CS 520
Theory and Practice of Software Engineering
Spring 2021
Software testing
March 9, 2021
Last week

- Requirements Engineering
- Collaborative Development

**Another example:** Single slot buffer requirements (written in Natural Language and as FSA) using pair programming
Example: Single slot buffer requirements (specified in Natural Language)

1. The initial size is 0. The maximum size is 1. The `size()` method returns either 0 or 1.
2. The single element has type $E$.
3. The `write(element : E)` method attempts to store the given element into the single slot buffer. This method returns void when the size is 0. It throws a `BufferOverflow` exception when the size is 1.
4. The `read()` method attempts to retrieve the stored element from the single slot buffer. This method returns that element when the size is 1. It throws a `BufferUnderflow` exception when the size is 0.
Example: Single slot buffer requirement (specified as an FSA alphabet)

- size(): 0
- size(): 1
- write(E): void
- write(E): BufferOverflow
- read(): E
- read(): BufferUnderflow

**NOTE** In the following, we’ll use ‘*’ to mean all events in the alphabet.
Example: Single slot buffer requirement (specified as an FSA)

size(): 0, read(): BufferUnderflow

size(): 1, write(E): BufferOverflow, read(): E

write(E): void
Example: Single slot buffer requirement (specified as an FSA)

size(): 0, read(): BufferUnderflow

Start state: 0
Accepting states: \{0, 1\}
Example: Single slot buffer requirement (specified as an FSA)

size(): 0, read(): BufferUnderflow

size(): 1, write(E): BufferOverflow, read(): E

write(E): void
Example: Single slot buffer requirement (specified as an FSA)

size(): 0,  
read(): BufferUnderflow

size(): 1,  
write(): BufferOverflow

read(): E

write(E): void

size(): 1,  
write(E): void,  
read(): BufferUnderflow

read(): E

write(E): void

*
Example: Single slot buffer requirement (specified as an FSA)

size(): 0, read(): BufferUnderflow

size(): 1, write(): BufferOverflow

write(E): void

read(): E

size(): 0, write(E): void, read(): BufferUnderflow

viol

size(): 1, write(E): BufferOverflow, read(): E
Example: Single slot buffer requirement (specified as an FSA)

\[
\text{size}() : 0, \\
\text{read}() : \text{BufferUnderflow}
\]

\[
\text{size}() : 1, \\
\text{write}() : \text{BufferOverflow}
\]

\[
\text{write}(E) : \text{void} \\
\text{read}() : E
\]

\textit{NOTE)} The violation state and its transition are now implied.
Example: Single slot buffer requirement (specified as an FSA)

Are the following event sequences accepted or rejected?
- []
- [size(): 1]
- [read(): E]
- [write(E): void, read(): E]
Example: Single slot buffer requirement (specified as an FSA)

size(): 0, read(): BufferUnderflow

size(): 1, write(): BufferOverflow

Are the following event sequences accepted or rejected?
• [] √
• [size(): 1] ×
• [read(): E] ×
• [write(E): void, read(): E] √
Thursday (March 11)

- 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; open from today until Thursday 9 PM
  - After closing, the remaining students will be randomly assigned to groups
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=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);
}
```
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
public void testNewRectangleSatisfiesPrecondition() {
    // 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.assertNotNull(nullRectangle);
    Assert.assertFalse(rectangle.equals(nullRectangle));
}
```
JUnit 4: Rectangle class other methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean equals(java.lang.Object obj2)</td>
<td>Checks if the object is equal to this rectangle.</td>
</tr>
<tr>
<td>int getArea()</td>
<td>Gets the area of this rectangle.</td>
</tr>
<tr>
<td>int getHeight()</td>
<td>Calculates the height of the rectangle.</td>
</tr>
<tr>
<td>int getWidth()</td>
<td>Calculates the width of the rectangle.</td>
</tr>
<tr>
<td>int hashCode()</td>
<td>Generates a hash code for the rectangle.</td>
</tr>
<tr>
<td>boolean isSquare()</td>
<td>Returns true if this rectangle is a square, false otherwise.</td>
</tr>
<tr>
<td>void setHeight(int height)</td>
<td>Sets the height of the rectangle to the given positive number.</td>
</tr>
<tr>
<td>void setWidth(int width)</td>
<td>Sets the width of the rectangle to the given positive number.</td>
</tr>
<tr>
<td>java.lang.String toString()</td>
<td>Returns a string representation of the rectangle.</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;

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 negativeRectangle = 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
})

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
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)
- testGetWidthSatisfiesPrecondition (0.000 s)
- testGetWidthViolatesPrecondition (0.029 s)
- testGetWidth (0.001 s)

Failure Trace
- java.lang.AssertionError: expected:<6> but was:<9>
  at RectangleTests.testGetAreaOfRectangle(RectangleTests.java:77)
Regression testing (rarely non-regression testing[1]) is re-running functional and non-functional tests to ensure that previously developed and tested software still performs after a change.[2] 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.[3] 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[4]).

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

```java
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]

false

num < 0

false

sum += num

true

sum -= num

++i

Normal exit

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

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

- Exceptional exit
  - num < 0  
    - false: sum += num
    - true: sum -= num
    - ++i

- **Normal exit**
  - Exceptional exit
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 \((a>0 \land \&\& b>0)\)
  - a=1, b=1
  - a=0, b=0
Decision coverage: example

Entry point

\[ a == \text{null} \ \lor \ a.\text{length} == 0 \]

\begin{align*}
\text{sum} &= 0 \\
\text{i} &= 0
\end{align*}

\text{i} < \text{a.length}

\begin{align*}
\text{num} &= \text{a[i]} \\
\text{num} &= \text{num} < 0
\end{align*}

\begin{align*}
\text{sum} &= \text{sum} + \text{num} \\
\text{sum} &= \text{sum} - \text{num} \\
\text{i} &= \text{++i}
\end{align*}

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

Exceptional exit

Normal 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

- `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
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?
Decision coverage vs. condition coverage

4 possible tests for the decision $a || b$:

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

|   |   | $a || b$ |
|---|---|---------|
| 0 | 0 | 0       |
| 0 | 1 | 1       |
| 1 | 0 | 1       |
| 1 | 1 | 1       |

Satisfies condition coverage but not decision coverage

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

```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;
}
```

Each mutant contains one small syntactic change
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) {
    float sum = 0;
    for (float num : data) {
    }
    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) {
    float sum = 0;
    for (float num : data) {
        sum += num;
    }
    return sum * data.length;
}
```
Mutation analysis: overview
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:
public int min(int a, int b) {
    return a < b ? a : b;
}

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

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
Commonly use the SUS survey for feedback

<table>
<thead>
<tr>
<th></th>
<th>The System Usability Scale Standard Version</th>
<th>Strongly disagree</th>
<th>Strongly agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I think that I would like to use this system.</td>
<td>○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>2</td>
<td>I found the system unnecessarily complex.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>3</td>
<td>I thought the system was easy to use.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>4</td>
<td>I think that I would need the support of a technical person to be able to use this system.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>5</td>
<td>I found the various functions in the system were well integrated.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>6</td>
<td>I thought there was too much inconsistency in this system.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>7</td>
<td>I would imagine that most people would learn to use this system very quickly.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>8</td>
<td>I found the system very cumbersome to use.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>9</td>
<td>I felt very confident using the system.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
<tr>
<td>10</td>
<td>I needed to learn a lot of things before I could get going with this system.</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
<td>○ ○ ○ ○ ○ ○ ○</td>
</tr>
</tbody>
</table>

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
Group selection

● Form 3-, 4-, or 5-person teams
  ○ If you need more members for your team, raise your hand and ask the instructor

● Use Moodle to self-select a team

● Select a team member responsible for submitting the completed exercise by next Wednesday 3/17, 9 AM
Set up

1. Make sure that you have Git (v2.7.4 or later) and Java (v8 or later)
2. Clone the cs520 git repository:
git clone https://github.com/LASER-UMASS/cs520 inclass2
3. In a terminal, change into the triangle directory (inclass2/triangle)