Testing & Verification

CS 2110: Software Development Methods

February 11, 2019 (Day 12)

Announcements

- Homework 3 due Friday 11:30pm
- Midterm Exam 1 next Monday (2/18)

Definitions

Specification: A precise and detailed statement about the required properties (the properties some thing must have)

- **Requirements specification**: Exact statements about what to build
- **Design specification**: Exact instructions for how to build something

Definitions

Verification: Showing that the code meets the requirement specification

- You *verify* that I wrote the program you asked me to write

Validation: Showing that the code meets the objectives / requirements

- You show that the program is a *valid* solution to the user's problem

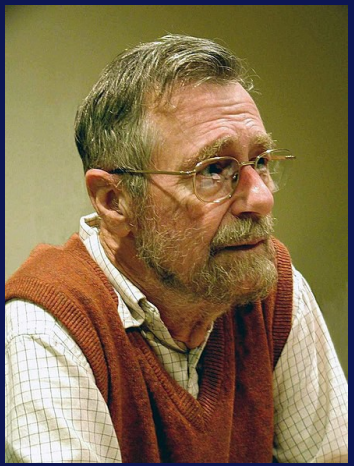
Definitions

Quality depends on specifications

- A project without specifications cannot be verified
- Your verification is only as good as your specifications!
- Quality is a *scale*

"Program testing can effectively show the *presence* of bugs but it is hopeless for showing their *absence*."

–Edsger Dijkstra



Some Types of Testing

- **Empirical**: Verified by observation
- **Unit**: Testing the smallest possible piece (unit) of the code
- **Regression**: Testing the impact of changed code on the rest of the application
- **Integration**: Testing separate units together
- **Acceptance** *(Beta Testing)*: Testing by the customer or user
- **Load**: Performance and capacity testing

Errors

Three fundamental types of errors: what can you test for?

- **Syntax**: "Grammatical" errors, caught by the compiler
  - Ex: missed a semicolon or forgot a `return` statement
- **Logical**: Program faithfully follows incorrect instructions
  - Ex: used multiplication of two numbers when it should have been subtraction
- **Runtime**: An error that occurs during the execution of a program
  - While the program runs
  - Ex: division by zero when the denominator is passed in by the user

Common Approaches to Testing

- Testing with the `main()` method
  - *"code a little, test a little"*
  - Writing code so that the class can be run to validate the methods within it
- Tester classes
  - Classes within the application dedicated to running tests and validating output
- Testing frameworks
  - Packaged tester class collections such as **JUnit**
  - *We'll use this extensively*

Empirical Testing

Instead of formal verification (formal proofs, etc), correctness is demonstrated through **empirical testing**

- Based on observed experience
- Uses a number of carefully crafted **test cases**
  - Use *several inputs* with *expected results*
- **Exaustive testing isusually *impossible*!**

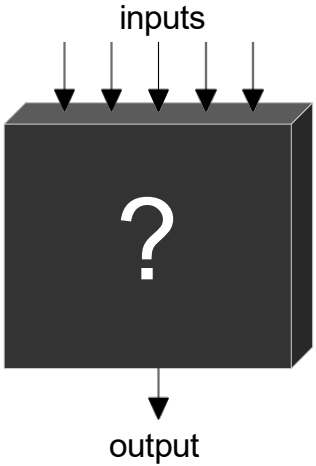
Unit Testing

Write code that tests the smallest possible units of the specification

- Must attempt to test every flow path
- Two main approaches to unit testing:
  - Black-box testing
  - Glass-box (or white-box) testing

Unit Testing

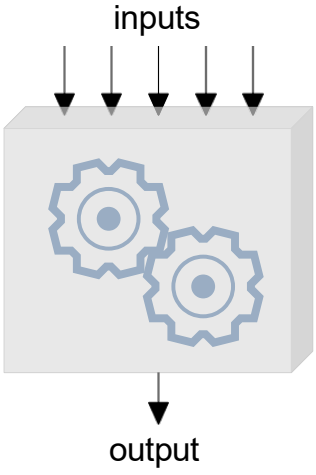
Black-Box Testing



- Test inputs and outputs (specification) without looking at the code
- Given x as input, do I get y as specified?

Unit Testing: Two Approaches

Glass-Box Testing



- Test based on the logic of the code (must study what it's doing)
- Principles of glass-box testing
  - Attempt to execute every line of code
  - Make sure loops get executed 0 times and > 0 times
  - *Pay attention to flow paths*

Flow Paths

Flow path: a unique execution sequence through a program unit

- A good set of test data might make sure that every possible path is followed (test every possible behavior)
- Virtually **infinite** number of flow paths
  - Loops can be executed an arbitrarily large number of times
  - *Therefore exaustive testing is usually impossible*

```
s1;
while (b1) {
    if (b2) {
        s2;
    } else{
        s3;
    }
}
```

Unit Testing

Unit testing is **NOT**

- Customer acceptance testing
  - Is this the right thing for the customer?
- Integration testing
  - Do all system components work together correctly?
- Performance testing
  - How well does the program work as a whole under varying conditions?
- ...other kinds of testing that could be added to the software development process

Unit Testing

Unit testing **IS**

- For the developer
  - Who better to test the code than the person who developed it?
- An integral component of the software project
  - Updated with code changes and any found bugs
- Valuable for helping the developer to fully understand the requirements
  - You simply cannot write any tests if you don't understand the requirements
- Always the #1 priority
  - New development **stops** until all the tests run without error

Planning Your Tests

How to do proper testing?

- **Plan a set of *test cases***: Define three parts first:
  - A purpose (why are we doing this particular test?)
  - Precise details about the state, inputs, etc
  - An expected result
  - *optional: an ID to allow us to refer to the test case*
- **Execute**: Run the tests to see if the *observed* output matches the **expected** result – *like a scientific experiment!*
- Have a plan when testing! (at least in your head)
  - For serious testing, write it out ahead of time
  - Don't test blindly or randomly – you'll waste your time

Planning Your Tests

Testing a 2-player board game... *how about these test cases?*

Test ID	Description	Expected Results	Actual Results
1	Player 1 rolls dice and moves	Player 1 moves on the board	
2	Player 2 rolls dice and moves	Player 2 moves on the board	

Planning Your Tests

Testing a 2-player board game... *Why are these tests better? Still problems?*

Test ID	Description	Expected Resuls	Actual Results
1	Precondition: Game is in test mode, GameBoard is at the start state, and game begins Number of Players: 2 Player 1 dice roll: 3	Player 1 moves to square Blue 3	
2	Precondition: TestID 1 completed successfully Player 2 dice roll: 3	Player 2 moves to square Blue 3	

Unit Testing: Things to Test

- **Boundary values**: often programs misbehave because we make mistakes dealing with the "extremes"
  - Ex: last or first item in a collection, start or end of a range of values
  - Ex: an empty list, an empty file, one item in a list,...
- **Off-by-one errors**: stopping at the wrong place, or going to far
  - Ex: array starting at 1 instead of 0, going to `size` instead of `size - 1`
  - Ex: executing a loop once when it should not be executed at all
- **Invalid inputs**: a method is called with inputs that are not valid
  - Ex: `null` instead of an Object
  - *Does the program handle these problems correctly?*

In-Class Activity

In-Class Activity

Testing Scenarios

- Gather in groups of 2, with people you haven't worked with before.
- Read the different scenarios on Collab and discuss which kind of testing would best have prevented the issue. Write your responses in the inline submission box on Collab.
- **Each person** must turn in their own copy of the solution!

In-Class Activity: Testing Scenarios

1. You've just launched your new startup, larpwithme.com, that matches complete strangers for costumed games. The site is wildly successful, with 100 visits on the second day and 3.2 million visits on the second day. Unfortunately, the site crashes whenever there are more than 1000 users on at the same time. What kind of testing would have prevented this?

2. You get a call from a dermatologist you did some work for recently, and she is livid. Your software accepts skin cancer measurements as width and height, and is computing the area for skin cancer using the formula for the area of a circle. The physician is expecting the calculation to use the formula for a rectangle, which means that all of the area data points are unusable. The program specification clearly indicates that the formula for a rectangle was desired. What testing should have caught this?

3. - 6. (See posted assignment…)

Quick Question!

When we test a class' method to see if its loop goes to `size() - 1` and not `size()`, this is clearly an example of:

1. A black-box test
2. A glass-box test
3. A unit test
4. Both 1 and 3
5. Both 2 and 3