

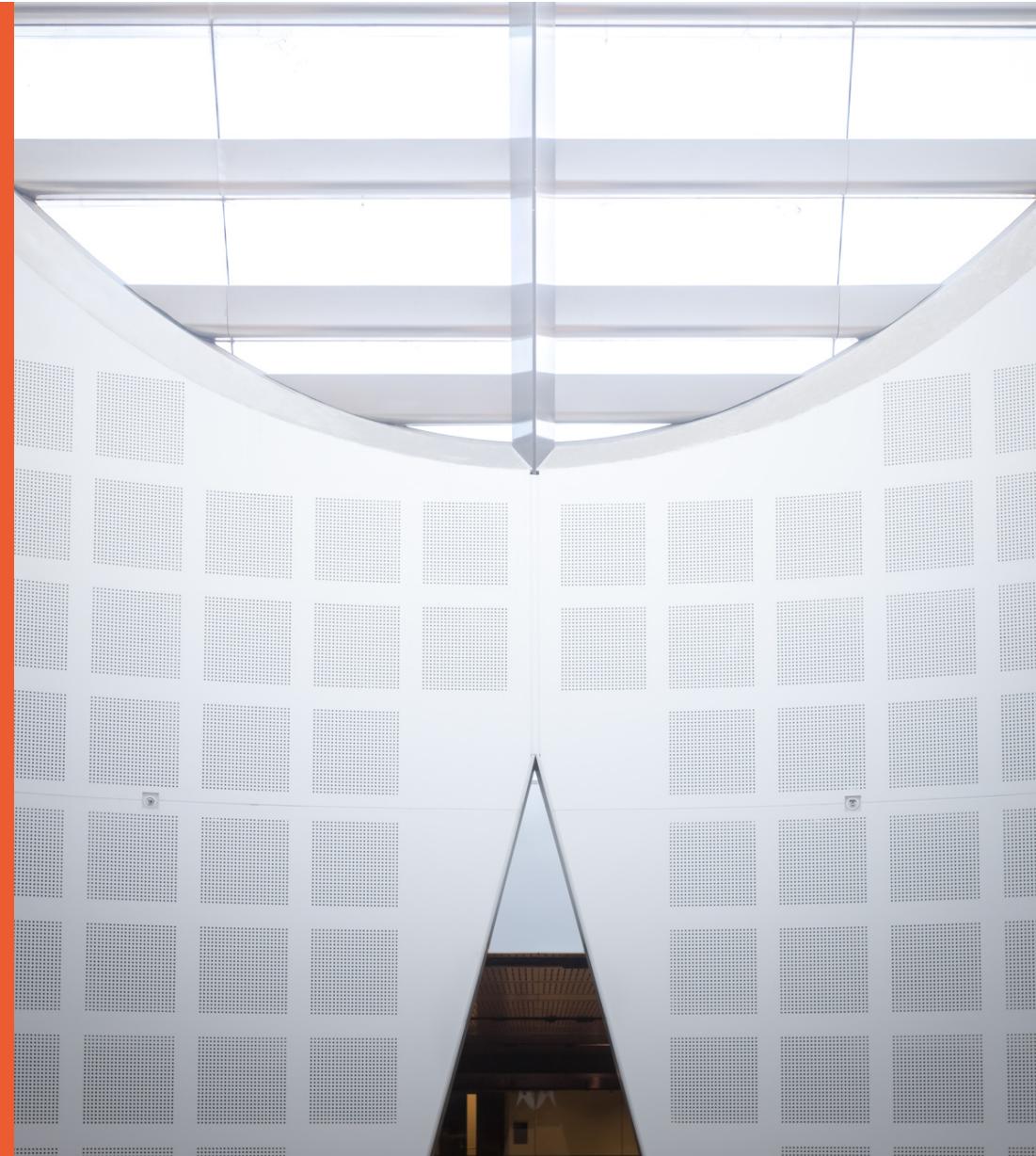
Software Design and Construction 2

SOFT3202 / COMP9202

Software Testing Theory

Prof Bernhard Scholz

School of Computer Science

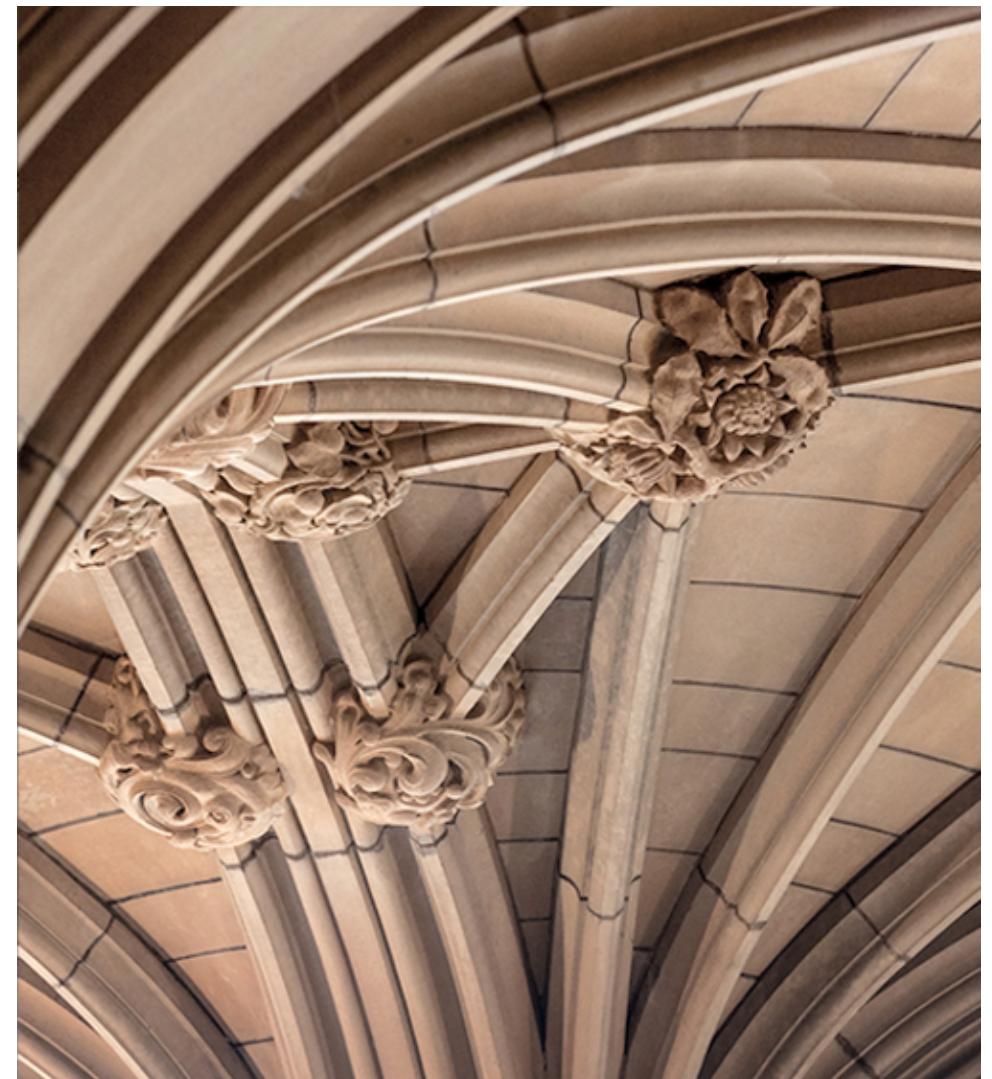


Agenda

- Theory of Testing
- Design of Tests
- Test Coverage

Software Testing

Revisit – Theory behind



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



“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



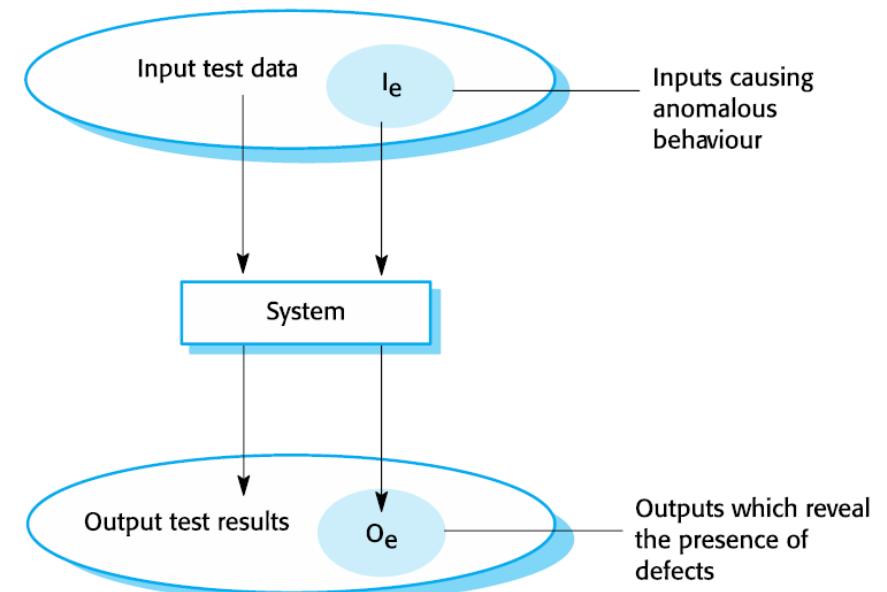
Testing Objectives

- State precisely and quantitatively for measuring and controlling test processes
- Test completeness is infeasible
 - Impossibility of exhaustive testing due to state space.
 - 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 output O_e (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
 - Testers often have different personality type from coders
 - Tester might be more productive finding software anomalies
- Real users, doing real work
 - Especially acceptance tests



Testing takes creativity

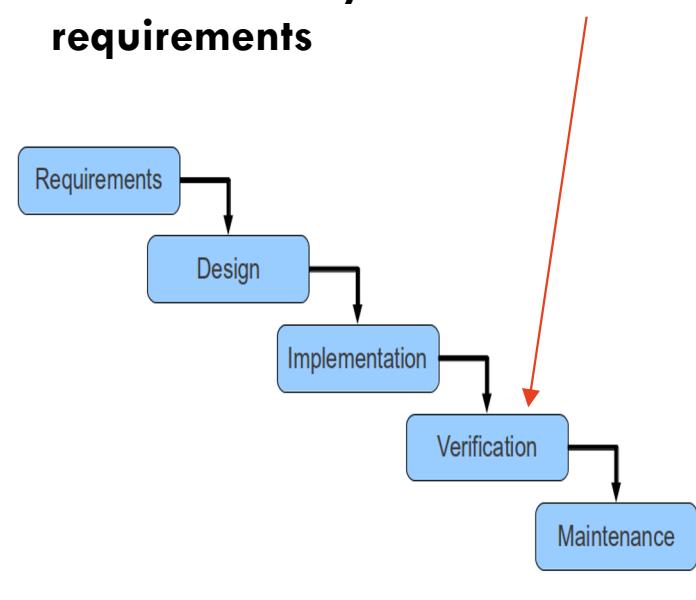
- To develop an effective test, one must have:
 - Detailed understanding of the system / understand the context of the software
 - Application and solution domain knowledge
 - Knowledge of the testing techniques
 - Skill to apply these techniques
 - Understand code
- Testing by independent testers
 - Programmers have a different mindset; think out-of-the-box for tests
 - Programmers often concentrate on standard use-cases; not on corner cases.
 - Testers reveal program behavior by using a different set of use-case.



When is Testing happening?

Waterfall Software Development

- Test whether system works according to requirements

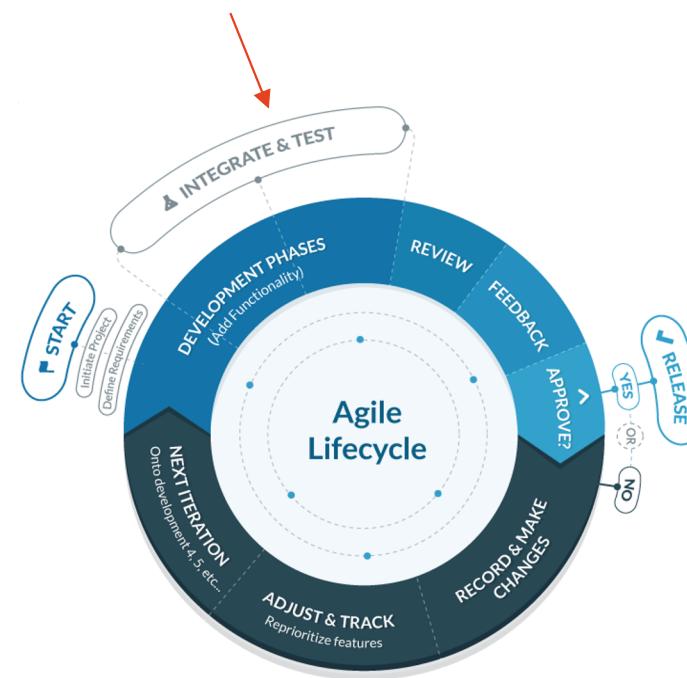


<https://www.spritecloud.com/wp-content/uploads/2011/06/waterfall.png>

<https://blog.capterra.com/wp-content/uploads/2016/01/agile-methodology-720x617.png>

Agile Software Development

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



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

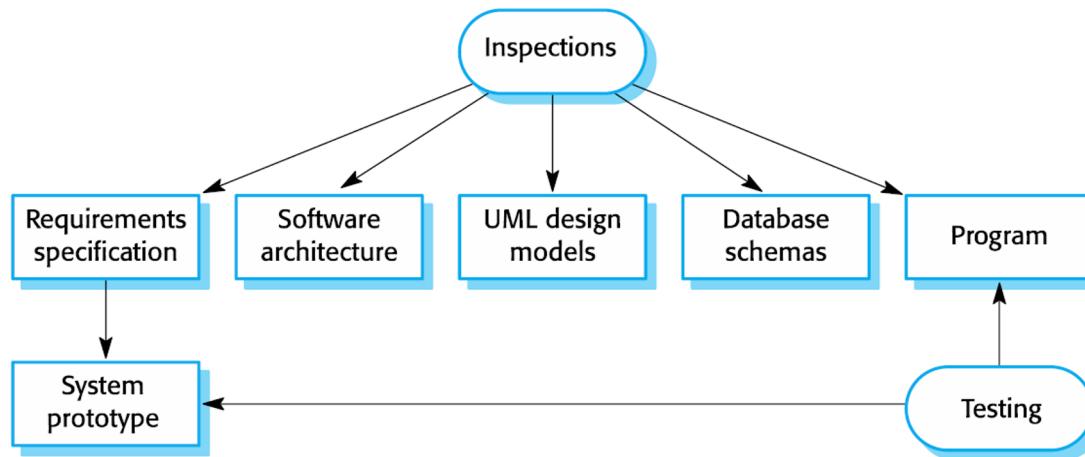
- Performance testing
- Load testing
- Security testing
- Stress testing
- Reliability testing
- Usability testing

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 Testing

- 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

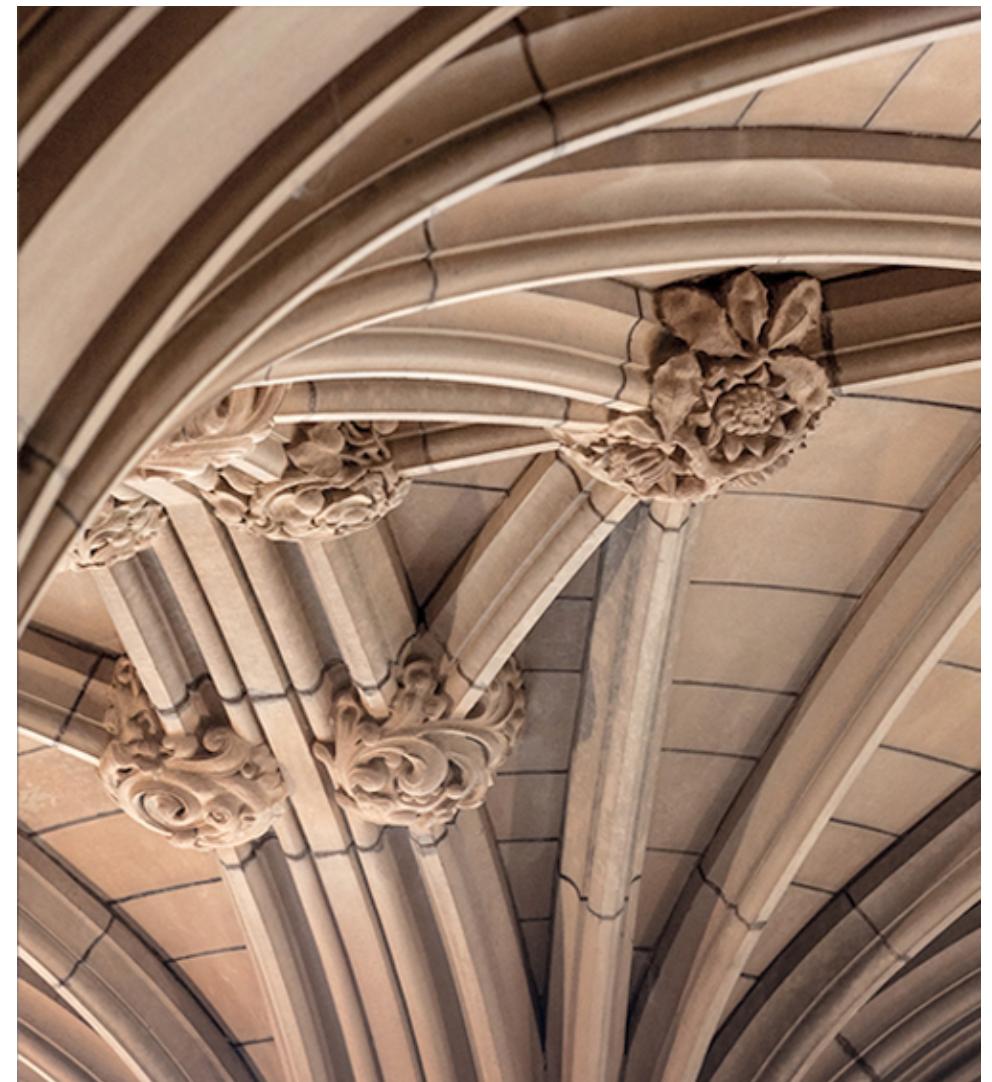


Ian Sommerville. 2016. *Software Engineering* (10th ed.). Addison-Wesley, USA.

Software Validation and Verification



THE UNIVERSITY OF
SYDNEY



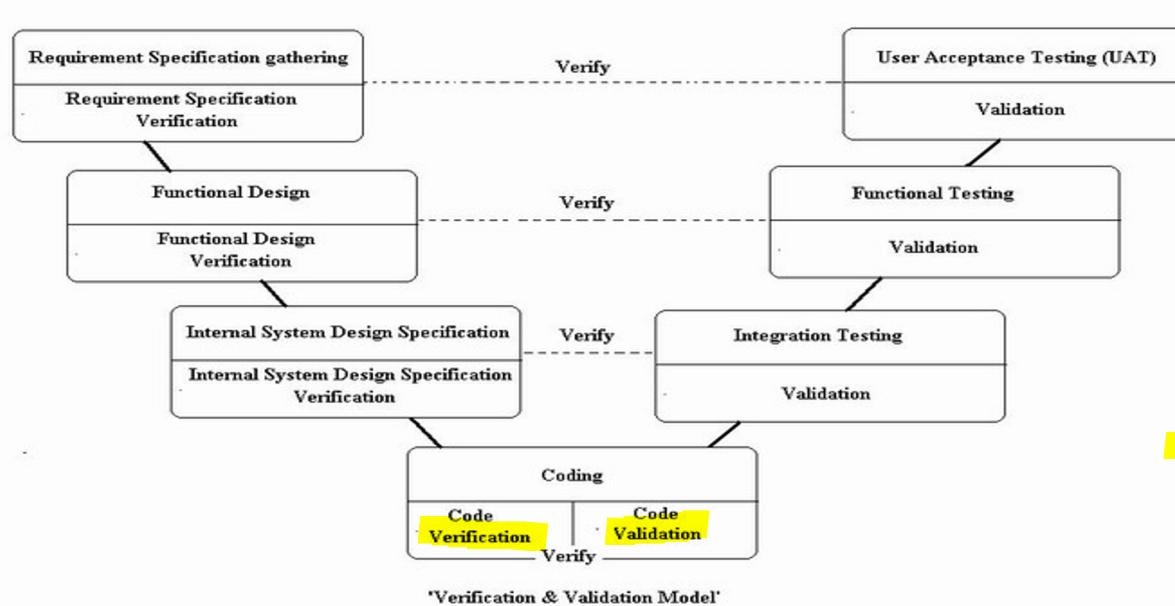
Software Verification and Validation

- 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 correct product?**
 - Process of checking whether the specification captures the customer’s needs
 - ***Did I build what I said I would?***
- Software **Verification**
 - Are we **correctly building the product?**
 - Process of checking that the software meets the specification
 - ***Did I build what I need?***



V-Model

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

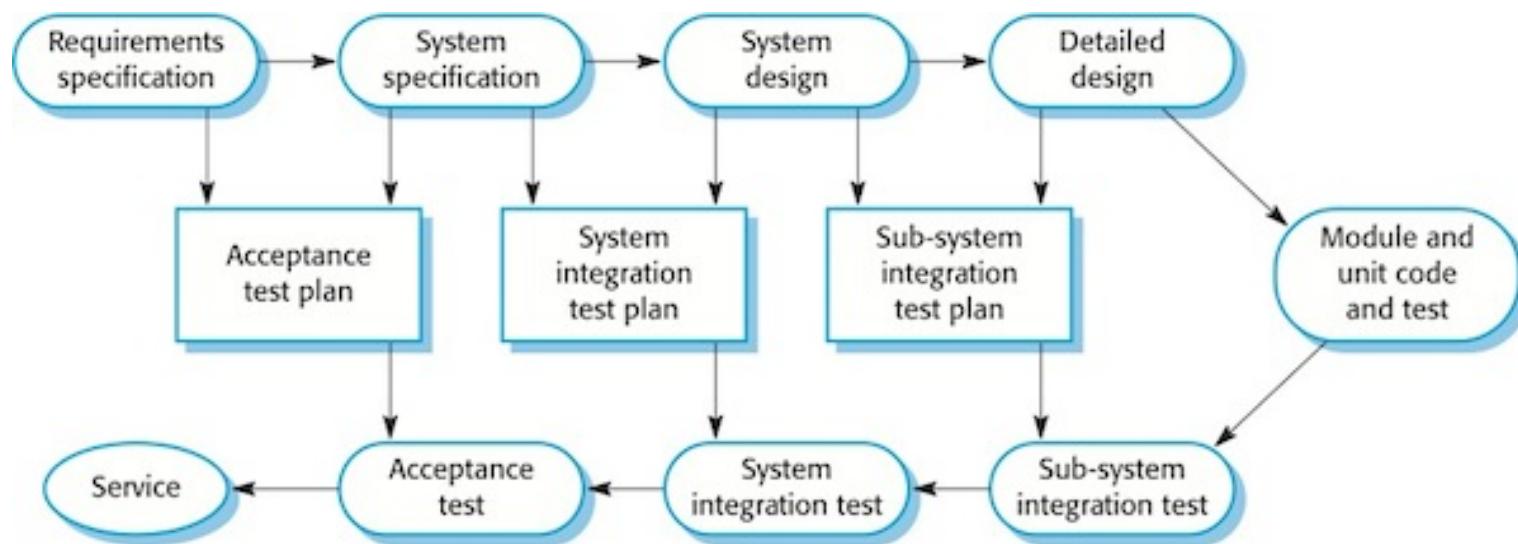


<https://www.buzzle.com/editorials/4-5-2005-68117.asp>

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



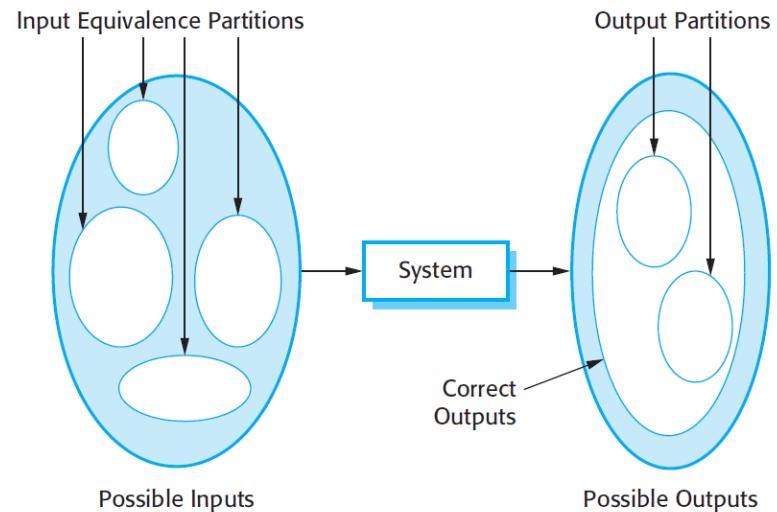
Choosing Test Cases – Techniques



- **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 }
```

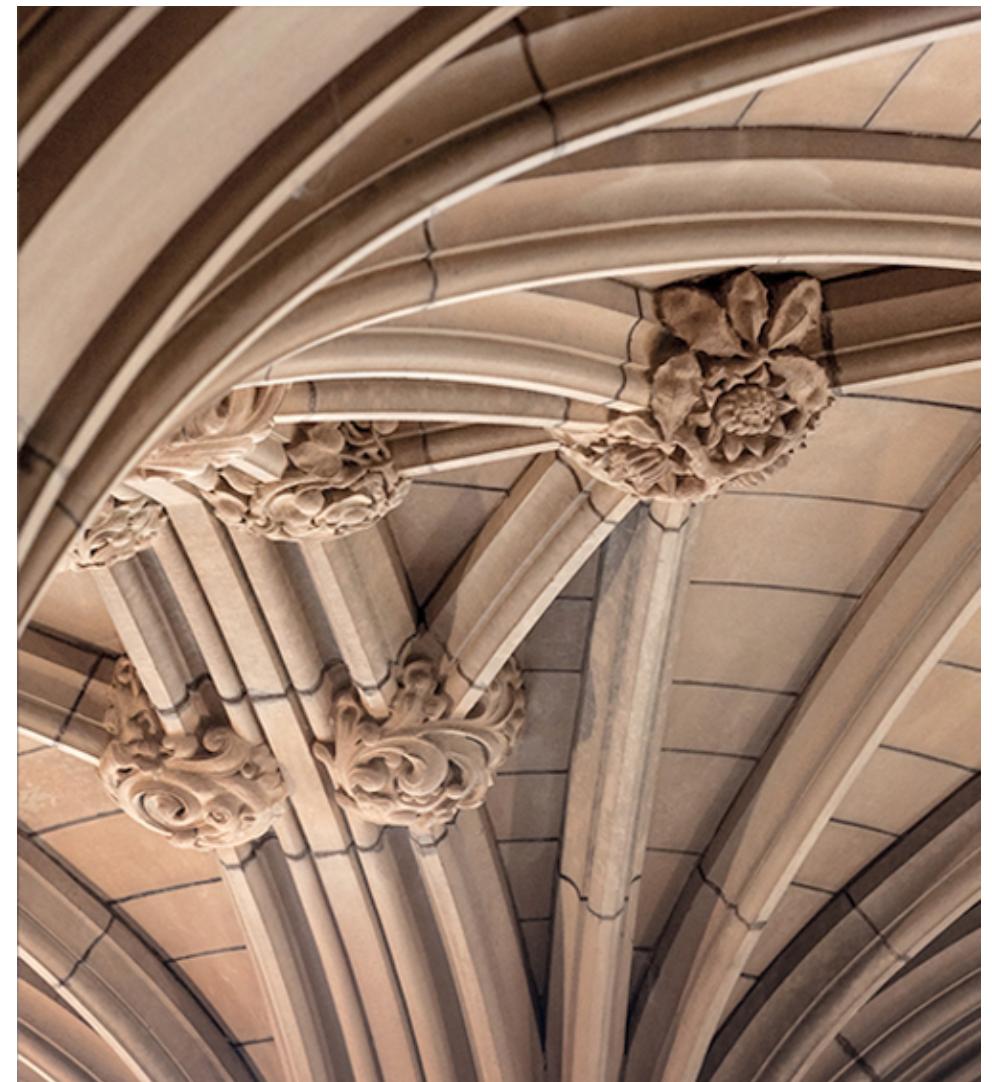
Test Case Selection

- Understanding developers thinking
 - Developers tend to think of typical values of input
 - Developers sometimes overlook atypical values of input
- Choose test cases that are:
 - On the boundaries of the partitions
 - Close to the midpoint of the partition

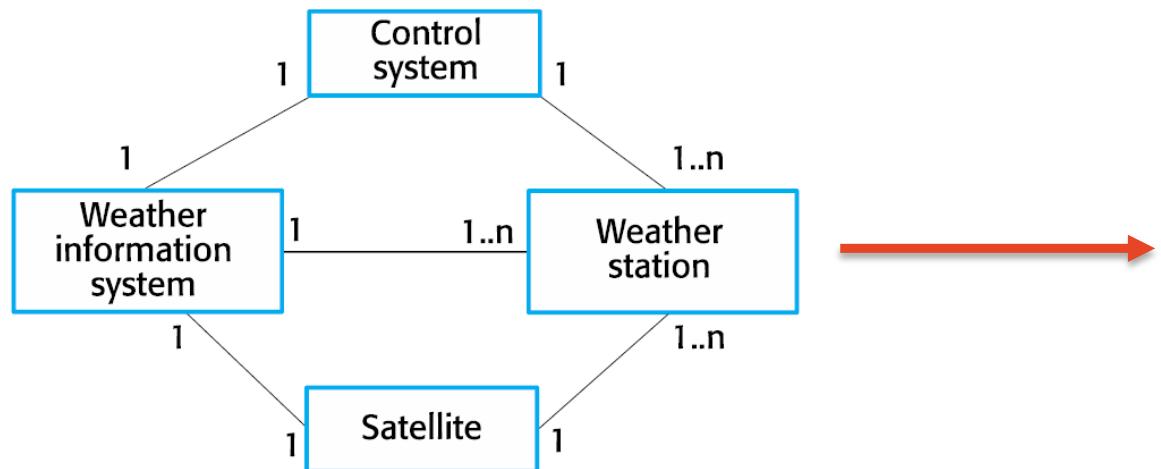


Unit Testing

Part 2



Unit Testing – Case Study



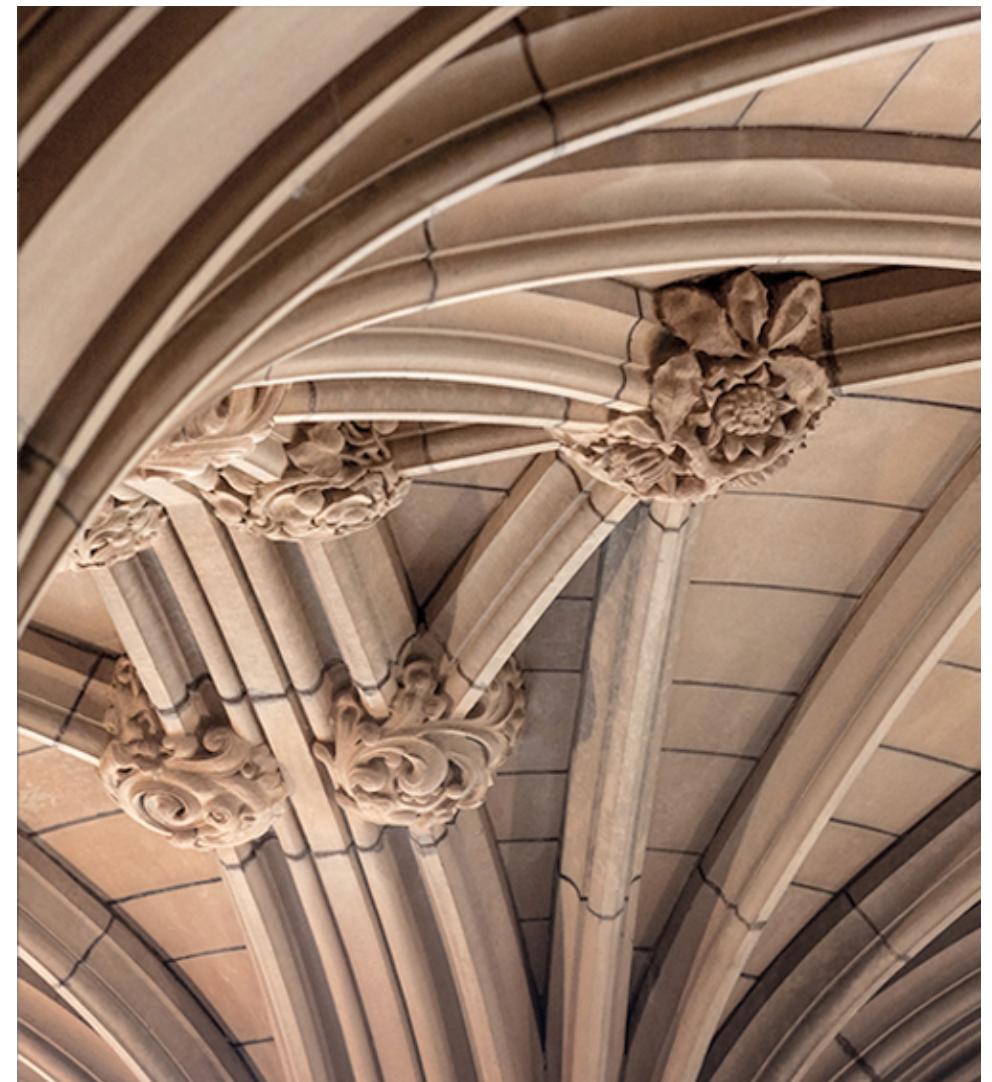
WeatherStation
identifier
reportWeather ()
reportStatus ()
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)

Unit Testing – Techniques

- Partitioning
- Boundary
- Path
- State testing
- Black and white testing

JUnit

Part 2



Junit – **Assert Class**

- Assert class provides *static* methods to test for certain conditions
- Assertion method compares the actual value returned by a test to the expected value
 - Specify the expected and actual results and the error message
 - Throw an *AssertionException* if the comparison fails

Junit – Test Execution Order

Exercise:

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

```
1 import org.junit.Test;
2 import org.junit.runners.MethodSorters;
3
4 public class TestMethodOrder {
5
6     @Test
7     public void testA() {
8         System.out.println("first");
9     }
10    @Test
11    public void testB() {
12        System.out.println("second");
13    }
14    @Test
15    public void testC() {
16        System.out.println("third");
17    }
18 }
```

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
 - lead into problems / poor test practices
- By default, Junit 4.11 uses 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/>

<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 as a parameter for the test method

Junit – Parameterized Test Example

```
1 package testing;
2 import org.junit.Test;
3 import org.junit.runner.RunWith;
4 import org.junit.runners.Parameterized;
5 import org.junit.runners.Parameterized.Parameters;
6 import java.util.Arrays;
7 import java.util.Collection;
8 import static org.junit.Assert.assertEquals;
9 import static org.junit.runners.Parameterized.*;
10
11 @RunWith(Parameterized.class)
12 public class ParameterizedTestFields {
13
14     // fields used together with @Parameter must be public
15     @Parameter(0)
16     public int m1;
17     @Parameter(1)
18     public int m2;
19     @Parameter(2)
20     public int result;
21
22     // creates the test data
23     @Parameters
24     public static Collection<Object[]> data() {
25         Object[][] data = new Object[][] { { 1, 2, 2 }, { 5, 3, 15 }, { 121, 4, 484 } };
26         return Arrays.asList(data);
27     }
28
29
30     @Test
31     public void testMultiplyException() {
32         MyClass tester = new MyClass();
33         assertEquals("Result", result, tester.multiply(m1, m2));
34     }
35
36
37     // class to be tested
38     class MyClass {
39         public int multiply(int i, int j) {
40             return i * j;
41         }
42     }
43
44 }
```

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

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
 - 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	Continue execution of test after occurrence of first problem
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

- Timeout Rule

```
1 import org.junit.Rule;
2 import org.junit.Test;
3 import org.junit.rules.Timeout;
4
5 public class HasGlobalTimeout {
6     public static String log;
7     private final CountDownLatch latch = new CountDownLatch(1);
8
9     @Rule
10    // 10 seconds max per method tested
11    public Timeout globalTimeout = Timeout.seconds(10);
12
13    @Test
14    public void testSleepForTooLong() throws Exception {
15        log += "ran1";
16        TimeUnit.SECONDS.sleep(100); // sleep for 100 seconds
17    }
18
19    @Test
20    public void testBlockForever() throws Exception {
21        log += "ran2";
22        latch.await(); // will block
23    }
24 }
```

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 in for the right situation – the goal is to write good tests (not test smell)

Testable Code



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?



Smell bad.

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

..

Testable Code

- Adhere to known **design principles**
 - **SOLID**: Single responsibility, Open-closed, Liskov Substitution, Interface Segregation, Dependency Inversion
 - **GRASP** (General Responsibility Assignment Software Patterns) principles: Creator, Information Expert, High Cohesion, Low Coupling, Controller (see revision slides at the end)
- Adhere to API design principles
 - KISS, YAGNI, DRY, Occam's Razor
 - Information hiding, encapsulation, documentation, naming convention, parameters selection
- ..
- 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/>

Theory of Testing, Design of Test Cases and....

Next Lecture/Tutorial

