CS 520
Theory and Practice of Software Engineering
Fall 2021
Software testing
October 5, 2021
Recap: Design patterns

- Addresses a recurring, common design problem.
- Provides a generalizable solution.
- Provides a common terminology.

1. Structural
   - Composite
   - Decorator
   - ...

2. Behavioral
   - Observer
   - Strategy
   - Template (method)
   - ...

3. Creational
   - Factory (method)
   - ...

● Addresses a recurring, common design problem.
● Provides a generalizable solution.
● Provides a common terminology.
Design pattern: Decorator

<<interface>>
Component
+operation()

1

- decorated: Component
+ operation(

CompA
+ operation()

CompB
+ operation()

Decorator
+ Decorator(d: Component)
+ operation()
One possible solution: Template method

AbstractMedian

{abstract}

+ median(a:double[]):double
# sort(a:double[])

- The template method (median) implements the algorithm but leaves the sorting of the array undefined.

SimpleMedian

# sort(a:double[])
Another solution: Strategy

```
<<interface>>
Median
+median(a:double[]):double

StrategyMedian
-sortStrategy:Sorter
+median(a:double[]):double
+setSorter(s:Sorter)
```

```
<<interface>>
Sorter
+sort(array:double[])

HeapSort
+sort(...)

QuickSort
+sort(...) ... 1
```

“median” delegates the sorting of the array to a “sortStrategy”
Template method pattern vs. strategy pattern

Two solutions to the same problem

Template method
- Behavior selected at compile time.
- Template method is usually final.

Strategy
- Behavior selected at runtime.
- Composition/aggregation over inheritance.
Design pattern 1

Here is the Java Arrays utility class: [https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Arrays.html](https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Arrays.html)

We’ll focus on the following method:

```java
public static <T> void sort(T[] a, Comparator<? super T> c)
```

This method takes as input a Comparator class. Which design pattern is being applied?

- Composite
- Decorator
- Strategy
Design pattern 2

Here is the Java AbstractList class:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/AbstractList.html

Here is the ArrayList class:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/ArrayList.html

Which design pattern is being applied?

- Decorator
- Strategy
- Template (method)
Design pattern 3

Here is the Java Collections utility class:
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/util/Collections.html

We’ll focus on the following method:

```java
public static <T> List<T> unmodifiableList(List<? extends T> list)
```

For this method, which design pattern is being applied?

- Composite
- Decorator
- Strategy
Thursday (October 7)

- Second in-class exercise
- On testing (today is a prelude with useful info)
- Form 3-, 4-, or 5- person teams
  - Use Moodle to self-select a team
Today

Introduction to software testing

- Blackbox vs. whitebox testing
- Unit testing (vs. integration vs. system testing)
- Test adequacy
  - Structural code coverage
    - Statement coverage
    - Decision coverage
    - Condition coverage
  - Mutation analysis
  - User acceptance testing
Software testing

What can testing do, and what can’t it do?

Software testing can show the presence of defects, but never show their absence! (Edsger W. Dijkstra)

- A good test is one that fails because of a defect.

How do we come up with good tests?
Two strategies: black box vs. white box

**Black box testing**
- The system is a black box (can’t see inside).
- No knowledge about the internals of a system.
- Create tests solely based on the specification (e.g., input/output behavior).

**White box testing**
- Knowledge about the internals of a system.
- Create tests based on these internals (e.g., exercise a particular part or path of the system).
Unit testing, integration testing, system testing

Unit testing
- Does each unit work as specified?

Integration testing
- Do the units work when put together?

System testing
- Does the system work as a whole?

Our focus: unit testing
Unit testing

- A **unit** is the **smallest testable part** of the software system.
- **Goal**: Verify that each software unit performs as specified.
- **Focus**:
  - Individual units (not the interactions between units).
  - Usually input/output relationships.
JUnit 4: Overview

- Provides the xUnit testing framework for Java
- Uses Java annotations to specify tests and test suites
JUnit 4: Sample Rectangle class constructor

```java
public Rectangle(int width,
                 int height)

Creates a new rectangle with the given width and height where both the
width and height are positive numbers.

Parameters:
width - The width for the new rectangle
height - The height for the new rectangle

Throws:
java.lang.IllegalArgumentException - when either the
width or height is a negative number
```

Available from here: https://github.com/LASER-UMASS/cs520-Spring2020.git
(Contained in the rectangle folder)
JUnit 4: Test – Normative behavior

A single unit test [@Test]

```java
@Test
public void testNewRectangleSatisfiesPrecondition() {
    // Given known inputs
    Assert.assertTrue(WIDTH > 0);
    Assert.assertTrue(HEIGHT > 0);
    // Test on those inputs
    Rectangle rectangle = new Rectangle(WIDTH, HEIGHT);
    // Check for expected output
    Assert.assertNotNull(rectangle);
    Assert.assertEquals(WIDTH, rectangle.getWidth());
    Assert.assertEquals(HEIGHT, rectangle.getHeight());
}
```
JUnit 4: Test – Exceptional behavior

```java
// Check expected output
@Test(expected=IllegalArgumentOutOfRangeException.class)
public void testNewRectangleViolatesPrecondition() {
    // Given known inputs
    int negativeWidth = - WIDTH;
    Assert.assertFalse(negativeWidth > 0);
    Assert.assertTrue(HEIGHT > 0);
    // Test on those inputs
    Rectangle newRectangle = new Rectangle(negativeWidth, HEIGHT);
}
```
JUnit 4: Test fixture

A fixed set of objects used as a baseline to run tests (to be able to replicate test results) can be run before/after:

- each method (test) [@Before/After]
- each class (often test suite) [@BeforeClass/AfterClass]

```java
@Before
public void setUp() {
    rectangle = new Rectangle(WIDTH, HEIGHT);
    square = new Rectangle(WIDTH, WIDTH);
}

@After
public void tearDown() {
    rectangle = null;
    square = null;
}
```
JUnit 4: Test –
Normative behavior (using a text fixture)

```java
@Test
d public void testNewLabelSatisfiesPrecondition() {
    // Given known inputs
    Assert.assertTrue(WIDTH > 0);
    Assert.assertTrue(HEIGHT > 0);
    // Check for expected output
    Assert.assertNotNull(rectangle);
    Assert.assertEquals(WIDTH, rectangle.getWidth());
    Assert.assertEquals(HEIGHT, rectangle.getHeight());
}

@test
public void testEqualsNullReturnsFalse() {
    Assert.assertNotNull(rectangle);
    Rectangle nullRectangle = null;
    Assert.assertNull(nullRectangle);
    Assert.assertFalse(rectangle.equals(nullRectangle));
}
```
JUnit 4: Rectangle class other methods

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>equals(java.lang.Object obj2)</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getArea()</td>
<td>Gets the area of this rectangle.</td>
</tr>
<tr>
<td>int</td>
<td>getHeight()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>getWidth()</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>hashCode()</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>isSquare()</td>
<td>Returns true if this rectangle is a square and false otherwise.</td>
</tr>
<tr>
<td>void</td>
<td>setHeight(int height)</td>
<td>Sets the height of this rectangle to the given positive number.</td>
</tr>
<tr>
<td>void</td>
<td>setWidth(int width)</td>
<td>Sets the width of this rectangle to the given positive number.</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>toString()</td>
<td></td>
</tr>
</tbody>
</table>
JUnit 4: Test suite - Implied

A test suite consists of one or more tests or test suites

```java
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.Assert;

public class RectangleTests {
    public static final int WIDTH = 3;
    public static final int HEIGHT = 2;
    private Rectangle rectangle;
    private Rectangle square;

    @Before
    public void setUp() {
        rectangle = new Rectangle(WIDTH, HEIGHT);
        square = new Rectangle(WIDTH, WIDTH);
    }

    @After
    public void tearDown() {
        rectangle = null;
        square = null;
    }

    // Check expected output
    @Test(expected=IllegalArgumentException.class)
    public void testNewRectangleViolatesPrecondition() {
        // Given known inputs
        int negativeWidth = - WIDTH;
        Assert.assertFalse(negativeWidth > 0);
        Assert.assertTrue(HEIGHT > 0);
        // Test on those inputs
        Rectangle newRectangle = new Rectangle(negativeWidth, HEIGHT);
    }

    @Test
    public void testNewRectangleSatisfiesPrecondition() {
        // Given known inputs
        Assert.assertTrue(WIDTH > 0);
    }
```
JUnit 4: Test suite - Explicit

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses(
    RectangleTests.class
)

public class RectangleTestSuite {
    // the class remains empty,
    // used only as a holder for the above annotations
}
JUnit 4: Running a test suite

From command line:
- “java org.junit.runner.JUnitCore –cp ../lib/junit-4.11.jar:bin RectangleTests”

From Eclipse UI:
1. Right click on RectangleTests.java
2. In the context menu, select “Run As > JUnit Test”

NOTE) Can replace RectangleTests.java with RectangleTestSuite.java in the above
JUnit 4: Interpreting the test suite results

Finished after 0.129 seconds

Runs: 8/8  Errors: 0  Failures: 2

RectangleTests [Runner: JUnit 4] (0.031 s)
- testNewRectangleViolatesPrecondition (0.000 s)
- testNewRectangleSatisfiesPrecondition (0.000 s)
- testEqualsNullReturnsFalse (0.000 s)
- testIsSquareReturnsTrue (0.000 s)
- testGetWidthSatisfiesPrecondition (0.000 s)
- testGetWidthViolatesPrecondition (0.029 s)
- testGetAreaOfRectangle (0.001 s)
- testGetAreaOfSquare (0.001 s)
JUnit 4: Interpreting the test suite results

Finished after 0.129 seconds

Runs: 8/8  Errors: 0  Failures: 2

RectangleTests [Runner: JUnit 4] (0.031 s)

- testNewRectangleViolatesPrecondition (0.000 s)
- testNewRectangleSatisfiesPrecondition (0.000 s)
- testEqualsNullReturnsFalse (0.000 s)
- testIsSquareReturnsTrue (0.000 s)
- testGetWidthSatisfiesPrecondition (0.000 s)
- testGetWidthViolatesPrecondition (0.029 s)

Failure Trace

java.lang.AssertionError: Expected exception: java.lang.IllegalArgumentException
JUnit 4: Interpreting the test suite results

Finished after 0.129 seconds

- Runs: 8/8
- Errors: 0
- Failures: 2

RectangleTests [Runner: JUnit 4] (0.031 s)
- testNewRectangleViolatesPrecondition (0.000 s)
- testNewRectangleSatisfiesPrecondition (0.000 s)
- testEqualsNullReturnsFalse (0.000 s)
- testIsSquareReturnsTrue (0.000 s)
- testSetWidthSatisfiesPrecondition (0.000 s)
- testSetWidthViolatesPrecondition (0.029 s)
- testGetAreaOfRectangle (0.001 s)

Failure Trace

java.lang.AssertionError: expected:<6> but was:<9>
at RectangleTests.testGetAreaOfRectangle(RectangleTests.java:77)
Regression testing is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change. If not, that would be called a regression. Changes that may require regression testing include bug fixes, software enhancements, configuration changes, and even substitution of electronic components. As regression test suites tend to grow with each found defect, test automation is frequently involved. Sometimes a change impact analysis is performed to determine an appropriate subset of tests (non-regression analysis).

Contents

1 Background
2 Techniques
   2.1 Retest all
   2.2 Regression test selection
   2.3 Test case prioritization
      2.3.1 Types of test case prioritization
   2.4 Hybrid
Software testing

Software testing can show the presence of defects, but never show their absence! (Edsger W. Dijkstra)

- A good test is one that fails because of a defect.

When should we stop testing if no (new) test fails?
Test effectiveness

Ratio of detected defects is the best effectiveness metric!

Problem
- The set of defects is unknowable

Solution
- Use a proxy metric, for example code coverage or mutation detection rate
Average of the absolute values of an array of doubles

```java
public double avgAbs(double ... numbers) {
    // We expect the array to be non-null and non-empty
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("Array numbers must not be null or empty!");
    }

    double sum = 0;
    for (int i=0; i<numbers.length; ++i) {
        double d = numbers[i];
        if (d < 0) {
            sum -= d;
        } else {
            sum += d;
        }
    }

    return sum/numbers.length;
}
```
Control Flow Graph (CFG): example

Entry point

- `a == null || a.length == 0`
  - true: throw new IllegalArgumentException("Array a must not be null or empty!") → Exceptional exit
  - false: sum = 0; i = 0

- `i < a.length`
  - false: return sum / a.length → Normal exit
  - true: num = a[i]

- `num < 0`
  - false: sum += num
  - true: sum -= num

++i
Statement coverage

- **Every statement** in the program must be executed at least once
- Given the control-flow graph (CFG), this is equivalent to node coverage
Statement coverage: example

Entry point

\[ a==null \lor a.length==0 \]

\[ \text{true} \Rightarrow \text{throw new IllegalArgumentException(} \text{"Array a must not be null or empty!"}) \]

\[ \text{false} \Rightarrow \text{sum = 0, i = 0} \]

\[ i < a.length \]

\[ \text{false} \Rightarrow \text{return sum/a.length} \]

\[ \text{true} \Rightarrow \text{num = a[i]} \]

\[ \text{num < 0} \]

\[ \text{false} \Rightarrow \text{sum += num} \]

\[ \text{true} \Rightarrow \text{sum -= num} \]

++i
Condition coverage vs. decision coverage

**Terminology**

- **Condition**: a boolean expression that cannot be decomposed into simpler boolean expressions.

- **Decision**: a boolean expression that is composed of conditions, using 0 or more logical connectors (a decision with 0 logical connectors is a condition).

- **Example**: if ((x<5) && (y>7)) { … }
  - (x<5) and (y>7) are *conditions*.
  - The boolean expression ((x<5) && (y>7)) is a *decision*. 
Decision coverage (a.k.a. branch coverage)

- **Every decision** in the program must take on all possible outcomes (true/false) at least once
- Given the CFG, this is equivalent to edge coverage
- Example: if \((a>0 \&\& b>0)\)
  - \(a=1, b=1\)
  - \(a=0, b=0\)
Decision coverage: example

Entry point

```
a == null ||
a.length == 0
```

- **true**
  - throw new IllegalArgumentException(“Array a must not be null or empty!”)
- **false**
  - sum = 0
  - i = 0

```
i < a.length
```
- **false**
  - return sum/a.length
- **true**
  - num = a[i]
  - num < 0
    - **false**
      - sum += num
    - **true**
      - sum -= num
  - ++i

Normal exit

Exceptional exit
Condition coverage

- **Every condition** in the program must take on all possible outcomes (true/false) **at least once**
- Example: \((a>0 \land b>0)\)
  - a=1, b=0
  - a=0, b=1
Condition coverage: example

Entry point

true

a==null || a.length==0

false

sum = 0
i = 0

false

i<a.length

true

num = a[i]

false

num < 0

true

num += num

false

sum-=num

++i

return sum/a.length

throw new IllegalArgumentException("Array a must not be null or empty!")

Exceptional exit

Normal exit
Subsumption

Given two coverage criteria A and B, 
A subsumes B iff satisfying A implies satisfying B

- Subsumption relationships:
  - Does decision coverage subsume statement coverage?
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?
4 possible tests for the decision $a \lor b$:

1. $a = 0, b = 0$
2. $a = 0, b = 1$
3. $a = 1, b = 0$
4. $a = 1, b = 1$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$a \lor b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Satisfies **condition coverage** but not **decision coverage**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>$a \lor b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Satisfies **decision coverage** but not **condition coverage**

Neither coverage criterion subsumes the other!
Structural code coverage: subsumption

Given two coverage criteria A and B, 
**A subsumes B** iff satisfying A implies satisfying B

- **Subsumption relationships:**
  - Decision coverage **subsumes** statement coverage
  - Decision coverage **does not subsume** condition coverage
  - Condition coverage **does not subsume** decision coverage
Code coverage: advantages

- Code coverage is easy to compute.
- Code coverage has an intuitive interpretation.

But, does coverage ensure effective testing?
Code coverage: drawbacks

- Code coverage does not require test assertions.
- Not all statements etc. are equally important.
- Coverage is not the same as behavior.

Are there any alternatives?
Mutation analysis: overview
Mutation analysis: overview
Mutation analysis: overview

Program

Generate mutants

Mutants

Each mutant contains one small syntactic change

```java
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```java
public float avg(float[] data) {
    float sum = 1;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```
Mutation analysis: overview

```java
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```java
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        // Modified code
    }
    return sum / data.length;
}
```
Mutation analysis: overview

```
public float avg(float[] data) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum / data.length;
}
```

```
public float avg(float[] data) {
    public float avg(float[] data) {
        public float avg(float[] data) {
            float sum = 0;
            for (float num : data) {
                sum += num;
            }
        }
    }
}
```

```
public float avg(float[] data) {
    return sum * data.length;
}
```
Mutation analysis: overview
Assumption: Mutant detection rate is a good proxy for fault detection rate.

What does it mean for a test to fail on a mutant program?
Mutation analysis: first example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return a;
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>Original</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Mutation analysis: second example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
public int min(int a, int b) {
    return a <= b ? a : b;
}
```

There is no such test that can detect the mutant...

The mutant is undetectable because it is equivalent to the original program!
Mutant detection rate

Search for a test case that passes on the original program but fails on the mutant

- If found, the mutant is not equivalent to the original program. This is called a **detectable mutant**.
- If not found, the mutant is equivalent to the original program. This is called an **undetectable mutant**.

\[
\text{mutant detection rate} = \frac{\# \text{ detectable mutants}}{\# \text{ all mutants}}
\]
User acceptance testing

- **Alpha testing**: The development team releases alpha versions of the UI to internal users in a simulated setting.
- **Beta testing**: The development team releases beta versions of the UI to target an external user group in a real-world setting.

The alpha and beta testers submit bug reports and provide their feedback on using the UI.

https://www.webopedia.com/TERM/B/beta_test.html
Summary

- Testing is an important way to measure code quality
- Black-box testing versus White-box testing
- Coverage metrics (Statement, Condition, Decision) and Mutation-based metric

For more, read:

“Are mutants a valid substitute for real faults in software testing?” in FSE 2014

- User acceptance testing: Alpha versus Beta, SUS
Thursday (October 7)

- Second in-class exercise
- On testing (today is a prelude with useful info)
- Form 3-, 4-, or 5- person teams
  - Use Moodle to self-select a team