AUTOMATIC CONCEPTUAL FEEDBACK ON SOFTWARE TESTS

A Paper
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Pranay Kumar

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Program:
Software Engineering

April 2017

Fargo, North Dakota

North Dakota State University
Graduate School

Title

AUTOMATIC CONCEPTUAL FEEDBACK ON SOFTWARE TESTS

By

Pranay Kumar

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Gursimran S. Walia
Chair

Dr. Kendall E. Nygard

Dr. Sudarshan K. Srinivasan

Approved:

04/03/2017	Dr. Brian Slator
Date	Department Chair

ABSTRACT

Students in introductory programming courses face many challenges. While the educators try their best to assist students with these challenges by adopting different approaches, majority of them end up overlooking the significance of software testing activities. However, these activities are crucial for any developer. This paper proposes and then develops a tool, Testing Tutor, developed by us to help improve student’s testing skills by providing them with conceptual feedback on their tests. We believe that such an approach will help students understand basic testing concepts and understand the reasons for their mistakes instead of offering them the option to simply tweak the source/test code at the erroneous location pointed out by test runner or coverage tool. This paper details the process of designing and building a prototype for such a tool and lessons learnt.

ACKNOWLEDGEMENTS

I would like to note the contributions and guidance of all those who have helped me along the way culminating in this paper. I'm grateful to my advisor Dr. Gursimran Walia, for his continuous guidance, support and patience not only while working on this paper but throughout my master's program journey as a professor and an advisor. His guidance and feedback has been very valuable and I'm certain it will enable me to be a better professional.

I would like to extend my sincerest thanks to Dr. Kendall Nygard and Dr. Sudarshan Srinivasan for taking the time to be part of my supervisory committee. I would also like to acknowledge their contributions to my academic success as my professors.

I would like to thank all the faculty and staff at the Computer Science department and North Dakota State University as a whole for all the help and assistance offered to me throughout my time here.

Finally, I would like to thank all my peers who I've met during my studies at North Dakota State University for all the good times and unmistakable sharing of learning through conversations, ideas, group study, group projects, etc.

DEDICATION

In dedication to my parents, for all they’ve done for me and their continued love, support and guidance.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ACM	Association for Computing Machinery.
API	Application Program Interface.
AST	Abstract Syntax Tree.
CS[#]	Computer Science [course level number].
IDE	Integrated Development Environment.
IEEE	Institute of Electrical and Electronics Engineers.
JDK	Java Development Kit.
MVC	Model View Controller.
RDBMS	Relational Database Management System.
SQL	Structured Query Language.
TT	Testing Tutor.
UI	User Interface.
V[#]	Version [number].

LIST OF APPENDIX TABLES

LIST OF APPENDIX FIGURES

1. INTRODUCTION

Testing is a very important part of the software development lifecycle. Software engineers are tasked with performing testing at various levels (e.g. unit, system) and at various stages of the software development lifecycle. This makes it critical for a student who may become a professional programmer to learn how to effectively test software. The importance of testing has been pointed out by organizations such as ACM and IEEE [1]. As noted by Marrero et al [2], introductory programming courses tend to focus on an object-first approach and teaching how to write code but overlook testing. Even if a test-suite is required as part of an assignment, they are often incomplete, incorrect, or missing. This leads to a need for better techniques, to be incorporated into introductory programming courses, to improve the testing skills or ability of students.

We hypothesized that a tool that provides conceptual feedback to students in programming assignments regarding their test cases will help students become better testers by allowing them to learn concepts they didn't apply properly, instead of providing them with the line number where the test case failed (or where coverage was missing). The current technique promotes students to try hit-and-trial method to simply fix the issue and move on without understanding the underlying reason that caused it. Our proposed solution will allow students to review a summary of missing and failed test cases in terms of concepts and provide learning materials so they can refresh or learn them better in order to avoid making the same mistakes again; something that may be overlooked if the student knows where the issue lies and simply tweaks the code and moves on. This should promote the students to make fewer mistakes as they progress (and avoid mistakes of the same type) by clearing their fundamental testing concepts and increasing their knowledge. As the test suites will be evaluated automatically (even though the overall assignments may be graded manually) there shouldn't be too much of a workload increase for the instructor or teaching assistants. On the contrary they can use the tool to reduce their workload by providing guidance through the tool with reduced work done individually explaining concepts to students and have more automated guidance on how to evaluate a student's tests.

Testing Tutor also takes its inspiration from Bloom's taxonomy for educational objects [3][4]. The current issue of not understanding why something was wrong with a test case and fixing it via hit-and-trial would likely require only the knowledge/remembering domain with respect to the error at hand and its location in a test case. This is the first in the set of 6 major incremental categories of cognitive processes. Testing Tutor's main aim is to enable users to be able to cross all of these categories, and not simply to attain some of the first few and stagnate the process. The conceptual feedback it provides combined with the supplemental learning materials and a lack of direct pointers to the problem should enable users to traverse the categories: remembering, understanding, and applying [4] in the cognitive domain. A student would run a test case and based on the feedback have to remember the concept that failed, understand how to test that concept, apply this understanding to fix the test case. Over repetitive use of the tool spanning multiple levels of study (e.g. CS1, CS2, CS3) the student may be able to achieve the remaining categories in the domain: analyzing, evaluating and creating, where they can eventually create their own programming and testing assignments which are thought out and help evaluate another students understanding.

The goal of this research is to develop Testing Tutor prototype tool that serves the purpose of improving students' testing ability with respect to the comprehensiveness (coverage) of their test cases from the point of view of the instructor in the context of introductory programming assignments. However, this research is not limited to just the development of the prototype tool, it also included determining the finalized tool that is ready for practical usage and performing empirical studies on such a tool. This paper details the process undergone to build the prototype, it's design, features and limitations. The expected result of students using Testing Tutor making fewer mistakes and creating more comprehensive test suites as they progress would display the effectiveness of conceptual feedback.

1.1. Testing Tutor Tool

This paper proposes development of Testing Tutor – A tool to improve students' acquisition of software testing skills – to help students improve their testing knowledge and skills. It will have two modes, *Training Mode* and *Development Mode.* The two modes are self-descriptive, with the Training Mode allowing students to practice writing tests against an instructor provided reference code and test suite while the Development Mode requires the instructor to first provide a reference test suite to the problem, to help evaluate the quality of the student's source code as well as test suite. The tool will allow students to submit assignments along with test-suites, to allow all tests to be run on the code. The tool will then categorize the reference tests into *covered* – matched by a student test, *redundant* – matched by more than one student test, *uncovered* – not matched by any student tests, and *failed* – failed on student implementation, tests and then provide conceptual feedback based on all-but-covered tests along with additional learning materials. The instructor can tailor the feedback provided based on the course, assignment or skill level of the students.

1.2. Prototype

While the tool will have many features, the entirety of it is outside the scope of this paper largely due to the size of the task and limited resources. The purpose of the prototype is to display that it is possible and feasible to create a tool that is able to provide conceptual feedback to the students on their tests based on the reference code and test class. This paper will detail a prototype based on the overall tool but with a subset of the functionality. The prototype only offers the Training Mode, where the instructor provides a reference source class and corresponding test class, to allow the student to learn how to write better test cases. The key and distinguishing features of this prototype include providing conceptual feedback when a test case fails or is ignored, as well as comparing it with the reference test class to determine efficiency and comprehensiveness. The reference test class allows the instructor to configure the level of testing desired from the students and base the evaluation of them on how closely they replicate it.

1.3. Explanation of Terms

- Java – A popular object-oriented and compiled programming language.
- Source Code – Refers to the cumulative actual programming language, such as Java, scripts that has instruction on what a computer or a particular environment should do. Generally used for .java files in this paper.
- Class – A class refers to an object-oriented programming concept defined as an extensible program-code-template for creating objects, providing initial value for state and implementations of behavior. [5][6] In this paper is refers to an un-compiled (in .java file) or compiled (in .class file) Java class.
- Test Suite – Refers to a collection of tests that are considered similar or should be executed together. In the context of this paper this is analogous to a test class.
- Construct – A concept commonly found in programming languages that is used to build code structure and allows the code to follow certain paradigms. In this paper it refers to branch conditions, loops, recursion, etc.

- Classpath – The root and sub-file directory path(s) where the source code resides.
- AST – It refers to Abstract Syntax Tree, which is a data structure that stores a tree representation of the abstract syntactic structure of source code written in a programming language (such as Java). [7]
- Visitor – A software engineering design pattern that allows addition of new methods to existing hierarchies without modifying them, thus separating algorithm from the object [8]. This pattern allows the implementing code to follow the object-oriented open/closed principle.
- MVC – An abbreviation for Model-View-Controller. This is an architecture or pattern that allows separation between model – generally data objects, view – the interface visible to the user, and controller – holder for the logic that controls and manages the view and model.
- Server-side – Generally used in web applications' terminology to specify the part of the application that resides on the server where the application is hosted or served from, as opposed to the part that is on the user (client) side such as a browser.
- REST(ful) – Type of web services that stands for Representational State Transfer. It allows interoperability between web applications over the Internet/HTTP via manipulation of web resources using a uniform and predefined set of stateless operations. [9]
- Persistence – In computer science persistence refers to a state that is not lost when the parent application or process ends. Generally, this refers to database systems that exists outside the application or process that generates or manipulates the persisted data.

- Javac – A (default/primary) compiler for the Java programming language that is bundled with the Java Development Kit (JDK).
- IDE – Integrated Development Environment is an application software that provides various useful features for working with a programming language. These generally include a source code editor, build automation tool and debugger. The main purpose is to make the developers work easier by loading features such as content assist, code generation and other plugins for development tools and frameworks.
- Eclipse – Eclipse is a free and open-source IDE by the Eclipse Foundation that is popular for use with Java programming language.
- SQLite – A specific light-weight, easy-to-use and generally embedded (as opposed to client-server) implementation of a RDBMS.
- JavaParser – A Java 1.0 to 1.8 parser with AST generation and visitor support. Allows recording of the source code structure in an AST, which can then be analyzed or modified. [10]

1.4. Organization of Paper

This paper provides some background about existing research based on similar approaches or concepts in section 2, details the prototype design in section 3, provides a thorough overview of development specifics, prototype versions and features in section 4, and finally provides a brief conclusion of the research and status of project objectives along with further improvements and work on this topic in section 5. Appendix A and B provide materials for reference such as samples of inputs/outputs and a sample of a hypothetical empirical study.

2. BACKGROUND

A brief literature review was performed to understand relevant and similar research that may have already been carried out. There are a lot of studies and research suggesting the importance of testing in computer science (CS) curriculum. A few of these are briefly explored in this section.

Marrero et al [2] argue that the short period of introductory programming courses and the reality that not all of the students will go on to take advanced programming courses means that while there is less time to fit in a bunch of concepts it also increases the importance of learning testing techniques. They propose a test-first approach and share some preliminary results. Similarly, Edwards [11] proposes a similar approach and supplements it with an automatic submission and grading tool called Web-CAT that allows students to test their code and find out coverage before submissions. Spacco et al [12] propose a tool called Marmoset and share their experience of using it in programming courses. The main features are the various types of tests supported that allow students to perform thorough testing before submission but also hide the actual tests used to grade the assignments. They argue this promotes students to start work early but smartly as release tests are a limited-rejuvenating resource. The literature review for each of these is detailed below.

2.1. Testing First: Emphasizing Testing in Early Programming Courses

This paper, by Will Marrero and Amber Settle, focuses on the requirement for emphasis on testing in early programming courses. Their motivation lies in primarily the short period of introductory programming courses and the reality that not all of the students will go on to take advanced programming courses. The first means there is less time to fit in a bunch of concepts and the second increases the importance of learning testing techniques. These coupled with the difficulties students face due to requirement of abstract thinking due to object-oriented focus forms the basis of this paper. The authors detail the challenges and the requirement for emphasis on testing in detail before providing a background for their approach to assignments in these courses, based on their experience at DePaul University's School of Computer Science, Telecommunications, and Information Systems. They opt for the simplest technique to emphasize testing, require submissions of tests before actual code submissions.

2.1.1. Main Research Question

The authors wanted to see if introducing testing as assignment(s) in introductory programming courses would help students learn more and improve their testing skills. Their hypothesis was that while students already face two significant tasks of learning to program and learning the features of the language itself, that introducing a third task of testing would ultimately help the students.

2.1.2. Motivations

This study was not directly motivated by the results of a previous study. The authors feel the need to emphasize on testing in introductory programming courses based on their observations and experiences at the School of CTI, DePaul University. However, they recognize that this is not a new observation, citing ACM and IEEE pointing out the importance of testing in their final report of computing curricula; introducing software engineering techniques in introductory programming courses by Alford, Hsia, and Petry in 1977; Schneider's suggestion that included testing and debugging among the ten essential objective of an initial programming course in 1978. The author's also list the research and suggestions already made to emphasize on testing that include test-first approach, tools for grading and measuring test coverage, etc.

2.1.3. Type of Study

They primarily conducted multiple experiments in introductory programming courses that were assignment based, noted student reaction and collected grade-based data from them. In general, these experiments of a test-first approach were standalone in terms of a course section or an assignment, like an independent study. The results of these were either compared to other sections, a previously taught class, or other assignments in the same class. Their study also included a survey-based approach to gather student response by the instructor.

2.1.4. Relevant Findings

The quantitative results for the testing based assignment did not indicate uniform improvement across the board. However, as explained with description of each assignment there were qualitative improvements and benefits noticed due to the introduction of testing in assignments. Particularly when they are provided with *.class* files while working on the testing document. It seems that the extra workload of testing assignment did not fare well with students in a CS1 equivalent course, however there are still reasons to believe that it will help them in the future. CS2 equivalent course provided better results, with students that had testing assignments performing better in testing related questions over the final exam and being generally more inquisitive about the application and its requirements in the view of the instructor. Another important note was to make testing assignments mandatory, as most students do not participate otherwise.

2.1.5. Validity Threats

The authors do not directly discuss any validity threats. Most of the justifications are to discuss why a particular result may not align with the theory. However, I believe these justifications were required because the studies conducted were not comprehensive or extensive, and hence had a fair amount of validity threats. Some of these included comparing data between the assignments with and without their technique, where one had an extra credit opportunity but the other didn't hence introducing additional motivation for a section that might skew results even if the actual extra credit score is factored out. They also compare scores between undergraduate and graduate students, which clearly is a major validity threat as their knowledge and skills will be disproportionate. Another instance was comparing scores on a test between sections that covered the material in different durations. The authors generally seem to recognize these shortcomings, however they do not necessarily acknowledge the actual validity threats they pose, focusing on justifying why a result may still be useful instead.

2.1.6. Interest

A very important reason for my selection and interest in this paper was the author's observation that students in introductory programming courses find it inconceivable to be able to write test cases for a class that doesn't exist. This is important to keep in mind when evaluating student's test cases. Another interesting point was that students that had to submit tests before submitting the class were more inquisitive. This could translate into better coverage of tests for their code.

2.2. Experiences with Marmoset: Designing and Using an Advanced Submission and Testing System for Programming Courses

The paper, by Jaime Spacco et al., details the features of the marmoset tool, a submission and testing system, and the authors' experiences with it in several introductory programming courses. The tool provides feedback to students and instructors and also allows for detailed research on the development processes of students by capturing a snapshot of their work from the first save all through to the final edit before submission, and saves it on a repository. Some of the initial research results are discussed in the paper. Due to the large amount of data, they have only just scratched the surface of the collected data. They also discuss some related work that has been done previously.

2.2.1. Main Research Question

The paper, by Jaime Spacco et al., details the features of the marmoset tool, a submission and testing system, and the authors' experiences with it in several introductory programming courses. The tool provides feedback to students and instructors and also allows for detailed research on the development processes of students by capturing a snapshot of their work from the first save all through to the final edit before submission, and saves it on a repository. Some of the initial research results are discussed in the paper. Due to the large amount of data, they have only just scratched the surface of the collected data. They also discuss some related work that has been done previously.

2.2.2. Motivations

This paper was not directly motivated by the results of a previous study. Although they do mention some of the related work done previously. They also mention their goals for research based on the CVS history data to be similar to those of Liu et al [13].

2.2.3. Type of Study

The major study type was a survey of the students. The authors' list their experiences and observations from using the tool in their courses, in the paper, which possibly counts as experimental study. They are also in the process of performing data analysis on the data collected in the CVS repository.

2.2.4. Relevant Findings

Marmoset front-loads the programming assignment development process by forcing completed projects including test cases. This promotes better projects, along with the fact that marmoset also forces instructors to think about the grading process. The test cases, however, needed to be changed after the project was handed out to students. Also, a key finding was that a test harness developed for Marmoset should be able to handle any valid implementation of the project and not just the reference copy. A large amount of data was collected, and about 12.5% of all test results were failures due to recognizable runtime exception.

2.2.5. Validity Threats

The authors do not discuss any validity threats. This seems appropriate as they mention that they did not compare results of semester using Marmoset against previous semesters without it because of major restructuring of introductory curriculum at University of Maryland. This would have been a major validity threat had they gone through with the study.

2.2.6. Interest

It was interesting to read about researcher's experiences as a first person narrative. A major point of interest was the overview of the architecture of a tool similar to what my research topic requires. The tool allows student's test cases to be evaluated by using different implementation of a common API, each seeded with a different bug, which they create test cases for. A fair amount of test results were recognizable runtime exceptions that should be easy enough to find and fix, and do not necessarily imply any conceptual learning issues for the students.

2.3. Rethinking Computer Science Education from a Test-First Perspective

The author recognizes that the students struggle in acquiring comprehension and analysis skills. He believes it to stem from an aggressive objects-first strategy and a lack of any motivation for the students to test the completeness and correctness of their own programs. The paper suggests a new vision for the computer science (CS) curriculum that is based on test-driven development (TDD) approach. This vision includes the use of industry-standard tools for automation of the submission and grading process that checks the programs as well as the validity, correctness, style, and quality of the student's code as well test cases. The key ingredient mentioned is the rapid and informative feedback that promotes thoroughness of student test cases thereby forcing them to think deeply about the assignment and its requirements. The paper details the test-first assignment approach along with the Web-CT tool. An example is provided to help understand how the tool works and will satisfy the goals set to ensure students are benefitted and can eventually see a cultural shift in their practices. The author mentions the benefits that student's will experience and lists some observations from the use of this tool in a junior level course. The paper outlines the vision in detail and also lists the related work that this work builds upon.

2.3.1. Main Research Question

The main aim of the paper is to propose a new vision for computer science courses that is based on the use of test-driven development for programming assignments. The student programmers face issues acquiring comprehension and analysis skills. The focus on aggressive object-first strategy ignores testing and correctness of programs and students merely ensure no compile issues and possible limited sample-data based tests, tweaking their code for fixes without understanding the underlying issues. The author stipulates that bringing a TDD approach to programming assignments with rapid, concrete and useful feedback, the students will gain a better understanding, analysis skills and eventually a cultural shift in how they approach programs. This progress would follow Bloom's taxonomy of learning.

2.3.2. Motivations

The author does not specifically mention that the motivations of this study stem from results of any particular previous study. However, he does state that the vision described builds on a large body of prior work. He refers to a fair few papers that cover or point to issues such as pursuing an aggressive objects-first strategy for its pedagogical value. Similarly, he also refers to papers that suggest software engineering concepts and testing skills should be integrated across the CS curriculum. The significance of different papers is discussed in more detail in the *Related Work* section, and these papers are listed in the *References* section of the author's paper. There isn't any one (or two) significant paper to list in this section.

2.3.3. Type of Study

While the author never mentions a specific study conducted, he does mention that the Web-CAT tool was used in a junior level class. Although no data is provided, it seems as though an experiment or independent study was performed.

2.3.4. Relevant Findings

The automated grading techniques devised were applied to different CS courses from freshman through to the junior level with success. The author also mentions that the students who used TDD, compared to prior offerings, are:

- More likely to complete assignments
- Less likely to turn the assignments in late
- Receive higher grades

- Submit more thoroughly tested programs

2.3.5. Validity Threats

The author does not discuss any validity threats.

2.3.6. Interest

One of the major reasons for interest in this paper is that this paper identifies a key issue that aligns with my research topic. Students don't focus on testing, they generally only test on limited sample data, when required to, and when issues are found, compiler-based or logical, they simply tweak the code just enough to get it to work without understanding why the issue existed in the first place.

Finally, the Taxonomy of Educational Objectives by Bloom et al. [3] and its revised version by Anderson et al. [4] help put these ideas in context of domains for educational activities or learning, primarily the cognitive domain that is most relevant. This governs how Testing Tutor may help a student progress faster through the domains in relation to testing.

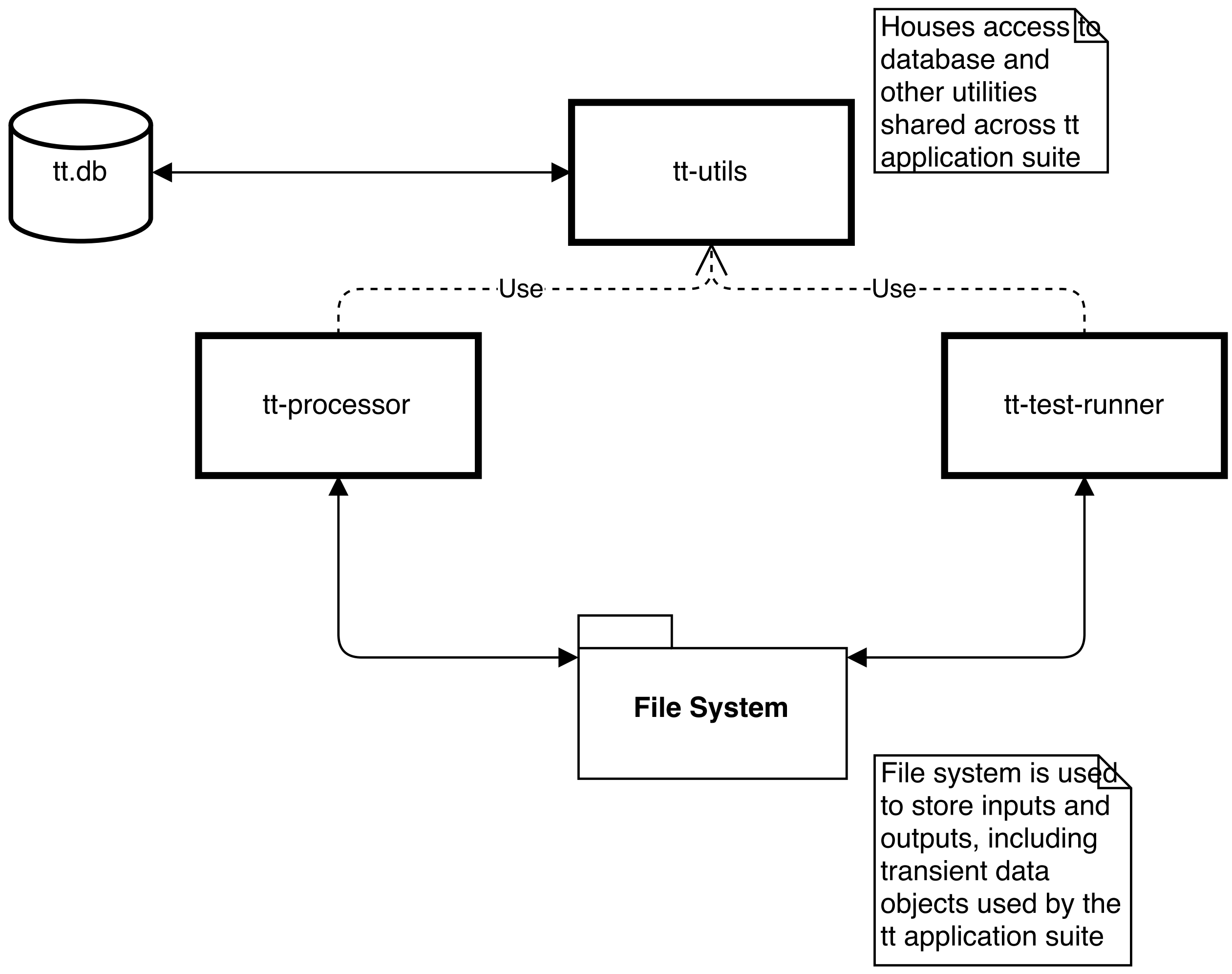
Reading different papers on different but similar tools have helped to get a better idea on how to think regarding the requirements and expectations of Testing Tutor. They have shed some light on some areas unclear to me while also making me think about the design and ask questions that I might not have otherwise. While all these papers might have different approaches, they all seek to promote the exposure of students to testing activities and attempt to address a variety of deficiencies in the current curriculum and education system. Two of them offer, propose or evaluate a tool to help solve these issues with varying degrees of success, however none of them approach it with providing conceptual feedback to students to improve their understanding of testing concepts like Testing Tutor. The research detailed in this paper aims to follow in the same path and evaluate the usefulness of the proposed approach.

3. DESIGN

This section details the overall design of the Testing Tutor (V2) prototype.

3.1. Architecture

The Testing Tutor (V2) prototype consists of three java projects that comprise of one application suite. We explore the application suite structure along with explanation of each application and activities it performs, database design and details about file system directory structure as well as inputs and outputs in this section.



Houses access to database and other utilities shared across tt application suite
tt.db
tt-utils
Use
Use
tt-processor
tt-test-runner
File System
File system is used to store inputs and outputs, including transient data objects used by the tt application suite

Figure 1. Testing Tutor application suite structure

3.1.1. Application Suite Structure

The three projects in the Testing Tutor application suite are: tt-utils – a dependency for other executable projects in the suite, comprises of shared resources including the database; tt-processor – this project is run initially to process the source code, reference test class and evaluation test class; tt-test-runner – this project is only run after tt-processor has processed the inputs, and its main purpose is to run the evaluation test class on the source code class and provide the feedback that comprises the final output of the application suite. Note that the user input acts as input to tt-processor, while tt-test-runner provides the eventual output to the user. The application suite utilizes a database to store persistent data, however the data's lifecycle is only required until tt-processor needs to run again – existing data can be cleaned up before it is run again. Additionally, the application suite uses the file system to store and retrieve files relating to Java source inputs, compiled classes, and output text files. Figure 1 displays the structure of the Testing Tutor application suite described here.

3.1.1.1. TT-Processor

The tt-processor project is the piece of the suite that takes in the user inputs and processes and prepares them for use by tt-test-runner. It uses an AST parser to parse through the source code class and reference test class to gather details that are used to provide the feedback after the evaluation test is run. The imports are resolved for source code class and evaluation test class and they are placed in the correct directories, since they will need to run as part of the tt-test-runner application. tt-processor uses tt-utils primarily to access the database to save and capture the information in a persistent state. It is also used for utility classes such as *DirExplorer* to assist with searching for files given a root directory, *CodeConstruct* enum and *TTConstants* to get relevant constants such as key paths.

One of the key features of tt-processor is the use of AST of the source code and visiting of the relevant nodes, which predominantly uses the visitor pattern to traverse the tree, to perform analysis and capture key information. The visitor pattern allows the visit() method's source implementation to be based on the dynamic type of the element being visited and the dynamic type of the visitor. For instance, in the case of Testing Tutor, this allows visiting different types of nodes within an AST of a class such as class or interface declaration, method declaration, method call, if statement, etc.

3.1.1.2. TT-Test-Runner

The tt-test-runner Java application should be executed after tt-processor has executed successfully on the user inputs. It will pick up the processed files and persisted data that were provided by tt-processor and attempt to run the evaluation test class as a JUnit test case class on the reference source code class provided, which is loaded as part of the tt-test-runner compiled application. This is the key part that requires two separate applications to be executed sequentially since the source code class needs to be on the class path and compiled and loaded with the rest of the project.

While executing the JUnit test class, if any test case (method) fails or is ignored then the Testing Tutor algorithm fetches data and determines which source method was invoked and what constructs it contains to determine the feedback to provide to the user. At the end of the test run a comparison between the reference and evaluation test classes is performed to provide feedback to the user based on the metrics of which one had more/less tests for each of the source code class methods. This provides guidance to the student based on whether their test class is complete/thorough as well as efficient. This is only possible due to JUnit's implementation of listeners that allow processing at certain stages of the test run.

3.1.1.3. TT-Utils

tt-utils is the non-executable shared dependency for the above two projects. The idea behind this separate project is to control access and have a consistent implementation for all shared resources such as the database and utility classes as well as avoid duplication of code. The key features and classes that it houses include: database access via data service classes that run SQL commands directly on the connected datasource, class for constants, enums code constructs and test types, model/domain classes for method, method call and construct to allow carrying data in objects, and finally utility classes for exploring directories and connecting to the datasource. The only dependency this project has is on datasource and its drivers.

3.1.2. Database Design

The main idea behind the database design here is to keep it simple and modular. This helps keeps things segregated so it's easy to identify the purpose of each as well as identify issues easily since they can be isolated to certain relation(s). A conventional RDBMS is preferred due to their popularity and relational nature that is easy to understand. This database stores information gathered by tt-processor about the source code class methods including any pre-determined code constructs found, reference test class methods, evaluation test class methods and any source code method calls the test methods make. Figure 2 provides an overview of the data model for the Testing Tutor application suite.

To accomplish this, basic information about each method is stored in three separate relational tables based on the class they belong to. While the data model for each of these tables is identical, separating them out provides the ability to easily distinguish by originating class visually as well as programmatically. In addition to these one table stores all the constructs found in the source code class and one table stores all the source code methods called by each test method, both reference and evaluation.

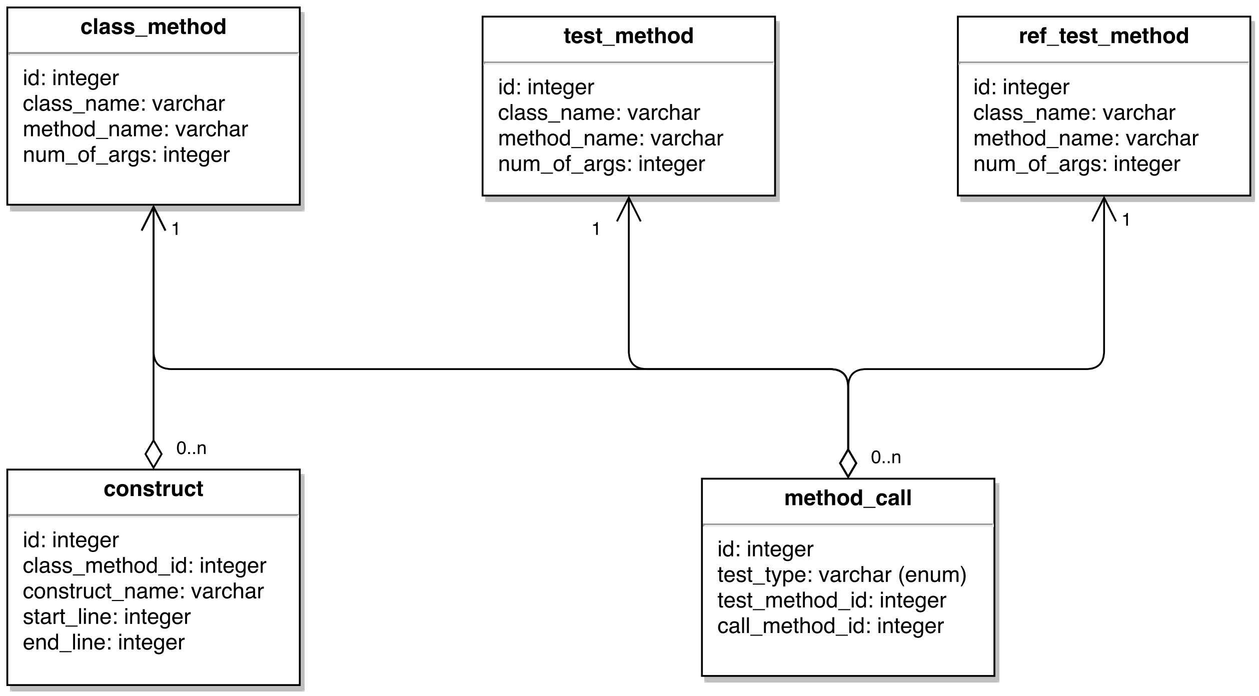


Figure 2. Testing Tutor data model

3.2. Supplemental File and Directory Structure

Note that all folders mentioned below exist in the root TestingTutorV2 directory by default. The root directory along with individual directories is configurable via the TTConstants.java class in tt-utils application.

3.2.1. Inputs

/input: Java class for source code under test in *sourceFiles* folder, java class for reference tests in *refTestFiles* folder, java class for (student) tests to be evaluated by the tool in *testFiles* folder.

3.2.2. Outputs

/output: *test-run-result.txt* and *tt-feedback.txt* in the output folder. Also, logs in the default system out console primarily for debugging and informative purposes.

3.2.3. Reserved Folders

- feedback – houses the pre-created files that house the construct-specific feedback.
- data – contains the data-related files such as the (SQLite) database and tt-*schema.sql* database schema file.
- processedTests – folder reserved for the application to store test (.java) files after processing them and .class times generated during run time.

3.2.4. Project Root Directories

The project root directories contain the actual source code for each Java Maven project. They are named the same as the project names: tt-processor, tt-test-runner and tt-utils.

3.3. Inputs/Outputs and User Interaction

To get any output the prototype requires at least three inputs – the reference source code class under test, reference test class, and student or evaluation test class. The outputs produced by this set of inputs include a JUnit test run result in a text file and a Testing Tutor feedback output text file. For samples of inputs and outputs see Appendix A.

There are also internal outputs and inputs used by the entire application suite. This primarily stems from tt-processor producing outputs of modified source code and test class file for tt-test-runner to consume. The tt-test-runner application also generates .class file for test classes that it dynamically runs within the same folder the un-compiled class exists. The persistent data stored in a database is also considered internal input and output for different parts of the application suite.

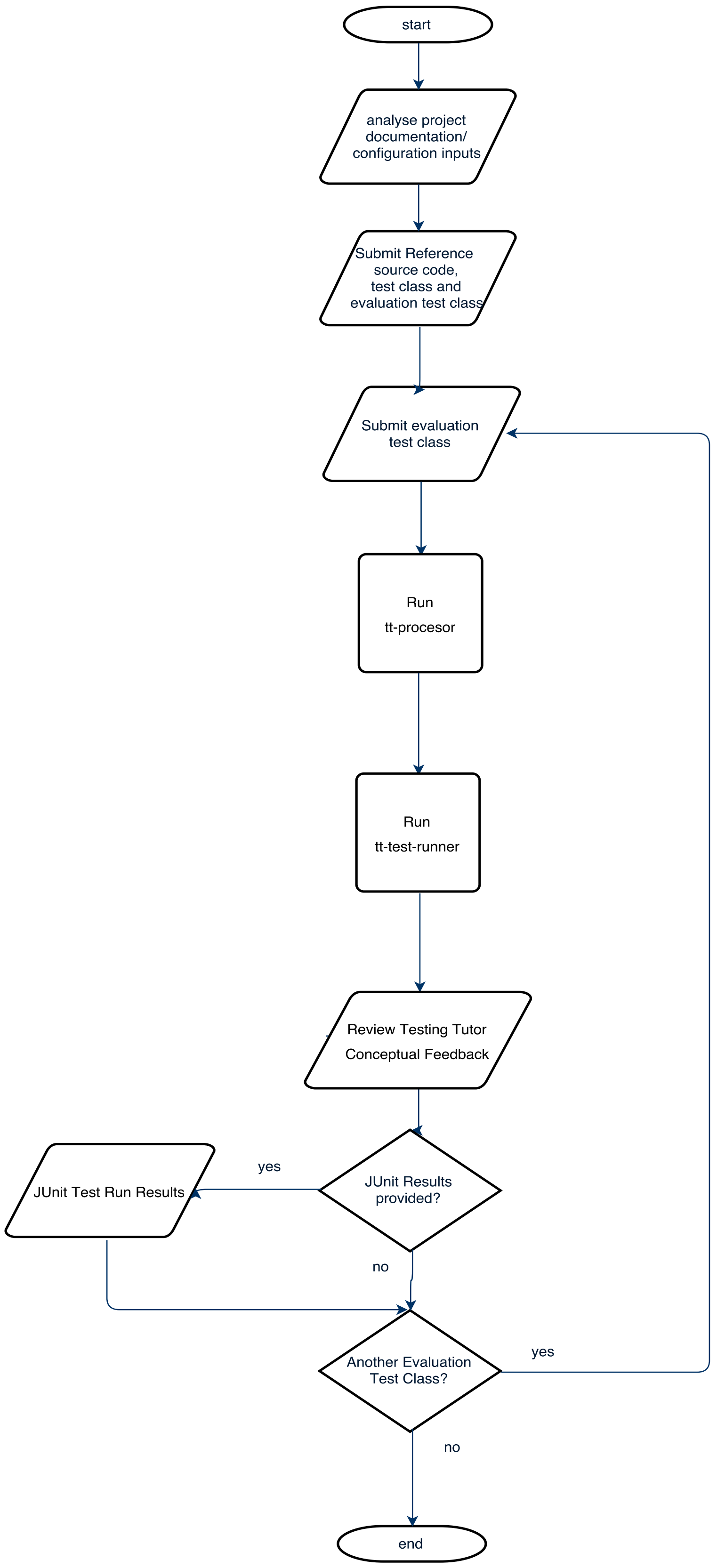

start
analyse project documentation/ configuration inputs
Submit Reference source code, test class and evaluation test class
Submit evaluation test class
Run tt-procesor
Run tt-test-runner
Review Testing Tutor Conceptual Feedback
JUnit Results provided?
yes
JUnit Test Run Results
no
Another Evaluation Test Class?
yes
no
end

Figure 3. User interaction flow for Testing Tutor V2 prototype

3.4. Constructs

Constructs in the context of this tool and paper refer to concrete, visible and identifiable elements of the source code. Since the prototype is only concerned with an object-oriented language, Java, the constructs have to be Java code structure elements that allow easily following logical and computer software paradigms. This generally translates to control flow statements, that essentially decide what path the execution of the program should follow, of which the most common are conditional branch statements. These have to be pre-defined to the tool to allow it to correctly search for them and perform actions accordingly. The prototype only cares about the pre-selected constructs that exist in the source code, they are associated to the test case via the source code method it invokes and by association the constructs it internally uses. For the prototype the following constructs were chosen, defined as an enum in tt-utils:

- IF
- FOR
- WHILE
- SWITCH
- FOREACH
- RECURSION

4. DEVELOPMENT

This section details everything required and accomplished during the development phase of the prototype. The development included two similar, but still separate applications (or suites). The primary reason for this was that the development of the first prototype application could not entirely support what the tool should be or evolved to yet it led to considerable amount of lessons learnt and choices made in the development of the final prototype, not to mention the effort spent. This also displays the evolution of the prototype. Due to these reasons, each of the two versions of the Testing Tutor prototype are explored below.

4.1. Testing Tutor V1

This refers to the initial attempt at developing a Testing Tutor prototype. This version had significant issues and was brittle to varying inputs, thus making it an incomplete or insufficient prototype for the desired Testing Tutor tool. However, this is still elaborated on in this paper because this was the first attempt and led to significant realizations that influenced choices and decisions made for future version.

4.1.1. Environment and Tools

This prototype was developed in multiple system environments – Mac OSX and MS Windows. Below is a list of tools used during the development:

- Language – Java 7
- Versioning System – Git (BitBucket.org)
- IDE – Eclipse 3.8 (Juno)
- Test runner – JUnit 4.12

4.1.2. Features

- Parse the source code and determine the existence of pre-determined code constructs.

- Instrument source code class to invoke internal APIs to track details during test run such as current executing method and line number. This will allow providing more details if required in terms of what executed successfully and what didn't. It also enables providing exact positioning of execution and issues, at a later time, that may be configurable by the instructor.
- Dynamically load and run an external JUnit test class on a pre-compiled and loaded source code class, and saving results.
- Select and provide pre-defined, configurable feedback based on construct occurrences within the source method called by the test method at the end of the test run. These are picked from a repository of conceptual feedbacks for pre-determined constructs.
- Provide JUnit test run result as output in a text file.

4.1.3. Known Limitations and Drawbacks

- Only supports Java programming language
- Supports a simple single source code class (that is under test) and reference test class at a time.
- Parses java code using Regular Expressions (RegEx).
    - For instance, *return* statement is only expected at the end of the method while instrumenting code.
- Feedback is limited to comparison with the reference test and a certain set of pre-configured constructs (IF, FOR, WHILE, SWITCH).
- The processed/instrumented source code class file needs to be manually added to a pre-determined source package of TestingTutorV1's pre-compiled code – then the application needs to be re-run.

4.1.4. Lessons Learnt

- RegEx, while simple, is not a suitable code parsing technique and is very inferior to AST parsing tools.
- Requirement for two separate applications due to issues dynamically loading a source code class and execute a loaded test class on this source code class in the same application. One application needs to parse the inputs and supply source class in the classpath of the second application to allow compilation, loading and test execution.
- Instrumentation may not be necessary, at least at this stage of tool development, and simply complicates things when simpler may be better.

4.2. Testing Tutor V2

This is the evolved and current version of the Testing Tutor prototype. There are significant improvements and additional features this version offers over the initial version. This version of the tool can be considered a true prototype since it accomplishes and displays that the goal of providing conceptual feedback based on desired inputs is feasible.

4.2.1. Environment

4.2.1.1. Tools

- Environment – Mac OSX
- Language – Java 8
- Versioning System – Git
- IDE – Eclipse 4.6.2 (Neon)
- Build Tool – Maven 3.3.9
- AST Parser – JavaParser 3.0.1
- Test runner – JUnit 4.12

- Database – SQLite3 3.16.1

4.2.1.2. Environment Decisions

Each of the tools and technologies used in this project were chosen after a considered evaluation of the options, particularly after the experience from the first prototype attempt. Latest versions of open-source, convenient and easily accessible/usable software and tools were preferred in general. In some cases, there may have been comparable or better alternatives available, however all decisions were made keeping the specific requirements of this research project and prototype in mind along with the author's strengths.

The environment or development operating system was chosen via convenience of access and the benefit of Mac OSX systems using a Unix kernel similar to many servers. The language was pre-decided due to prior knowledge, Java's variety of third-party tools and frameworks access, convenience of use as a server side web application compiled and object-oriented language and the extensive use of Java language in introductory programming courses. The version control system for code was again a decision based on ease of access and free access to platforms such as BitBucket.org and GitHub along with prior familiarity with Git. Eclipse is an IDE preferred by many in academia and the industry while being open-source making it an easy decision. The build tool maven is very popular and allows compiling code along with dependency management of third party tools as well as in-suite dependencies such as *tt-utils*. Generally, coverage and source code analytical tools use AST or byte code parsing, AST is preferred here since it uses raw pre-compiled source code while byte code uses compiled code. Working with understandable source code and not compiled .class byte code is very beneficial for learning, reference and debugging purposes since this is the first attempt at the tool. JavaParser is a light-weight, performant and easy-to-use open-source Java source parser that provides modifiable AST, with sufficient documentation, blogs, answered questions, and supporting tools over the web making it a worthwhile option [10]. JUnit is also one of the most common and open-source testing framework that is most commonly used in schools, the more students and instructors that already use the framework the more easily Testing Tutor cane be adopted. Finally, SQLite3 was chosen as the database due to its ease, convenience, and light-weight nature. The type of database isn’t a priority at this stage of the prototype and other databases such as Oracle, MySQL, PostgreSQL, etc. require more of a setup process and management that may take focus away from the actual important functionality of the prototype. Moreover, SQLite can be bundled with the application in many forms and much easier than other fully-featured database servers.

4.2.2. Configuration

The configuration of this application suite is pretty basic. First of all, ensure that the environment mentioned in this paper is matched. Key prerequisites are JDK 1.8, maven, git and preferably an IDE such as Eclipse. Fetch the latest version of the application suite source code (preferably via git). Then open the TTConstants class in the tt-utils project and modify the TT_ROOT_PATH constant value to point to the root TestingTutorV2 directory on the system. That should be all the configuration required in a typical case. You may choose to modify other directory path constants should you wish, however this is not advised unless the use and impact of each of them is clearly understood. Continue with providing the inputs as described in setions 3.2 and 3.3, then use javac or Eclipse IDE to run the applications. See section 4.2.4 for more on the user interaction flow.

4.2.3. Execution

This section details the normal flow, in simple steps, of the application suite in its entirety. The flow of the application suite as a whole may jump across applications and classes within the suite, especially in the case of access to shared resources within the tt-utils project. It is important to note that entire tt-utils is available when either of tt-processor or tt-test-runner is running.

- Execution starts in tt-processor's main method, where first logging level is set and a clean-up of processed evaluation test class and the reference source code class is performed so there are no conflicts during the current run.
- A file search algorithm looks for the source code class file in the input folder for processing.
- Processing involves fetching the name of the class, updating its package to reflect tt-test-runner class path, and determining existence of each of the relevant constructs by visiting each method declaration and each node inside the method down to the statement level.
- Each time a key construct statement is found, it is recorded in the database along with its name, parent class name and start and end line of code.
- Finally, the updated source code is saved in a new file named the same as the class in the class path of the tt-test-runner project to be compiled and loaded as part of the application.
- The name of this new file is fetched and used later when saving test method details.
- The reference test class is then fetched and imports are updated to allow running of the tests later.

- The class is then visited as an AST using customization of JavaParser's use of visitor pattern. Each method that calls a source class method is then saved along with the specific calls in the database for analysis and feedback later.
- A similar process to reference test class is followed for the evaluation test class. Additionally, the updated class (with the imports) is saved separately for use by tt-test-runner. This concludes the tt-processor execution.
- If tt-processor executed without issues, then tt-test-runner should be run. This will launch its main method that first sets the logging level and cleans up any existing output files to avoid overwriting or conflicts.
- It then fetches the previously processed version of the evaluation test class and then dynamically compiles, loads and runs the class as a JUnit test.
- A test run listener is used to process feedback in case of key events such as test case failure. There are two key algorithm that provide the feedback. The first runs if a test case fails or is ignored and fetches all the constructs that are contained within all the source code methods called by the test method. The feedback is fetched from pre-existing files and appended to the output.
- Once the test run is finished, the second key Testing Tutor algorithm loads the data from reference test class and compares it with the evaluation test class to determine tests per source method (tpsm). Testing Tutor makes inferences based on which test class has more/fewer tpsm and provides that as feedback.
- The overall JUnit test run result and Testing Tutor feedback are saved to two separate text files concluding the execution of tt-test-runner and the Testing Tutor application suite.

4.2.4. Code Excerpts and Enumerations

This section lists some examples of key code and pseudo code from the prototype along with enumerations used.

4.2.4.1. Compare Tests per Source Method

```java
// Map data structure to store source method name and # of tests for it
Map<String, Integer> sourceMethodRefTests = null;
List<Method> refTestMethods = tss.getMethods(TestType.REFERENCE);
// Determine source methods and tests for each from stored data
if (refTestMethods != null && !refTestMethods.isEmpty()) {
      sourceMethodRefTests = new HashMap<>();
      for (Method method : refTestMethods) {
            for (MethodCallDetail methodCall : method.getMethodCalls()) {
                  String sourceMethod = methodCall.getMethodName();
                  Integer methodTestNum = 1;
                  if (sourceMethodRefTests.containsKey(sourceMethod)) {
                        methodTestNum =
sourceMethodRefTests.get(sourceMethod) + 1;
                  }
                  sourceMethodRefTests.put(sourceMethod, methodTestNum);
            }
      }
}

// Determine source methods and tests for each from stored data
// Code omitted but is similar to above
Map<String, Integer> sourceMethodEvalTests = …;

for (String methodName : sourceMethodRefTests.keySet()) {
      if (sourceMethodEvalTests.containsKey(methodName)) {
            Integer refTestCount = sourceMethodRefTests.get(methodName);
            Integer evalTestCount =
sourceMethodEvalTests.get(methodName);
            if (refTestCount > evalTestCount) {
                  // Feedback for too few tests for this method
                  // Can update grade here too
            } else if (refTestCount < evalTestCount) {
                  // Feedback for too many tests for this method
                  // Can update grade here too
            } else {
                  // Feedback for optimal number of tests
                  // Can update grade here too
            }
      } else {
            // Feedback for not testing a source method at all
            // Can update grade here too
      }
}
```
Figure 4. Code excerpts from TestSuiteRunListener.java in tt-processor

4.2.4.2. Enumerations

- TestType – Used to differentiate between code running on reference and evaluation test class as they use similar code in many cases. This helps keep data model separate while keeping it easy to re-use code.
    - REFERENCE
    - EVALUATION
- CodeConstruct – Used to define each construct/statement/expression that the tool cares about. Note that to support new constructs, currently, JavaParser should be able to determine its type or it should have a differentiating factor from other AST nodes.
    - IF
    - FOR
    - WHILE
    - SWITCH
    - FOREACH
    - RECURSION

4.2.5. Improvements and Additional Features

Even though this paper lists a bunch of limitations for the Testing Tutor V2 prototype, that is simply because it is a prototype and not user/production ready. What is more important is the improvements made over Testing Tutor V1 prototype that show progression, and the overall features the prototype supports that proves what is possible and gives an idea of feasibility.

4.2.5.1. AST Parser

V2 makes use of an AST-based parser tool, JavaParser, as opposed to customized RegEx based (brittle) code parsing. This allows feeding the un-compiled class into the tools framework and getting an AST representation of the entire class source code and its nodes down to the statement level. This can then be processed to accomplish the tasks required of modifying the code with correct import statements and visiting each method to save all relevant method details.

4.2.5.2. Persistent Data

The prototype saves all relevant information in a persistent, light-weight SQLite database. Key information about source code class and methods, reference test class methods and evaluation test class methods are stored safely and separately. This allows reference code and test class to be submitted separately from the student/evaluation test class. Due to the same reason, it will easily be possible to submit multiple student test classes for evaluation against the same set of reference code and test class even though the current prototype has not been tested to support this.

4.2.5.3. More Code Constructs Supported

Support for 6 different constructs including dummy feedback with additional learning web links for each one. The supported constructs are IF, FOR, WHILE, SWITCH, FOREACH, and RECURSION. These cover some of the most basic concepts of programming, such as branch conditions and loops, and are frequently used in introductory programming courses. If any of these are encountered in a failed/ignored test case, then feedback/guide is provided on how the specific construct(s) may be tested along with additional learning web links provided. Since these are picked from some sort of storage (file system currently) they can easily be updated by the instructor to configure and customize the feedback based on the course, student's ability and the reference source code and test classes. The current dummy feedback attempts to inform the student that a construct exists and generally when testing such a construct in isolation they may want to test a certain number of branches to be thorough. Additional learning materials can also be supplied with this feedback that provide the student detailed information on what the construct is and how it should be tested, this can either link to internet web link or the instructor/university's own learning materials. Some samples of the feedback are available in Appendix A.

4.2.5.4. Training Mode with Reference Test Comparison

The prototype tool performs a comparison of evaluation test class with reference test class to determine comprehensiveness and efficiency of the student's test efforts. This feature along with the feedback provided for construct occurrences provides the key distinguishing ability of this tool and prototype. The number of tests methods (or cases) per source code class method is compared and inferred. The general idea is that if the student has more test cases per source method than the reference test class then their tests may be inefficient, alternatively if they have less test cases then they may have missed testing some condition(s). Since tests per source method, which may have different combinations of constructs, is variable it is hard to capture a particular number of pre-determined test cases per construct. The process of using reference test class to determine the correct number of tests per source method offers a solution by accounting for variability and allowing the instructor to configure how a student test class is evaluated. A key benefit is that it allows the same source code to be used to evaluate different levels of students' testing ability by changing the number of reference tests.

4.2.5.5. Suite of Applications

A complete suite of (three total, two executable) applications where providing input files (source code class, reference test class and student's/evaluation test class), leads to output files containing the JUnit Test Runner results and the conceptual Testing Tutor feedback.

4.2.5.6. Build Tool

Build tool with dependency management such as Maven to support a suite of tools and simplify the build run/deploy process. This also allows easier extension to other third party tools and dependencies later. For instance, to convert this into an MVC application or server-side application that supports access via RESTful APIs one could easily include Spring Boot and create Rest Controllers that invoke the appropriate application suite methods.

4.2.6. Prototype Limitations

- Only supports Java programming language
- Lack of a user-friendly interface – file system based input and output
- Supports a simple single source code class (that is under test) and reference test class at a time.
- A single test class can be run at a time, however multiple test classes can be evaluated for the same source code and reference test by updating only the desired inputs.
- Feedback is limited to comparison with the reference test and a certain set of pre-configured constructs (IF, FOR, WHILE, SWITCH, FOREACH, RECURSION).
- Requires each actual test method in the test class to be annotated with @Test Junit annotation or that the JUnit test class *extends* the TestCase class.
- Under-test class name association with test method is done based on the type name of the source code class.
- Does not support multiple classes within a single class file.
- Methods are matched simply based on name, which means that if a source code class method is named a common java method name such as *"println()"* then all *"System.out.println()"* methods will be matched and counted as source method calls

as well. So no source code methods should be a name used by any JDK library class' methods.

- Comprehensive individual test feedback is only provided for tests that either failed, were assumed failed or ignored. This means that unless a test falls under one of these categories, it is not evaluated for testing effectiveness of constructs.
- General assumption is that each source code method is simple with inputs that affect the output. Additionally, inputs are generally able to change the flow of code when branch or loop conditional statements are present in the source code.

5. CONCLUSION

The idea of providing students with conceptual feedback on their test suites to help them address any knowledge deficiencies is a novel one. This paper introduces this concept along with the Testing Tutor tool that implements this idea via a prototype. Part of the conclusions and summary of this research project are already covered in sections such as Lessons Learnt from Testing Tutor V1 and Improvements made in Testing Tutor V2.

5.1. Objectives Summary

Objective 1:	Ability of the instructor to submit reference source code and reference test classes.
Status:	Complete.
Objective 2:	Allow student to submit a test class for evaluation.
Status:	Complete.
Objective 3:	Parse source code class and capture relevant information, particularly relating to constructs important for tests and this paper.
Status:	Complete.
Objective 4:	Parse reference test class and capture relevant information.
Status:	Complete.
Objective 5:	Process and run evaluation test class as a JUnit test on the reference source code class.
Status:	Complete.
Objective 6:	Provide feedback based on constructs that each evaluation test class method tests when the test does not successfully complete.
Status:	Complete.

Objective 7: Provide feedback to the user/student based on comparison of their evaluation test class with the reference test class provided by the instructor.

Status: Complete.

5.2. Future Work and Guidance

As part of the research concerning Testing Tutor, the current prototype tool wasn't the only focus. The research work included considering the final Testing Tutor and estimating its functionality as well as its use and practicality. As such this section includes some guidance on how the current prototype can be expanded upon to make it a useful working tool and how that tool can be evaluated by use of empirical studies. For guidance and a sample research paper on potential empirical studies please refer to the Appendix B.

5.2.1. Potential/Envisioned Tool Improvements

This section details the potential future work to convert this prototype into a practical tool, and brief insight into how that may be accomplished or by using which technologies. A lot of consideration when working on the current prototype went into how to make it extensible and scalable in the future, thus plugging in extra functionality and support for user-friendly features should be easy and convenient.

- Provide a Web UI interface for user interaction. ReactJS with NodeJS offers a simple way of constructing client-side code. The basic elements would be to allow user authentication with roles (e.g. student, instructor, admin), allow uploading of source code and test code based on roles, execute test command, and provide a feedback/result page that allows download/saving of the output.
- Extending on the basic user interface the tool could allow the instructor to create and customize the feedback based on the construct type. It could also allow the instructor

to submit only one fully-comprehensive reference test class and then allow different student testing levels that would automatically reduce or increase the strictness of the comparison.

- To support this web interface, the current code can be made web complaint by the use of Spring technologies. Spring boot offers a simple and convenient way to convert any Java project to a web project if desired. Using spring would not only allow creation of rest APIs that can access the existing Java code for processing or running the test class, it would also enable simplification of some of the existing code such as using Spring Data entities instead of SQL queries and manual Java object translation.
- Convert the tool suite into one project via Maven parent project. Alternatively, a sound deployment strategy for the suite combined with a separate layer (either UI or application) can encapsulate the underlying suite of projects and provide a single interface to interact with and orchestrate the entire tool suite.
- Evaluation test class processing should be made optional and a separate process when running tt-processor. This will allow user to configure if they want to run tt-processor to process reference Java files only or evaluation Java test class only (will require reference files to be previously processed) or do both at the same time (as it runs currently). Alternatives to this include moving the evaluation test class processing to a separate project or to tt-test-runner.
- Addition of other desired metrics and functionalities is possible. Some of these may be:
    - Determine the minimum number of test each type of construct should require for effective testing and provide that as feedback per method. This

could include evaluating the implications of nested constructs and provides the tool with inherent intelligence.

- Determine the type of object used to call the source method from the test method and associate it to the call. This may be possible by using the java-parser-symbol-solver open source utility.
- Use a more generalized way, such as application.properties configuration file or environment variables, to allow the user to configure required directory paths. Ensure that the purpose of each is explained clearly before allowing configuration of the paths.

REFERENCES

[1] The Joint Task Force on Computing Curricula, *Computing Curricula 2001 Computer Science Education*, 2002, pp. 271-275.

[2] W. Marrero and A. Settle, “Testing first: Emphasizing Testing in Early Programming Courses,” in *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2005, pp. 4–8.

[3] B.S. Bloom, M.D. Engelhart, E.J. Furst, W.H. Hill, and D.R. Krathwohl, *Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain*, New York: David McKay Co Inc., 1956.

[4] L.W. Anderson, D.R. Krathwohl, P.W. Airasian, K.A. Cruikshank, R.E. Mayer, P.R. Pintrich, J. Raths, and M.C. Wittrock, *A Taxonomy for Learning, Teaching, and Assessing: A revision of Bloom's Taxonomy of Educational Objectives*, New York: Pearson, Allyn & Bacon, 2001.

[5] Gamma, Helm, Johnson, and Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995, pp. 14.

[6] Kim B. Bruce, *Foundations of Object-Oriented Languages: Types and Semantics*, Cambridge: MIT Press, 2002, pp. 18.

[7] Free On-Line Dictionary Of Computing, “Abstract Syntax Tree”, 1994, http://foldoc.org/abstract%20syntax%20tree.

[8] Robert C. Martin, “Visitor”, 2002, https://web.archive.org/web/20151022084246/http://objectmentor.com/resources/articles/visitor.pdf.

[9] W3C Web Services Architecture Working Group, Web Services Architecture, 2004, https://www.w3.org/TR/ws-arch/#relwwwrest.

[10] JavaParser contributors, “Java Parser and Abstract Syntax Tree”, 2017, https://github.com/javaparser/javaparser/blob/master/readme.md.

[11] S. H. Edwards, “Rethinking computer science education from a test-first perspective,” in *Companion of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, October 2003, pp. 148-155.

[12] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, “Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses,” in *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2006, pp. 13–17.

[13] Y. Liu, E. Stroulia, K. Wong, and D. German, “Using CVS historical information to understand how students develop software,” in *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.

APPENDIX A. SAMPLE TESTING TUTOR V2 PROTOTYPE INPUTS AND OUTPUTS

This section displays the content of a sample set of three inputs – the reference source code class under test, reference test class, and student or evaluation test class – and corresponding conceptual feedback output generated by the application suite in a text file. Note that the JUnit run feedback is omitted here since it's a supplement and likely will not be provided to the student as feedback, if it is then only a subset may be provided.

A.1. Inputs – Java Code Files

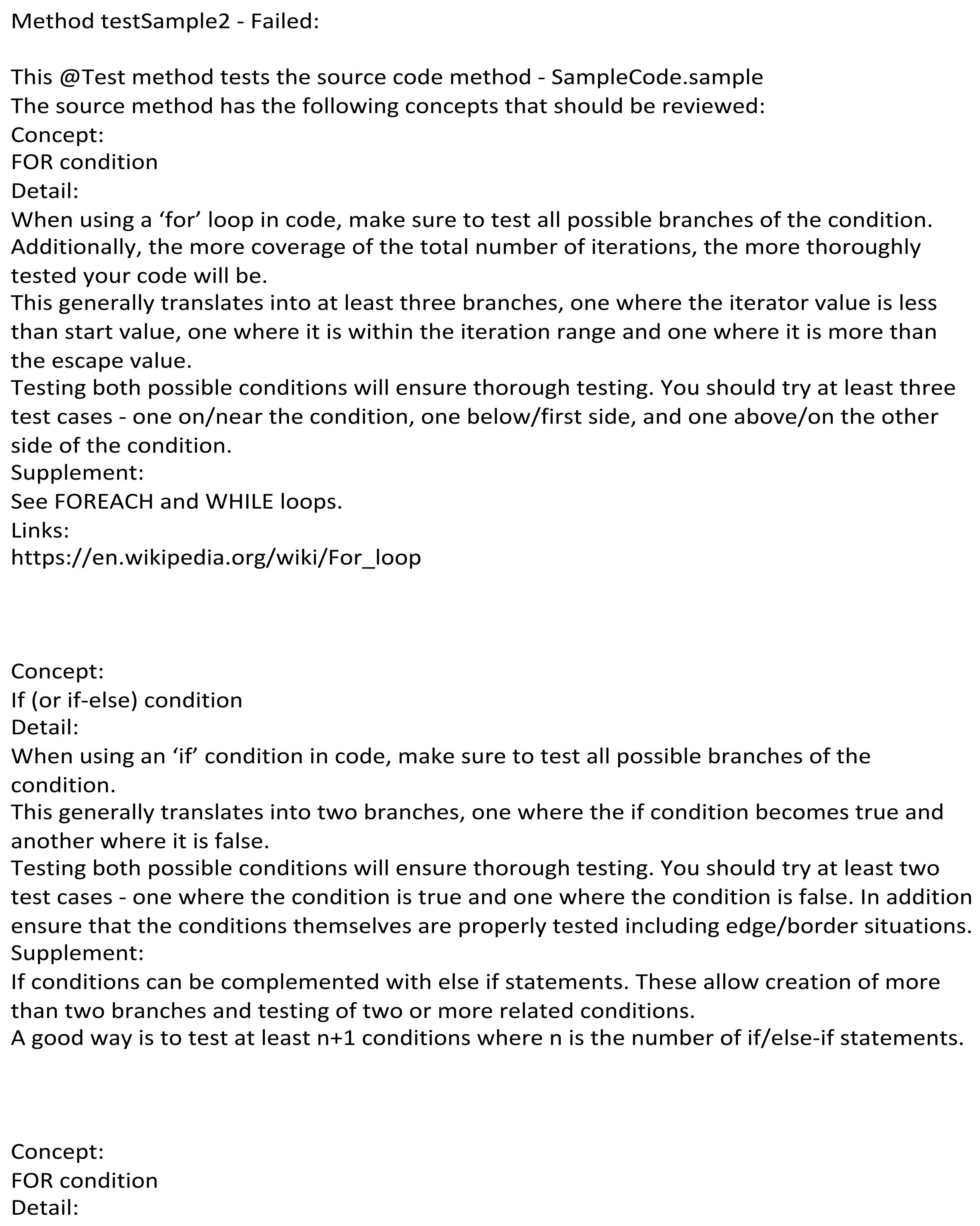
```java
package com.tt.code;

import java.util.ArrayList;
import java.util.List;

/*This is a test source code file*/

public class SampleCode {
      int a;

      public List<Integer> sample(int a, int b) {
            for (int i=0; i < 5; i++) {
                  if (i == a) {
                        System.out.println("SampleCode: a value - " + a);
                  }
            }

            for (int i=10; i < 25; i++) {
                  if (i == b) {
                        System.out.println("SampleCode: b value - " + b);
                  }
            }
            List<Integer> values = new ArrayList<>();
            values.add(a);
            values.add(b);
            return values;
      }
}
```
Figure A.1. Sample reference source code class

```
/**
 *
 */


import static org.junit.Assert.*;

import java.util.List;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import com.app.pckg.SampleCode;

/**
 * @author prakumar
 *
 */
public class SourceCodeTestSuite {

      private static SampleCode classToTest;
      private int a, b;

      /**
       * @throws java.lang.Exception
       */
      @BeforeClass
      public static void setUpBeforeClass() throws Exception {
            classToTest = new SampleCode();
      }

      /**
       * @throws java.lang.Exception
       */
      @AfterClass
      public static void tearDownAfterClass() throws Exception
{
      }

      @Test
      public void testSample() {
            a = 2;
            b = 21;
            boolean success = false;
            List<Integer> result = classToTest.sample(a, b);
            assertTrue(result.get(0) == a && result.get(1) ==
21);
      }

}
```
Figure A.2. Sample reference test class

```
import static org.junit.Assert.*;

import java.util.List;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

import com.app.pckg.SampleCode;

public class SourceCodeTestSuite {

      private static SampleCode classToTest;
      private int a, b;

      /**
       * @throws java.lang.Exception
       */
      @BeforeClass
      public static void setUpBeforeClass() throws Exception {
            classToTest = new SampleCode();
      }

      @Test
      public void testSample() {
            a = 2;
            b = 21;
            boolean success = false;
            List<Integer> result = classToTest.sample(a, b);
            assertTrue(result.get(0) == a && result.get(1) == 21);
      }

      @Test
      public void testSample2() {
            a = 2;
            b = 20;
            boolean success = false;
            List<Integer> result = classToTest.sample(a, b);
            assertTrue(result.get(0) == a && result.get(1) == 21);
            //fail("Not yet implemented");
      }

      @Test
      public void testSample3() {
            a = 2;
            b = 26;
            boolean success = false;
            List<Integer> result = classToTest.sample(a, b);
            assertTrue(result.get(0) == a && result.get(1) == 21);

      }
}
```
Figure A.3. Sample evaluation test class

A.2. Outputs – Content from Text Files

A.2.1. Testing Tutor Conceptual Feedback



```
Method testSample2 - Failed:

This @Test method tests the source code method - SampleCode.sample
The source method has the following concepts that should be reviewed:
Concept:
FOR condition
Detail:
When using a ‘for’ loop in code, make sure to test all possible branches of the condition.
Additionally, the more coverage of the total number of iterations, the more thoroughly
tested your code will be.
This generally translates into at least three branches, one where the iterator value is less
than start value, one where it is within the iteration range and one where it is more than
the escape value.
Testing both possible conditions will ensure thorough testing. You should try at least three
test cases - one on/near the condition, one below/first side, and one above/on the other
side of the condition.
Supplement:
See FOREACH and WHILE loops.
Links:
https://en.wikipedia.org/wiki/For_loop



Concept:
If (or if-else) condition
Detail:
When using an ‘if’ condition in code, make sure to test all possible branches of the
condition.
This generally translates into two branches, one where the if condition becomes true and
another where it is false.
Testing both possible conditions will ensure thorough testing. You should try at least two
test cases - one where the condition is true and one where the condition is false. In addition
ensure that the conditions themselves are properly tested including edge/border situations.
Supplement:
If conditions can be complemented with else if statements. These allow creation of more
than two branches and testing of two or more related conditions.
A good way is to test at least n+1 conditions where n is the number of if/else-if statements.



Concept:
FOR condition
Detail:
```

Figure A.4. Testing Tutor feedback

When using a ‘for’ loop in code, make sure to test all possible branches of the condition. Additionally, the more coverage of the total number of iterations, the more thoroughly tested your code will be.
This generally translates into at least three branches, one where the iterator value is less than start value, one where it is within the iteration range and one where it is more than the escape value.
Testing both possible conditions will ensure thorough testing. You should try at least three test cases - one on/near the condition, one below/first side, and one above/on the other side of the condition.
Supplement:
See FOREACH and WHILE loops.
Links:
https://en.wikipedia.org/wiki/For_loop

Concept:
If (or if-else) condition
Detail:
When using an ‘if’ condition in code, make sure to test all possible branches of the condition.
This generally translates into two branches, one where the if condition becomes true and another where it is false.
Testing both possible conditions will ensure thorough testing. You should try at least two test cases - one where the condition is true and one where the condition is false. In addition ensure that the conditions themselves are properly tested including edge/border situations.
Supplement:
If conditions can be complemented with else if statements. These allow creation of more than two branches and testing of two or more related conditions.
A good way is to test at least n+1 conditions where n is the number of if/else-if statements.

Method testSample3 - Failed:

This @Test method tests the source code method - SampleCode.sample
The source method has the following concepts that should be reviewed:
Concept:
FOR condition
Detail:
When using a ‘for’ loop in code, make sure to test all possible branches of the condition.
Additionally, the more coverage of the total number of iterations, the more thoroughly tested your code will be.

Figure A.4. Testing Tutor feedback *(continued)*

This generally translates into at least three branches, one where the iterator value is less than start value, one where it is within the iteration range and one where it is more than the escape value.
Testing both possible conditions will ensure thorough testing. You should try at least three test cases - one on/near the condition, one below/first side, and one above/on the other side of the condition.
Supplement:
See FOREACH and WHILE loops.
Links:
https://en.wikipedia.org/wiki/For_loop

Concept:
If (or if-else) condition
Detail:
When using an ‘if’ condition in code, make sure to test all possible branches of the condition.
This generally translates into two branches, one where the if condition becomes true and another where it is false.
Testing both possible conditions will ensure thorough testing. You should try at least two test cases - one where the condition is true and one where the condition is false. In addition ensure that the conditions themselves are properly tested including edge/border situations.
Supplement:
If conditions can be complemented with else if statements. These allow creation of more than two branches and testing of two or more related conditions.
A good way is to test at least n+1 conditions where n is the number of if/else-if statements.

Concept:
FOR condition
Detail:
When using a ‘for’ loop in code, make sure to test all possible branches of the condition.
Additionally, the more coverage of the total number of iterations, the more thoroughly tested your code will be.
This generally translates into at least three branches, one where the iterator value is less than start value, one where it is within the iteration range and one where it is more than the escape value.
Testing both possible conditions will ensure thorough testing. You should try at least three test cases - one on/near the condition, one below/first side, and one above/on the other side of the condition.
Supplement:
See FOREACH and WHILE loops.
Links:

Figure A.4. Testing Tutor feedback *(continued)*

https://en.wikipedia.org/wiki/For_loop

Concept:
If (or if-else) condition
Detail:
When using an 'if' condition in code, make sure to test all possible branches of the condition.
This generally translates into two branches, one where the if condition becomes true and another where it is false.
Testing both possible conditions will ensure thorough testing. You should try at least two test cases - one where the condition is true and one where the condition is false. In addition ensure that the conditions themselves are properly tested including edge/border situations.
Supplement:
If conditions can be complemented with else if statements. These allow creation of more than two branches and testing of two or more related conditions.
A good way is to test at least n+1 conditions where n is the number of if/else-if statements.

Overall Test Class Guidance:
3 number of tests for the method sample may be too many, consider an optimal way of reducing the number of tests for this method while testing all relevant conditions.

Figure A.4. Testing Tutor feedback *(continued)*

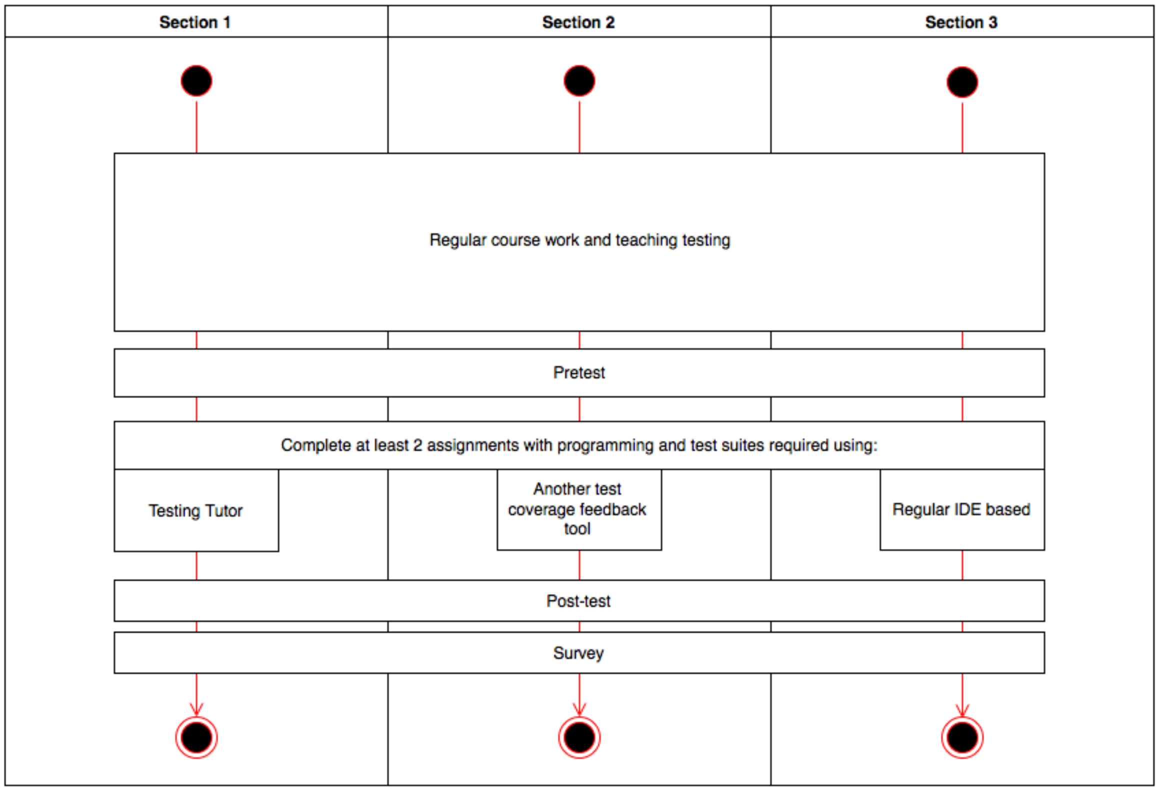
APPENDIX B. EMPRICAL STUDY TO EVALUATE TESTING TUTOR

This section details on how a prospective empirical study of the Testing Tutor tool may be carried out, along with data analysis, to determine the usefulness and fulfillment of its purpose detailed in this paper. An important disclaimer that this is a hypothetical study with generated data and is only presented as a sample of how a potential future study to evaluate the tool may be carried out.

B.1. Study Design

To evaluate the effect of the Testing Tutor tool on students' software testing skills and also determine the overall usefulness or presence of any issues in the tool itself for further improvement, we propose a longitudinal study that covers students enrolled in CS1, CS2 and CS3 in that particular sequence. Due to dual goals of the study the overall longitudinal study is further divided into three consecutive studies that are conducted in three courses over three semesters of introductory programming courses. This allows for a comprehensive evaluation of both, the tool itself – and any need for improvement – as well as its effect on student's testing ability. An overview for each of the three studies is provided below, followed by the hypotheses, design elements, experiment procedure and the data to be collected from these studies.

The first study (Figure B.1) will be of control group type and conducted in CS1. One group of students will be using Web-CAT and another group will be using a simple IDE (e.g. Eclipse) with a testing framework (e.g. JUnit) and instructor provided feedback, constituting the two control groups. It is also considered a pilot study for the Testing Tutor tool, as it will be the first experimental or practical use of the application by real users. From previous experiences of the author(s) of this, and already mentioned, papers we realize that there can only be some tasks at the end of the course semester that involve the students in terms of writing a class and complementing test suites. Keeping this in mind the study is conducted only during the final two assignments when the students have the required knowledge to accomplish the required tasks. The main goals of this study are to get an idea of the difference in testing ability of students caused by the use of different tools used and the difference in the type of feedback provided, along with determining any improvements that may be required to Testing Tutor.


Section 1
Section 2
Section 3
Regular course work and teaching testing
Pretest
Complete at least 2 assignments with programming and test suites required using:
Testing Tutor
Another test coverage feedback tool
Regular IDE based
Post-test
Survey
Figure B.1. First (control group) study – conducted in three different sections of CS1

The second study (Figure B.2) will also be of control group type and conducted throughout CS2 in the semester right after CS1. The main goals of this study are to compare the effects of Testing Tutor on students' testing skills against those of an existing test coverage tool such as Web-CAT. The secondary goal is to validate any improvements made to Testing Tutor and determine any further improvements that may be required. Along with CS1, this is the course the tool is eventually aimed at and hence the results of this study would be the most important. This study also allows us to track students' progress throughout a semester and continued progress of any students from CS1. This will include students who initially used Web-CAT and now use Testing Tutor and data pertaining to their progress can be analyzed, among all other such combinations.

The third is an independent study (Figure B.2) and is to be conducted in CS3 (or the logical successor to CS2 in the curriculum at a university where CS3 does not exist) when the enrollment of students progressing from the specific CS2 course where the second study was conducted, is expected to be maximum. The main goal of this study is to collect substantial data of students who have used Testing Tutor or Web-CAT previously (in CS1/2) and see how consequently using Testing Tutor affects their abilities to test code. Continually the study will also help evaluate the tool itself.

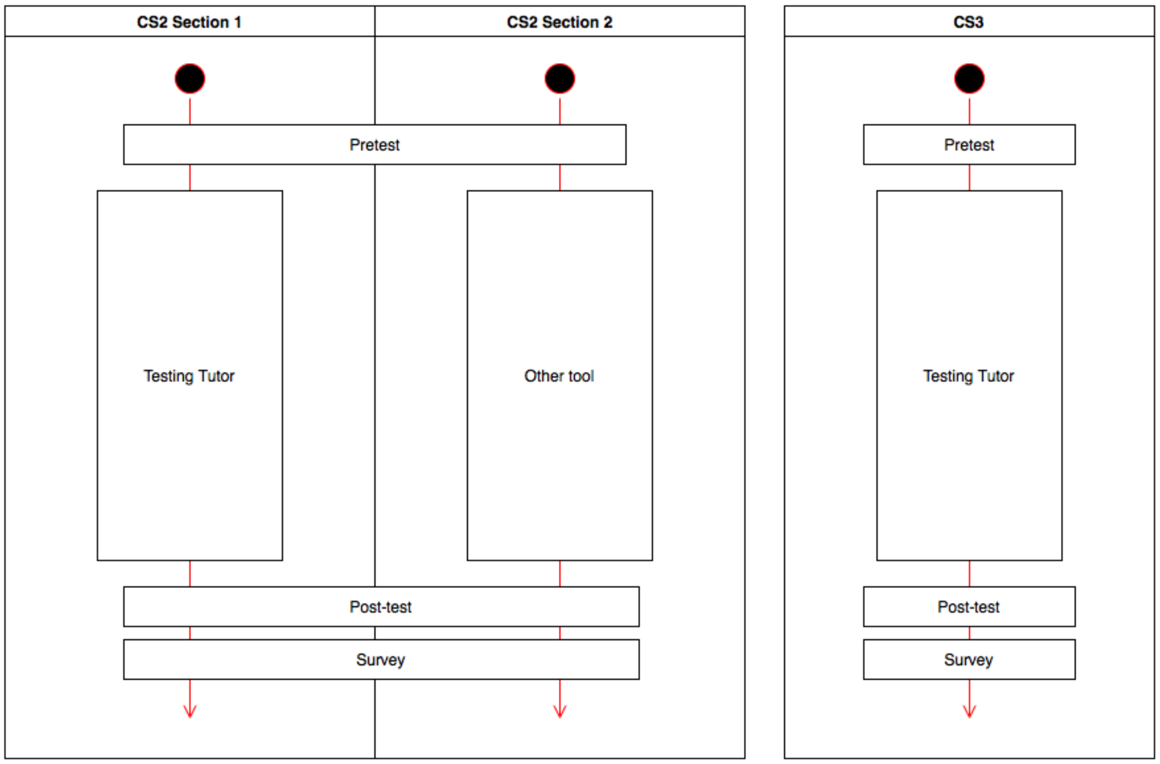


Figure B.2. Second and third studies – conducted in two sections of CS2 and CS3 respectively

B.1.1. Study Hypotheses and Variables

The following are the hypotheses we intend to test as part of this research:

Hypothesis 1*: Introducing the Testing Tutor tool (providing conceptual feedback) in introductory programming courses will help students improve their understanding of concepts and testing better than by traditional implementations.*

Hypothesis 2*: As the students continue to use the Testing Tutor tool (receiving conceptual feedback) they will make fewer mistakes or avoid making the same mistakes again.*

Hypothesis 3*: As the students continue to use the Testing Tutor tool (receiving conceptual feedback) they will create more comprehensive test suites.*

The following are the variables that are controlled and measured as part of this study:

Independent Variable(s): The tool used as a medium for completing the assignments and receiving feedback.

Dependent Variable(s): Correctness – based on failed tests, Comprehensiveness – based on test coverage, of the student test suites to determine their overall testing ability, Usefulness – based on logs along with instructor and student feedback.

B.1.2. Artifacts

The primary artifacts that are used as part of this study are assignment related. This includes the assignment and the reference test suite, both of which are provided by the course instructor after discussion with the researchers. These are generally basic due to the introductory programming level, but complex enough to include coding entire classes with multiple methods to support writing a test suite.

Other artifacts include the pre/post-tests for each study to determine students' understanding of testing, and the survey for each study that provides researchers with student feedback in relation to the study and the tool. The tests are small with 5-7 short answer questions in general, and the surveys are a total of 10-15 questions with 5-point Likert scale responses and two comment-based questions.

B.1.3. Participants

The primary participants of each of the studies are the undergraduate students enrolled in introductory programming courses (CS 1/2/3) at a university. These courses expect little to no knowledge (starting from CS1) of programming and build up the students' knowledge base incrementally. The number of participants differs based on the courses as the enrollment in them generally differs, decreasing as the level increases. This study assumes a conservative estimate of students enrolled in each, CS1 – three sections with 20 students each (60 total), CS2 – two sections with 20 students each (40 total), CS3 – 30 students in a single section.

B.1.4. Experiment Procedure

Each of the three studies follows a similar procedure and pattern for the experiment and thus a common procedure is described. The key point to remember is that while the first and second are control group studies, the third is an independent study and thus some minor procedural differences are present and noted in the relevant step. The major steps of the procedure are briefly described below.

1) *Step 1 – Pretest:* In order to determine the general software testing knowledge of students the instructor conducts a pretest before the study begins (beginning or middle of the course). The questions in this pretest concern general questions pertaining to software testing scaled to the level of the course, and are designed by the instructor in collaboration with the researcher. These are graded by the instructor on pre-decided scale such as 50 points and passed

on to the researchers with any instructor comments. These results do not factor in the student grades.

2) *Step 2 – Assignments:* This is a repetitive step in all the studies. The instructor creates a programming assignment and corresponding reference test suite that is uploaded to Testing Tutor. These are reviewed by the researchers to validate compatibility with the tool prior to incorporating them in the course. The students then work on the assignment and submit the code along with their test suites on the respective tool they are working with. The tool computes the results and provides feedback to the students. This feedback is aimed at reducing similar mistakes and improving testing knowledge for the same or consequent tasks.

3) *Step 3 – Post-test:* Is conducted at the end of the study/course and is similar to *Step 1 – Prestest*. The questions are of a similar type but different from the pretest.

4) *Step 4 – Survey:* At the end of the course the students are asked to answer a survey questionnaire that asks them about their experiences with the tool. Questions are similar to "How much did the tool hinder or assist you in completing the assignment tasks successfully?" to collect responses on the Likert scale, and also including a couple of questions that are open ended to allow students to explain their experiences in their own words. The researchers collect these responses.

5) *Step5 – Discussion:* After each study concludes, the researchers have a brief discussion with the instructor to gain insight into how the study design and the tool fared from their perspective. Some student survey responses may also be discussed at this time, either collectively or on an individual basis.

B.1.5. Data Collected

As detailed in the study design, a lot of different types of data is available. The data to be collected in all of the mentioned studies is more or less the same. Below we describe the crucial sets of data that we will collect, how it is collected (if not explained already) and what precisely that data is useful for, for all studies collectively.

1) *Assignment scores:* The most important and obvious data to be collected as part of all the studies is the individual scores/grades of each student in all the assignments that are part of the study. This data will be collected from the instructor at the end of the course. This data can help us track the continuous progress of each of the students through each of the courses and also help compare the performance of one group with another.

2) *Total number of test cases:* This is the number of test cases a each student writes per assignment. This data will also be available from the instructor, or alternatively it could be tracked through Testing Tutor if required. It will help us determine the trends in how effective (coverage/total) a student is when writing test cases.

3) *Test coverage:* The test coverage will part of the output from both Testing Tutor and Web-CAT tools, however Testing Tutor will likely be configured to provide this output only to the instructor. This is collected from the instructor for each student per assignment. It helps analyze how comprehensive a student's test suites are.

4) *Failed tests:* The number of tests for each assignment that fail for a student is determined through the tool(s) used and collected by the instructor. The failed tests help measure the correctness quality of each student's test suite that is analyzed to determine if the students are improving by making fewer mistakes.

5) *Testing Tutor logs:* The logs track overall test related data (that was submitted) for analysis such as types of concepts uncovered or failed, etc. This can help determine if students tend to avoid making similar mistakes after receiving feedback. These logs also provide data on any warnings, errors or general issues encountered when the tool was running which can help us troubleshoot and improve the tool.

6) *Pretest and post-test scores:* The sole purpose of these tests is to generally evaluate if the testing ability of the students experienced any change over the course of the study. The level of improvement (or unlikely worsening) in their scores is useful to track the effectiveness of the tool used.

7) *Survey results:* The researcher conducts the survey in class for all students. The responses are collected and will eventually help understand student experience of the study as a whole and their impressions of the tool used.

8) *Final grade:* The final grade is provided by the instructor after the course ends. The purpose of this is somewhat similar to that of post-tests and individual assignments. It provides an overview of how the students fared, and through the three studies it also provides incremental evaluation of the student.

B.2. Data Analysis

The data analyzed below is considered to be from the pilot study instead of the complete proposed experiment design for the longitudinal study. The pilot study, of control group type, is conducted in only two sections of CS1, one using Testing Tutor and one using Web-CAT, with 20 students each. The collected and analyzed data includes the scores for two assignments and corresponding test coverages (percentage value, based on number of lines of code tested) of each student, along with survey results for *Usefulness* of the tool used. The survey question was "*How* *useful did you find the tool used in achieving the goals of the assignments, particularly performing testing activities?*", which the students answer on a 5-point Likert scale that is assumed to have equally spaced intervals (Minimally Useful, Somewhat Useful, Fairly Useful, Very Useful, Extremely Useful).

The collected data allows us to compare the students' assignment scores (assignment 1 and assignment 2), test coverage (for each assignment) and survey responses for Testing Tutor as well as Web-CAT. We ran various statistical tests on it to confirm or reject our hypotheses.

B.2.1. Assignments

The assignments provided to both groups of students were the same. This allows us to compare the scores of students over both assignments. The means of the assignment scores for each student are listed in Table B.1. Comparing just the means shows that there is a noticeable difference between the performance of students in the two sections, with the students using Testing Tutor outperforming the students using Web-CAT. Running an Independent Samples T-Test on the two assignment scores for students in both the sections, however, shows that while the difference in Assignment 1 isn't significant, the difference in Assignment 2 scores definitely is. The mean increase in scores for students using Testing Tutor (**77.95-82.36**) is also noticeable when compared to students using Web-CAT (**75.92-75.35**), for whom it remains pretty much the same. However, the increase isn't significant as shown by a p value of 0.089 when running a One-Sample T-Test on Assignment 2 scores for students using Testing Tutor. The results from the statistical tests hence lead us to conclude that the performance of students in assignments did not significantly differ based on the tool, although Assignment 2 delivered promising results for students using Testing Tutor.

Table B.1. Mean assignment scores comparison

Ind. Samples T-Test	The Tool Used	Mean	Sig. (2-tailed)
Assignment 1 scores	Testing Tutor	77.9455	0.656
	Web-CAT	75.9225	
Assignment 2 scores	Testing Tutor	82.3590	**0.049**
	Web-CAT	75.3480	

B.2.2. Test Coverage

Test Coverage seems to have similar results as assignments, although even Test Coverage for Assignment 2 had no significant difference between the two sections. This should be expected as Assignments and Test Coverage would be correlated. Another test that made sense was whether the test coverage for students using Testing Tutor significantly increased for Assignment 2 compared to their Assignment 1 coverage and also compared to students using Web-CAT (Assignment 2 coverage). However, a One-Sampled T-Test using the means of Testing Tutor Assignment 1 (**77.5**) and Web-CAT (**71.3**) Assignment 2 as cut-offs also failed to show any significant results as shown in Table B.2 (means shown in Figure B.3). This leads us to conclude that while the Coverage for students using Testing Tutor was higher, as well as it increased from Assignment 1 to Assignment 2 as opposed to decreasing as seen for students using Web-CAT, we cannot conclude any significant impact of Testing Tutor on the students' testing ability, or on their progressive learning curve.

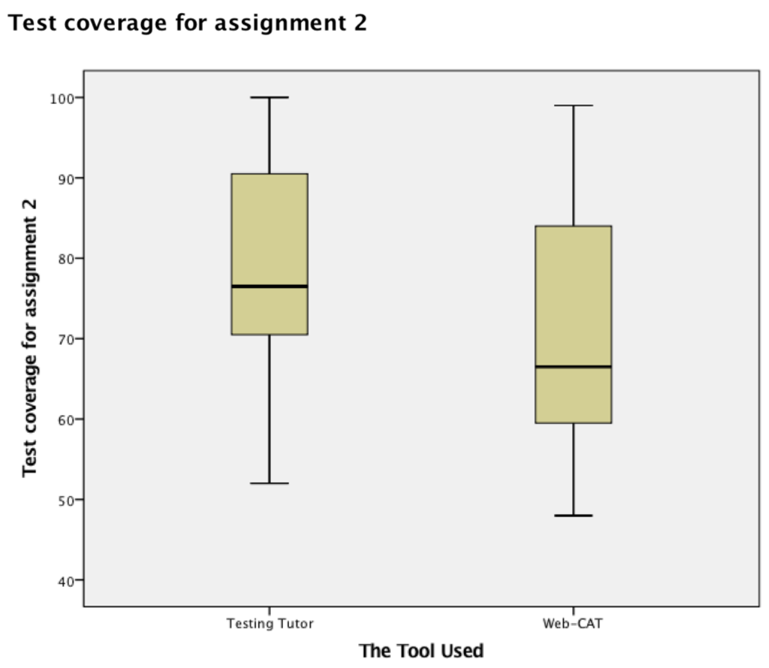


Figure B.3. Means of assignment 2 coverage for Testing Tutor and Web-CAT

Table B.2. Mean coverage comparisons of Testing Tutor assignment 2 coverage

One-Sample T-Test	Test Value	Sig. (2-tailed)
Testing Tutor Assignment 1 Coverage	77.5	0.952
Web-CAT Assignment 2 Coverage	71.3	0.064

B.2.3. Usefulness Survey

Responses to the survey were promising for Testing Tutor, shown in Figure B.4. Students using Testing Tutor averaged a score of **3.45** on the 5-point Likert scale, while students using Web-CAT averaged a score of only 2.80. However, a Mann-Whitney Test concludes that this difference is insignificant, as seen in Table B.3. Again, this result is promising since the mean score on the survey for Testing Tutor was higher than that of Web-CAT, and also higher than the midpoint. This indicates the students leaned towards a favorable response for a tool that is still in its infancy. These results are easily seen in the graph in Figure B4, where no students gave

Testing Tutor a rating below 2, and quite a few gave it a 5 (more than students who gave Web-CAT a 5). Further survey of instructors and students in planned future studies are likely to make this difference significant, particularly since the tool is expected to have evolved further and the studies would be more comprehensive.

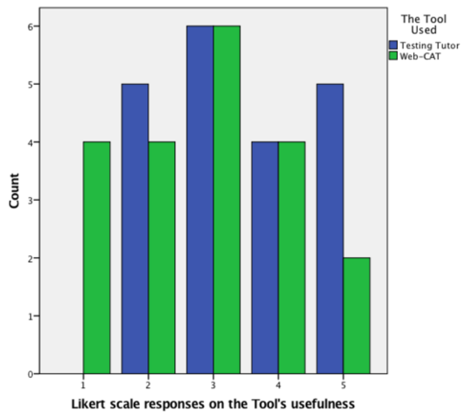


Figure B.4. Survey results by tool

Table B.3. Survey response significance

Mann-Whitney Test	Asymp. Sig. (2-tailed)	Exact Sig. [2*(1-tailed)]
Testing Tutor Usefulness Survey	.126	.142

B.3. Threats to Validity

One of the primary validity threats is that the tools used in the study operate differently. While Testing Tutor reduces instructor workload by analyzing the submissions but leaving the grading to the instructor, Web-CAT takes care of the entire process. Using these tools on separate sections and possibly having the same instructor for all may mitigate this threat, as there is some consistency applied in grading. Similarly, Web-CAT allows students to run their code against provided tests any number of times while Testing Tutor provides a Training Mode to exercise on and customizable feedback. Determining a specific submission process such as none or only two tries allowed for each submission can control the amount of feedback.

Another major threat to validity is that during the first study (at least) the tool will not be perfect and unlikely to be at its user-tested version. Even though it is expected that the tool be thoroughly tested before it is actually used by instructors and students in a real course environment, some issues are to be expected. This is partly a study design flaw as the first study is also used as a pilot run for the Testing Tutor tool. The effect could be that the results from the first study (assuming the issues found are fixed before further studies) vary significantly from any consequent studies. However, this validity threat is actually biased against the Testing Tutor tool, as an incomplete (worse) tool should decrease the students' performance.

Comparing data from the sections that used a tool in the first study against the section that relies on manual grading and feedback introduces the threat of a positive bias towards Testing Tutor in the results due to a lack of motivation for that particular section. Since subjects that work on or with something new tend to display more motivation for the tasks. Having them use an IDE with a testing framework to write tests mitigates this threat with the assumption that it will still be a new tool for new students of computer science.

Although unlikely, there may be students who may be aware of or might have used the tool used for the control group. This may positively or negatively bias the study in equal measure. Their familiarity could improve the results of the control group since they do not need to worry about the environment and tool used and can solely focus on the task at hand, or it may cause them to not gain sufficient motivation a new tool may provide and have a negative effect on their results.

Another validity threat is that a bunch of students may drop the course or refuse to participate in the study. This would reduce the available data points for analysis. Similarly, a number of students that were part of the study in CS1 may not be part of the study in CS2 (same applies from CS2 to CS3) due to drops, change of majors, different course paths etc.

As the study progresses to CS3 some assignments may start to become too complex for Testing Tutor to be of use in its current proposed form. To avoid this to impact the results the authors will consult with the instructor before the study is conducted and may decide to restrict the study only to certain assignments that conform to the tool's proposed application. While this potentially introduces another threat of researcher bias in the study, this is mitigated by the involvement of the course instructor.

B.4. Summary and Discussion of Results

All the statistical results simply point to the fact that the analyzed data is inconclusive. Partly such results are to be expected as this was only a pilot run for Testing Tutor, which is still in its infancy, as compared to Web-CAT that has been available for a while and already used in courses at at-least one university.

We cannot reject the null hypotheses that there is no significant difference between Testing Tutor and Web-CAT in terms of their impact on students' understanding of testing concepts, making fewer consequent mistakes and their ability to create more comprehensive test suites. These results are discussed in detail below.

1) ***Hypothesis 1****: Introducing the Testing Tutor tool (providing conceptual feedback) in introductory programming courses will help students improve their understanding of concepts and testing better than by traditional implementations.*

Based on the results of the study we cannot reject the null hypothesis that there is no significant difference caused to students' understanding of the concepts and testing when using Testing Tutor in introductory programming courses when compared to traditional implementations. This can be seen from the insignificant differences in their assignments scores and test coverages over the two sections using different tools. However, even if insignificant the performances of students using Testing Tutor were better which is promising considering this was only a pilot study.

2) ***Hypothesis 2****: As the students continue to use the Testing Tutor tool (receiving conceptual feedback) they will make fewer mistakes or avoid making the same mistakes again.*

Since the test coverage increase from Assignment 1 to Assignment 2 was insignificant for students using Testing Tutor, we cannot reject the null hypothesis that Testing Tutor has no effect on the number of mistakes students make progressively or prevents them from making the same mistakes again. However, a noticeable improvement was seen among students from one assignment to the next, even if insignificant, showing the potential of conceptual feedback. It is our belief that carrying such an experiment over a longer period of time and tasks will definitely show the benefits of conceptual feedback. A more in depth analysis is also required to see how individual students fared from one assignment to the next, and which concepts they made mistakes in all the assignments.

3) ***Hypothesis 3****: As the students continue to use the Testing Tutor tool (receiving conceptual feedback) they will create more comprehensive test suites.*

This hypothesis wasn't specifically tested for in the pilot study that has been detailed and analyzed in this paper. However, looking at the lack of significant improvement in test coverage for students using Testing Tutor over those using Web-CAT, and students using Testing Tutor from Assignment 1 to Assignment 2, we cannot reject the null hypothesis that students using Testing Tutor tool has no impact on the comprehensiveness of their test suites. However, looking at the means for test coverage for Testing Tutor students, particularly in Assignment 2, we can see the improvements over their own scores in Assignment1 and students using Web-CAT.

These results are definitely promising, however at this point insignificant. We need to build on the current study and follow up with multiple further studies to completely and comprehensively evaluate Testing Tutor and its potential benefits.