Overview of Software Testing Methods and Applications

Glenn Skawski
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, Arizona, Maricopa County
gskawski@asu.edu

1 Introduction

Software testing means to validate and verify software systems. The purpose for doing so arose in the 1960s in response to growing software development problems [1]. The observed problems can be generally addressed by answering the questions "*Are we building the right product?*" (validation) and "*Are we building the product correctly?*" (verification) at all levels of the software development process [1]. Validation is addressed by building software from the perspective, specifications, and/or needs of a user / client while also incorporating established software testing standards and ethics. The first step of the software development process is translating and organizing specifications into a combination of objective representations (requirement list, safety criteria) and abstractions (use-cases, scenarios). Verification views and tests software from a multi-dimensional perspective with dimensions categorized into functional and non-functional requirements. The goal of verification is to signal when software is ready for release based on a defined exit criteria of measurable quality goals based on an accepted confidence level. Verification is approached through software testing generally described as investigations to provide stakeholders with information on software quality [2]. Test investigations vary in terms of testing aspects of testing levels, testing types, granularity, coverage, aspect being tested, approach, etc. Test results such as defects, code review, speeds, security gaps, etc. are used to improve software and determining when software is ready for release. Provided testing all aspects of software is impractical, testing needs to be guided by assigning risk to system features based on failure likelihood and severity and prioritizing testing accordingly. The paper herein describes the approach and results of several testing methods that is an introduction to the breadth of the testing process from planning to testing to reliability estimation.

2 Specification-Based Testing

Specification-based testing was conducted on a Java file with the purpose of locating 10 requirement-based seeded defects. The goal is to check system functionality by comparing inputs and resulting outputs to their respective requirements. Specific tests include equivalence partitioning, boundary value analysis, and cause & effect analysis. Each test case is an input set systematically created based on the test method, specifically:

- *Equivalence Partitioning*: Set of valid and invalid inputs based on equivalence partitions specific to the requirements. Input sets are either all valid or contain one invalid input;
- *Boundary Testing*: Input set includes values below, on, and above the equivalence partitions input and output edges; and
- *Cause & Effect Analysis*: Set of valid inputs tested in combination with one another based on requirement dependencies.

2.1 Specification-Based Testing Results

Executed test cases are tracked using a test case matrix and traceability between test case identified defects and requirements are tracked through a defect tracking sheet. An example of a test case record where a defect was identified and the record mapping defect to the applicable requirement is provided in Table 1.

Table 1. Test Case & Defect Tracking Example

Test Case Example				
Name	***Age***	***User Status***	***Reward Status***	***Season***
Glenn	32	Returning	Bronze	Spring
Product Category		***Rating***	***Pass/Fail***	***Defect #***
Electronics		1	Fail	1
Defect Tracking Example				
Defect Number & Description				***Requirement***
1: Test case inputs output 5% when 10% is required.				9.4.1

3 Design of Experiments Pairwise Testing

Design of experiments (DOE) pairwise test case generation was conducted from a set of provided specifications for an application. The purpose is to aid test case development by creating a set of test cases that includes every possible 2-wise combination of inputs. The reason being: 1) to reduce the number of test cases from all possible input combinations (800 for this application) to a more manageable test case volume [3]; and 2) by examining a systems behavior when inputs are systematically in combination with each other. While pairwise testing does not cover all possible test cases, confidence can be considered high as previous studies have demonstrated that a significant percent of system faults can still be detected [4]. Pairwise test cases were created using Inductive's Pairwiser-Pairwise Testing Tool [5].

3.1 DOE Pairwise Testing Results

The pairwise analysis identified 35 input sets (test cases) which satisfies the condition that all possible 2-wise combination of inputs

is included at least once in the 35 test cases. An example of a created pairwise test case is provided in Table 2.

Table 2. Pairwise Test Case Example

Test Case Example				
Phone	***Parallel Task***	***Connectivity***	***Memory***	***Battery Level***
iPhone 8	Yes	Wireless	2 GB	80 – 100%

4 Control Flow Testing

Control flow testing was conducted on the Vending Machine Java program. The purpose is to aid in the development of and check effectiveness of dynamic test cases. The approach is to analyze the internal code structure and create test cases that result in control flow coverage which includes four coverage levels; statement, decision / condition, and multiple condition. Control flow testing is measured as a percent of the coverage level executed from test cases compared to the total number of control flow items. Coverage results that do not meet objectives signifies undocumented requirements and/or poor test cases.

This example looks to maximize statement and decision coverage of the Vending Machine program by devising a set of test cases. The EclEmma JaCoCo tool was used to carry out the control flow analysis [6]. JaCoCo has the added user advantage of providing several options for viewing coverage and coverage progress. One such view colors each line according to the line's achieved coverage as test cases are executed. Color designations include red (no coverage), yellow (partial coverage), and green (full coverage).

4.1 Control Flow Testing Results

Control flow test cases were developed by evaluating the provided code and choosing input combinations that would most effectively achieve coverage. The final statement coverage of 100% and the final decision coverage of 93% was achieved after executing seven test cases. 100% decision coverage could not be achieved because the last "else" statement cannot be entered unless the input amount is less than the cost of the item selected. As coffee is the most expensive item any input less than all of the three costs will always be less than the cost of coffee. Therefore, it is infeasible to go the path of else being True and the following if being False.

5 Static Analysis

Data flow testing is a static analysis method to identify data flow anomalies by modeling data flow through where variables are defined and used. The purpose is aid test case development for dynamic analysis based on anomalies detected. IntelliJ's Data Flow tool was used to conduct static analysis of the provided StaticAnalysis program to locate two seeded data flow anomalies. The tool detects error-types and data flow inconsistencies that are not "compilation errors" such as variables defined then redefined without being referenced, referencing undefined variable, and not using defined variable [7] [8].

5.1 Static Analysis Results

The tool identified the two variables "*weight*" and "*length*" in the StaticAnalysis() method as data flow anomalies which were defined in the program but never used.

6 Reliability Prediction Approach

Reliability is the probability that a software program operates for some given time period without software error [1]. The goal is to assess and inevitably demonstrate that reliability objectives have been met. Testing techniques include operational profile, error detection and recovery, and serviceability to guide test case development in order to target software aspects that impact reliability. Reliability testing begins at a designated entry point and continues throughout software development. Test data can take several forms interrelating defect count and time of occurrence. Test data is used to conduct reliability trend analysis and reliability growth model analysis. Both show how reliability changes over time. The combination of testing, trend analysis, and modeling results are used as evidence of when reliability objectives have been met [12]. Predicting reliability can be facilitated by the use of existing tools to transform reliability test data via trend analysis or reliability growth model. SoRel is one such tool that follows the general approach for predicting reliability and by the end can provide reliability measures and model validation to support reliability testing (Table 3).

Table 3. SoRel Reliability Growth Analysis Tool [13]

Step & Inputs	**Outputs**	**Assumptions**
1. Trend Analysis Select Trend Test: (arithmetic Mean (τ_k), Laplace (u)) Data Inputs: (inter-failure time, failures per unit time) Time / interval unit (k)	• Reliability trend graph of τ_k / u / $u(k)$ against k • Deduce if reliability is growing, decreasing, or stable	• τ_k: Results directly related to metrics inputted • u: Practically applied using significance level
2. Reliability Growth Model Select Growth Model: (hyperexponential, exponential, S-Shaped, double stochastic) Data Inputs: (see Step 1) Model Specific Inputs	• Reliability Measures (MTTF, failure intensity, cumulative failures, residual failure rate) • Retrodictive or predicative model validation	• Reliability growth model is selected based on trend analysis results

7 Risk Based Testing

A testing strategy was determined for four tasks; legacy code, outsourced code, new code, and APIs. Each task is distinct in its development history and functionality. Creating the testing strategy includes: 1) assigning priority to each task; 2) assigning testing intensity to each task; and 3) developing a contingency plan to mitigate risk. Provided that full testing of a system is impractical, the software testing strategy includes evaluating tasks according to their risk posed to users and the project. The evaluated risk for each task is used to assign testing priority when creating the schedule and testing intensity. Prioritizing tasks and identifying testing

intensity satisfies the best practice of focusing on high risks tasks by testing these early and more thoroughly [9]. Risk is the calculation of risk exposure (RE) and equals the product of a task's failure probability (likelihood) and its failure severity. For this example, the risk exposure levels (low, medium, and high) are determined based on the perceived likelihood (low, medium, and high) and severity (marginal, critical, and catastrophic) (Table 4). Each tasks likelihood and severity are qualitatively evaluated based on task descriptions provided in the project description. Priority order and testing intensity (marginal, average, maximum) is assigned according to each task's risk exposure.

Table 4. Risk Exposure (RE) Chart

Likelihood		Severity		
		Marginal	***Critical***	***Catastrophic***
	Low	Low	Medium	Medium
	Medium	Low	Medium	High
	High	Low	Medium	High

Also considered when prioritizing is evaluating the potential for a task to adversely impact the progress and effectiveness of testing [10]. A task determined to have this potential impact may receive higher testing priority.

In the event the testing strategy is insufficient at meeting the testing objectives, a contingency plan can be developed to mitigate risk through additional strategies to fulfill the testing objectives [11]. Contingencies are most applicable to highest risk tasks.

7.1 Risk Based Testing Results

This risk-based plan's priority and intensity results are summarized in Table 5. Supporting evidence for each task's priority and intensity conclusion is provided in the following sections by detailing each tasks likelihood, severity, and project impact findings from the provided project description.

Table 5. Test Case & Defect Tracking Example

Task	Likelihood, Severity, RE	Priority	Intensity
Legacy	High, Catastrophic, High	1	Maximum
New	Medium, Catastrophic, High	2	Maximum
APIs	High, Critical, Medium	3	Average
Outsourced	Low, Marginal, Low	4	Marginal

The contingency plan applies to testing legacy and new code. Its key features include adding contingency time to the schedule, avoid tests leading to combinatorial explosion, attempt a model-based test to facilitate auto-generated test creation, and/or regression testing.

8 Lessons Learned

Reflecting on the course material, I realize the practiced skills can be considered to have covered the testing development process with applications to real-world software development projects. Specifically, each phase of the test development process was touched on by at least one assignment. The test objectives phase as it relates to planning was covered in the assignment by creating a risk-based strategy for a set of tasks that included prioritized scheduling and testing intensity characterization. Another planning consideration that is part of the objectives phase is defining when to stop testing. Reliability testing is a vital aspect when software testing as it aids in identifying when to stop testing based on reliability metrics collected during testing and development. The objectives of when to stop testing will guide the planning of subsequent steps. Another aspect of the test objectives phase is determining what testing needs to accomplish. This was practiced in most of the assignments simply by disseminating directions into what I wanted to achieve for a given test. Class examples included finding 10 seeded defects using specification testing and finding data flow anomalies using static analysis. While the examples are simple in comparison to practical applications, they nonetheless are representative of this phase.

Following the objectives phase is the test design phase whereby we address how to meet the objectives. While this was straight forward as each assignment specified the test, in hindsight I can look outside the box of individual assignment directions and visualize how and why a variety of tests are used in conjunction to support each other and create a robust testing strategy that adequately addresses test objectives. This test network, as it relates to practiced tests, begins with the requirements list from the specification-based testing assignment. The test design phase would specify performing specification-based testing for the purpose of testing functional coverage. This was a practice in static testing as we were testing requirements from a list, but it can be seen how a behavioral testing format could also be applied. Then DOE pairwise testing could be applied to evaluate the quality of the original test cases. Testing is further strengthened through control flow and data flow testing where the former evaluates code coverage achieved by test cases and the latter facilitates dynamic test case development based on anomalies detected. DOE, control flow testing, and data flow analysis all provide unique feedback that can be fed back to functionality test methods strengthening the overall testing process. There is also the added benefit of these tests providing functional coverage information. This test network results in a cycle of testing to improve software, testing to improve tests, and updated tests repeating the cycle.

The phases writing and executing tests was practiced in several assignments. The standout lesson is the availability and range of testing tools available and that they should be taken advantage of in order to conduct testing efficiently while also being cautious by understanding tool assumptions, limitations, and confidence thresholds. Last is the maintenance phase which entails updating previous phases as new requirements come in and would mean circling back to the start updating phases accordingly.

This high-level review of lessons learned relating class assignments to the test development process illustrates the complex nature of software testing in terms of testing knowledge, software multi-dimensionality, inter-disciplinary collaboration, and planning. The degree to which a testing strategy succeeds will depend on satisfactorily addressing and having answers to these complexities in accordance with industry standards and best practices.

9 References

[1] Collofello, James. *CSE 565 Software Verification and Validation Testing Background History and Background*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/GhJEV/history-and-background.

[2] Kaner, Cem. Exploratory Testing. QAI, 17 Nov. 2006.

[3] Collofello, James. *CSE 565 Software Verification and Validation Design of Experiments*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/fyJ1C/design-of-experiments.

[4] Kuhn, Richard D., et al. "Combinatorial Coverage as an Aspect of Test Quality." *Journal of Defense Software Engineering*, vol. 28, no. 2, 31 Mar. 2015, pp. 19–23.

[5] "Pairwiser - Pairwise Testing and Test Generation Tool." *Pairwiser*, Inductive, 7 June 2020, inductive.no/pairwiser/.

[6] Coverage Counters." *JaCoCo*, Mountainminds GmbH & Co. KG, 2020, www.jacoco.org/jacoco/trunk/doc/counters.html.

[7] *Static Code Analysis*, Jetbrains, www.jetbrains.com/idea/docs/StaticCodeAnalysis.pdf.

[8] Collofello, James. *CSE 565 Software Verification and Validation Structural Based Testing Strategies Static Analysis*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/LkSR9/static-analysis

[9] Collofello, James. *CSE 565 Software Verification and Validation Test Management Part 1 Risk Based*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/BPXZr/risk-based-testing.

[10] Collofello, James. *CSE 565 Software Verification and Validation Test Documentation: Developer Mentality*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/Pj0BK/test-documentation-developer-mentality.

[11] Collofello, James. *CSE 565 Software Verification and Validation Specification of the Test Environment*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/WjMa1/specification-of-the-test-environment.

[12] Collofello, James. *CSE 565 Software Verification and Validation Testing Software Quality Characteristics – Part 2 Reliability Testing*. *Coursera*, Arizona State University, 2020, https://www.coursera.org/learn/565/lecture/9bejq/reliability-testing.

[13] K. Kanoun, M. Kaaniche, J. -. Laprie and S. Metge, "SoRel: A tool for reliability growth analysis and prediction from statistical failure data," *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, Toulouse, France, 1993, pp. 654-659, doi: 10.1109/FTCS.1993.627370.