Dependable Software Systems

Topics in Software Testing

Material drawn from [Beizer, Sommerville]

CS576 (Introduction)
**Software Testing**

- Software testing is a critical element of software quality assurance and represents the ultimate review of:
  - specification
  - design
  - coding
- Software life-cycle models (*e.g.*, waterfall) frequently include software testing as a separate phase that follows implementation!
Software Testing (Cont’d)

• Contrary to life-cycle models, testing is an activity that must be carried out throughout the life-cycle.

• It is not enough to test the end product of each phase. Ideally, testing occurs during each phase.
Software Testing Terminology

- **Error:** A measure of the difference between the actual and the ideal.
- **Fault:** A condition that causes a system to fail in performing its required function.
- **Failure:** The inability of a system or component to perform a required function according to its specifications.
- **Debugging:** The activity by which faults are identified and rectified.
Software Testing Myths

- If we were really good at programming, there would be no bugs to catch. There are bugs because we are bad at what we do.
- Testing implies an admission of failure.
- Tedium of testing is a punishment for our mistakes.
Software Testing Myths (Cont’d)

• All we need to do is:
  – concentrate
  – use structured programming
  – use OO methods
  – use a good programming language
  – ...

**Software Testing Reality**

- Human beings make mistakes, especially when asked to create complex artifacts such as software systems.
- Studies show that even good programs have 1-3 bugs per 100 lines of code.
- People who claim that they write bug-free software probably haven’t programmed much.
Goals of Testing

• Discover and prevent bugs.
• The act of designing tests is one of the best bug preventers known. (Test, then code philosophy)
• The thinking that must be done to create a useful test can discover and eliminate bugs in all stages of software development.
• However, bugs will always slip by, as even our test designs will sometimes be buggy.
Phases in a Testers Mental Life

- Testing *is* debugging.
- The purpose of testing is to show that the software works.
- The purpose of testing is to show that the software doesn’t work.
- The purpose of testing is to reduce the risk of failure to an acceptable level.
Testing Isn’t Everything

- Other methods for improving software reliability are:
  - **Inspection methods**: Walkthroughs, formal inspections, code reading.
  - **Design style**: Criteria used by programmers to define what they mean by a “good program”.
  - **Static analysis**: Compilers take over mundane tasks such as type checking.
  - **Good Programming Languages and Tools**: Can help reduce certain kinds of bugs (e.g., Lint).
Testing Versus Debugging

- The **purpose of testing** is to show that a program has bugs.
- The **purpose of debugging** is to find the faults that led to the program’s failure and to design and implement the program changes that correct the faults.
- Testing is a demonstration of failure or apparent correctness.
- Debugging is a deductive process.
Testing Versus Debugging (Cont’d)

- Testing proves a programmer’s failure.
- Debugging is a programmer’s vindication.
- Testing can be automated to a large extent.
- Automatic debugging is still a dream.
- Much of testing can be done without design knowledge (by an outsider).
- Debugging is impossible without detailed design knowledge (by an insider).
**Function Versus Structure**

- **Functional Testing:**
  - Program is treated as a black box.
  - Program is subjected to inputs, and its outputs are verified for conformance to specified behavior.
  - Implementation details do not matter.
  - Takes a user’s point of view.
  - In principle, can detect all bugs in an infinite amount of time.
Function Versus Structure (Cont’d)

- **Structural Testing:**
  - Aims at exercising the different control and data structures used in the program.
  - Criteria are precise as they are based on program structures (i.e., are quite precise).
  - Looks at implementation details.
  - Takes a developer’s point of view.
  - Is inherently finite but cannot detect all faults.
**Designer Versus Tester**

- The designer and the tester can be completely separated if only functional testing is performed.
- In structural testing the tester, like the designer, has intimate knowledge of the structure of the program.
Designer Versus Tester (Cont’d)

• The more the tester knows about the design, the more likely he will eliminate useless tests (functional differences handled by the same code).

• Testers that have design knowledge may have the same misconceptions as the designer.
Designer Versus Tester (Cont’d)

• Lack of design knowledge may help the tester to develop test cases that a designer would never have thought of.

• Lack of design knowledge may result in inefficient testing and blindness to missing functions and strange cases.
Small Versus Large Systems

• For small systems with one user, quality assurance may not be a major concern.
• As systems scale up in size and number of users, quality assurance becomes more of a concern.
• As systems dramatically scale up in size (e.g., millions of lines of code), our quality criteria may have to change as exhaustive testing may not be economically possible.
  – E.g., a 75% code coverage may be acceptable.
Programs and their Environment

• A program’s environment is the hardware and systems software required to make it run.

• Programmers should learn early in their careers that it is not smart to blame the environment for bugs.

• Bugs in the environment are rare because most bugs have been found over a long period of usage by a large number of users.
**Myths About Bugs**

- **Benign Bug Hypothesis:** Bugs are nice, tame, and logical.

- **Bug Locality Hypothesis:** A bug discovered within a component affects only that component’s behavior.

- **Control Bug Dominance:** Most bugs are in the control structure of programs.

- **Corrections Abide:** A corrected bug remains correct.
Myths About Bugs (Cont’d)

- **Silver Bullets:** A language, design method, environment grants immunity from bugs.
- **Sadism Suffices:** All bugs can be caught using low cunning and intuition. (Only easy bugs can be caught this way.)
Test Cases

• Test cases are formal procedures:
  – inputs are prepared
  – outcomes are predicted
  – tests are documented
  – commands are executed
  – results are observed

• All of these steps are subject to mistakes.

• Tests may have bugs themselves.
Levels of Testing

- **Unit Testing:** A unit is the smallest testable piece of software (*e.g.*, function).
- **Component Testing:** A component is an integrated aggregate of one or more units (*e.g.*, module).
- **Integration Testing:** This testing is used to demonstrate that a combination of successfully tested components has bugs.
Levels of Testing (Cont’d)

- **System Testing**: A system is a very large component. System testing includes testing for:
  - performance
  - security
  - system recovery from failure
Testing Oracles

• A testing oracle is any program, process, or body of data that specifies the expected outcome of a set of tests.

• Oracles are often defined as a set of input/expected outcome pairs.
Sources of Testing Oracles

- **Regression Test Suites:** Test software using the test suites developed for previous versions of the same software.

- **Purchased Test Suites:** Highly standardized software (compilers, mathematical routines) often have commercially available test suites.

- **Existing Program:** A working, trusted, program that is being re-hosted to a new language or O/S.
Is Complete Testing Possible?

- **NO.** Complete testing is both practically and theoretically impossible for non-trivial software.
Complete Functional Testing

- A complete functional test would consist of subjecting a program to all possible input streams.
- Even if the program has an input stream of 10 characters, it would require $2^{80}$ tests.
- At 1 microsecond/test, exhaustive functional testing would require more time than twice the current estimated age of the universe!
Complete Structural Testing

• One should design enough tests to ensure that every path is executed at least once.
• What if the loops never terminate?
• Even if loops terminate, the number of paths may be too large.
How About Correctness Proofs?

• Requirements are specified in a formal language.
• Each program statement is used in a step of an inductive proof.
• In practice, such proofs are time consuming and expensive.
How About Correctness Proofs? (Cont’d)

- Proving the consistency and completeness of a specification is a provably unsolvable problem, in general.
- Proofs may have bugs.
A Theory of Program Testing

• A program is a function:
  \[ P : D \rightarrow R \]

• A program specification is the function:
  \[ S : D \rightarrow R \]

• Input domain is D, output range is R.

• Proving program correctness is the process of demonstrating that:
  \[ P(d) = S(d), \forall d \in D \]
A Theory of Program Testing (Cont’d)

• A program $P$ is correct with respect to the specification $S$ if the program behavior matches the specified behavior for all possible inputs in $D$.
• This is impossible to achieve for any reasonably complex program.
Software Testing is the process of demonstrating program correctness by selecting a subset of set $D$ called $T$ such that:

- $T$ is a finite subset of $D$.
- If $S(t)$ is incorrect for some $t$ in $T$, $P(t)$ must be incorrect also.
- If $S(t)$ is correct for some $t$ in $T$, $P(t)$ must be correct also.
A Theory of Program Testing (Cont’d)

- A testing strategy is a set of rules that outline:
  - the criteria by which $T$ is selected (test selection criteria)
  - the properties of the program that must be exercised (test coverage criteria)
Program Testing

- Can reveal the presence of faults NOT their absence.
- A successful test is a test which discovers one or more faults.
- Only validation technique for non-functional requirements.
- Should be used in conjunction with static verification.
Defect Testing

- The objective of defect testing is to discover defects in programs.
- A successful defect test is a test which causes a program to behave in an anomalous way.
- Tests show the presence not the absence of defects.
Testing Priorities

• Only exhaustive testing can show a program is free from defects. However, exhaustive testing is impossible.

• Tests should exercise a system’s capabilities rather than its components.

• Testing old capabilities is more important than testing new capabilities.

• Testing typical situations is more important than boundary value cases.
Test Data and Test Cases

• **Test data**: Inputs which have been devised to test the system.

• **Test cases**: Inputs to test the system and the predicted outputs from these inputs if the system operates according to its specification.
Testing Effectiveness

- In an experiment, black-box testing was found to be more effective than structural testing in discovering defects.
Black-box Testing

• Approach to testing where the program is considered as a “black-box”.
• The program test cases are based on specifications.
• Test planning can begin early in the software process.
Black-box Testing
Equivalence
Partitioning

CS576 (Introduction)
**Search Routine Specification**

**procedure** Search (Key : INTEGER ; T: array 1..N of INTEGER; Found : BOOLEAN; L: 1..N) ;

**Pre-condition**
-- the array has at least one element
1 <= N

**Post-condition**
-- the element is found and is referenced by L
( Found and T (L) = Key)
or
-- the element is not in the array
( not Found and not (exists i, 1 >= i >= N, T (i) = Key ))
Search Routine - Input Partitions

- Inputs which conform to the pre-conditions.
- Inputs where a pre-condition does not hold.
- Inputs where the key element is a member of the array.
- Inputs where the key element is not a member of the array.
Testing Guidelines (Arrays)

- Test software with arrays which have only a single value.
- Use arrays of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the array are accessed.
- Test with arrays of zero length (if allowed by programming language).
## Search Routine - Input Partitions

<table>
<thead>
<tr>
<th>Array</th>
<th>Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single value</td>
<td>In array</td>
</tr>
<tr>
<td>Single value</td>
<td>Not in array</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>First element in array</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Last element in array</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Middle element in array</td>
</tr>
<tr>
<td>More than 1 value</td>
<td>Not in array</td>
</tr>
</tbody>
</table>
# Search Routine - Test Cases

<table>
<thead>
<tr>
<th>Input array (T)</th>
<th>Key (Key)</th>
<th>Output (Found, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>false, ??</td>
</tr>
<tr>
<td>17, 29, 21, 23</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>41, 18, 9, 31, 30, 16, 45</td>
<td>45</td>
<td>true, 6</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 41, 38</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>21, 23, 29, 33, 38</td>
<td>25</td>
<td>false, ??</td>
</tr>
</tbody>
</table>
Structural Testing

- Sometime called white-box testing.
- Derivation of test cases according to program structure. Knowledge of the program is used to identify additional test cases.
- Objective is to exercise all program statements (not all path combinations).
White-box Testing

CS576 (Introduction)
class BinSearch {

    // This is an encapsulation of a binary search function that takes an array of
    // ordered objects and a key and returns an object with 2 attributes namely
    // index - the value of the array index
    // found - a boolean indicating whether or not the key is in the array
    // An object is returned because it is not possible in Java to pass basic types by
    // reference to a function and so return two values
    // the key is -1 if the element is not found

    public static void search ( int key, int [] elemArray, Result r )
    {
        int bottom = 0 ;
        int top = elemArray.length - 1 ;
        int mid ;
        r.found = false ; r.index = -1 ;
        while ( bottom <= top )
        {
            mid = (top + bottom) / 2 ;
            if (elemArray [mid] == key)
            {
                r.index = mid ;
                r.found = true ;
                return ;
            } // if part
            else
            {
                if (elemArray [mid] < key)
                    bottom = mid + 1 ;
                else
                    top = mid - 1 ;
            } //else part
        } //while loop
    } //search
} // BinSearch

Binary search (Java)
Binary search flow graph

1. bottom > top
2. while bottom <= top
3. if (elemArray[mid] == key
4. (if (elemArray[mid] < key
5. 6
6. 7
7. 8
8. 9
9.
Binary Search - Equivalence Partitions

- Pre-conditions satisfied, key element in array.
- Pre-conditions satisfied, key element not in array.
- Pre-conditions unsatisfied, key element in array.
- Pre-conditions unsatisfied, key element not in array.
- Input array has a single value.
- Input array has an even number of values.
- Input array has an odd number of values.
Binary Search Equivalence Partitions
## Binary Search - Test Cases

<table>
<thead>
<tr>
<th>Input array (T)</th>
<th>Key (Key)</th>
<th>Output (Found, L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>17</td>
<td>0</td>
<td>false, ??</td>
</tr>
<tr>
<td>17, 21, 23, 29</td>
<td>17</td>
<td>true, 1</td>
</tr>
<tr>
<td>9, 16, 18, 30, 31, 41, 45</td>
<td>45</td>
<td>true, 7</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 38, 41</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>17, 18, 21, 23, 29, 33, 38</td>
<td>21</td>
<td>true, 3</td>
</tr>
<tr>
<td>12, 18, 21, 23, 32</td>
<td>23</td>
<td>true, 4</td>
</tr>
<tr>
<td>21, 23, 29, 33, 38</td>
<td>25</td>
<td>false, ??</td>
</tr>
</tbody>
</table>