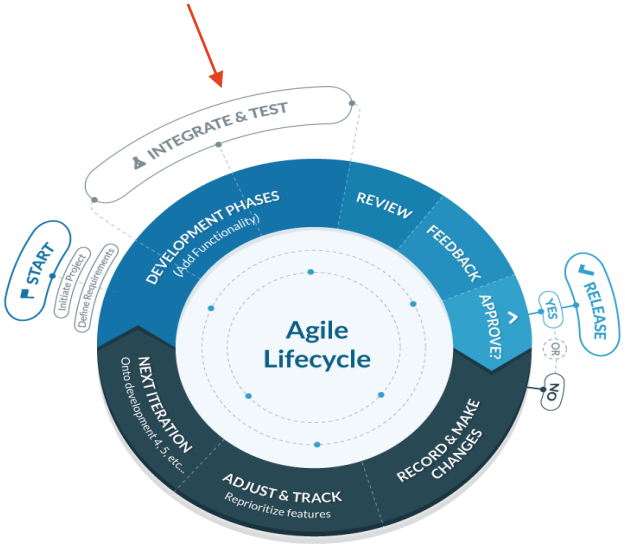
Waterfall Software Development

- **Test whether system works according to requirements**



Agile Software Development

- Testing is at the heart of agile practices
- 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

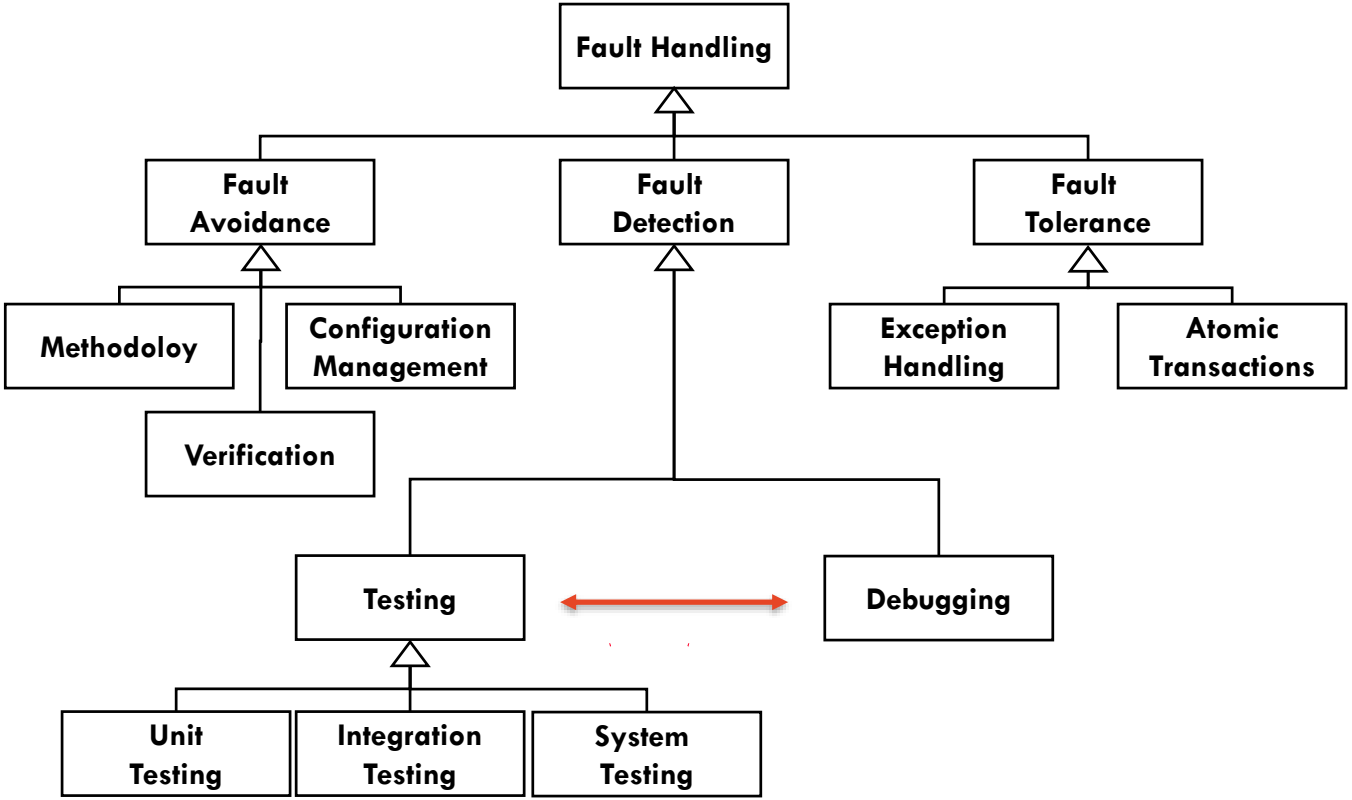
Testing Terminology

- **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 Errors 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

Fault Handling Techniques



Software Testing Classification

Functional Testing

- Unit testing
- Integration testing
- System testing
- Regression testing
- Interface testing
- User Acceptance Testing (UAT) – Alpha and Beta testing
- Configuration, smoke, sanity, end-to-end testing

Non-Functional Testing

- Performance testing
- Load testing
- Security testing
- Stress testing
- Reliability testing
- Usability testing

Testing Objectives

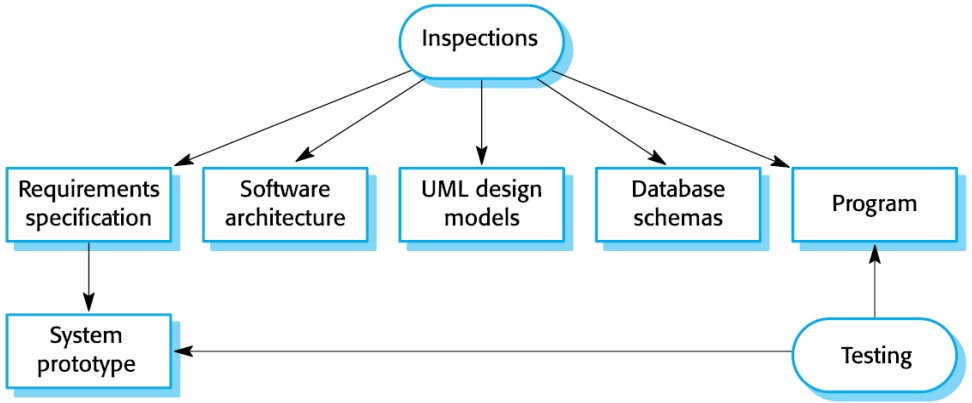
Testing Type	Objective
Alpha / Beta testing	Identify possible issues (bugs) before releasing the product to end users
Regression testing	Verify that a software behaviour has not changed by incremental changes to the software
Performance testing	Verify system's performance characteristics (e.g., speed)
Security testing	Verify confidentiality and integrity of the system and its data
Stress testing	Analyse the system's behavioural limits under max. possible load
Interface testing	Verify behaviour of software interfaces of interacting components to ensure correct exchange of data and control information
Usability (HCI)	Evaluate how easy to learn and use the software by end users
Configuration	Verify the software behaviour under different user configurations

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

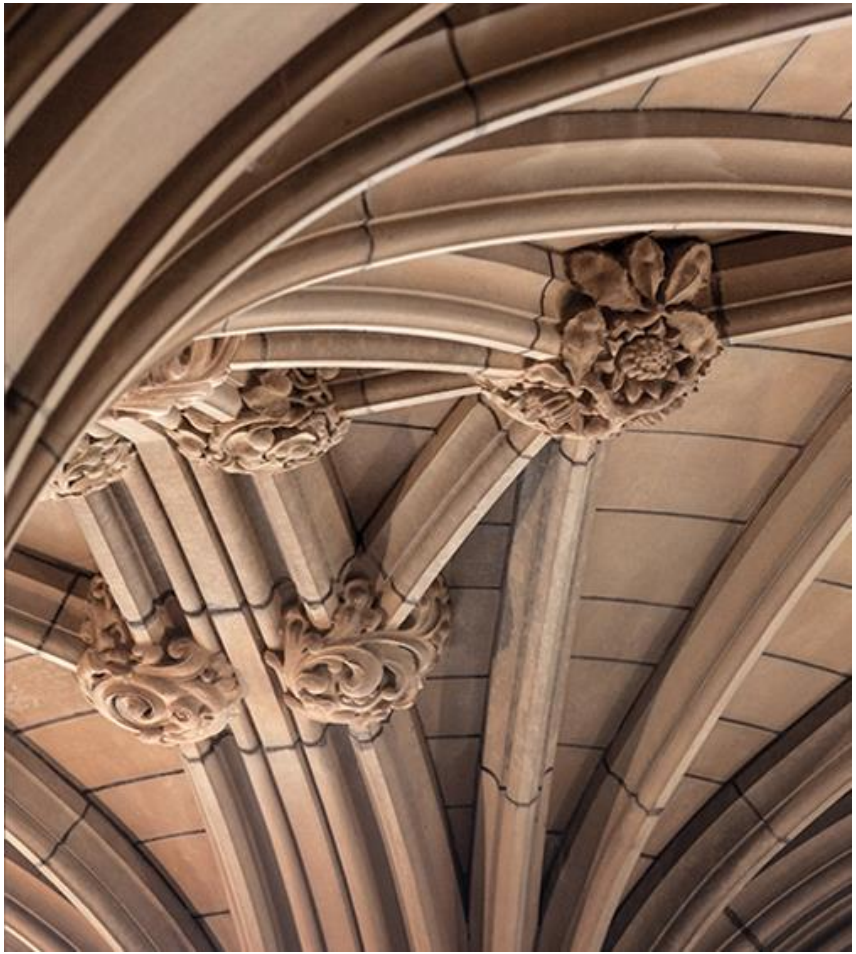
Static Verification

- Static Verification/testing
  - Static system analysis to discover problems
  - May be applied to requirements, design/models, configuration and test data
- Reviews
  - Walk through
  - Code inspection



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

Software Validation and Verification



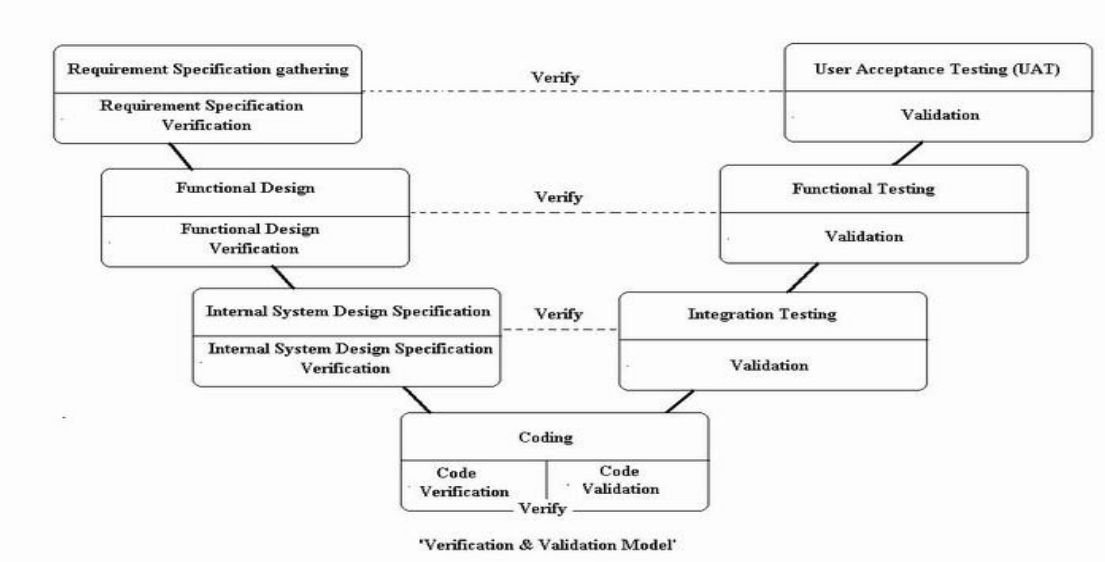
THE UNIVERSITY OF SYDNEY

Software Verification and Validation

- Software testing is part of software Verification and Validation (V&V)
- The goal of V&V is to establish confidence that the software is "*fit for purpose*"
- Software Validation
  - Are we building the right product?
  - Ensures that the software meets customer expectations
- Software Verification
  - Are we building the product right?
  - Ensures that the software meets its stated functional and non-functional requirements

V-Model

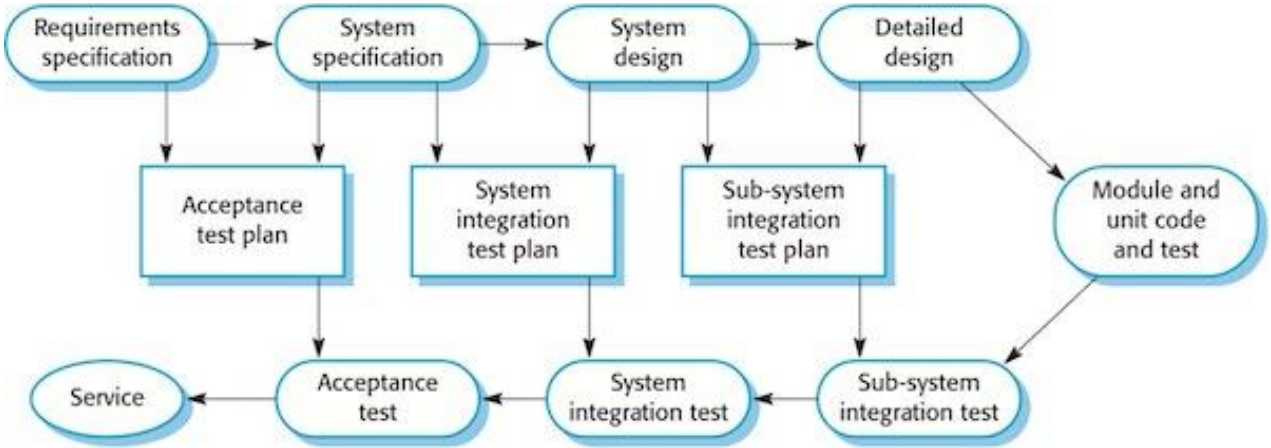
- Link each phase of the SDLC with its associated testing phase
- Each verification stage relates to a validation stage



https://www.buzzle.com/editorials/4-5-2005-68117.asp

V-Model

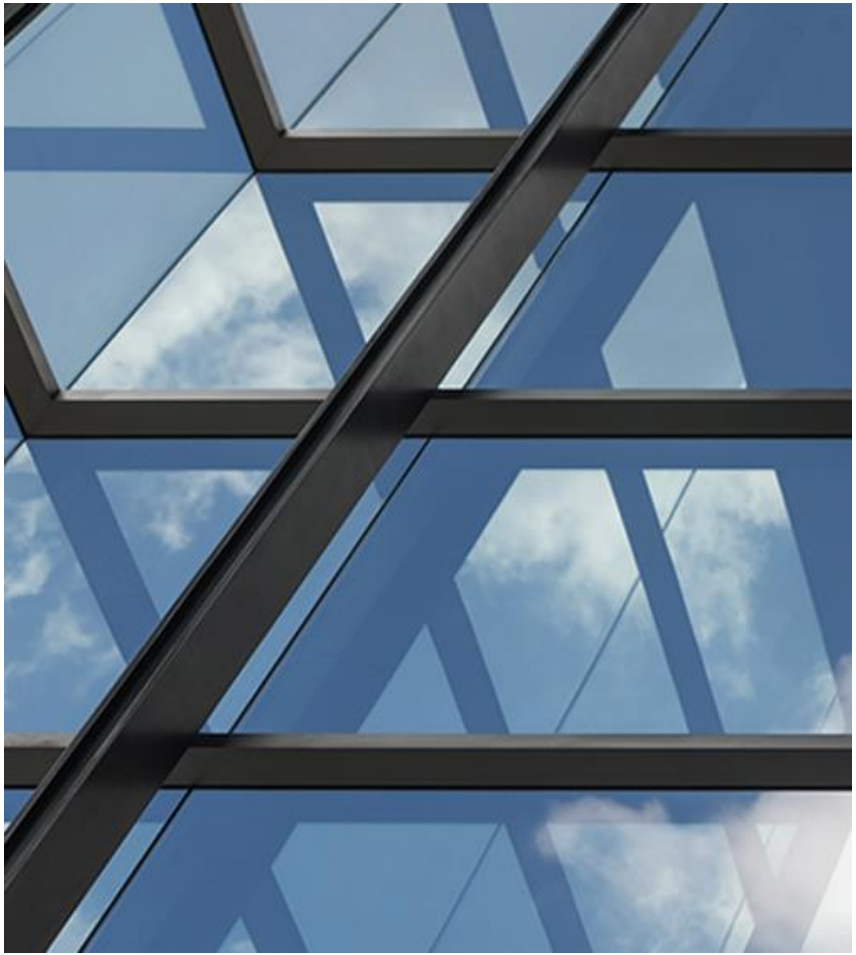
- Link each phase of the SDLC with its associated testing phase
- Each verification stage relates to a validation stage



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

Choosing Test Cases



THE UNIVERSITY OF SYDNEY

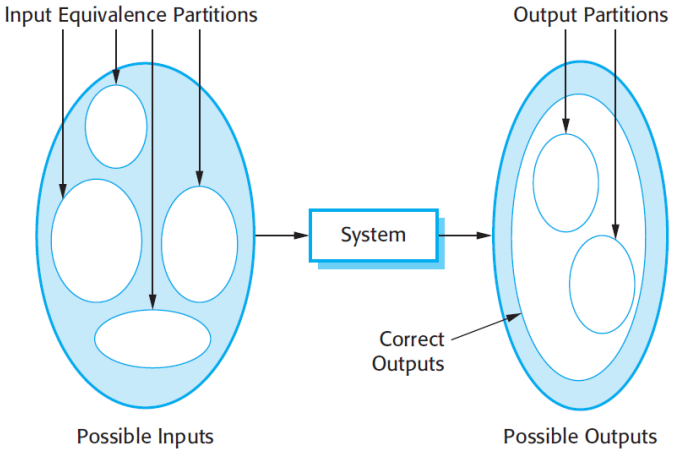
Choosing Test Cases – Techniques

- Partition testing (equivalence partitioning)
  - Identify groups of inputs that have common characteristics
  - From within each of these groups, choose tests

- Guideline-based testing
  - Use testing guidelines based on previous experience of the kinds of errors often made

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
  - Developers tend to select normal/typical cases



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 – Sample Solution

- Equivalent classes for the 'month' parameter
  - 31-days months, 30-days months, and 28-or-29-days month
  - non-positive and positive integers larger than 12
- Equivalence classes for the 'year' parameter
  - Leap years and no-leap years
  - negative integers
- Select valid value for each equivalence class
  - E.g., February, June, July, 1901 and 1904
- Combine values to test for interaction (method depends on both parameters)
  - Six equivalence classes

Choosing Test Cases – Sample Solution

- Equivalence classes and selected valid inputs for testing the *getNumDaysInMonth()* method

Equivalence Class	Value for month Input	Value for year Input
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

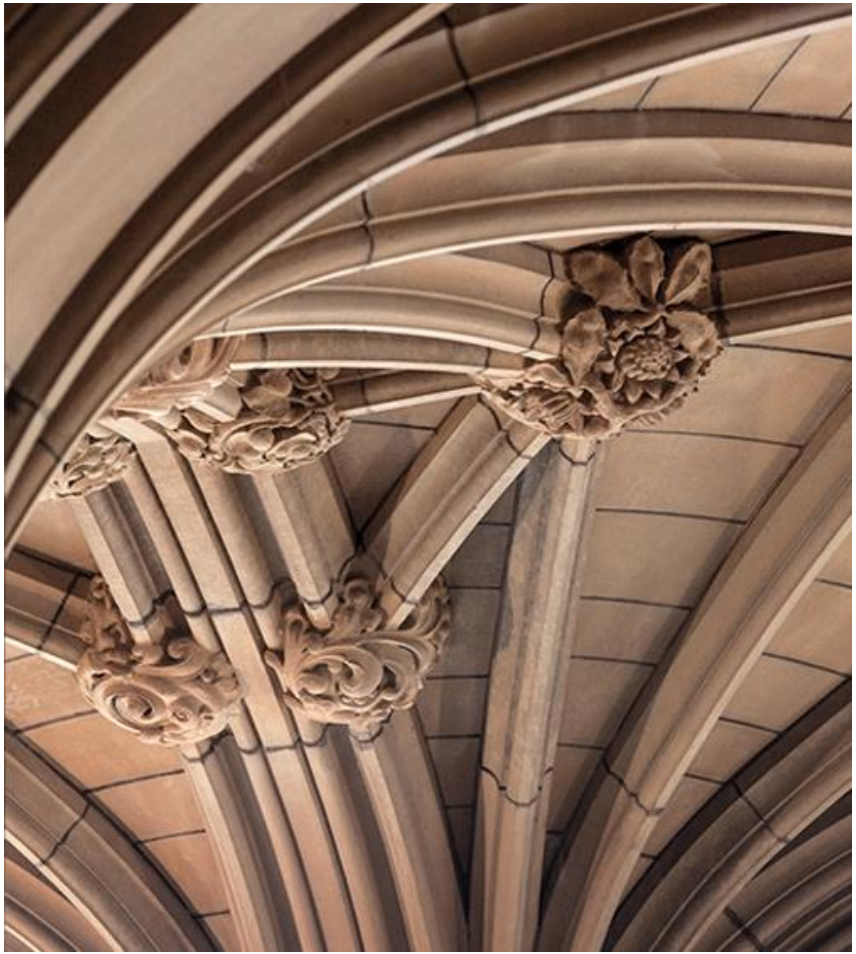
Choosing Test Cases – Sample Solution

- Additional boundary cases identified for the *getNumOfDaysInMonth()* method

Equivalence Class	Value for month Input	Value for year Input
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

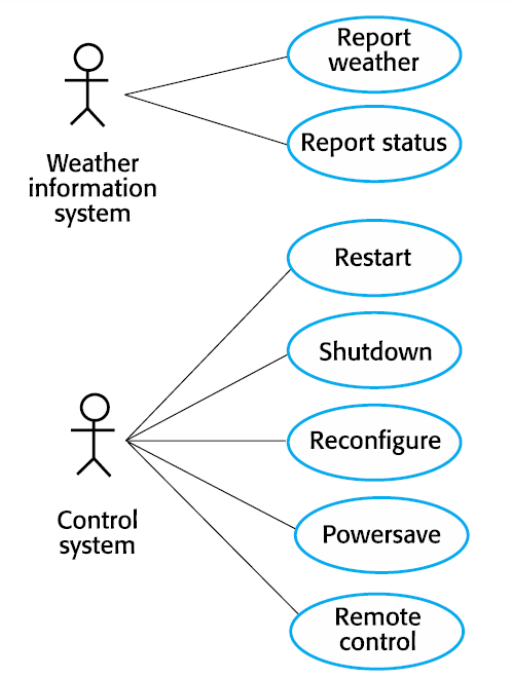
Unit Testing

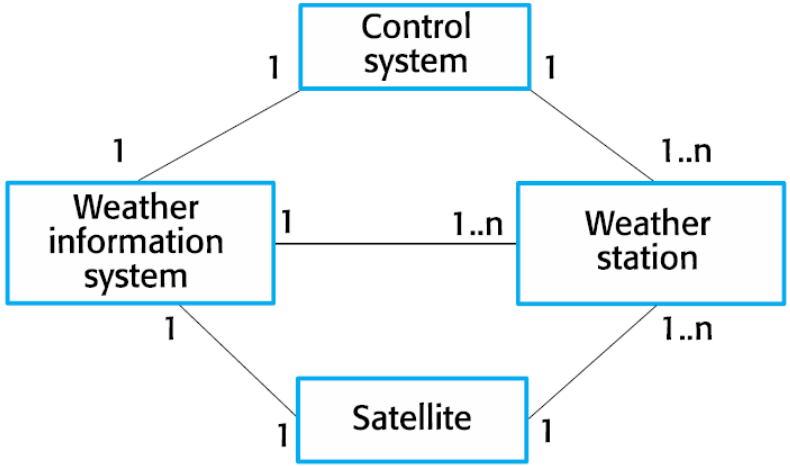
Part 2



THE UNIVERSITY OF SYDNEY

Unit Testing – Case Study

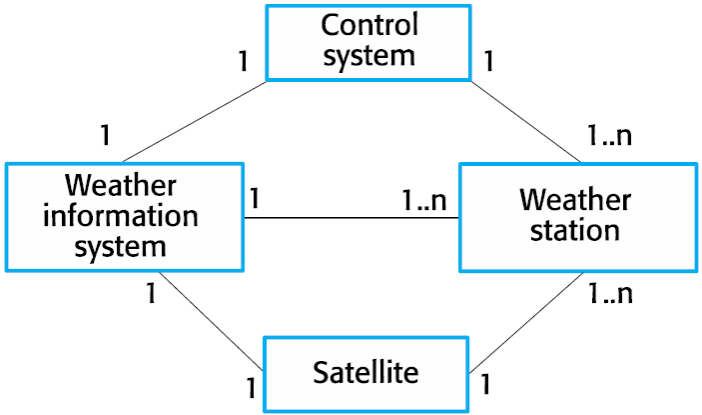


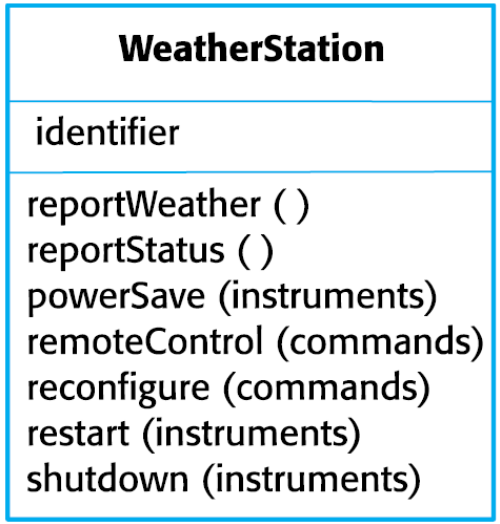


Unit Testing – Case Study

Exercise:

In groups, define various unit tests for the Weather Station object



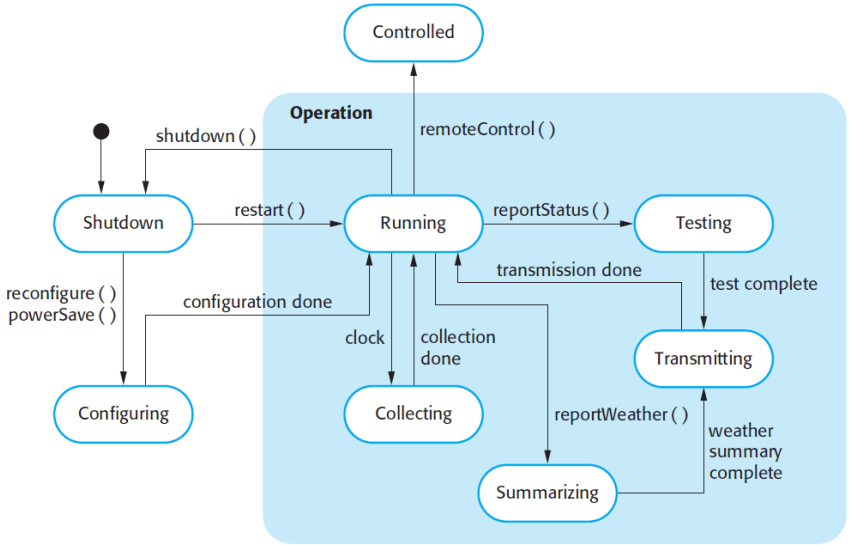


Unit Testing – Techniques

- Attributes
  - identifier: check if it has been set up properly
- Methods
  - Perform its functionality correctly
  - Input and output of each method
  - Not always possible to test in isolation, test sequence is necessary
    - Testing *shutdown(instruments)* require executing *restart(instruments)*
- Use system specification and other documentation
  - Requirements, system design artefacts (use case description, sequence diagrams, state diagram, etc.)

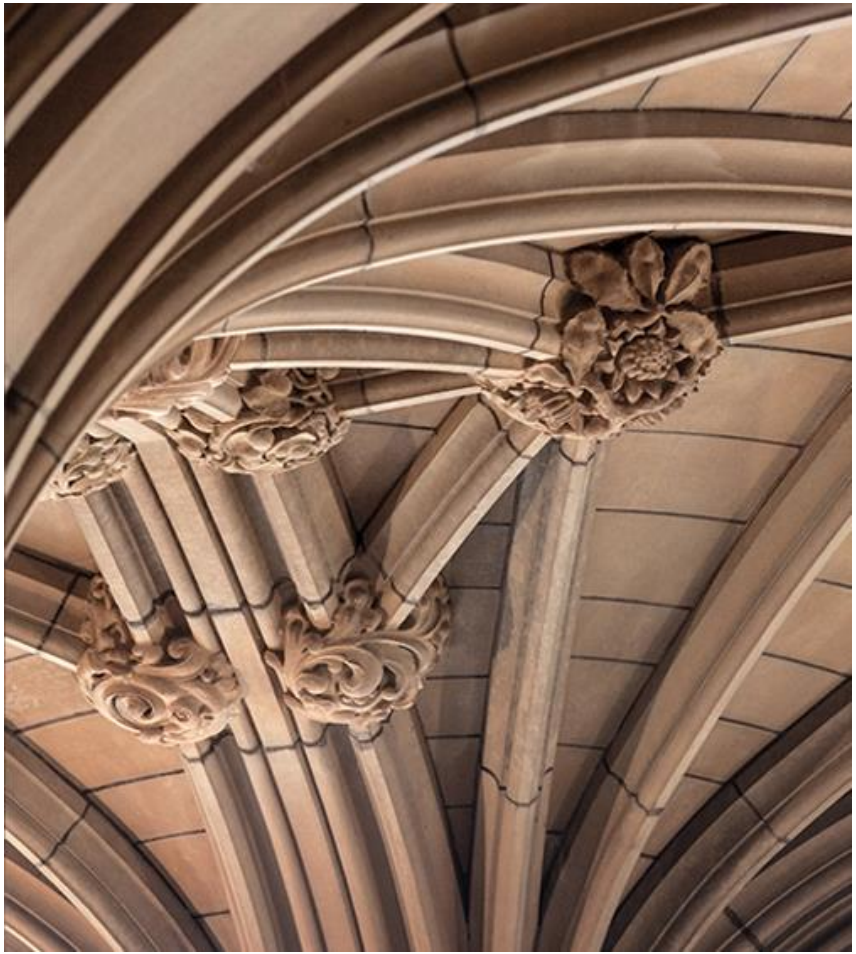
Unit Testing – State Sequences

- Testing states of *WeatherStation* using state model
  - Identify sequences of state transitions to be tested
  - Define event sequences to force these transitions
- Examples:
  - Shutdown → Running → Shutdown
  - Configuration → Running → Testing → Transmitting → Running
  - Many others…



JUnit

Part 2



THE UNIVERSITY OF SYDNEY

Junit – Annotations and Test Fixtures

- Four fixture annotations; class-level and method-level
- Time of execution is important to use it properly

JUnit 4*	Description
@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. 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

Test Fixture – Exercise

- Examine the code snippet (line 11-54) and write down the output that will be produced after executing the code.

```
11 // All required imports ommited for simplicity
12 public class TestFixturesExample {
13   static class ExpensiveManagedResource implements Closeable {
14     @Override
15     public void close() throws IOException {}
16   }
17   static class ManagedResource implements Closeable {
18     @Override
19     public void close() throws IOException {}
20   }
21   @BeforeClass
22   public static void setUpClass() {
23     System.out.println("@BeforeClass setUpClass");
24     myExpensiveManagedResource = new ExpensiveManagedResource();
25   }
26   @AfterClass
27   public static void tearDownClass() throws IOException {
28     System.out.println("@AfterClass tearDownClass");
29     myExpensiveManagedResource.close();
30     myExpensiveManagedResource = null;
31   }
32   private ManagedResource myManagedResource;
33   private static ExpensiveManagedResource
34                  myExpensiveManagedResource;
35
36   private void println(String string) {
37     System.out.println(string);
38   }
39   @Before
40   public void setUp() {
41     this.println("@Before setUp");
42     this.myManagedResource = new ManagedResource();
43   }
44   @After
45   public void tearDown() throws IOException {
46     this.println("@After tearDown");
47     this.myManagedResource.close();
48     this.myManagedResource = null;
49   }
50   @Test
51   public void test1() {
52     this.println("@Test test1()");
53   }
54 }
```

Test Fixture – Exercise

- Examine the code snippet (line 11-54) and write down the output that will be produced after executing the code.

@BeforeClass setUpClass
@Before setUp
@Test test1()
@After tearDown
@AfterClass tearDownClass

Junit – Test Execution Order

Exercise:

Examine the test code and identify the execution order of the included test methods.

```java
import org.junit.Test;
import org.junit.runners.MethodSorters;

public class TestMethodOrder {

    @Test
    public void testA() {
        System.out.println("first");
    }
    @Test
    public void testB() {
        System.out.println("second");
    }
    @Test
    public void testC() {
        System.out.println("third");
    }
}
```

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 might lead into test problems (*poor test practices*)
- By default, Junit 4.11uses a deterministic order *(MethodSorters)*
  - Java 7 (and older) return a more or less random order
- @FixMethodOrder to change test execution order (not recommended practice)
  - *@FixMethodOrder(MethodSorters.JVM)*
  - *@FixMethodOrder(MethodSorters.NAME ASCENDING)*

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

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

Parameterized Test Example

- Write a unit test that consider different parameters for the following class method *compute(int)*

```
1  public class Fibonacci {
2      public static int compute(int n) {
3          int result = 0;
4
5          if (n <= 1) {
6              result = n;
7          } else {
8              result = compute(n - 1) + compute(n - 2);
9          }
10
11         return result;
12     }
13 }
```

Junit – Parameterized Test Example

```
 7 import org.junit.runner.RunWith;
 8 import org.junit.runners.Parameterized;
 9 import org.junit.runners.Parameterized.Parameter;
10 import org.junit.runners.Parameterized.Parameters;
11
12 @RunWith(Parameterized.class)
13 public class FibonacciTest {
14     @Parameters
15     public static Collection<Object[]> data() {
16         return Arrays.asList(new Object[][] {
17                 { 0, 0 }, { 1, 1 }, { 2, 1 }, { 3, 2 }, { 4, 3 }, { 5, 5 }, { 6, 8 }
18            });
19     }
20
21     @Parameter // first data value (0) is default
22     public /* NOT private */ int fInput;
23
24     @Parameter(1)
25     public /* NOT private */ int fExpected;
26
27     @Test
28     public void test() {
29         assertEquals(fExpected, Fibonacci.compute(fInput));
30     }
31 }
```

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 }
```

Junit – Verify Tests Timeout Behaviour

- 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
  - Test method runs in separate thread
  - *Optionally specify 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	Lets 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 – Timeout Rule

- Applies the same timeout rule to all test methods in a class including @Before and @After

```java
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.Timeout;

public class HasGlobalTimeout {
    public static String log;
    private final CountDownLatch latch = new CountDownLatch(1);

    @Rule
    // 10 seconds max per method tested
    public Timeout globalTimeout = Timeout.seconds(10);

    @Test
    public void testSleepForTooLong() throws Exception {
        log += "ran1";
        TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds
    }

    @Test
    public void testBlockForever() throws Exception {
        log += "ran2";
        latch.await(); // will block
    }
}
```

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 – ExpectedException Rule

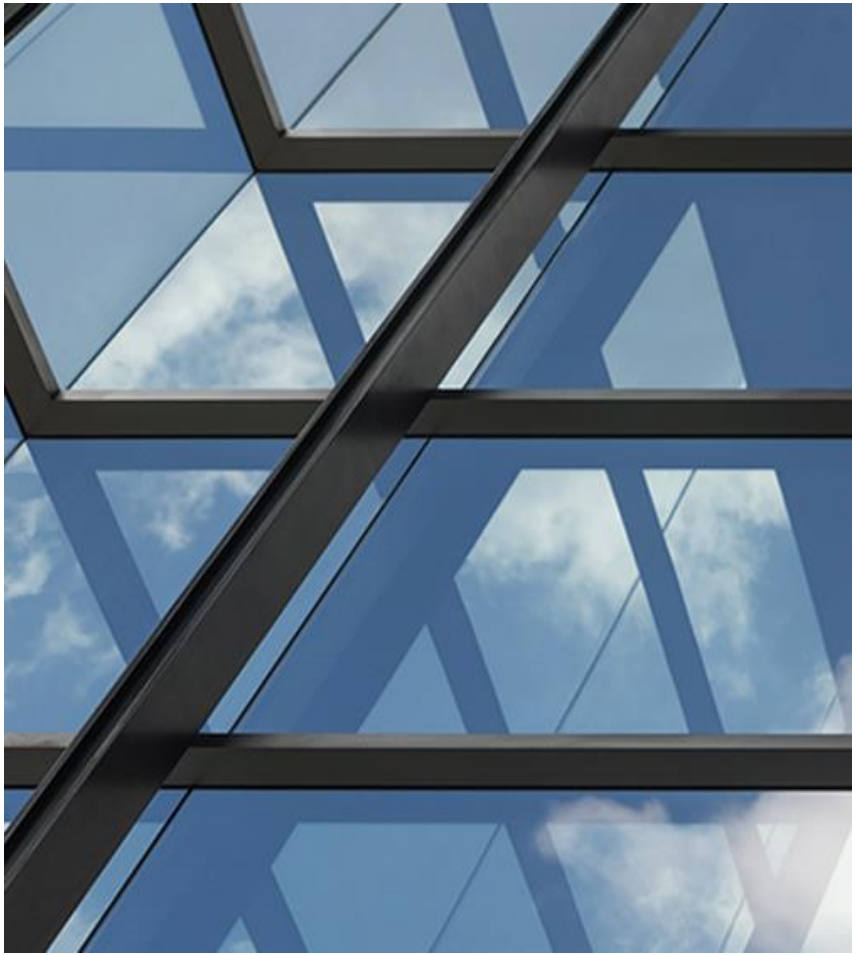
- How to test a message value in the exception or the state of a domain object after the exception has been thrown?
- *ExpectedException* rule allows specifying expected exception along with the expected exception message

```
1  @Rule
2  public ExpectedException thrown = ExpectedException.none();
3
4  @Test
5  public void shouldTestExceptionMessage() throws IndexOutOfBoundsException {
6      List<Object> list = new ArrayList<Object>();
7
8      thrown.expect(IndexOutOfBoundsException.class);
9      thrown.expectMessage("Index: 0, Size: 0");
10     list.get(0); // execution will never get past this line
11 }
```

Junit – Examples of other Rules

- Check Junit documentation for more examples on rules implementation
- Make sure you use them for the right situation – the goal is to write good tests (not Test Smell)

Testable Code



THE UNIVERSITY OF SYDNEY

Writing Testable Code

- Testable code: code that can be easily tested and maintained

- What makes code hard to test (untestable)?
  - Anti-pattern
  - Design/code smells
  - Bad coding practices
  - Others?

http://www.codeops.tech/blog/linkedin/what-causes-design-smells/

..

Testable Code

- Adhere to known design principles (e.g., SOLID)
  - Single responsibility
    - Small pieces of functionality that are easier to test in isolation
  - Open-closed
    - All existing tests should work even with un-extended implementation
  - Liskov Substitution
    - Mocked object substituted for real part without changing unexpected behavior
  - Interface Segregation
    - Reduce complexity of SUT
  - Dependency Inversion
    - Inject mock implementation of a dependency instead of real implementation

For more details See revision slides on Canvas

Testable Code

- Adhere to known design principles (GRASP)
  - Creator
  - Information Expert
  - High Cohesion
  - Low Coupling
  - Controller

- For more details See revision slides on Canvas

Testable Code

- Adhere to API design principles
  - Keep It Simple Stupid (KISS)
  - You Aren't Gonna Need It (YAGNI)
  - Don't Repeat Yourself
  - Occam's Razor

For more details See revision slides on Canvas

Testable Code

- Adhere to other OO design principles
  - Information hiding
  - Encapsulation
  - Documentation
  - Naming convention
  - Parameters selection
  - Others …

- For more details See revision slides on Canvas

Testable Code – Industry/Expert Guide

- Google's guide for 'Writing Testable Code'
  - Guide for Google's Software Engineers

- Understanding different types of flaws, fixing it, concrete code examples before and after
  - Constructor does real work
  - Digging into collaborators
  - Brittle global state & Singletons
  - Class does too much

http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf
http://misko.hevery.com/code-reviewers-guide/

References

- Ian Sommerville. 2016. Software Engineering (10th ed.) Global Edition. Pearson, Essex England
- Bernd Bruegge and Allen H. Dutoit. 2009. Object-Oriented Software Engineering Using Uml, Patterns, and Java (3rd ed.). Pearson.
- Junit 4, Project Documentation, [https://junit.org/junit4/]
- Jonathan Wolter, Russ Ruffer, Miško Hevery Google Guide to Writing Testable Code [http://misko.hevery.com/attachments/Guide-Writing%20Testable%20Code.pdf]

Next Lecture/Tutorial…

W3 Tutorial: More on unit Testing +
W3 quiz
W3 Lecture: Advanced Testing Techniques
Testing Assignment A1 release

THE UNIVERSITY OF SYDNEY