Software Testing

Note: This document aims to cover a subset of the Software Engineering modules. Do not assume that CS4218 – Software Testing covers only the information in this document.

Document Outline

Introduction to Software Testing

Given the increasing complexity of the software in the market, software testing is undoubtedly an important aspect in the implementation phase. One might adopt random testing, which generates test cases randomly based on the requirements and input descriptions. This method is easy to generate test cases, but is it timely and effective to cover all possible permutations and conditions? Thus, the reliability of the software is hard to determine through random testing. Typically, a small sample of the enormously large software is tested so that it is easier to localise any errors. However, is it easy to construct these samples? How should we construct them?

By reading this document, you will realise that there is a systematic process of testing a software. There are also various ways to generate all the possible test cases. The importance of continuous integration and automated testing will be elaborated as well. Lastly, the document illustrates some key ideas of debugging using an IDE.

Why is Software Testing important?

- Obtain measurable confidence
    - Coverage based – Line coverage, Branch coverage, Path coverage
    - Fault based – Mutation testing
    - Failure based – Representations by control flow graphs & symbolic executions
- Test for **comprehension** of requirements, for **completeness** and for **correctness**
- Maximise user experience

Terminologies

- Fault, Error and Failure

Fault	An anomaly in the source code of a program that may lead to an error.
Error	The runtime effect of executing a fault, which may result in a failure.
Failure	The manifestation of an error external to the program.

- Testing

Test Case	A group of input values that cause a program to take some defined action
Test Suite	A collection of test cases
Test Oracle	A mechanism for determining if the actual behaviour of a test case execution matches the expected behaviour
Test Effectiveness	The degree to which testing reveals faults or achieves other objectives.
Test Plan	A document describing the scope, approach, resources and schedule of intended activity.

- Testing vs. Debugging

  Testing reveals faults, while debugging is used to remove a fault. Debugging is part of testing.

- Random vs. Systematic Testing

Random Testing	• Randomly pick possible inputs • Minimises programmer bias • Treat all inputs as equally valuable
Systematic Testing	• Attempts to select inputs which are very valuable • Typically by picking representative values which are pertinent to fail /pass

Types of Testing

Unit Testing	Testing a single, simple component in an isolated environment. (e.g. procedure, class) Problem: Modules might involve other interactions. • Calling procedures in other modules • Receiving procedure calls from other modules • Sharing variables Solution: Drivers and Stubs • Driver: A program that calls the interface procedures of the module being tested and reports the results • Stub: A program that has the same interface as a module that is being used by the module being tested, but is simpler.
Integration Testing	Testing parts of the system by combining the modules. Problem: What should be the order to combine modules into partial system? Solution: Approaches to integration testing • Bottom-up approach (Integrate drivers) • Top-down approach (Integrate stubs) • Big-bang approach • Sandwich (Both bottom-up & top-down)
System Testing	Testing based on requirements specifications.
Acceptance Testing	Testing based on use cases and use case scenarios. • Alpha Testing – Performed within the developers • Beta Testing – Performed by a select target audience • Stress Testing – Push system to extreme situations (e.g. data, users, performance)
Regression Testing	Testing based on previous preserved test cases, when a change is made. Automated regression testing • Re-run test cases that have been affected by a change. • All relevant test cases are run after each change.

Approaches to Testing

Black Box Testing (Functional Testing)	Testing which ignores the internal mechanism of a system / component, and focuses solely on the outputs generated in response to the selected inputs and execution conditions. Uses the requirements to partition the input space, and test each category and boundaries between categories. Types of errors detected: • Incorrect / Missing functions • Interface errors • Errors in data structure / external database access • Performance errors • Initialization & termination errors
White Box Testing (Structural Testing)	Testing which takes into account the internal mechanism of a system or a component, with respect to some well defined coverage criterion. Coverage Criterion: • Line Coverage – Test every statement of code • Branch Coverage – Test every line, and every branch on multi-branch lines • Path Coverage – Test every path through the program, from entry to exit Fault-based criteria: • Mutation Testing[#] – A slightly changed version of the original program Failure-based criteria: • Control Flow Graph – A type of program representation • Symbolic Execution[#] – Method to analyse a program to determine what inputs cause each part of a program to execute [#] Not covered in this document

Methods to generate test cases

Functional Testing – Part 1

Deriving test cases from program specifications.

Specification Based	Category Partition Testing 1. Decompose the specification into independently testable features (i.e. Check that <input> is a valid <output>) 2. Select representative values based on input and/or model (e.g. boundary values, equivalence partitioning) 3. Form test specifications (through combinatorial approach or model behaviours) 4. Produce and execute actual tests
Model Based	State Machine Models (e.g. UML diagram) • Object States that can be modified by methods (i.e. Every state should be covered by at least 1 test case) • Transitions between 1 state to another (Every transition between states should be covered by at least 1 test case) Decision Structure Models (e.g. flow charts, decision trees) 1. Come up with a Decision Table a. Rows – Basic Condition b. Columns – Combinations of Basic Conditions c. Output (Last row of table) – Expected output for each combination (that column) 2. Generate test cases for each column. Fill up "don't care" values as much as possible without violating constraints. 3. Modified Condition Decision Coverage (MC-DC) (Aim - Flipping 1 output from a column should lead to another output) a. Systematically flip 1 value from each column – see if the column can be merged with an existing column without violating any constraint. b. If not, create an additional column – output should be different from the original column whose value was flipped. (Note: Output can represent "error", meaning that the flipped column is infeasible)

Functional Testing – Part 2

Generate various combinations of values / models from different testable features.
Minimise combinatorial explosion.

Category Partition Testing
 	
Exhaustive, systematic approach to manually identify characteristics and values, as well as to automatically generate combinations.
	1. Identify independently testable units and categories
	  a. For each feature, identify parameters and environment elements.
	  b. For each parameter and environment element, identify categories.
	2. Identify relevant values for each category:
	  a. Boundary values (Valid)
	  i. Extreme values within a class
	  ii. Values outside but as close as possible to the class
	  iii. Non-extreme values in the class
	  iv. Special values (e.g. NULL, "")
	  b. Special & Error values (Invalid)
	3. Introduce constraints for 2a and 2b to:
	  a. Rule out impossible combinations
	  b. Minimise the size of test suite
	4. Enumerate the test cases

Pairwise Testing
 	
Combine values systematically but not exhaustively.
 	
Generate combinations that efficiently cover all pairs (triples) of classes.
 	
Rationale: Most failures are triggered by single values or combinations of few values.
	1. Sort the values in a main table such that:
	  a. 1st variable: one with most number of values
	  b. Last variable: One with the least number of values
	2. If there are constraints, generate cartesian product with the affected variables in separate tables.
	3. For the 1st column, if the 1st variable has 3 values and 2nd variable has 2 values, the 3 values should be written 2 times each.
	4. Consider the values pairwise in a horizontal manner (by looking at main & separate tables), and fill in the values column by column accordingly.
	5. Identify the missing pairwise products, adjust the values starting from this row onwards - 1st variable, 2nd value.
	6. Check the values pairwise before proceeding to the next column.

Catalog Based Testing

Exhaustive, systematic approach to identify attribute values based on test designer's experience.

1. Identify the following elements from the specification:
    a. Pre-conditions – Conditions that must be true before execution
        i. Validated pre-conditions by the system
        ii. Assumed pre-conditions by the system
    b. Post-conditions – Results of the execution
    c. Definitions – Abbreviations
    d. Variables – Elements used for the computation
    e. Operations – Main operations on variables & input
2. Derive a first set of test specifications from pre-conditions, post-conditions and definitions.
    a. Validated pre-conditions
    Simple preconditions: Valid or invalid inputs
    Compound preconditions: Apply MC-DC criterion
    b. Assumed pre-conditions
    Apply MC-DC only to "OR" preconditions
    c. Post-conditions & Definitions
    If given as conditional expressions, consider conditions as if they were validated pre-conditions.
3. Complete the set of test case specifications using test catalogs.
    a. Scan the catalog sequentially
    b. Apply the catalog entry
    c. Delete redundant test cases

Sample catalog:

- Boolean
    - [in/out] true
    - [in/out] false
- Enumeration
    - [in/out] each enumerated value
    - [in] values outside of the enumerated set
- Range L..U
    - [in] L-1
    - [in/out] L
    - [in/out] A value between L and U
    - [in/out] U
    - [in] U+1
- Numeric Constant C
    - [in/out] C
    - [in] C - 1
    - [in] C + 1
    - [in] Any other constant in the same data type.
- Non-Numeric Constant C
    - [in/out] C
    - [in] Any other constant in the same data type
    - [in] Some other value of the same data type
- Sequence
    - [in/out] Empty
    - [in/out] A single element
    - [in/out] More than one element
    - [in/out] Maximum length (in bounded) or very large
    - [in] Longer than max length (if bounded)
    - [in] Incorrectly terminated
- Scan with action on element P
    - [in] P occurs at beginning of sequence
    - [in] P occurs in interior of sequence
    - [in] P occurs at end of sequence
    - [in] P appears twice in a row
    - [in] P does not occur in sequence

Structural Testing

A program can be represented using a Control Flow Graph (CFG).

	Intra-Procedural CFG	**Inter-Procedural CFG (Call Graphs)**
Nodes	Maximum code region with 1 entry point & 1 exit point	Procedures (e.g. methods, functions)
Directed Edges	Flow from 1 code region to another	Call relations

Identifies cases that may not be identified from specifications alone.

- Natural differences between specifications and implementation
- Flaws in the software or its development process

Note: Executing all control flow statements does not guarantee all faults are found – Errors might be masked in the code!

Advice: Create functional test suite first, then measure structural coverage to identify missing cases.

	Coverage	**Rationale**
Statement Testing	$\frac{\text{No. of executed statements}}{\text{Total no. of statements}}$	A fault in a statement can only be revealed by executing the faulty statement.
Branch Testing	$\frac{\text{No. of executed branches}}{\text{Total no. of branches}}$	Traversing all edges → All nodes are visited. (Converse is not true)
Condition Testing	$\frac{\text{No. of truth values consumed by all basic conditions}}{2 \times \text{No. of basic conditions}}$	Apply MC-DC where necessary
Path Testing	$\frac{\text{No. of executed paths}}{\text{Total no. of paths}}$ The total number of paths might be infinitely huge. • Limit the number of loop traversals • Limit the path length to be traversed • Limit the dependencies among selected paths	Sequences of branches might cause faults.

Note: Sometimes criteria may not be 100% satisfiable. Why?

Automated Testing

Test driver is a module written specifically to invoke the software with test inputs and verifying the expected output with actual output.

JUnit is a tool for automated testing of Java programs. Similar tools are available for other languages. Most modern IDEs (e.g. IntelliJ, Eclipse) come packaged with integrated support for testing tools.

Test-Driven Development (TDD)

Typically test cases are written after the code implementation is completed. However, TDD advocates the converse.

1. Decide what behaviour to implement
2. Write test cases to exhibit the behaviour → Run these test cases to see them fail
3. Implement code behaviour
4. Run the test cases again → Keep modifying the code & re-run test cases until they all pass
5. Refactor code to improve code quality
6. Repeat the cycle for each small unit of behaviour

Build Automation

Build refers to the process that converts code and other assets into a final, consumable software product. Some of the build steps such as to import assets, compile source codes, link and package into relevant formats can be automated in most IDEs (e.g. IntelliJ, Eclipse).

These steps are repeatable automatically, and can be performed at any time without knowing any information other than what is stored in the central repository.

Build Automation Tools: Maven, Gradle

Continuous Integration (CI)

CI is an extreme application of build automation in which integration, building and testing happens automatically after each code change.

CI Tools: Travis CI, Jenkins

Debugging Using an IDE

https://www.jetbrains.com/help/idea/debugging-code.html

1. Set a breakpoint, which are source code markers used to trigger actions during a debugging session.
2. Run in Debug mode.
3. The programs runs from the start until it hits a breakpoint. The line of code that contains a set breakpoint, is marked with a colour stripe; once such line of code is reached, the marking stripe changes to another colour.
4. Notice the variables & its corresponding values in the debugger view. As you complete one of the following, the view is updated up till that line of code.
    a. Step Into – Step deeper into the code implementation in a method
    b. Step Over – Jump over the method, with inner code implementation ran.
5. Remove the breakpoint when it is no longer useful.

End of Document