Software testing

Sudipto Ghosh
CS314

Reference: Chapter 8 of custom textbook, which is Chapter 15 (Implementation) in the original text.
Read Section 15.9—15.25
Pre-learning for Monday’s (Sep 14) flipped classroom

- Slides 1-26: including testing terminology (error, fault, failure, program, specification, test input, test case, test set, test oracle) what is an input domain, how to partition input into equivalence classes, what is a boundary
  - Specifically, slides 8, 9, 12, 13, 14, 21-26
- YouTube videos (see next slide)
YouTube Videos on Testing

- What black box testing is all about
  - https://www.youtube.com/watch?v=Wi75S5TTfQ0

- What goes into a test case
  - https://www.youtube.com/watch?v=BBmA5Qp6Ghk
Plan for Monday (Sep 14)

- Teams of 2 (create them on Friday)
- Quiz to check for pre-class learning
  - Do the quiz individually
  - Discuss differences with teammate
  - Submit quiz (IQ4)
- Discussion of IQ4
- Quiz for team activity
  - Work on quiz as a team; OK to ask for help
  - Submit quiz (GQ1)
- Discussion of GQ1
- Reflection of learning (to be submitted as IQ5 beginning of next class on Wednesday, Sep 16)
What is testing

- Act of checking if a part or product performs as expected.

- Why do we test?
  - Check if there are any errors?
  - Increase confidence in the “correctness”

- It is very easy to forget to test some aspects of a software system:
  - “Running the code a few times” is not enough.
  - Forgetting certain types of tests diminishes the system’s quality.

- Conflict between achieving adequate quality levels, and “getting the product out of the door”
  - Create a separate department to oversee QA.
  - Publish statistics about quality.
  - Build adequate time for all activities.
What to test and when?

- All products generated during lifecycle

<table>
<thead>
<tr>
<th>Phase</th>
<th>Product</th>
<th>How to test?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification</td>
<td>Req. doc</td>
<td>Scenario construction, simulations</td>
</tr>
<tr>
<td>Design</td>
<td>Design doc</td>
<td>Simulations</td>
</tr>
<tr>
<td>Implementation</td>
<td>Modules</td>
<td>Run against test cases</td>
</tr>
<tr>
<td>Integration</td>
<td>System</td>
<td>Run against test cases</td>
</tr>
<tr>
<td>Maintenance</td>
<td>System</td>
<td>Regression testing</td>
</tr>
</tbody>
</table>

- During each phase, simultaneously with all development and maintenance activities
QA terminology

- **Verification**: Process of determining whether a phase has been correctly carried out
  - Are we building the product right?

- **Validation**: Intensive evaluation process that takes place just before the product is delivered to the client
  - Are we building the right product?

- **V&V denotes testing**

- **Formal verification**: mathematically prove program correctness <<WHY IS THIS HARD?>>

- **Inspection and reviews**: Human beings go over code manually.
Definitions

- Error: slip-up or inappropriate decision by a software developer that leads to the introduction of a fault
- Fault: flaw in any aspect of the system that contributes, or may potentially contribute, to the occurrence of one or more failures
  - Several faults together may cause a failure
- Failure: unacceptable behaviour exhibited by a system
  - Frequency of failures measures the reliability
Program testing goals

- Testing is a demonstration that faults are not present.
- Program testing can be an effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. – Dijkstra
- Other goals:
  - Effectiveness: uncover as many defects as possible
  - Efficiency: find the largest possible number of defects using the fewest possible tests
Program testing types

- Black-box testing: Testers provide the system with inputs and observe the outputs
  - No code, internal data, documentation for system internals available
  - Specifications are used to derive test inputs

- White-box testing: Testers have access to the system design
  - Code, design documents, access to run-time internal data available
  - Code structure used to derive test inputs
Levels of testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing
Terminology

- Program - Sort
  - Collection of functions (C) or classes (Java)

- Specification
  - Input: $p$ - array of $n$ integers, $n>0$
  - Output: $q$ - array of $n$ integers such that
    - $q[0] \leq q[1] \leq ... \leq q[n]$
    - Elements in $q$ are a permutation of elements in $p$, which are unchanged
  - Description of requirements of a program
More terminology

- **Test input**
  - A set of values given as input to a program (along with the expected output)
  - Includes environment variables
  - \{2, 3, 6, 5, 4\}

- **Test case**
  - Test input with expected output

- **Test set**
  - A set of test cases
Test oracle

- Function that determines whether or not the results of executing a program under test are as per the program’s specifications
- Why is this hard?
Categories of errors

- Faults in ordinary algorithms
  - Incorrect logical conditions
  - Performing a calculation in the wrong part of a control construct
  - Not terminating a loop or recursion
  - Not setting up the correct preconditions for an algorithm
  - Not handling null conditions
  - Off-by-one errors
  - Operator precedence errors
  - Use of inappropriate standard algorithms (e.g., inefficient bubble sort)
Categories of errors

- Faults in numerical algorithms
  - Not using enough bits or digits
  - Not using enough places after the decimal point or significant figures
  - Ordering operations poorly so errors build up
  - Assuming a floating point value will be exactly equal to some other value
Categories of errors

- Faults in timing and coordination
  - Deadlock and livelock
  - Critical races
- Faults in handling stress and unusual conditions
  - Insufficient throughput or response time on minimal configurations
  - Incompatibility with specific configurations of hardware or software
  - Defects in handling peak loads or missing resources
  - Inappropriate management of resources
  - Defects in the process of recovering from a crash
Formal test plans

- A test plan is a document that contains a complete set of test cases for a system
  - Along with other information about the testing process.
- The test plan is one of the standard forms of documentation.
- If a project does not have a test plan:
  - Testing will inevitably be done in an ad-hoc manner.
  - Leading to poor quality software.
- The test plan should be written long before the testing starts.
- You can start to develop the test plan once you have developed the requirements.
What should be in a test plan

A. Identification and classification:
   - Number each test case; give a descriptive title.
   - Clearly indicate the system, subsystem or module being tested.
   - Indicate the importance of the test case.

B. Instructions:
   - Tell the tester exactly what to do.
   - The tester should not normally have to refer to any documentation in order to execute the instructions.

C. Expected result:
   - Tells the tester what the system should do in response to the instructions.
   - The tester reports a failure if the expected result is not encountered.

D. Cleanup (when needed):
   - Tells the tester how to make the system go ‘back to normal’ or shut down after the test.
Blackbox testing
Input domains

- Identify the input domain of $P$.
  - The set of all valid inputs that a program $P$ can expect is the input domain of $P$.
- Execute $P$ against each element of the input domain.
- For each execution of $P$, check if $P$ generates the correct output as per its specification $S$.
- <<Class exercise – example from sorting>>
Input domains

- The *size* of an input domain is the number of elements in it.
- An input domain could be finite or infinite.
- Finite input domains may be large!
- Exhaustive testing is not feasible.
- It is inappropriate to test by *brute force*, using *every possible* input value.
  - Takes a huge amount of time
  - Is impractical
  - Is pointless!
Equivalence classes

- You should divide the possible inputs into groups which you believe will be treated similarly by all algorithms.
  - Such groups are called \textit{equivalence classes}.

- A tester needs only to run one test per equivalence class

- The tester has to
  - understand the required input,
  - appreciate how the software may have been designed
Examples of equivalence classes

- Valid input is a month number (1-12)
  - Equivalence classes are: \([-\infty..0], [1..12], [13.. \infty]\)

- Valid input is one of ten strings representing a type of fuel
  - Equivalence classes are
    - 10 classes, one for each string
    - A class representing all other strings
Partition testing (or equivalence class testing)

- You should first make sure that at least one test is run with every equivalence class of every individual input.
- You should also test all combinations where an input is likely to affect the interpretation of another.
- You should test a few other random combinations of equivalence classes.
Boundary value testing

- More errors in software occur at the boundaries of equivalence classes

- The idea of equivalence class testing should be expanded to specifically test values at the extremes of each equivalence class
  - E.g. The number 0 often causes problems

- Example: If the valid input is a month number (1-12)
  - Test equivalence classes as before
  - Test 0, 1, 12 and 13 as well as very large positive and negative values
Unit testing classes with JUnit

Using JUnit

- Integrated with Eclipse – several advantages
  - Environment variables set up automagically
  - Execution tied with the Eclipse debugger
  - Test driver skeleton generated – just fill in test cases

- As a separate tool/framework
  - Install, setup classpath, run command line, etc, etc

- Either way, **YOU** write the test cases
JUnit as an Eclipse plugin

- Detailed instructions available at O’Reilly’s OnJava site:

- Run Eclipse IDE.

- Create a new workplace project
  - Click **File -> New -> Project**
  - Choose **Java** and click **Next**.
  - Type in a project name -- for example, ProjectWithJUnit.
  - Click **Finish**.
  - The new project will be generated in your IDE.
Including JUnit in Eclipse build path

Configure the Eclipse IDE, so it will add the JUnit library to the build path. This step isn’t really needed. If you don’t do this, Eclipse will ask you later when you create your first JUnit test.

- Click on **Project -> Properties**
- Select **Java Build Path, Libraries**
- Click **Add External JARs** and browse to directory where your JUnit is stored.
- Pick `junit.jar` and click **Open**. You will see that JUnit will appear on your screen in the list of libraries.
- By clicking **Okay** you will force Eclipse to rebuild all build paths.
Writing test cases in JUnit (Eclipse)

- Right-click on the ProjectWithJUnit title
- Select **New -> Other**
- Expand the "Java" selection, and choose **JUnit**.
- On the right column of the dialog, choose **Test Case**
- Click **Next**.
- Run the JUnit file as an application/debug mode.
Steps for using JUnit

- Simple framework to write repeatable tests.
- Write a test:
  - Import `junit.framework.*`
  - Extends class `junit.framework.TestCase`
  - Modify the following methods:
    - `protected void setUp()` – set up the fixture of the test.
    - `protected void tearDown()` – release resources allocated in `setUp()`.
Steps (contd)

- Write test methods for the test case (method name should begin with `test`):
  
  ```java
  public void testMoney(){ ...}
  ```
  
  - Use the following method for test:
    
    - `assertEqual(Object, Object)` – check if two objects are equal
    
    - `assertTrue(boolean)` – check if boolean expression is true

- Run the test using three different test runners:
  
  ```java
  java junit.awtui.TestRunner ClassName
  java junit.swingui.TestRunner ClassName
  java junit.textui.TestRunner ClassName
  ```
Example

- MoneyTest example can be found in junit directory:

  - Source file:  $JUNITHOME/junit/samples/money
  - Document:
    $JUNITHOME/doc/testinfected/testing.htm
What to do and not to do

☐ Do’s:

- Create new objects using constructors that are known to be correct.
- Use the equals() method if it is known to be correct.

☐ Don’ts:

- Suppose you want to test method $\text{foo}()$. **DO NOT** use methods $\text{bar}()$ and $\text{foo}()$ in the same test case if $\text{bar}()$ is not known to be correct.
  - If the test failed, you don’t know if it is because of $\text{foo}()$ or $\text{bar}()$. 
Example

- Class Under Test: LinkedList
- Constructors known to be correct:
  - LinkedList() – empty list with capacity 10
  - LinkedList(int c) – empty list with capacity c
  - LinkedList(int[] l) – list with numbers obtained from l, capacity equal to the length of l
- Test set up – need a list with numbers [1]–>[2]–>[3]–>[4]
  - WRONG method:
    - LinkedList l = new LinkedList(4);
    - l.add(1); l.add(2); l.add(3); l.add(4); // add may be faulty
  - CORRECT method:
    - int[] list = {1, 2, 3, 4};
    - LinkedList l = new LinkedList(list);
Whitebox testing
Testing to code

- White-box testing
- Glass-box testing
- Logic-driven testing
- Path-oriented testing
- Specifications are not used to generate the test cases, but used to check the outputs
- Test "to code," i.e. the internal structure of the code is important
Structural testing

- Intent is to exercise the different programming structures and data structures used in the program
- Intent is *not* to exercise all the different input and output conditions
  - *though this may be a by-product*
- Achieve test cases that force the desired “coverage” of different structures
- Criteria are formal and precise
Control-flow based criteria

- Statement coverage
  - Run a series of test cases and ensure that every statement is executed at least once
  - Simplest form of glass box testing
  - What is the weakness? Consider the example:

```c
int abs (int x) {
    if (x >= 0)
        x = 0 - x;
    return x;
}
```

Test inputs:
- x = 0: What is coverage?
- x = 5: What is coverage?
Statement coverage

- Not very strong, may leave errors undetected.
- Examples:
  - if statement without else part
  - conjunctions of predicates
- In all these cases, all branches were probably not exercised.
- Can you think of a better criterion based on the above observation?
Branch coverage

- Require that every decision is evaluated to true and false values at least once during testing
- Branch coverage implies statement coverage
  - Each statement is part of some branch
Control flow graph of a program

- Let $G$ be the graph of a program $P$.
- Node:
  - Represents a block of statements that are always executed together
- Edge $(i, j)$ from node $i$ to node $j$:
  - Represents a possible transfer of control after executing the last statement of the block represented by node $i$ to the first statement in the block represented by node $j$. 
Control flow graph of a program

- Start node:
  - Node corresponding to a block whose first statement is the start statement of P.

- Exit node:
  - Node corresponding to a block whose last statement is an exit statement of P.

- Path:
  - Finite sequence of nodes \((n_1, n_2, ..., n_k)\), \(k > 1\) such that there is an edge \((n_i, n_{i+1})\) for all nodes \(n_i\) in the sequence, except the last node \(n_k\).
Control flow graph of a program

- Complete Path:
  - Path whose first node is a start node and the last node is an exit node.

- All-nodes criterion (statement coverage)
- All-edges criterion (branch coverage)
An example ($x^y$)

1. `scanf(x, y); if(y < 0)`
2. `pow = 0 - y;`
3. `else pow = y;`
4. `z = 1.0;`
5. `while(pow != 0)`
6. `{ z = z * x; pow = pow - 1; }`
7. `if ( y < 0 )`
8. `z = 1.0/z;`
9. `printf(z);`
Control Flow Graph of example
Problems with branch coverage

- What if a decision has many conditions (using *and*, *or*)
- Decision may evaluate to true or false without actually exercising all the conditions

```java
int check (int x) {
    if ((x >= 5) && (x <= 200))
        return TRUE;
    return FALSE;
}
```

Test inputs:
- x = 5:
- x = -5:

Error (should be 100)
Solution?

- Require all individual conditions to evaluate to true and false
- Problem:
  - Even if individual conditions evaluate to true and false, the decision may not get both true and false values
- Solution:
  - Require both decision / condition coverage!!
- Still there is a problem.
Path testing

- Some errors are related to some *combinations of branches*.
- Presence revealed by an execution of a path that includes those branches.
- Solution:
  - Require all possible paths in the CFG to be executed during testing.
  - Path-coverage criterion, or all-paths criterion
  - Path coverage implies branch coverage
Problems with path testing

- With each predicate, there is a combinatorial explosion in the number of paths.
- Potentially infinite number of paths because of loops.
- Not all paths a feasible, i.e., there may not be inputs for which the path is executed.
- Suppose that somehow, we satisfied this criterion. Would we find all the faults?
More problems

- A path is tested only if it is present!!
- Consider the example:

  ```java
  double logBase19 (double x) {
    return ln(x)/ln(19);
  }
  ```

- Missing condition is:

  ```java
  if(x > 0)
  ```
More problems

- It is possible to test every path without detecting the fault in the product.

- Function to test the equality of 3 integers:
  - (Fallacious) assumption:
    - If the average of the 3 numbers is equal to the first, then they are equal.

```java
boolean areEqual(int x, int y, int z) {
    if ((x+y+z)/3 == x) return TRUE;
    else return FALSE;
}
```

- Exercise: Think of test cases.