Lecture 17: Program Verification

Kenneth M. Anderson

Software Methods and Tools

CSCI 3308 - Fall Semester, 2004

Today's Lecture

- Introduce the concept of program verification
  - specifications
  - terminology
  - debugging
  - testing
- Cover "Passing the Word" in Brooks' Corner
  - This chapter will set us up for the next lecture

Program Verification

- Program Verification is the **process of demonstrating** that a particular program **meets its specification**
  - If a program meets its specification it is considered "correct"

Program Correctness

- To repeat: **a program is correct only when it meets (i.e. implements) its specification**
- This does not mean that the program is actually useful!
  - In order for it to be useful, the specification has to match the needs of the program's users
- Furthermore, what happens if the specification contains an error (i.e. it doesn't specify the user's requirements correctly)
  - The program is still "correct"; but the program is not actually meeting the needs of its users

Program Specifications

- You can view a program's specification abstractly as a function that maps the program's inputs to its expected outputs
  - F(input) = output
  - e.g. if you click on this button, a menu pops up
    - F(click on button) = menu pops up
- Remember that this way of thinking is "an abstraction"
  - The "real world" is much more complex. For instance, a program may be "correct" on a machine with 64MB of memory, but fail on a machine with 32MB of memory
    - This type of error is related to the "non-functional" requirements of a system
    - In this class, we will be focusing on "functional" reqs. only

Testing Terminology

- **Error** - a mistake made by a programmer
  - implies that for some input i, F(i) ≠ expected output
- **Fault** - an incorrect state of a program that is entered because of the error
  - **Some errors don't cause failures right away**, every state between the error and the failure are faults
  - For this class, however, you can think of a "fault" as being the **location in the code where the error exists**
- **Failure** - a symptom of an error
  - e.g. a crash, incorrect output, incorrect behavior, …

Testing Terminology, cont.

- Discussion
  - A **failure occurs only if a fault occurs**, and a **fault occurs only if an error exists**
  - Note: not all faults are detected
    - because you may need to execute a specific portion (e.g. state) of the program for the failure to appear…
    - …and it may be impossible to execute all "states" of a program
  - Recall that Fred Brooks in No Silver Bullet talked about complexity and one aspect of complexity is the sheer number of states associated with software systems

An Example

- If a program contains an error, it does not necessarily lead to a failure

```
if (x < y) /* should be <= */
    …
else
    …
```

- The error may be a typo, or the error could be the result of the programmer not understanding the problem
- The fault is the location of the error, e.g. the expression contained in the if statement, or more explicitly the missing "="
- A failure may occur if x==y and this if statement is executed

Creating Correct Programs

- Strategy One: Error Prevention
  - Employ techniques to avoid errors in the first place
    - Software Reuse!
    - Create Solid Designs
      - before you even write code!
      - UML class diagrams, sequence diagrams, etc.
    - Smart Programming Environments
      - auto-balance, syntax coloring, etc.

Creating Correct Programs, cont.

- Strategy Two: Debugging
  - The process of discovering and eliminating faults
  - Tools can help with this
    - compiler warnings, source debuggers
  - Review can help as well
    - Code Inspections

Creating Correct Programs, cont.

- Strategy Three: Testing
  - The process of **discovering failures** and **locating the faults** which cause them
  - Give input to program, compare with expected output
    - The input / expected output is known as a test case
    - Testing involves creating test cases that “cover” (or rather “uncover”) different types of failures
  - Module and Integration Testing is done by developers (or QA), system testing is done by the customer!

Creating Test Cases

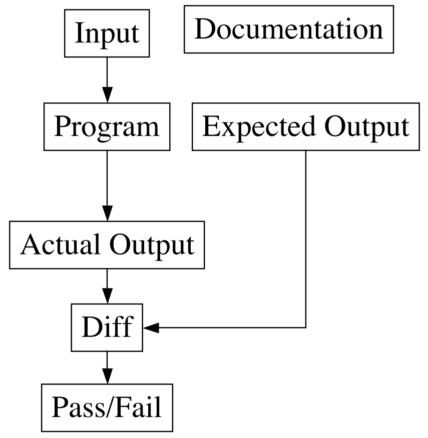
- A test case consists of
  - **Input**
    - The specific values given to the program
  - **Expected output**
    - The output predicated by the program’s specification
  - **Documentation**
    - What type of failure is this test case testing?

Test Runs

- Test cases are applied to a program during a **test run**. A test run consists of:
  - **Actual Output**
    - The output generated by the program when given the input of the test case
  - **Pass/Fail Grade**
    - Did the actual output match the expected output
- Test runs are typically supported by a "testing harness" or "test scaffolding"
  - This refers to the software that helps you perform (or sometimes automate) testing runs

Testing Process

1. For each class of failure defined in the documentation
2. For each test case in that class
3. Apply the input and compare the output to the specification
4. Record results
5. Fix problems
6. Repeat until all test cases pass



Creating Test Cases

- How do you pick test cases?
  - We will look at two strategies for doing this
    - Black Box Testing
    - White Box Testing
  - For now, think of trying to pick "categories" of input that test the same thing
    - e.g. its impossible to "exhaustively" test a program, but if your categories contain values that all test the same thing, you can get by with using just a single value from each category

Example

int GreatestCommonDivisor(int x, int y)

- **x=6 y=9, returns 3, tests common case**
- **x=2 y=4, returns 2, tests when x is the GCD**
- **x=3 y=5, returns 1, tests two primes**
- **x=9 y=0, returns ?, tests zero**
- **x=-3 y=9, returns ?, tests negative**

- **To test exhaustively is impossible (both parameters can take on an infinite number of values)**
  - **but with 5 categories identified, we can get by with only 5 test cases!**

Brooks' Corner: Passing the Word

- Communicating Design Decisions
  - Written Specifications
    - "The Manual"
      - Answers questions
      - Conceptual Integrity
      - Demands high precision
  - Formal Definitions
    - Natural language is not precise
    - Formal notations have been developed to help

Formal Definitions

- Notations help express precise semantics
  - However, natural language is often needed to "explain" the meaning to the uninitiated
- What about using an implementation?
  - Benefits: Precise specification
  - Disadvantages: Over-prescription, potential for inelegance, may be modified!

Communicating Design, continued

- Meetings
  - Weekly half-day meetings
    - Problems and change proposals distributed beforehand
    - Chief architect has final say
  - Annual "Supreme Court" sessions
    - Typically lasts two weeks
    - Agenda typically had 200 items!

Communicating Design, continued

- Multiple Implementations
  - Inconsistencies between implementations can identify problems in the specs;
    - With only one implementation, its easier to change the manual!
- The Telephone Log
  - Or, be sure to capture all decisions made by the chief programmer!
- Product Test
  - An external test group keeps the implementation honest