

Testing

Introduction

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Feb. 27, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 Midterm Review
- 2 Midterm Exam #1

Homework Assignment

- 1 Programming assignment 2 is posted on your lab Canvas page.

Concepts

- ▶ **Failure:** any deviation of the observed behavior from the specified behavior
- ▶ **Erroneous State:** the system is in a state such that further processing of the system will result in a failure.
- ▶ **Fault:** the mechanical or algorithmic cause of the erroneous state. (AKA a defect, bug, error)
 - ▶ Fault leads to erroneous state which leads to failure
- ▶ **Test component:** part of the system that can be isolated for testing

Testing

- ▶ Testing is the systematic attempt to find faults/bugs in a planned way in the implemented software
 - ▶ Not the process of demonstrating that faults aren't present (which is impossible via testing)
 - ▶ A **successful** test shows us a failure
 - ▶ Tells us a bug exists
 - ▶ But may not tell us it's exact location in the code
- ▶ There is no way to demonstrate that faults are not present through testing unless we can test every possible input
- ▶ Similar to falsification of scientific theories
 - ▶ Design experiments where the goal is to falsify the theory
 - ▶ If the theory is not falsified, you now have more confidence in the theory
- ▶ Our goal in testing is to find the errors so we can correct them
 - ▶ If we are unable to find any, we are more confident in our software

Central Limitation of Testing

“Program testing can be used to show the presence of bugs, but never to show their absence!”

— Edsger W. Dijkstra (1972)

- ▶ We will discuss formal reasoning in a future lecture. . .

Test Case

- ▶ A **test case** contains:
 - ▶ A set of inputs
 - ▶ Includes the state of the object!
 - ▶ Expected results for those input
- ▶ We use them to detect the existence of faults by causing failures
- ▶ A failure is a deviation from the expected behavior, so we need to know the expected results
- ▶ A test case does not directly give us the fault, it just shows us the failure
 - ▶ We have to identify the underlying fault ourselves.

Unit Testing: Dealing with Scale

- ▶ **Best practice:** Test individual *units* or *components* of software (one class, one method at a time)
 - ▶ This is known as **unit testing**
 - ▶ Testing what happens when multiple components are put together into a larger system is known as **integration testing**
 - ▶ Testing a whole end-user system is known as **system testing**

Unit Tests

- ▶ Just focus on one small part of the system at a time
 - ▶ Reduces complexity of the tests
 - ▶ Focuses the test
 - ▶ Easier to pinpoint faults
 - ▶ Independence of testing components
- ▶ **Unit Test Techniques:**
 - ▶ Equivalence Testing
 - ▶ Boundary Testing
 - ▶ Path Testing¹

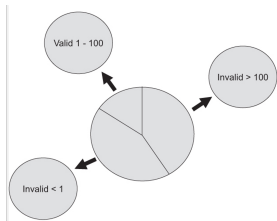
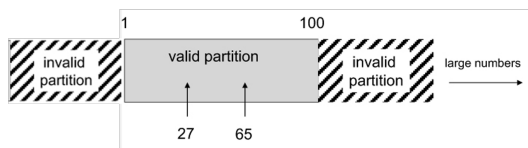
¹We will dicuss this in a future lecture

Equivalence Partitioning Testing

- ▶ Inputs to the software or system are divided into groups that are expected to exhibit similar behavior (i.e., likely to be processed in the same way)
- ▶ Tests can be designed to cover all valid and invalid partitions.
 - ▶ **Although in this class, we will only write test cases for the valid partitions.**
- ▶ Equivalence partitioning is applicable at all levels of testing.
- ▶ It can be applied to
 - ▶ human input
 - ▶ input via interfaces to a system
 - ▶ interface parameters in integration testing.

Equivalence Partitioning Example

- ▶ Valid input: integers in the range 1 to 100.
 - ▶ **Valid partition:** 1 to 100 inclusive.
 - ▶ **Invalid partitions:** less than 1, more than 100, real (decimal) numbers and non-numeric characters
 - ▶ Only need 1 value in each partition²



²For this class, you only need to worry about the valid partitions

Boundary Value Testing

- ▶ Considered an extension of equivalence partitioning
- ▶ Behavior at the edge of each equivalence partition is more likely to be incorrect than behavior within the partition
 - ▶ boundaries are an area where testing is likely to yield defects
 - ▶ why?
- ▶ The maximum and minimum values of a partition are its boundary values
- ▶ Tests can be designed to cover both valid and invalid boundary values
 - ▶ **Although in this class, we will only write test cases for the valid partitions.**
- ▶ When designing test cases, a test for each boundary value is chosen.

Equivalence Partitioning Example

- ▶ A real number in range -99.99 to 999.99 in increments of .01

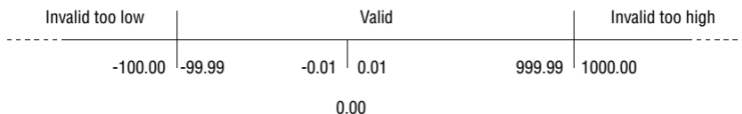


Figure 10-2: Real number equivalence partitions and boundary values

Test Stub and Test Driver

- ▶ **Test Stub**

- ▶ Partial implementation of components on which the tested components depend
 - ▶ Replaces lower level of code
 - ▶ Can avoid test stubs by testing lower levels of the system first (commonly done).

- ▶ **Test Driver**

- ▶ Partial implementation of component that depends on the test component
- ▶ Simulates the part of the system that uses the test component
- ▶ Call's the test component and then checks the input
- ▶ Replaces the user interface and higher levels of code
- ▶ Both can be used to isolate possible locations of the fault.

Designing a Test Plan

- ▶ A **test plan** is the set of test cases for a specific unit.
- ▶ To make testing most likely to succeed in revealing defects, **best practices** include:
 - ▶ **Test boundary cases:** “smallest”, “largest”, “special” values based on the contract
 - ▶ **Test routine cases**
 - ▶ **Test challenging cases:** i.e., ones that, if you were writing the code (maybe you didn’t write the code being tested!), you might find difficult or error-prone
- ▶ We want *distinct* test cases
 - ▶ Is there a scenario where you would expect one test case to pass and one to fail?
- ▶ Our test cases should follow Design by Contract
 - ▶ We don’t use inputs that go against the preconditions.
 - ▶ Our postconditions help us identify our expected output.
- ▶ Also don’t give test cases that compilers would prevent
 - ▶ Compilers prevent unmatching data types, etc.
 - ▶ However, we still need to test when testing UI.

Example Method Contract³

```
/**  
 * Returns some factor of a number.  
 * ...  
 *  
 * @pre n > 0  
 * @post n > 0 and n mod aFactor = 0  
 */  
private static int aFactor(int n) {...}
```

³ $n \bmod aFactor = 0$ means: “n is divisible by aFactor”

Partial Test Plan

Inputs	Results	Reason
<code>n = 1</code>	<code>aFactor = 1</code>	boundary
<code>n = 2</code>	<code>aFactor = 1</code> <code>aFactor = 2</code>	routine challenging? (prime)
<code>n = 4</code>	<code>aFactor = 1</code> <code>aFactor = 2</code> <code>aFactor = 4</code>	challenging? (square)
<code>n = 12</code>	<code>aFactor = 1</code> <code>aFactor = 2</code> <code>aFactor = 3</code> <code>aFactor = 4</code> <code>aFactor = 6</code> <code>aFactor = 12</code>	routine

Practice Problem #1

Create test cases for:

```
/**
 * ...
 *
 * @pre arr.length > 0
 * @post getAvg = [average of values in the array] and arr = #arr
 */
double getAvg(double[] arr)
```

Psychology of Testing

- ▶ Design and coding are *creative* activities
- ▶ Testing is a *destructive* activity
 - ▶ The primary goal is to “break” the software, i.e., to show that it has defects
- ▶ Very often the same person does both coding and testing
 - ▶ **Not a best practice** outside of unit testing
 - ▶ **You need a “split personality”:**
 - ▶ When you start testing, become paranoid and malicious
 - ▶ **It’s surprisingly hard to do**
 - ▶ Developers don’t like finding out that they made mistakes

Blackbox vs Whitebox Testing

- ▶ **Blackbox**

- ▶ Focus on the input and output behavior
- ▶ Ignore internal details or structure of the component
- ▶ Use the contracts to create our test plans

- ▶ **Whitebox**

- ▶ Focuses on internal structure of the component.
- ▶ Every state of the object and interactions are also tested
- ▶ Every possible path in the code can be tested
 - ▶ 100% code coverage
- ▶ Look at the code while developing test cases

Blackbox Testing

If we are doing **blackbox testing**, we don't use implementation details.

- ▶ We just rely on the interface to test our code
- ▶ **Pros:**
 - ▶ One test plan can work for any implementation of the same interface
 - ▶ If implementation changes, or a new implementation is developed, we don't have to change our test plan
- ▶ **Cons:**
 - ▶ We can't guarantee 100% code coverage
 - ▶ We don't look at the code, so we don't know if we are hitting every path
 - ▶ Techniques for developing good test cases are more conceptually difficult

Whitebox Testing

If we are doing **whitebox testing**, we look/use implementation details.

- ▶ **Pros:**

- ▶ We can guarantee 100% code coverage
- ▶ Conceptually easier to develop

- ▶ **Cons:**

- ▶ We can miss errors of omission
 - ▶ Coder leaves out a logical step in the process
 - ▶ Our test cases are based on their code, so we leave the out as well
- ▶ Can't develop before the code is written
- ▶ When the code changes, we may need to change our test plan
- ▶ New implementations of the same interface require new test plans

Tracing and Debugging

- ▶ **Objective**

- ▶ Locate errors using test inputs

- ▶ **Approach**

- ▶ Analyze the code on sample test inputs to understand why code fails to behave as specified in its contract

- ▶ **Tracing and Debugging**

- ▶ **Tracing:** Analyze, but do not execute code
 - ▶ **Debugging:** Execute code on selected inputs and follow the execution

Test Driven Development (TDD)

- ▶ A common practice in Software Engineering
- ▶ Create the automated test cases *before* writing the code based on the contracts
- ▶ Then write the code needed to pass your test cases
 - ▶ **Pros:**
 - ▶ You are ready to test your code immediately after writing it
 - ▶ Reducing the time between writing the code and discovering the existence of a fault makes it easier to locate the fault
 - ▶ Developing test cases first forces you to consider challenging inputs and how to handle them
 - ▶ **Cons:**
 - ▶ Only works with blackbox testing
 - ▶ Coding to the test plan relies on having a great test plan

Practice Problem #2

Create test cases for `IntegerDeque` interface we have made in lab:⁴

```
/**
 * @pre size < MAX_LENGTH
 * @post self = [x will be added to the end of #self] and x = #x
 */
void enqueue(Integer x)

/**
 * @pre size > 0 and 0 < pos <= size
 * @post self = [#self with the value at position pos removed] and
 *         removePos = [the value that was at position pos] and
 *         pos = #pos
 */
Integer removePos(int pos)
```

⁴ `size` is the number of elements in my deque (i.e. there is constraint that says: `size = |self|`) and `MAX_LENGTH` = 100

Practice Problem #3

Create test cases for `IntegerDeque` interface we have made in lab:⁵

```
/**
 * @pre 0 < pos <= size + 1 and size < MAX_LENGTH
 * @post self = [#self except x is inserted at position pos]
 *       pos = #pos and x = #x
 */
void insert(int pos, Integer x)

/**
 * @pre 0 < pos <= size
 * @post self = #self and get = [the value at position pos] and pos = #pos
 */
Integer get(int pos)
```

⁵`size` is the number of elements in my deque (i.e. there is constraint that says: `size = |self|`) and `MAX_LENGTH` = 100

Why does testing matter?

- ▶ Engineering comes with responsibility
 - ▶ You're responsible for the product you produce
 - ▶ In Software Engineering, you're responsible for the software you produce
 - ▶ You want it to be high quality software
- ▶ Bad software can lead to
 - ▶ Angry customers
 - ▶ Lost money and time
 - ▶ Even loss of life
- ▶ Let's consider some historical examples

Software Engineering Mistakes

- ▶ Mariner 1
- ▶ Supposed to do a flyby of Venus
- ▶ Cost 18.5 million in 1962 (\$146,476,299 today)
- ▶ NASA lost control of the craft, and had to destroy it 294.5 seconds after launch.
- ▶ Apparent cause? Incorrectly transcribed a hand written equation into the code.

The diagram illustrates a transcription error. It shows a handwritten-style equation $\dot{\bar{R}}_n$. An orange arrow points from a horizontal line to the bar over the R , indicating the error in transcribing the equation into the code.

Software Engineering Mistakes

- ▶ Lt. Col. Stanislav Petrov 1982
 - ▶ Soviet officer monitoring the missile detection system
- ▶ Alarms went off indicating incoming missiles from America
 - ▶ NATO involved in provocative exercises
 - ▶ Russians had recently shot down a South Korean airliner that went into its air space
- ▶ Petrov thought it was odd there were only 5 missiles, and told his superiors it was a false alarm, instead of telling them to fire the retaliatory missiles
- ▶ False alarm caused by the reflection of sunlight off the top of the clouds

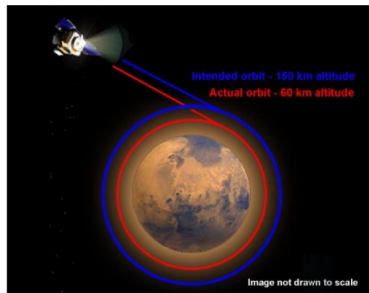
Software Engineering Mistakes

- ▶ Ariane 5 rocket
- ▶ First test launch ended in automatically triggered self destruct
- ▶ Velocity was computed as a 64-bit float
- ▶ Tried to store velocity in a 16-bit memory register
- ▶ The number was too large for the register, creating an overload error.
- ▶ Overload error caused a drastic and unnecessary flight correction
- ▶ Flight correction caused damage to booster rockets
- ▶ Damage to the booster rockets resulted in self destruct sequence being activated.



Software Engineering Mistakes

- ▶ Mars Climate Orbiter
- ▶ Two teams worked on the software for the orbiter
 - ▶ Lockheed Martin's team used Imperial units
 - ▶ NASA's JPL team used Metric units
- ▶ When the orbiter tried to calculate the thrust to enter the orbit of Mars the unit difference caused a miscalculation
- ▶ The orbiter crashed into Mars
- ▶ The orbiter cost \$125 million dollars



Software Engineering Mistakes

- ▶ Boeing 737 Max

- 1 <https://youtu.be/H2tuKiiznsY>

- 2 <https://youtu.be/mh-U6v2Cyco>

Homework Assignment

- 1 Programming assignment 2 is posted on your lab Canvas page.

Summary

- 1 Testing and Testing Concepts
- 2 Central Limitation of Testing
- 3 Test Case
- 4 Unit Testing
- 5 Designing a Test Plan
- 6 Blackbox and Whitebox Testing
- 7 Tracing and Debugging
- 8 Test Driven Development (TDD)
- 9 Why Testing Matters?