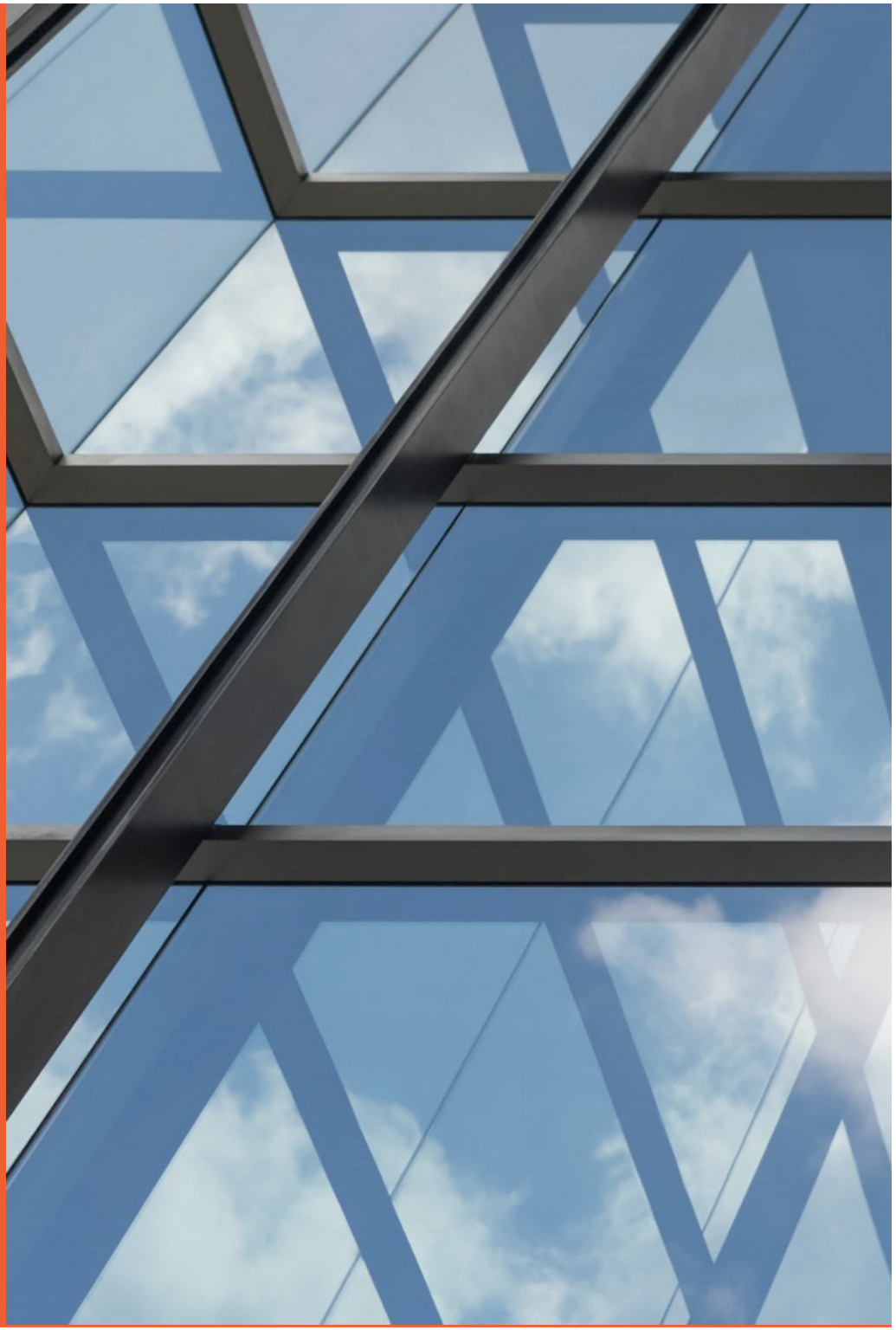
SOFT2201/COMP9201: Software Construction and Design 1

Testing

Dr. Xi Wu
School of Computer Science



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

- Software Testing
- Unit Testing

Software Engineering Body of Knowledge

- Software Requirements
- **Software Design / Modelling**
- **Software Construction**
- **Software Testing**
- Software Maintenance
- Software Configuration Management
- Software Engineering Process
- Software Engineering Tools and Methods
- Software Quality

Software Engineering Body of Knowledge (SWEBOK) https://www.computer.org/web/swebok/

Why Software Testing?

THE UNIVERSITY OF SYDNEY

Software is Everywhere!

- Societies, businesses and governments depend on SW
  - Power, Telecommunication, Education, Government, Transport, Finance, Health
  - Work automation, communication, control of complex systems
- Large software economies in developed countries
  - IT application development expenditure in the US more than $250bn/year[1]
  - Total value added GDP in the US[2]: $1.07 trillion
- Emerging challenges
  - Security, robustness, human user-interface, and new computational platforms

[1] Chaos Report, Standish group Report, 2014
[2] softwareimpact.bsa.org

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

Why Software Testing?

- Software development and maintenance costs
  - Financial burden of failure

- Total costs of imperfect software testing for the US in 2002 was AUD86 billion*
  - One third of the cost could be eliminated by 'easily' improved software testing

- Need to develop functional, robust and reliable software
  - Human/social factor
    - Dependence on software in many aspects of their lives
  - Small software errors can lead to disasters

* NIST study 2002

What is Software Testing?

THE UNIVERSITY OF SYDNEY

Software testing

- Software process to
  - demonstrate that software meets its requirements (validation testing)
  - Find incorrect or undesired behaviour caused by defects (defect testing)
    - e.g. crashes, incorrect results, data corruption
- Part of the software Verification and Validation (V&V) process

Types of testing

- **Unit testing**
  - Verify functionality of software components independent of the whole system
- **Integration testing**
  - Verify interactions between software components
- **System Testing**
  - Verify functionality and behaviour of the entire software system
  - Includes security, performance, reliability, and external interfaces
- **Acceptance testing**
  - Verify desired acceptance criteria are met from the users point of view

Software Verification and Validation

- Software testing is part of software 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 correctly
  - Ensure the software meets the stated functional and non-functional requirements

Black box or White box

- **Black box testing**
  - The internals of the software system is unknown
  - Only inputs to the system are controlled, and outputs from the system are measured
  - Specification-based testing
  - May be the only choice to test libraries without access to internal

- **White box testing**
  - The internals of the software system are known
  - The internal structure is tested directly
  - Unit, integration, system testing

Types of testing

Functional testing

- Unit
- Integration
- System
- Regression
- Interface
- User Acceptance
- Configuration
- Sanity

Non-functional testing

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

Who should design and run tests?

THE UNIVERSITY OF SYDNEY

Test engineer

- **Independent testers**
  - Independent testers do not have the same biases as the developer
  - Different assumptions
  - Domain specific knowledge of testing

- **Developer**
  - Understands the system being developed
  - Domain specific knowledge of the system
  - Cheaper
  - Can finish writing the system faster without tests since they won't make mistakes

Unit Testing

THE UNIVERSITY OF SYDNEY

Unit testing

- The process of verifying functionality of software components independently
  - Unit can mean methods, functions, or object classes
  - Verify that each unit behaves as expected
  - Carried out by developers and software testers
  - First level of testing

Why unit testing

- Maintain and change code at a smaller scale
- Discover defects early and fix it when its cheaper
- Simplify integration testing
- Simplify debugging
- Code reusability

How to do the unit test

- Identify the unit that you want to test
- Design test case
- Prepare test data (input and expected output)
- Run test case using test data
- Compare result to expected output
- Prepare test reports

Designing test cases

- Effective test cases show:
  - The unit does what it is supposed to do
  - Reveal defects, if they exist (does not do what it is not supposed to do)

- Design two types of test case
  - Test normal operation of the unit
  - Test abnormal operation (common problems)

Designing test cases - techniques

- Partition testing (equivalence partitioning)
  - Identify groups of tests that have common characteristics
  - From each group, choose specific tests
  - Use program specifications, documentation, and experience

- Guideline-based testing
  - Use testing guidelines based on previous experience of the kinds of errors made
  - Depends on the existence of previous experience (developer/product)

Equivalence partitioning

- Groups of test input that have common characteristics
  - Positive numbers
  - Negative numbers
  - Boundaries
- Program is expected to behave in a comparable way for all members of a group
  - Control flow should be similar for all members
- Choose test cases from each partition

Test case selection

- Understanding developers thinking
  - Easy to focus on typical values of input
    - Common case, and what was asked for
  - Easy to overlook a typical value of input
    - Users, other developers, new features, all have different expectations
- Choose test cases that are
  - On boundaries of partitions
  - In 'midpoint' of partitions
  - NB: Boundaries may be unclear (-1, 0, 1, 0.5)

Test cases – identifying partitions

- Consider this specification:
  - The program accepts 4 to 8 inputs that are five digit integers greater than 10,000

- Identify the input partitions and possible test inputs

?

Test cases – identifying partitions

- Consider this specification:
  - The program accepts 4 to 8 inputs that are five digit integers greater than 10,000

- Identify the input partitions and possible test inputs

- How many values
  - <4, 4-8, >8
- How many digits
  - < 5, 5, > 5, non-digits
- How big
  - > 10000
  - etc.

Test case selection guidelines

- Knowledge of types of test case effective for finding errors
- If testing sequences, arrays, lists:
  - Single value
  - Different sequences of different sizes
  - Test partition boundaries (first, middle, last)
  - Consider order of values

Test case selection guidelines

- Choose inputs that force the system to generate all expected error messages
- Design inputs that cause buffer overflows
- Repeat input
- Force invalid outputs to be generated
- Force computations results that are too large or too small
- Domain specific knowledge!

Acquiring domain specific knowledge

- Be an expert on the system, or type of system
- **or,**
- Make many mistakes
- Identify mistakes
- Write tests to identify mistakes
- Fix mistakes
- Be an expert on the system, or type of system

- Regression testing!

Regression testing

- If a defect is identified in software it can be fixed
  - How did it get there?
  - How do you stop it happening again?

Regression testing

- Regression: a defect that has been fixed before, happens again
  - Human error
  - Version control problems
  - Specific case is fixed, but the general case remains
  - Convergent evolution

Regression testing

- As defects in software are fixed, tests are written that demonstrate that the software is fixed (at least in regard to that particular defect)

  - Tests can be re-run with each change in the software system

    - Regression testing

    - Frequently automated

When to test

THE UNIVERSITY OF SYDNEY

When to test

- Continuously

- When the software system changes
  - Code changes
  - Design changes
  - Infrastructure changes
  - At regular intervals in case the above missed a change

How to test

THE UNIVERSITY OF SYDNEY

How to test

- Write testable code

```
public static void main(String[] args) {
    // All the code
    // All
}
```

How to test

- Write testable code

```
public static void main(String[] args) {
    Application app = new Application();
}

Public class Application {
    Application() {
        // All the code
    }
}
```

How to test

- Write testable code

```
public static void main(String[] args) {
    Application app = new Application();
    app.doEverything();
}

public class Application {
    Application() {
        // Construct the application
    }
    public void doEverything() {
      // All the code
    }
}
```

How to test

- Write testable code

```
public class Application {
    Application() {
        // Construct the application
    }
    public void doEverything() {
      // Most of the code
      doSomeOfTheThings();
    }
    public void doSomeOfTheThings() {
        // Some of the code
    }
}
```

How to test

- Write testable code

```
public class Application {
    Application() {
        // Construct the application
    }
    public void doEverything() {
      // Some code
      Thing = doSomeOfTheThings(thing);
      // More code
    }
    public BigThing doSomeOfTheThings(LittleThing littleThing) {
        // Some of the code that deals with LittleThings
    }
}
```

How to test

- Write testable code

```
public class Application {
    // ...
    public void doEverything(LittleThingFactory littleThingFactory) {
      LittleThing firstThing= littleThingFactory.makeThing();
      LitteThing secondThing = doStuff(firstThing);
      doStuff(secondThing);
      doStuffWithTwoThings(firstThing, secondThing);
      doSomeOfTheThings(thing);
      // ...
    }
    protected BigThing doSomeOfTheThings(LittleThing littleThing) {
        // Some of the code that deals with LittleThings
    }
    // ...
}
```

Unit Testing in Java

Unit testing terminology

- **Unit test**
  - A piece of code written by a developer that executes a specific functionality in the code under test and asserts a certain behaviour or state as correct
  - Small unit of code (method/class)
  - External dependencies are removed
    - (Mocking)
- **Test fixture**
  - Testing context
    - Shared test data
    - Methods for setting up test data

Unit testing frameworks for Java

- JUnit
- TestNG
- Jtest
- Many others
- Custom, developer-written, tests

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1));
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1));
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1));
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertEquals(2, calculator.add(1, 1));
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        int expected = 2;
        int actual = calculator.add(1, 1);
        assertEquals(expected, actual);
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        int expected = 2;
        int actual = calculator.add(1, 1);
        assertEquals(expected, actual);
    }
}
```

JUnit

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        int expected = 2;
        int actual = calculator.add(1, 1);
        assertEquals(expected, actual);
    }
}
```

JUnit constructs

- **JUnit test**
  - A method only used for testing
- **Test suite**
  - A set of test classes to be executed together
- **Test annotations**
  - Define test methods (e.g., @Test, @Before)
  - JUnit uses the annotations to build the tests
- **Assertion methods**
  - Check expected result is the actual result
  - e.g., assertEquals, assertTrue, assertSame

JUnit annotations

- **@Test**
  - Identifies a test method
- **@Before**
  - Execute before each test
- **@After**
  - Execute after each test
- **@BeforeClass**
  - Execute once, before all tests in this class
- **@AfterClass**
  - Execute once, after all tests in this class

JUnit assertions

- assertEquals(expected, actual)

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        int expected = 2;
        int actual = calculator.add(1, 1);
        assertEquals(expected, actual);
    }
}
```

JUnit assertions

- assertEquals(message, expected, actual)

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        int expected = 2;
        int actual = calculator.add(1, 1);
        assertEquals("Expected value != actual", expected, actual);
    }
}
```

JUnit assertions

- assertTrue

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertTrue(2 == calculator.add(1, 1));
    }
}
```

JUnit assertions

- assertTrue

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import mypackage.Calculator;

class CalculatorTest {
    @Test
    void addition() {
        Calculator calculator = new Calculator();
        assertTrue("Can't do 1 + 1 :(", 2 == calculator.add(1, 1));
    }
}
```

JUnit assertions

```
import ...

class CalculatorTest {
    Calculator calculator
    @Before
    void setup() {
        calculator = new Calculator();
    }
    @Test
    void additionBothPositive() {
        assertEquals(2, calculator.add(1, 1));
        assertEquals(5, calculator.add(4, 1));
         assertEquals(5, calculator.add(2, 3));
    }
    ...
```

Tasks for Week 11

- Submit weekly exercise on canvas before 23.59pm Saturday
- Continue assignment 3 and ask questions on Ed platform.
  - All assignments are individual assignments
  - Please note that: work must be done individually without consulting someone else's solutions in accordance with the University's "Academic Dishonesty and Plagiarism" policies

What are we going to learn next week?

- Creational Design Pattern
  - Singleton
- Structural Design pattern
  - Decorator and Façade