SPEST - A TOOL FOR SPECIFICATION-BASED TESTING

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Corrigan Johnson

January 2016

COMMITTEE MEMBERSHIP

TITLE:	SPEST - A Tool for Specification-Based Testing
AUTHOR:	Corrigan Johnson
DATE SUBMITTED:	January 2016
COMMITTEE CHAIR:	Gene Fisher, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	David Janzen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Associate Professor of Computer Science

ABSTRACT

SPEST - A Tool for Specification-Based Testing

Corrigan Johnson

This thesis presents a tool for SPEcification based teSTing (SPEST). SPEST is designed to use well known practices for automated black-box testing to reduce the burden of testing on developers. The tool uses a simple formal specification language to generate highly-readable unit tests that embody best practices for thorough software testing. Because the specification language used to generate the assertions about the code can be compiled, it can also be used to ensure that documentation describing the code is maintained during development and refactoring.

The utility and effectiveness of SPEST were validated through several experiments conducted with students in undergraduate software engineering classes. The first experiment compared the understandability and efficiency of SPEST generated tests against student written tests based on the Java Modeling Language (JML)[25] specifications. JML is a widely used language for behavior program specification. A second experiment evaluated readability through a survey comparing SPEST generated tests against tests written by well established software developers. The results from the experiments showed that SPEST's specification language is at least understandable as JML, SPEST's specification language is more readable than JML, and strongly suggest that SPEST is capable of reducing the effort required to produce effective tests.

ACKNOWLEDGMENTS

Deepest gratitude to Gene Fisher for the inspiration and motivation to pursue specification testing and the continued support in design and development of the project. Additional thanks David Janzen and John Clements for taking the time to evaluate and analyze this work. Further thanks to Daniel Gerrity for assistance in development and design of components used in the final product.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

Introduction

Testing is a fundamental part of the software engineering process and is an area of continued research [39]. As software becomes a larger part of people's daily lives, the need to verify that programs perform as expected has grown. Advances in testing strategies from the last decade have led to many significant improvements; however, many challenges in this area remain unsolved [39].

Specification languages can be used to provide an unambiguous description of code's behavior and as a result can be used to generate black-box software tests [20]. Research on the benefits and applications of formal specifications has primarily focused on formal verification; however, there is less research on the use of specifications for testing. The primary distinction between verification and testing is the level of correctness guaranteed. While specification languages can reduce time spent testing, their complexity has led to limited adoption by the development community[50]. An easy to use specification based test generation tool has the potential to increase the usage of specification languages and relieve some of the burden of writing software tests by hand.

This thesis describes SPEST, a black-box test generation tool designed to increase the value of formal specification comments in code and facilitate quicker and easier generation of unit tests. The tool works by using specifications in comments above Java methods to create unit tests. The syntax used within SPEST comments is derived from predecessor specification languages such as Z [46] and Eiffel [31] but is most similar to the Java Modeling Language (JML) [25]. The simplicity of SPEST differentiates it from its predecessors; the syntax used by SPEST is designed specifically for testing and is general enough to be adaptable to support multiple source languages with minimal changes to the syntax. SPEST's syntax allows the user to create boolean expressions that describe the expected behavior of their code. Additionally, the user can validate that their comments are syntactically and type correct by using SPEST to compile their specifications.

A black-box test generation tool, that is easy to use, may help encourage developers to write specifications, increase the quality of their code, and reduce development time [52]. Furthermore, by creating a tool that is capable of being used in conjunction with the latest versions of Java, developers following the design by contract design paradigm will be able to use the tool to validate their contracts as well as their source code.

1.1 Description of the Problem

The specific problem addressed in this thesis is how to make a black-box test generation tool which improves upon the current state of the art tools for Java. The current problems not addressed by the state of the art black-box test generation tool for Java, the Java Modeling Language Unit Next Generation (JMLUnitNG)[52], include generating data for generics, bounding unbounded sets of data, creating data sets for universal quantifiers, creating tests that can be executed in reasonable amounts of time, creating data that meets a user's specifications, and creating tests that are human readable. SPEST aims to provide a solution to these problems. While there are other specification based test generation tools for languages other than Java, these are outside the focus of this thesis, except as topics for discussion in related work.

The more general problem addressed in this thesis is how to create a specification language for Java that is simpler than existing specification languages and that focuses specifications on testing as opposed to formal program verification. Identifying and removing the keywords within the Java Modeling Language that only pertain to formal verification and identifying ways to bring the specification language closer to the source language will likely reduce the effort required to learn the language. Furthermore, identifying unnecessarily Java-specific features that prevent the specification language from working across multiple source languages will reduce cross language barriers and may help attract a larger community of users.

Specification languages are used to formally describe the behavior of a system at a higher level than a programming language. In this context, the problem of improving development is to allow developers to write fewer lines of testing code through the use of specifications while still maintaining the highest quality code possible. The problem of generating inputs for unit tests is of particular interest in this thesis. The more general problem discussed throughout this thesis is how black-box test generation can be used to produce high quality unit tests, particularly in a teaching environment.

Writing unit tests by hand can be a challenging and tedious process. Developers must understand what the code under test is doing and put careful thought in to what values should be tested while writing the same boilerplate template code around their assertions. Developers must not only understand the object under test but the data structures that it relies on and how to construct them as well.

Black-box test generation can effectively reduce the amount of domain knowledge needed to write unit tests while removing the tedium of writing boiler plate code [35] [51]. This can help reduce the amount of time spent writing tests and analyzing objects that are only indirectly relevant while forcing the developer to document their assumptions. Per Runeson warned "it's difficult to motivate the developers to execute unit tests" [44]. Unit testing takes time and can be difficult to justify the benefit-cost ratio because of the short term delays and increased short term costs associated with it. Developers' lack of interest in black-box test generation is a testament to the fact that black-box test generation frameworks have not progressed to a mature state where developers can clearly evaluate the benefits they give. To facilitate faster and stronger unit testing, tools and methods that are capable of removing the tedium of writing boiler plate code and reducing the delays created by testing are needed.

1.2 Overview of the Solution

This thesis aims to provide a means to provide high quality unit tests in less time than they could be written by hand as well as demonstrate that the implementation of this tool is capable of performing as well as the state of the art tool in Java, JMLUnitNG, in delivering readable and performant unit tests that students are able to understand while being flexible enough to be expanded to languages other than Java in the future.

The work of this thesis is to create a black-box test generation tool, SPEST, that adds new capabilities that have not been available in its predecessor tools. This tool provides a new specification language, a parser to interpret the language, a test generation mechanism, and a data generation tool. The parser is created by defining a grammar that describes valid tokens and the order the tokens can appear using the tool ANother Tool for Language Recognition (ANTLR)[42]. The test generation mechanism is a collection of modular services that are responsible for analyzing the specifications to determine information such as assertions and outputting the information to a test file. The data generation tool uses Java's reflection to dynamically instantiate objects and modify their state to meet the requirements interpreted from parsing a user's specifications.

The primary foundations for SPEST are the Java Modeling Language and the corresponding test generation tool, the Java Modeling Language Unit Next Generation[52]. SPEST's language was built by picking only the keywords essential to testing from the Java Modeling Language and modifying this subset to more closely resemble the languages it describes. SPEST's data generation is a modified version of the approach taken in JMLUnitNG that enforces smaller data generation limits to reduce the runtime of tests. Several of the limitations of JMLUnitNG also drove core design decisions such as not making SPEST's parser a full Java compiler as well as storing additional information, such as generics, that a typical Java compiler does not retain.

1.3 Outline of the Thesis

What follows in Chapter 2 is a description of background and related work, which covers testing, specification languages, and test generation tools. Chapter 3 describes the design of the language and its intended goals while Chapter 4 dives into the subsequent process created from the design. Chapter 5 demonstrates and describes SPEST's capabilities. Chapter 6 discusses the experiments used to validate that SPEST's readability, simplicity, and test efficiency as well as the results from the experiments. Chapter 7 concludes with a summary of contributions and chapter 8 lists potential future work opportunities.

Chapter 2

Background and Related Work

2.1 Unit Testing

Addressing how to improve software quality has been a long standing concern amongst developers. Modern programming languages now support unit testing frameworks designed to facilitate the execution of small units of code to verify the source code's behavior [44]. Two fundamental factors in the success of unit tests are quick feedback and the ability to isolate the smallest modules in the system that can be tested [44]. Tests that run quickly allow developers to run them more often and get quicker feedback. Isolation means the dependencies of a method are mocked so that the behavior shown can only be the result of the method under test. Mock objects are substitutions for implementations of code that aid with testing by creating objects that provide known results for known inputs in tests [28].

Java has two prominent testing frameworks, JUnit [6] and TestNG [7]; similar xUnit frameworks are also available in other languages. These frameworks are designed to support writing and running tests and provide a graphical user interface that displays information about which tests passed and failed. JUnit was launched in 2005 and distinguished itself through its simplicity and its functionality which removed the need for testing print statements and made tests reusable and extensible [6]. TestNG was launched after JUnit in 2007 with the goal of providing functionality that JUnit did not include such as defining dependencies between tests. JUnit and TestNG also serve as the foundation for collecting additional information such as line coverage and mutation scores.

2.2 Specification Languages

Specification languages are a tool that allow developers to model and specify requirements [30]. Behavioral interface specification languages are a subset of specification languages that use annotations

such as preconditions and postconditions containing assertions to describe the expected behavior of a code block [19]. Preconditions and postconditions can be as simple as english statements that define what must be true before and after a program is run. Formal languages that go beyond natural language have the capability to remove natural language's ambiguities and allow specifications to be analyzed for completeness and consistency.

Preconditions and postconditions were first introduced in 1967 in a seminal paper by Robert Floyd where he described an approach to formally prove things about the code being described [15]. Floyd's work was a purely mathematical presentation, not based on a compilable language. Compilable languages would become the next key development in the evolution of specification languages. Early examples of specification languages that could be compiled included Larch [17] and Z [46]. While these tools provided a means to describe the code and they were able to be syntactically validated and type checked, they failed to produce anything directly usable for testing. This problem was later addressed by the Assertion Definition Language (ADL) [9]. ADL was originally designed to work with C programs and had a companion tool, the ADL Translator (ADLT), that had the capability to generate program tests from the specifications [9]. Most of these specification languages have gone widely unused in large part because of the lack of desire for testing until more recent years and an uncertainty about the value added by the tools relative to their costs.

2.3 Software Processes

To help organize the software development process, many developers follow software design processes such as design by contract [32], test driven development [5], and specification driven development [40]. These processes describe how requirements will be collected and when development and testing should take place in order to create an optimal process for developers to follow [43]. Each approach has strengths and weaknesses that result from the prioritization of some tasks before others as well as the lengths of the cycles before the processes are repeated. This section describes several design processes that prioritize early testing and defining specifications early.

2.3.1 Design By Contract

Design by contract, a concept first introduced by Bertrand Meyer, is a software methodology whereby preconditions and postconditions are used to document the behavior of a method [32]. This methodol-

ogy was proposed to counter the idea of defensive programming. In defensive programming, developers check that all inputs are valid. Validating all inputs leads to redundant checks and unnecessary code. Design by contract reduces the amount of validation done within a method and can be used to reduce unit testing if the contracts are executable assertions [18]. If the contracts in this technique are written in an assertion language, they can be compiled and run with the source code. Contracts are one of the strongest forms of documentation because their ability to be executed with the source code improves their maintainability. Design by contract suffers from challenges of converting requirements into specifications that concretely describe the exact behavior in all situations [18].

2.3.2 Test Driven Development

Test driven development is a software process in which a developer must write a unit test before writing production code and then writing just enough production code to satisfy the unit test as well as refactoring [5]. This development process gained popularity as a fundamental part of eXtreme Programming. The goal of this style is not pure testing but instead to form the specifications necessary for development before attempting to develop code. Because the tests are the specifications, documentation is a secondary consideration of this approach. While studies have shown test driven development to have little impact on code coverage or mutation score [48], test driven development may reduce computational complexity and provide other design benefits [21]. One of the key problems with this approach is that the goal is passing all the tests instead of having the desired behavior. Ideally the tests should reflect the desired behavior but this is not always the case. Tests are best for capturing low level desired behavior but fail to paint a larger picture for the program as a whole [40] [33].

2.3.3 Specification Driven Development

Specification driven development attempts to take advantage of both design by contract and test driven development. While some have argued that the two concepts are opposites, the effectiveness of the collaboration of these concepts has been shown in the context of the Eiffel language and environment [40]. Both techniques have the same goal, converting requirements into verifiable constructs, and using specifications to describe this goal. While the approaches taken by each of the techniques differs, both can be applied to different aspects of the same project. When a formal specification is too complex, tests can break the problem into smaller units before development; however, when documentation is needed

to identify larger aspects, design by contract can be used. The key to this approach is deciding which technique to use based on the given situation instead of choosing one and using it for the whole project.

2.4 Approaches to Automated Testing

Unit testing is essential to quality control and maintaining large systems. There are several approaches to automating the task of unit testing: white-box, grey-box, black-box, fuzzy, and many other techniques [39]. The largest difference between these approaches is the technique by which the tool determines the expected behavior. Black-box testing is a testing process in which the test ignores the implementation of the code and focuses solely on values that are returned [8]. In black-box test generation, the developer specifies the expected behavior for each method as boolean expressions above each method, usually in the form of a comment. Comments can be as simple as informal or semi-formal english descriptions or formal and compilable boolean expressions. The black-box test generation tool is responsible for parsing a developer's specifications and creating unit tests.

2.4.1 Machine Learning

In many environments, the information required for a test generation tool, such as the source code, are not available. Machine learning offers a new solution to analyzing the models and dynamically generating effective tests. A model inference engine can be trained from existing test inputs and outputs and used to guide future selection of test inputs and the expected test outputs [41]. Model inference is extremely powerful because it does not require the developers to perform additional tasks to improve the quality of their tests; however, the success of this approach is strongly tied to the availability and strength of existing tests.

A unified test representation and advice framework can expand the pool of test information that can be analyzed and used to improve machine learning approaches that rely on model inference to guide their test generation. Analysis from frameworks such as the Test Advice Framework [49] can provide information about combinations of invocations that lead to different states within the model as well oracle generation. While this approach can be used in conjunction with machine learning approaches to produce more accurate oracles, the approach places a larger burden on developers by requiring them to transform their tests to meet the new standard unified test representation.

2.4.2 Black-Box Test Generation Tools

Two tools, Eiffel [31] and JMLUnitNG [52], have grown into major efforts and have greatly influenced the development of specification languages and black-box test generation tools. This section describes the fundamental structure of these tools and how they are used to verify the behavior of code.

2.4.2.1 Eiffel

Eiffel is an object-oriented programming language that was designed to produce high quality software [31]. To achieve its goal, Eiffel has formal specification constructs built into the language that are capable of being compiled and executed with the code. There are three levels of execution for Eiffel specifications: none, preconditions only, and preconditions in conjunction with postconditions [31]. Because the assertions are not part of the comments but are instead a part of the language, Eiffel provides the keyword "deferred" to allow specifications to be written for methods that do not have an implementation yet. Eiffel has influenced and been cited by other specification languages such as iContract [24], Java with Assertions [3], jContractor [23], and JML [25].

In additional to Eiffel's internal tools for verifying preconditions and postconditions at runtime, external tools, such as AutoTest [27], can be used to create and execute tests. These tools use the preconditions and postconditions in the source code as the assertions in the tests and randomly generate input values to test the method with. The tests that are generated are bound by user constraints that specify the cutoff time for the invocations, the maximum number of invocations, the timeout time for the routine, and a seed used when randomly generating the input data.

While Eiffel provides strong support for testing, there are several limitations to determining the source of the problem through debugging. Known limitations of the debugger include:

- No support for Agents
- Some objects are not able to be instantiated
- The wrong assertion tag may be shown
- Evaluating some types of expressions may cause the debugger to crash

[47].

2.4.2.2 JMLUnitNG & JML

The state of the art technologies for creating black-box tests in Java are the Java Modeling Language (JML) in conjunction with the Java Modeling Language Unit Next Generation (JMLUnitNG). JML is a design by contract language that works with Java methods and objects [25]. JMLUnitNG relies on a JML translator known as OpenJML. OpenJML is responsible for translating JML specifications into SMT-LIB format but is only compatible with versions of Java up to version 7 [11]. JMLUnitNG is a testing framework for Java code and is able to create unit tests for code using JML annotations [52]. The tests generated by JMLUnitNG are executable by JUnit 4, which is widely used for testing Java-based systems.

JMLUnitNG was built as a successor to JMLUnit [10]. JMLUnit, like JMLUnitNG, transformed JML specifications into unit tests. JMLUnit had three fundamental problems: lack of input generation, memory utilization, and copyleft licensing. JMLUnit was limited to generating test data for primitive types and relied on the developer to generate test data for non-primitive data types. Furthermore, JMLUnit performed exhaustive testing which led to memory exhaustion and an unreadable set of test cases. JMLUnit's memory issues stem from the fact that the tool creates the entire JUnit test suite in memory before running any of the test.

JMLUnitNG was created to solve the existing issues with JMLUnit. The tool uses Java's reflection to generate non-primitive data types [52]; however, this approach has its own limitations. The generated objects have an arbitrary depth for recursive data structures and the objects do not automatically respect preconditions. In order to better utilize memory, JMLUnitNG uses a lazy approach to generate required parameters for the methods [52]. Furthermore, unlike JMLUnit, JMLUnitNG can record the results of failed tests on the disk with as much detail as necessary which conserves memory usage [52]; however, as of January 2015, JMLUnitNG does not support systems built with Java 8 or later [11].

JMLUnitNG was a large step forward for black-box test generation prior to Java 8, but the tests were still far from usable. In a comparison between JMLUnit tests and JMLUnitNG tests, JMLUnit tests took 10 seconds to run, as opposed to 3 hours taken by the JMLUnitNG tests. In test and specification driven development, running the tests after making small changes is imperative to verifying the behavior of the code; however, because JMLUnitNG takes so long to run unit tests, it is unable to provide any real time feedback to developers. Furthermore, JMLUnitNG's inability to apply preconditions to object generation severely complicates the generation process. The development of SPEST is

strongly motivated by the desire to overcome the limitations of JMLUnitNG.

Chapter 3

Design

SPEST has a language and a collection of services that can be used to generate unit tests dynamically and provide services to assist with object generation and Java's reflection. The language is used to help users describe the expected conditions before a method is run and the results of running a method. This chapter describes how SPEST uses feedback and minimalism to reduce the learning curve required to use the tool and how the services composing SPEST can be used to lessen the effort required to unit test.

3.1 Language

SPEST's keywords are a refined and revised subset of keywords from well-known notations for formal specifications and are influenced specifically by JML's keywords. Table 3.1 shows SPEST's keywords, the equivalent JML keyword, the keyword's meanings, and an example usage of the keyword. SPEST specifications are found in Java document style comments denoted by "/*" to start and "*/" to end the comment. By keeping the SPEST specifications in comments, they do not interfere with the source compiler. This style of comment, as opposed to JML's comments denoted by "/*@" and "@*/", was chosen because SPEST specifications supplement or even replace Java document comments since they are very explicit about expected inputs and outputs. Other modifications to the JML keywords were chosen to make the language feel closer to the target language. For example, "\result" was replaced with "return" because "return" is a keyword that the most developers will recognize.

Keyword	JML Keyword	Purpose	Example Usage
pre:	requires	Precedes statements that must be true before running the method.	pre: field == true
post:	ensures	Precedes statements that must be true after running the method.	post: field == true
forall	\forall	Precedes expressions that must be true for all objects from the universal collection of objects bound by a constraint.	forall(int i; i > -1 & i < 100; list[i] < list[i + 1])
exists	\exists	Precedes expressions that must be true for at least one object bound from the universal collection of objects bound by a constraint.	forall(int i; i > -1 & i < 100; list[i] < list[i + 1])
' (Prime)		Describes the value of the variable after running the method.	field != field'
return	\result	The returned value from running the method.	return == 4
if/else	a ==> b	Describe expressions that must be true under given conditions.	if(a > b) (a == 4) else (a == 5)
iff	a <==> b	Describes two expressions that must both be true or must both be false.	(a == 4) iff (b == 5)

Table 3.1: SPEST keyword meanings

To encourage developers to test all of their code's functionality through unit tests, SPEST implicitly ignores access restrictions on fields and methods. JML allows users to do bypass privacy restrictions for individual fields and methods through the annotation "/*@ spec_public @*/" after the Java privacy setting [26]; however, annotating every private field and method can be tedious and results in code littering. Code littering is the result of adding comments that hold no value to a user reading them but provide information to a tool that is using them. Ideally all comments should provide some direct use to the developer. Specifications should not be perceived as code littering because they provide users with exact knowledge of what restrictions are placed on the inputs and outputs.

3.2 Usability

Predecessors to SPEST, such as JMLUnitNG, have relied on command line interfaces to allow tools to specify file selections, validate their specifications, and generate unit tests. While command line interfaces do facilitate quick development for advanced users, they are unintuitive to first time users. Furthermore, command line interfaces have limited formatting capabilities which can make it more

difficult for users to understand their output.

Clear feedback is essential for users learning how to use a tool and in iterative processes that allow users to quickly make changes and check their work. The feedback users receive should be concise and describe only the relevant information to the error. In the case of the current state of the art tool for Java black-box test generation, JMLUnitNG, the error message produced on Windows 7, shown in figure 3.1, is cryptic and excessive. The error message only reveals information about an internal problem with the tool while the real problem was that the tool did not support all of the functionality of Java 8. When asking the creator of JMLUnitNG, Daniel Zimmerman, what the errors meant, his response was more questions about the environment and versions being used. Zimmerman's response made it clear that the feedback was not useful for determining how to fix the problem [2]. Further complicating the problem was the fact that the error message differed depending on the operating system it was running on. Figure 3.2 shows a different error message from running the same version of the tool on the same source code on Mac OS X. When working in a collaborative environment, it is imperative to have repeatable results so that procedures can be documented and developers are able to help each other overcome common errors.

```
Exception in thread "main" java.lang.AssertionError
    at com.sun.tools.javac.util.Assert.error(Assert.java:126)
    at com.sun.tools.javac.util.Assert.check(Assert.java:45)
    at com.sun.tools.javac.code.Scope.leave(Scope.java:150)
    at com.sun.tools.javac.comp.Attr.visitBlock(Attr.java:912)
    at com.sun.tools.javac.comp.JmlAttr.visitBlock(JmlAttr.java:613)
    at com.sun.tools.javac.tree.JCTree$JCBlock.accept(JCTree.java:781)
    at com.sun.tools.javac.comp.Attr.attribTree(Attr.java:431)
    at com.sun.tools.javac.comp.Attr.attribTree(Attr.java:418)
    at com.sun.tools.javac.comp.Attr.attribStat(Attr.java:480)
    at com.sun.tools.javac.comp.Attr.visitIf(Attr.java:1276)
    at com.sun.tools.javac.tree.JCTree$JCIf.accept(JCTree.java:1140)
    at com.sun.tools.javac.comp.Attr.attribTree(Attr.java:431)
    at com.sun.tools.javac.comp.Attr.attribTree(Attr.java:418)
    at com.sun.tools.javac.comp.Attr.attribStat(Attr.java:480)
    at com.sun.tools.javac.comp.Attr.attribStats(Attr.java:496)
    at com.sun.tools.javac.comp.Attr.visitBlock(Attr.java:911)
    at com.sun.tools.javac.comp.JmlAttr.visitBlock(JmlAttr.java:613)
    at com.sun.tools.javac.tree.JCTree$JCBlock.accept(JCTree.java:781)
    at com.sun.tools.javac.comp.Attr.attribTree(Attr.java:431)
    at com.sun.tools.javac.comp.Attr.attribTree(Attr.java:418)
    at com.sun.tools.javac.comp.Attr.attribStat(Attr.java:480)
    at com.sun.tools.javac.comp.Attr.visitMethodDef(Attr.java:829)
    at com.sun.tools.javac.comp.JmlAttr.visitMethodDef(JmlAttr.java:896)
    at
com.sun.tools.javac.comp.JmlAttr.visitJmlMethodDecl(JmlAttr.java:5161)
```

Figure 3.1: JMLUnitNG Exception on Windows 7

```
org.jmlspecs.jmlunitng.JMLUnitNGError: Could not construct OpenJML API
    at org.jmlspecs.jmlunitng.JMLUnitNG.processAllCompilationUnits(JMLUnitNG.java:543)
    at org.jmlspecs.jmlunitng.JMLUnitNG.run(JMLUnitNG.java:414)
    at org.jmlspecs.jmlunitng.JMLUnitNG.main(JMLUnitNG.java:177)
Caused by: java.lang.NullPointerException
    at com.sun.tools.javac.code.Type.isCompound(Type.java:333)
    at com.sun.tools.javac.code.Types.isSubtype(Types.java:384)
    at com.sun.tools.javac.code.JmlTypes.isSubtype(JmlTypes.java:194)
    at com.sun.tools.javac.code.Types.isSubtype(Types.java:372)
    at com.sun.tools.javac.code.Types.isConvertible(Types.java:285)
    at com.sun.tools.javac.code.JmlTypes.isConvertible(JmlTypes.java:158)
    at com.sun.tools.javac.code.Types.isConvertible(Types.java:294)
    at com.sun.tools.javac.comp.JmlAttr.visitJmlMethodInvocation(JmlAttr.java:2936)
```

Figure 3.2: JMLUnitNG Exception on Mac OS X

SPEST attempts to reduce the cognitive load of users by providing a clear graphical user interface that guides users through the input process with quick feedback and clear error messages. Figure 3.3 shows the front end of SPEST and explains the functionality of the different components. The tool gives users the ability to edit some inputs by hand or use the operating system's selection dialog for that input. Figure 3.4 shows how users can expand directories to select individual files to validate and generate unit tests from. The file selection dialog filters out files that are not of the appropriate file type as well as test files to simplify the users' experience.

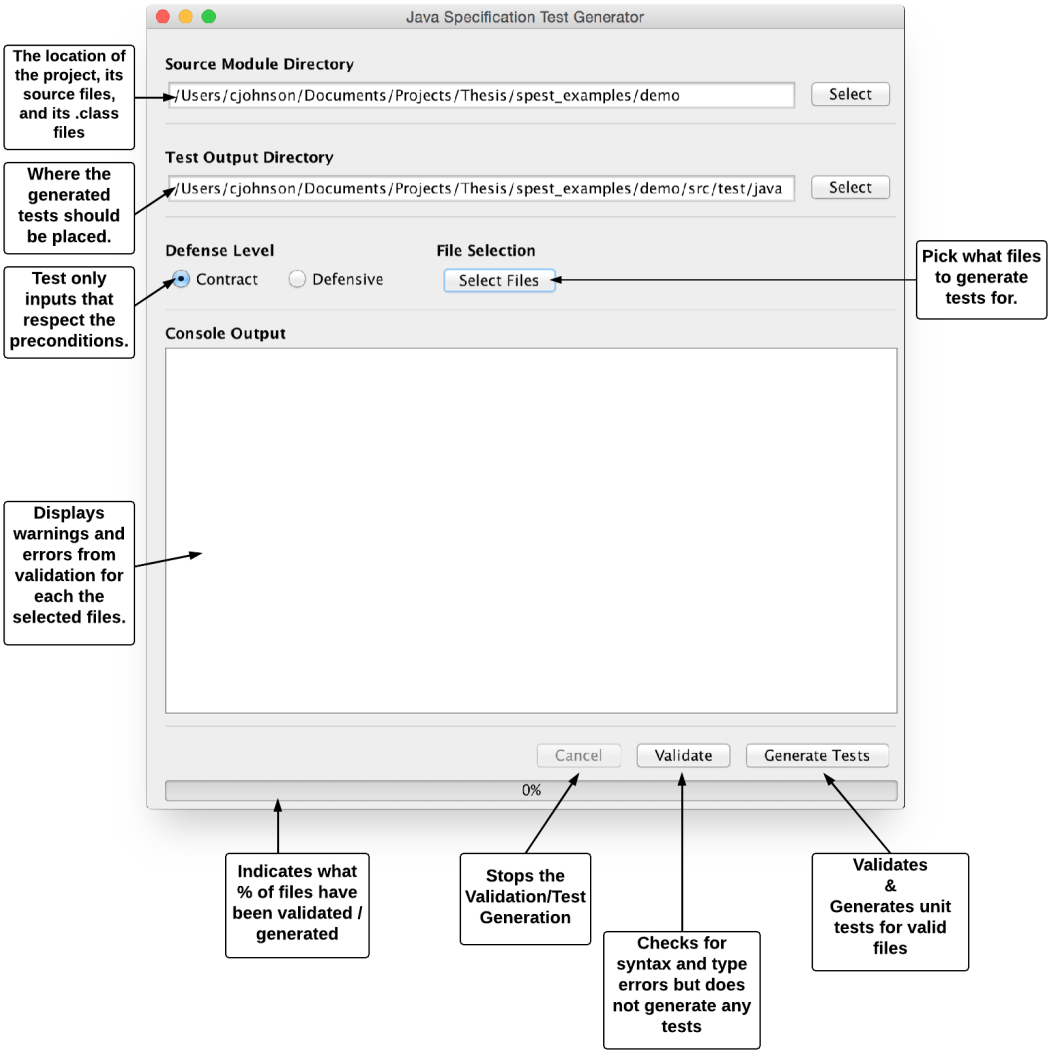


Figure 3.3: SPEST Graphical User Interface

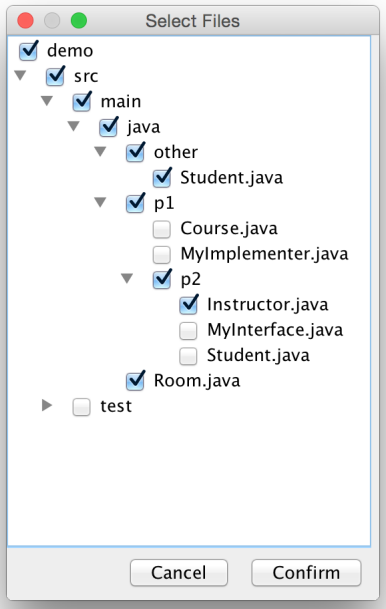


Figure 3.4: SPEST File Selection Dialog

To help users understand what inputs are invalid, SPEST also provides feedback on users' directory selections as well as their file selections. Figure 3.5 shows the feedback users recieve when they enter an invalid "source module directory" or whenever they enter an invalid "test output directory". In this situation, invalid means that the path does not refer to a directory. The tool highlights the fields that are invalid in red, provides a warning icon to show what fields need to be corrected, and also provides an error message in the console describing why the fields were invalid. Figure 3.6 shows the feedback SPEST provides when the user selects no files to validate or generate tests for. When no files are selected, the tool highlights the "File Selection" label with an asterisk and describes how to resolve the error.

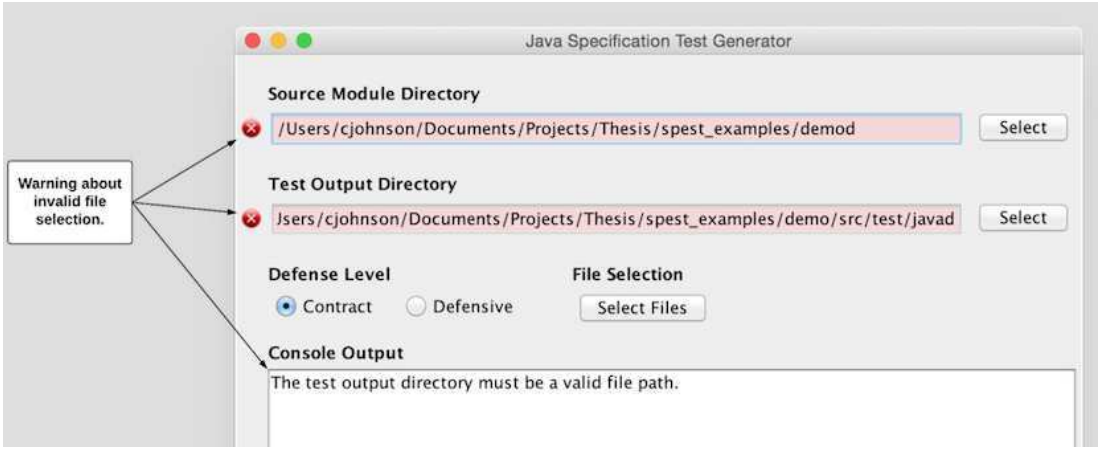


Figure 3.5: Invalid Source Module Directory / Test Output Directory

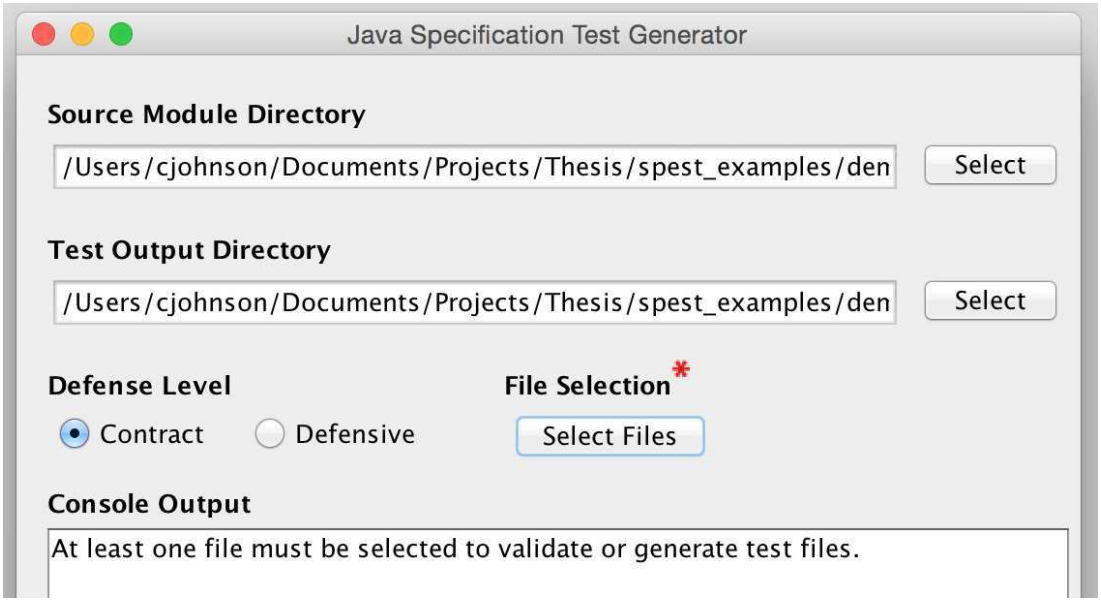


Figure 3.6: No files selected

Once the user has successfully filled out all of the fields in SPEST, they are ready to validate and generate unit tests. Both validation and generation perform syntax and type checking on a user's selected source files. Figure 3.7 shows the tool after the user has validated with a type error. The tool uses HTML formatting to distinguish warnings and errors, highlight the file the error occurred in, and lists the cause for the error. The error messages explain the reason for the failure as well as the line number and character position at which the error occurred. The user feedback is concise, and wraps internal failures in user understandable messages so that the user is able to resolve the problem.

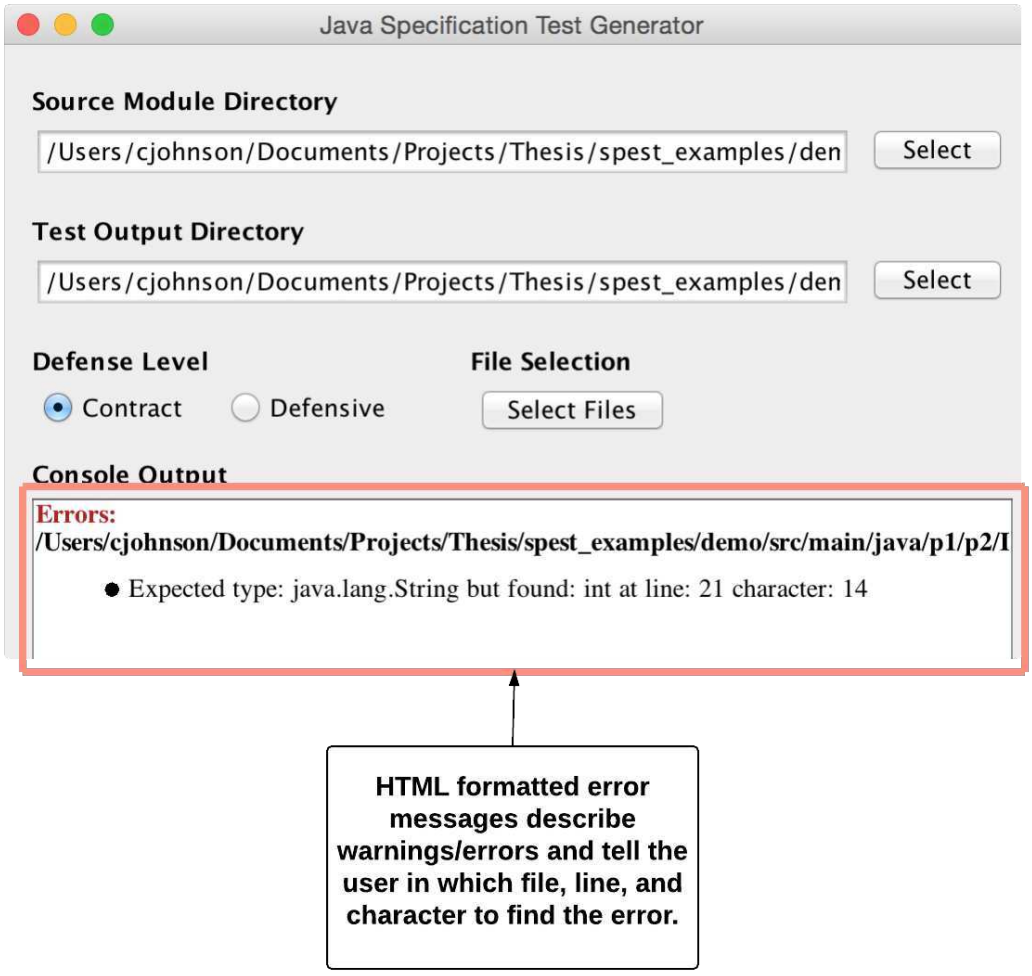


Figure 3.7: Formatted Error Messages

3.3 Multilingual

Learning a new language is a difficult task, even for experienced developers [45]. The learning curve for new languages is part of the reason that specification languages are not part of every developers day-to-day testing process [50]. SPEST attempts to alleviate some of the stress caused by this problem by providing a minimal language as well as creating a framework which can be expanded on to support more source languages with minimal changes to SPEST's specification language. Because there are simply fewer keywords to remember, and the keywords are close to keywords already found in the language, SPEST requires less time to learn and understand.

For the scope of this thesis, services that could be used to parse and generate unit tests were created for Java; however, these services could be replaced with alternate services to achieve a similar goal in additional languages. As a framework, SPEST collects a user's information, passes it through its parser, passes information from the parser along to the test writing mechanism, and requests information about how to create the various levels of test information within a unit test file. Beyond the parser, the operation of SPEST is largely language independent.

3.4 A Tool Suite for the Advanced Developer

Using SPEST to generate tests may not be the best solution in all situations; however, the components of SPEST can still be useful in most situations. There are known cases where SPEST may fail to generate data or may not generate data that targets a specific edge case a user wants to test. If a developer, who is not familiar with formal specifications, is willing to rethink how they test, SPEST can be extremely powerful. While SPEST is designed to generate unit tests, it also provides two other major capabilities: reflexive support and object generation.

A prominent difficulty when writing a new unit test is "isolating the unit under test" [13]. In Java, reflection can help resolve this by providing a means to isolate methods under test. Reflection is a tool built into Java that allows developers to access all information, including private fields and methods, within a class. By exposing private application programming interfaces (APIs), developers may be able to use reflection to break up the tests for public methods by also testing private support methods associated with them or verifying the values in fields are set appropriately. SPEST provides a facade that can simplify reflection, for the purposes of testing, by reducing the reflexive API to a subset of

invocations that are relevant to a testing environment. There are third party JUnit plugins that attempt to provide similar functionality, such as DP4J [22]; however, these plugins either require a specific compiler within an integrated development environments (IDEs) [22] or require additional annotations above each private field and method similar to JML [25].

Maintainability should not be limited to source code, "unit tests need to be maintainable" too [13]. Because manual data generation will not dynamically update with changes in assumptions, APIs, or as new implementations become available, tests can break or fail for reasons other than the source code not performing the desired functionality. One way to reduce how brittle current unit tests are is to remove the dependence on specific inputs always being used for a given test. By using the object generator in SPEST to create sample data, users are able to ensure that the test data will always remain up to date with the latest assumptions the code under test is making.

Chapter 4

Phases of Test Generation

Each of SPEST's services is part of one of three phases. The first phase is responsible for validation of a user's specification comments, the second phase is responsible for the test file generation, and the last phase is responsible for running the tests as well as data management.

Figure 4.1 shows a high level overview of the phases SPEST goes through before tests can be generated and evaluated for a Java program. The process starts with a Java object's class file. From the supplied class file, an internal representation of the fields and methods within the class object are passed along as a symbol table to the Parser and Type Checker. The Parser is responsible for syntax checking, creating an abstract syntax tree (AST), and passing the AST to the Type Checker. Using the AST and symbol table, the Type Checker verifies that the subtypes within the expressions match the expected type while also collecting additional language specific information to pass on. After the program passes the type checker, the test information collected can be used to generate a test file for the user or to execute the tests in a user's test file. The test information is needed in the test execution stage to ensure valid objects are generated for parameters. The subsequent subsections break down this process into a more detailed explanation of how each step performs the tasks described above.

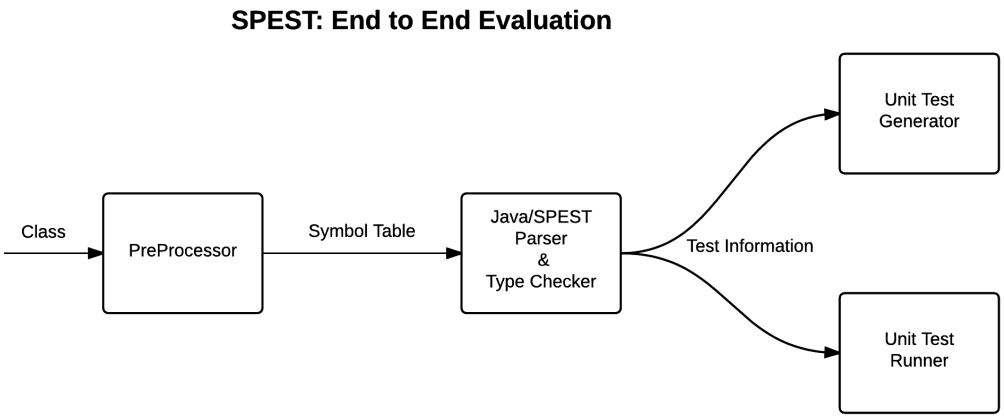


Figure 4.1: Phases of Generation

4.1 Steps Prior to Compilation

Software languages are rapidly evolving. Versions 5 - 7 of Java have had lifespans of only two to four years [38]. Because of these short lived life cycles, predecessors to SPEST that have relied on their own custom Java compilers are not compatible with the latest versions of Java. For instance, OpenJML, a toolset used to translate JML comments into a standard data format only supports versions of Java up to version 7 update 40 [11] while the latest version of Java, as of January 2015, is version 8 update 65[36]. When specification languages are one or more steps behind the latest and greatest features of the languages they work with, their functionality and acceptance will remain limited.

Relying on a custom compiler facilitates easier development of specific features; however, these features are not unique to a custom compiler. Having a custom compiler provides tools greater control over the information being used to validate and generate information for assertions. While a custom full compiler does provide access to all of the information available, most of the information is not needed. If additional information cannot be collected through the byproduct of an external standard compiler, such as javac [37] in the Java Development Kit, then constraining the parser to parse only the necessary information mitigates the consequences of changes to the source language. This targeted approach reduces the coupling between the data produced by a compiler and the service using the data while still providing access to the information required by the tools. Additionally, custom compilers facilitate the insertion of runtime assertions into the bytecode of the compiled source [10]; however,

bytecode injection is not restricted to compile time processes. For example, assertions can be injected into code during the launch of the JVM or even after the program has been launched [29].

Before the Java/SPEST Parser and Type Checker are able to begin their work, the information collected from the source language's standard compiler is translated into an internal form representing a symbol table. A symbol table is a data structure which maps variable names and method names to their types. For Java, the symbol table is created by reflexively looking at the fields and methods within the class objects in a particular project. Java's reflection allows SPEST to find not only the public fields and methods but also all private, protected, and package protected fields and methods, as well as all inherited fields and methods. By gaining access to otherwise inaccessible fields, SPEST provides a way for users to test whatever they believe needs to be tested regardless of the restrictions that were designed to prohibit tight coupling instead of facilitating stronger testing. Because field and method names are not guaranteed to be unique, method names are mangled [34] before being stored to avoid name clashes. To mangle the method names, SPEST takes the method's identifier and appends each of the parameter's types to the end of it. The mangled names are guaranteed to be unique since the source compiler will throw an error if two methods had the exact same signature.

4.2 Compilation

The syntax checking and type checking is done with a context free grammar written for ANother Tool for Language Recognition (ANTLR). ANTLR is a tool that facilitates tree construction, tree walking, error recovery, as well other services [42]. ANTLR translates the grammar into a lexer which is responsible for tokenizing the source file, a parser which is responsible for creating an AST, and AST traversal mechanisms so that the source code can be type checked and additional information can be collected.

4.2.1 Grammar

The grammar used by ANTLR is broken down into several categories: tokens, lexer rules, and tree traversal rules. The tokens and lexer rules are responsible for what will be used to do the syntax checking on a user's source code. The tree traversal rules are responsible for defining the rules used to do the type checking. The tokens define the primitive types, language specific keywords, and SPEST keywords that the user has access to. These basic building blocks are fed through the lexer rules which represent complex interactions between the building blocks and expand the language. All of the gram-

mars also have reference to a log which records warnings and errors that occur during the syntax and type checking.

The rules defined in the lexer are used to parse the source code and produce the AST. In SPEST the lexer rules attempt to minimize the coupling to the target language by skipping any information that is not critical to the success of the tool. To achieve this goal, SPEST only parses imports, SPEST comments, and method signatures. Additional information is gathered from the class files generated by a standard Java compiler run on the source code. The lexer is also responsible for ensuring precedence is respected when generating the AST. The grammar uses a non left recursive descent pattern, to evaluate high precedent expressions before evaluating anything else.

4.2.2 Evaluating Types

The Java parser implementation in SPEST uses class objects associated with the types being compared to analyze relationships. SPEST relies on the class objects because the classes provide information about the polymorphism in the code. Class look up is broken down into two steps: identifying relevant import statements and collecting the paths of all class files and jars needed to build the project. Imports are used to help identify the fully qualified class name. Classes are loaded in Java using the Java ClassLoader that is a part of the Java Runtime Environment.

In order to gain access to the classes that are not in the current classpath, SPEST uses a uniform resource locator (URL) classloader where the URLs come from the paths of the classes and the jars. Class objects can be retrieved from the classloader by providing the fully qualified name of the object to the class loader. The fully qualified name of most objects is the class name appended to the end of package path of the object. Special cases, such as arrays, require special formatting to match the name. Array's class names are prepended with a "[" for each dimension of the array followed by an "L".

Primitive classes are able to be looked up using a lookup table, but arrays require more thought. In order to retrieve the class for an array, two things need to be known: the component type and the number of subarrays. Given this information the fully qualified name can be recreated using the aforementioned format.

Because SPEST only views a part of the fully qualified object's name, it must first attempt to identify the fully qualified name before it can retrieve the class object. Identifying the fully qualified name occurs over several steps. First the explicit imports and implicit imports such as "java.lang.*"

are analyzed to see if any of them ends with the partial name that SPEST parsed. If an import with a matching ending is found, then the fully qualified name can by found by prending the import to the class name. If no imports are found, then the object is in the same package as the object that is being parsed; however, since the fully qualified name of the current object is unknown this only leads to another step in place of a solution. Because the object is in the same package, it can be inferred that the fully qualified name will contain the class name prepended with some number of the parent directories' names. Because directory structures are relatively shallow, the name can be deduced through a brute force attempt of starting with the class name and prepending each level of the parent directory and its children to the name.

Class retrieval is complicated by the subtleties of generics. Generics provide additional compile time information about the types of data within an object, interface, or method [4]. Unfortunately, most generic information is lost at runtime so additional generic information must be parsed before the type checking can complete. Type checking without this information would result in very weak types and could lead to runtime failures that should have been caught at compile time. For example, if the expression list.get(0) $>$ 4 was type checked without generics, the comparison would result in an Object being compared to an Integer; however, there would be no way to determine if the object was actually an Integer and this would pass the type checker even though it has the potential to fail. To overcome this, the parser collects the missing generic information before it begins type checking and passes this information along with the type of the object containing the generic information. This generic information is only collected for parameters in methods because member variables maintain enough information to determine their generic type.

4.2.3 Type Checking

The expressions composing a specification are examined to evaluate if they are composed of the valid types. The expected types for an expression are predetermined based on the operator or looked up using the symbol table. The types determined using the above methods are compared against the expression type using several different approaches. If the expected type requires a primitive, then null checks must be performed as well as straight class comparisons between the expected type and actual type. Additionally, when comparing primitives in Java, it is important to account for autoboxing, where primitives can be promoted or demoted to or from objects. Characters are also a special case with primitives, since they can be used wherever an int can be used. When an object is expected, null can be used as

a wild card. Additionally, object comparisons must consider polymorphism to account for inheritance and implementations of interfaces.

4.3 Test Generation

Generating unit tests from specifications occurs though several steps: interpreting the meaning of the specification, writing the test information to a file, and incorporating the state of the art techniques into the tests to ensure that the tests run as quickly as possible while finding as many bugs as possible.

4.3.1 File Generation

The first step in generating a test file is to verify if a file with the same name already exists. If there is a name clash, then the generated tests will be added to the bottom of the file in a section marked by "/*Start generated tests*/" and "/*End generated tests*/". Any changes the user makes in the section that is marked by the previously mentioned comments will be overwritten during the next file generation; however, anything that is not between those comments will not be changed or deleted. If a file does not already exist then several additional steps are taken: file generation, adding header information, and adding the test methods. The service handling the file I/O delegates to a second layer of services to retrieve header information and test method information. Abstracting out these steps into multiple layers allows for the services to be easily replaced to adapt the tool to meet the different needs of different source languages and xUnit Frameworks. The Future Work chapter further discusses the idea of adapting SPEST's functionality to apply to other languages.

When generating the file, the main challenge is determining what package the test file should be placed in. While the files could all be placed in the root of the test directory, SPEST follows the Apache Software Foundation's standard directory structure [16] and places the test file in the same package structure as the source file with only the root packages differing.

For the scope of this thesis, a Java implementation was created to generate header information and test method information. The header generator is responsible for determining all necessary imports and declaring the class. The test method generator assumes the remaining responsibilities. During test method generation, variables need to be declared, the method under test needs to be called, and assertions need to be added. Declarations are needed for private variable access, private method value access, parameters, and references to variables values from before the method is called. To ensure there

are no compile time errors, these declarations must use generic information whenever available.

When assigning values to declarations, calls to a facade are used. The facade improves readability in two ways: hiding implementation and reducing the need for excessive casting. The facade is responsible for making calls into the SPEST internal services that are responsible for generating primitives and objects. To reduce the amount of casting necessary, the facade uses generics at the method level so that the caller does not need to cast the object that is returned. Once the necessary declarations are in place, the next step is to decide how to insert the assertions.

4.3.2 Interpreting Specifications

Specifications can be broken down into preconditions and postconditions. These two groups are composed of combinations of boolean expressions, logical branches, and quantifiers. Logical branches and quantifiers act as boolean expressions, but require additional information for generation. Quantifiers are boolean expressions that describe conditions over all elements of a given type while logical branches describe conditions that must be true under given circumstances. A quantifier can be either a *forall* statement or an *exists* statement. *Forall* implies that the assertions are true for every object of a given type. *Exists* implies that the assertions are true for at least one object of a given type. Figure 4.2 illustrates a high level break down of the specifications in SPEST.

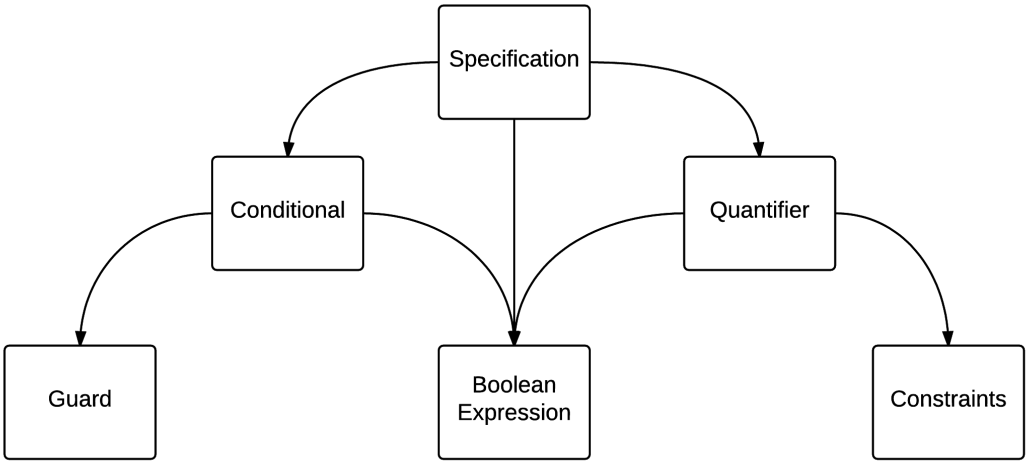


Figure 4.2: Structure of specifications

4.3.3 Writing Assertions from Specifications

Assertions should be as small as possible to allow developers to pin point the origin of failures in their code. To reduce the size of assertions, boolean expressions that are joined by *&&* are divided into individual assertions. Conditional expressions are inserted as if-then-else statements; if the guard is met one set of assertions will be run, otherwise, another set of assertions will be run. For specifications involving the bidirectional implication operator, *iff*, the assertions are broken out into two if statements that follow the previous rule. The two if statements represent a true implies true relationship and a false implies false relationship.

Conditions for quantifiers depend on if the quantifier is a *forall* statement or *exists* statement. While logically the two quantifiers can be transformed into each other, translating the quantifiers into one format reduces the readability of the tests that are generated. When generating assertions for a *forall* statement, SPEST adds a for-each loop that iterates over all of the objects in the quantified value set and adds the assertions to the loop. For an *exists* statement, SPEST adds a boolean before adding a for-each loop that iterates over all of the objects in the quantified value set, sets the value of the boolean to be the current boolean value or the result of running the assertions, and after the end of the loop asserts that the boolean is true.

Before the actual assertions can be written to the file, they must first be transformed. In order to transform the expressions a Java Parser, the Java Expression Language (JEXL) [12], is used to tokenize

the string, and then a visitor pattern is applied to each of the tokens. The transformation process involves two steps: adding field access where necessary, and substituting the appropriate variable names for the identifiers in the assertion. Field access needs to be added to the postconditions because the postconditions are written from the perspective of the object under test while they are evaluated from the perspective of the test object. What this means is that to reference the value of a variable in the object, the specification may only describe the field; however, in the test, the specification must describe the field with relation to the test object. This step has two cases: the field is accessible through direct access or the field has restricted access. If a field is public, then the field reference will have the identifier of the test object prepended to it. If a field has restricted access, a call to a facade for reflexive access is used to wrap the field's identifier.

Variable names need to be substituted to ensure that variable declarations are unique for references to the parameters' identifiers. To guarantee unique names are used, SPEST declares the names of parameters locally with unique identifiers. Subsequently, SPEST must map the names of the parameters to these new names. This mapping must consider different scopes, described through the prefix *this*, as well as method identifiers that can contain the same identifier. The tokenizer and visitor pattern applied in the parser distinguish between shared field and method identifiers. If SPEST detects the *this* keyword, then it will not apply the mapping to the identifier name because the name refers to a local variable and not a parameter.

4.3.3.1 Verifying Specifications & Generated Test Cases

Because specifications serve as documentation, drive the parameter selection in tests, and specify what assertions should be made in tests, verifying their correctness is essential to verifying the integrity of the generated tests. Although verifying a tool that verifies source code is unintuitive, developers already perform this task to manually verify that their hand written test cases meet their expectations.

Specifications can be verified through several different approaches. First, the specifications can be compared against the source code. If there is an obvious discrepancy between what the specifications describe and the behavior of the source code then the developer knows there is a problem in one of the two elements. When the source code is not available or too complex to compare against the specification, then evaluating the test file that is generated from the specification can provide developers with additional information. Before the tests are run the developer can see their postconditions broken up into

the smallest assertions possible. If the developer needs additional information then they can execute the unit tests. Running the unit tests provides two pieces of additional information: restrictions on the input parameters and verification of the described behavior of the code. While the test is running, developers are able to see what inputs SPEST has generated. By seeing concrete values for input parameters, users can quickly verify if their preconditions missed an edge case or did not include appropriate restrictions. If running the tests results in an unexpected failure then developers can quickly identify a mismatch between their specifications and their source code.

Verifying the specifications can be as difficult a process as verifying tests manually. If a developer's specifications are too weak, then the generated tests may not detect an error within the source code. This is not a problem of SPEST but rather a fundamental challenge of verifying the behavior of code in general. Developers looking to further their confidence in their tests should use additional external metrics, such as code coverage or mutation scores, to evaluate how effective their tests are and identify areas that may need to be strengthened.

4.3.4 Reducing Time & Complexity

SPEST generated tests should feel like a natural part of a developer's test suite. To achieve this goal, SPEST allows its tests to be inserted into a developer's existing tests or new files to be generated as previously discussed. Furthermore, the number of tests generated is bounded by the number of methods in the source file. Test methods are named after the method they are testing with a number appended to them to reduce the likelihood that they will conflict with a user's own tests. All of the code in the tests is formatted to use proper indentation and any extensive or complicated lines of code are hidden behind facades that do casting or make more calls behind the scenes. By making the tests as readable as possible, the user should be able to understand what caused the tests to fail, whether it was a poor specification or an error in their code.

Predecessors to SPEST, JMLUnit and JMLUnitNG, have both failed to solve the problem of creating unit tests that are capable of running in a "reasonable amount of time". JMLUnit struggled with this problem because it exhaustively tested all data [52]. JMLUnitNG improved the inputs that were used for testing but suffered even longer test completion times due to the overhead of generating data. SPEST has takes two steps to limit the amount of time tests take to run: restricting the objects generated to objects that will have the most meaningful impact [51] and the use of pairwise parameter selection

Parameter Name	Value 1	Value 2	Value 3	Value 4
Color	Red	Blue	*	*
Number	1	2	3	*
Description	Broken	In Progress	Stable	Perfect

Table 4.1: Example parameters and parameter ranges

[14].

When selecting objects for generation, SPEST selects data that represents edge cases, near edge cases, and average values. While this approach can and will miss key inputs, it reveals a large portion of user errors while significantly reducing the number of tests generated. Pairwise parameter selection is used to avoid the combinatorial nightmare of trying all combinations of parameters. Pairwise parameter selection covers all combinations of two parameters rather than all combinations of all parameters. While pairwise covers only a fraction of all of the combinations, it has been found to have comparable block coverage to exhaustively testing [14]. In a trial of an internal e-mail system, pairwise combinations achieved 97% coverage of branches with less than 100 test cases, where exhaustive testing would have required 27 trillion test cases. An example of pairwise generation is shown with the parameter identifiers and their values in table 4.1, with the pairwise combinations in table 4.2, and the full set of combinations in table 4.3. In this example the pairwise combinations results in only twelve combinations to test while the exhaustive combination results in twenty four.

Color	Number	Description
Red	1	Broken
Red	2	Stable
Red	3	Broken
Blue	1	Stable
Blue	2	Perfect
Blue	3	Perfect
Red	2	In Progress
Red	1	Perfect
Blue	1	In Progress
Blue	2	Broken
Red	3	In Progress
Red	3	Stable

Table 4.2: Pairwise combinations

Color	Number	Description
Red	1	Broken
Red	1	In Progress
Red	1	Stable
Red	1	Perfect
Red	2	Broken
Red	2	In Progress
Red	2	Stable
Red	2	Perfect
Red	3	Broken
Red	3	In Progress
Red	3	Stable
Red	3	Perfect
Blue	1	Broken
Blue	1	In Progress
Blue	1	Stable
Blue	1	Perfect
Blue	2	Broken
Blue	2	In Progress
Blue	2	Stable
Blue	2	Perfect
Blue	3	Broken
Blue	3	In Progress
Blue	3	Stable
Blue	3	Perfect

Table 4.3: Exhaustive combinations

4.3.5 Evaluating Preconditions

Before primitives and objects can be generated, a user's preconditions need to be simplified to one of two forms: identifier operator value, or methodInvocation(...). In order to simplify the expressions, the Java Parser JEXL is used. JEXL is capable of evaluating basic arithmetic, indexing into arrays, evaluating field access with context, and evaluating method invocations with context. The context used in the field access and method invocations is a SPEST reflexive support tool that takes an instance of an object, finds the corresponding field or method, and then uses that information to determine the value. SPEST acquires the method objects through iterating over the methods within the object, both private and public, and loosely comparing the parameter types parsed through JEXL to those in the method signature. In this context, loose comparison means that the types provided by the parser could be one of several types. For instance an int can be used in place of a double. As a result of the loose type checks, the exact method that is invoked is not guaranteed; however, the tests remain deterministic because the order in which the methods are searched will always remain the same across multiple invocations.

4.3.6 Data Generation

Data generation is broken into two major categories: primitives and objects. Primitives form the building blocks for objects and are thus the source of variability for the data. Within each category, the data is distinguished between arrays and non arrays. By default five test values are generated for each input request; however, the API supports unbounded return sizes. If the constraints specified by the user require specific values, only these values will be returned; otherwise, the values will represent the two extremes of a value's domain, the average of the maximum and the minimum, as well as two random values inside the boundaries. Random values are created using a seeded random number generator, so the values will be consistent across multiple runs of the tests. The boundaries are added to test edge cases of the code while the average and random values test the average cases of the code. If any of the values generated are banned by a user's specifications, then the value will be removed and the next logical number will be included.

For n-dimensional objects, such as arrays and collections, six objects of sizes: empty, 1, 8, 64, 512, 4096 are used to try cover cases of small to large data sets. Because these sizes represent the total number of elements, the number of elements within each dimension of the n-dimensional object is determined by taking the nth root of the total size at given level. For example, if the total size was 4096

and the objects were being generated for a two-dimension array, then the dimensions of the array would be 64 x 64. These sizes were chosen because they allow for maximum data variance while maintaining the performance that is generally expected from unit tests.

4.3.6.1 Primitive

Primitives are broken down into two categories for generation: numbers and booleans. Because characters can be interpreted as integers, they are grouped with the numbers category. This distinction is made because of how the preconditions for this data are interpreted. For numeric primitives, the preconditions are parsed to find the bounds, the required values, and the banned values. Because the preconditions have been simplified before this point to the form identifier operator value or method invocation, the preconditions can be parsed by splitting the string around the operator, if one is present, and using switch statements based on the operator or storing the method invocation. For example, the string "x $<$ 5" would yield the constraint that the maximum inclusive value of x is four. If two constraints conflict with each other, then a warning is logged and the last condition will be used. The current implementation of SPEST does not interpret the preconditions for booleans; however, this lack of functionality is not likely to hinder the developer since there will most likely not be any restrictions on the two options for booleans.

4.3.6.2 Objects

The object generation algorithm is built to take any kind of class specification and return an object of that type. The class specification can be an interface, abstract class, or a class that can be instantiated. The objects generated must fulfill two requirements: compliance and variety. Compliant objects must abide by the preconditions in the SPEST specifications and abide by whatever minimum form their default constructors require. Variety is needed to ensure that a reasonable test space is covered. SPEST does not claim to cover even a significant portion of all possible objects but instead defines a set of representative values and edge-case values for primitives and recursively builds objects from those.

The first step when instantiating the class, is finding a class that is assignable from the target class and can be instantiated. If target class can be instantiated, then the class will be used. If the class is an interface or an abstract class, then an assignable class that can be instantiated is found from a developer's classpath. ArrayLists are prioritized for Collections and HashMaps are prioritized for Maps

because there are often implementations of these interfaces that have limited or undesired functionality in the classpath.

When instantiating the object, SPEST uses different approaches depending on the type of object. Each of the different instantiation approaches use a seed to ensure a unique value will be generated. If the seed used for generation is greater than the maximum number of objects of that type, then the seed mod the maximum number of elements will be used. For enumerations, SPEST does not instantiate the object but instead pulls values from the enumeration using its implicit *T[] values()* method. Collections and Maps are constructed using the default constructor if available or through one of the other constructors otherwise. Native Java objects are constructed by indexing the constructors in the object and using the seed to select the constructor. User objects are constructed using the constructor with the fewest parameters and modified after construction. Strings are a special case and are pulled from a user accessible and modifiable dictionary. The default dictionary for Strings is a subset of the most commonly used passwords from 2015.

Once a default map or collection is generated it has to be populated. Collections and maps are treated as n-dimension data structures and so their sizes are predetermined as previously mentioned. The objects that are used to populate the collections and maps are recursively generated.

For a user defined object, the fields of the object are inspected and the maximum number of objects or primitives for each field type are recursively determined. The maximum number of unique objects or primitives able to be generated for each field is the amount each field can contribute to the variety of the object being generated. A list of these amounts is constructed for the new object's fields, thereby characterizing the new object's potential variety. The boundaries that are calculated are used when determining the current value of the seed. Once the seed is determined, a value for each field is recursively determined if the field is an object, or a value is assigned from the primitive generator described previously.

At its current stage of development, SPEST only partially fulfills the compliance requirement. Object generation does not yet respect preconditions describing nested objects; however, preconditions are respected for primitives and non nested fields. Variety is supported for primitives, objects, arrays, and Java's data structures.

While generating objects, if a recursive field or constructor is encountered, the object generator will use the variable seed used to vary objects in SPEST to determine the depth of the deepest recursive

field to prevent infinite loops. Furthermore, the object generation algorithm may not generate objects that provide full branch coverage. In both of these cases, as well as when the user has specific inputs that they believe should be tested, SPEST provides a work around to inject data into the methods being run in a particular test class. This approach can potentially affect all test methods in the class that are using objects of the injected type though. If a user's needs are still not met by the workaround then the developer can add a supplementary unit test that provides the desired functionality.

4.3.7 Quantifier Evaluation

Universal quantifiers are extremely powerful tools because they allow developers to describe relations that encapsulate all objects of a given type. These tools enable developers to create stronger executable specifications that can serve as test oracles. Because the data being described can be unbounded, universal quantifiers also allow developers to describe the behavior of their code very precisely without worrying about the limitations the hardware would impose on the source code.

Because the number of objects that can be generated for a given class is only limited by the hardware, an artificial limitation must be imposed on the objects that are used in the quantified set of values. To limit the number of objects generated for the quantifiers input sets while also providing the largest set of objects to use for a universal quantifier, SPEST caches all of the objects that are dynamically generated in the generated unit tests and uses these as its quantified value set. SPEST is thereby able to maintain a quantified value set without making the unit tests run indefinitely or until they hit the memory limitations of the hardware. If a minimum number of objects, by default 1000, has not been generated, SPEST will attempt to generate at least the minimum number of objects before continuing. This number was chosen to prevent tests that fail to cover an appropriate range of values while also preventing too many objects from being generated for complex data structures such as maps and lists that can contain thousands of additional references to other complex objects that also need to be generated. To ensure that all objects have been built before the quantified value sets are able to run their assertions, all unit tests are run twice using a custom test runner. On the first pass the test runner ignores all failures while all of the objects are being cached. On the second pass, the runner runs the unit tests through the standard JUnit version 4 test runner.

Chapter 5

Demonstration of SPEST's Capabilities

SPEST offers a diverse set of services to help developers improve the quality of their code including generating test files and generating parameter data. Because of the different natures of the keywords in SPEST's syntax, the unit tests that are generated can be composed of different logical structures as well as helper variable declarations. This chapter describes how SPEST interprets users' specifications in order to create flexible unit tests and the techniques used to help reduce the runtime of large parameter sets.

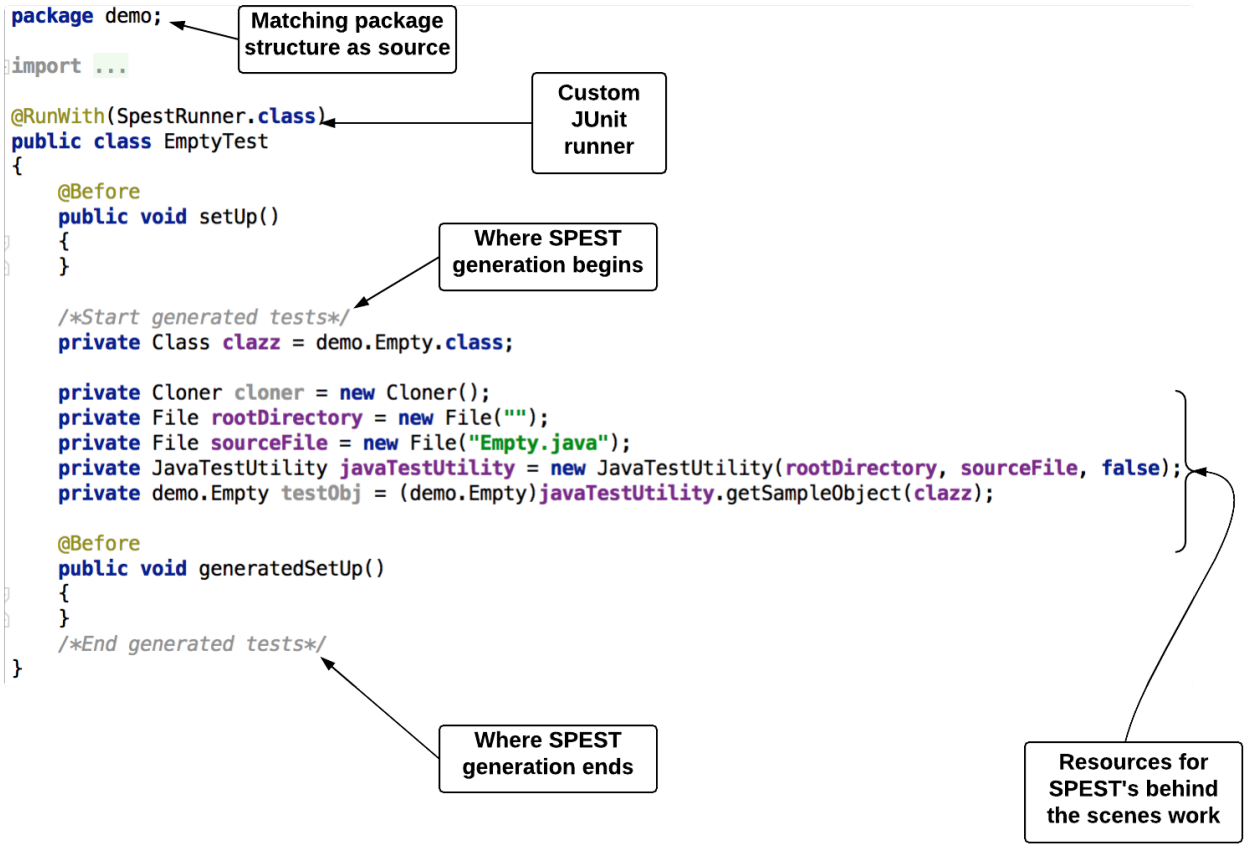
5.1 Creating the Test File

If a test file with the name of the source file appended with the word "Test" does not already exist under the user specified test directory a new file will be created with the appropriate package structure, imports, class declaration, a JUnit 4 setup method for the user to interact with, and a setup method that only SPEST should interact with. This file also contains information that is used to identify preconditions, and test the source file. Figure 5.1a shows an empty class object that was used to generate tests with SPEST, and figure 5.1b shows the results of generating a test file from the source.

If a test file with the name of the source file appended with "Test" already exists under the user specified test directory, the file will be modified to include all of the original content as well as test information and methods generated by SPEST. Figure 5.2a shows a class with two sample methods, that have SPEST specifications associated with them. A test file with hand written test methods was created and is shown in figure 5.2b and the SPEST modified version of the same file can be seen in figure 5.3.

```java
package demo;

public class Empty
{
    //This class has no methods
}
```

(a) Java file with no methods



(b) SPEST generated test with no test methods

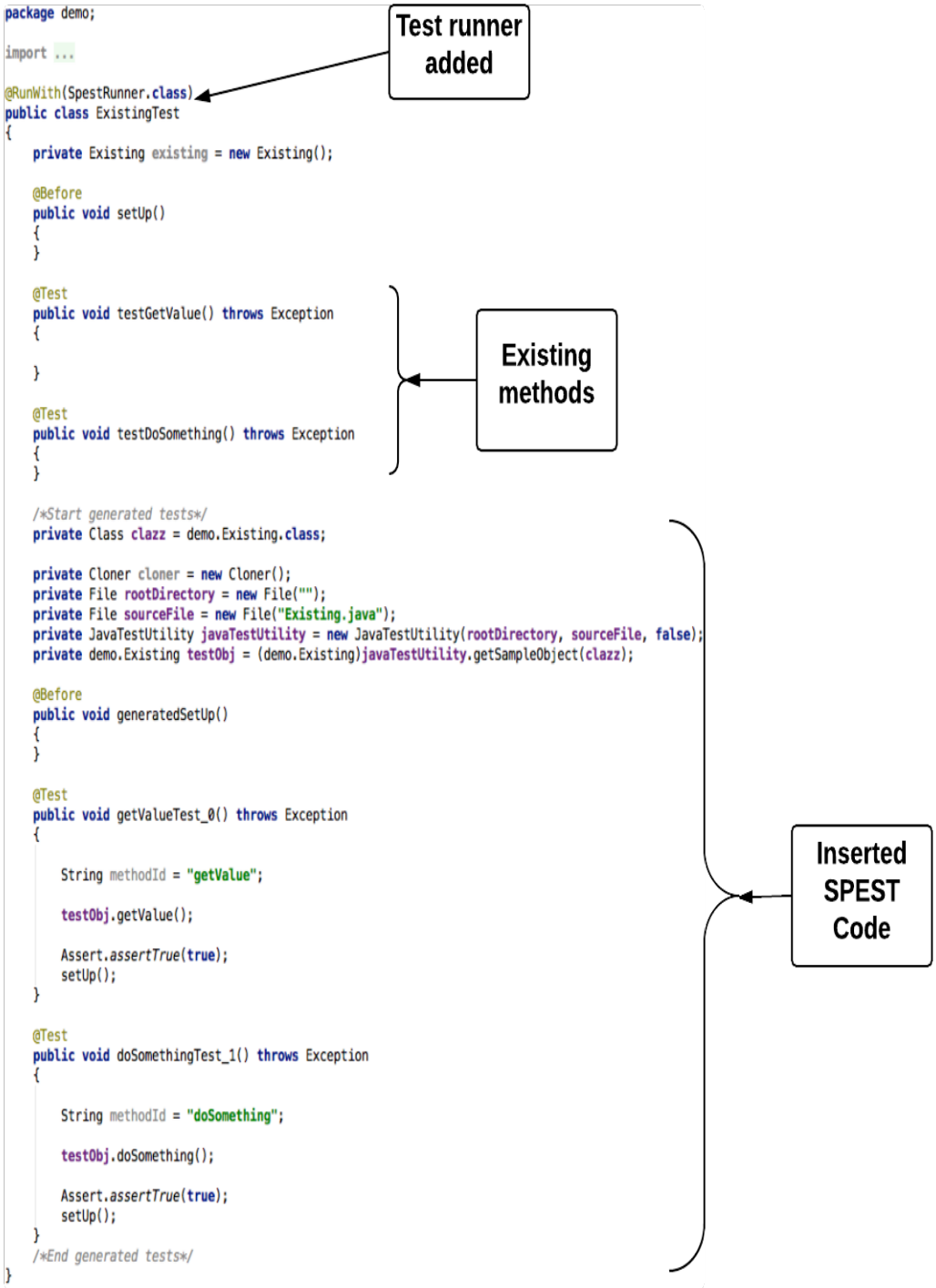
Figure 5.1: Creating a new test file

```java
package demo;

public class Existing
{
    /*
        POST: true
     */
    public int getValue()
    {
        return 0;
    }
    /*
        POST: true
    */
    public boolean doSomething()
    {
        return true;
    }
}
```

(a) Java file with SPEST comments

```java
package demo;

import ...

public class ExistingTest
{
    private Existing existing = new Existing();

    @Test
    public void testGetValue() throws Exception
    {

    }

    @Test
    public void testDoSomething() throws Exception
    {
    }
}
```

(b) Existing test file




```
package demo;

import ...

@RunWith(SpestRunner.class)
public class ExistingTest
{
    private Existing existing = new Existing();

    @Before
    public void setUp()
    {
    }

    @Test
    public void testGetValue() throws Exception
    {

    }

    @Test
    public void testDoSomething() throws Exception
    {
    }

    /*Start generated tests*/
    private Class clazz = demo.Existing.class;

    private Cloner cloner = new Cloner();
    private File rootDirectory = new File("");
    private File sourceFile = new File("Existing.java");
    private JavaTestUtility javaTestUtility = new JavaTestUtility(rootDirectory, sourceFile, false);
    private demo.Existing testObj = (demo.Existing)javaTestUtility.getSampleObject(clazz);

    @Before
    public void generatedSetUp()
    {
    }

    @Test
    public void getValueTest_0() throws Exception
    {

        String methodId = "getValue";

        testObj.getValue();

        Assert.assertTrue(true);
        setUp();
    }

    @Test
    public void doSomethingTest_1() throws Exception
    {

        String methodId = "doSomething";

        testObj.doSomething();

        Assert.assertTrue(true);
        setUp();
    }
    /*End generated tests*/
}
```

Figure 5.3: SPEST generated test interacting with existing test file

The figures shown below illustrate how different specifications are translated into unit tests by SPEST. Figure 5.4 shows the class and its associated fields that will be used to show the results of translating specifications into tests. The figures below containing test methods generated in SPEST do not show the class package, imports, and declaration for brevity; however, this information is present in the test files that are generated.

Figure 5.5a shows a Java method with an associated SPEST specification. The specification shown in the figure has two parts, but this example focuses on the precondition that is denoted by the *PRE:* keyword. This precondition specifies that the input to the method under test must be in the exclusive range of 45 to 100. Figure 5.5b shows the SPEST generated test method from the *precondition* method in figure 5.5a. Inside the generated code, the values for the parameter are generated at runtime which allows the values to be dynamically updated without regenerating the tests. Table 5.1 shows the SPEST generated values used to test the method. All of the values shown in table 5.1 are within the precondition defined boundaries.

Figure 5.6a shows a Java method with a simple postcondition. The specification asserts that the value in *publicField1* will be less than the value in *publicField2* after the method under test has been run. Figure 5.6b shows the SPEST generated test method from the *basicPostConditions* method in figure 5.6a. The postcondition is tested in a JUnit 4 assertion statement. If the assertions successfully complete, then the setup method is called to ensure that the data is properly reset before the next parameter is selected.

```
public class Translations
{
    public int publicField1;
    public int publicField2;
    public Translations recursiveField;

    private int privateField1;
    private List<Integer> list;

    public Translations(int publicField1, int publicField2, Translations recursiveField, int privateField1,
                        List<Integer> list)
    {
        this.publicField1 = publicField1;
        this.publicField2 = publicField2;
        this.recursiveField = recursiveField;
        this.privateField1 = privateField1;
        this.list = list;
    }
```

Figure 5.4: Sample object for specification translations

```
/*
        PRE:
            parameter > 45 && parameter < 100
        POST:
            true
 */
public void precondition(int parameter)
{
    //...
}
```

(a) Example method with a precondition restricting a parameter

```
@Test
public void preconditionTest_0() throws Exception
{
    int testComboIndex;

    String methodId = "precondition_int";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "parameter",
            java.lang.Integer.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size());

    int param_0;
    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);

        testObj.precondition(param_0);

        Assert.assertTrue(true);
        setUp();
    }
}
```

(b) Example test method with preconditions respected

Figure 5.5: Example method with preconditions and SPEST generated test

Parameter Identifier:	Type	Value 1	Value 2	Value 3	Value 4	Value 5
parameter	int	66	98	99	73	46

Table 5.1: SPEST generated primitives respecting preconditions

```
/*
    POST:
        publicField1' < publicField2'
 */
public void basicPostConditions()
{
    //...
}
```

(a) Example method with a postcondition

```
@Test
public void basicPostConditionsTest_1() throws Exception
{

    String methodId = "basicPostConditions";

    testObj.basicPostConditions();

    Assert.assertTrue(testObj.publicField1 < testObj.publicField2);
    setUp();
}
```

(b) Example test method with postcondition tested

Figure 5.6: Example method with a postcondition and SPEST generated test

```
/*
    POST:
        publicField1' < publicField2'
        && (publicField1' > 40 || publicField2' < 20)
 */
public void complexConditions()
{
    //...
}
```

(a) Example method with multiple postconditions

```
@Test
public void complexConditionsTest_2() throws Exception
{

    String methodId = "complexConditions";

    testObj.complexConditions();

    Assert.assertTrue(testObj.publicField1 < testObj.publicField2);
    Assert.assertTrue(testObj.publicField1 > 40 || testObj.publicField2 < 20);
    setUp();
}
```

(b) Example test method with postconditions tested

Figure 5.7: Example method with postconditions and SPEST generated test

Figure 5.7a shows a Java method with multiple boolean expressions contained within a postcondition. Figure 5.7b shows the SPEST generated test method from the *complexConditions* method in figure 5.7a. The postcondition is broken up into two assertions to aid the developer with identification of the source of the problem. In general postconditions containing "&&"ed boolean expressions will be divided into individual expressions for JUnit 4 assertions.

Figure 5.8a shows a Java method with a reference to the value of a member variable before the test method is run and the value of a member variable after the test method is run within a postcondition. Figure 5.8b shows the SPEST generated test method from the *increment* method in figure 5.8a. In this example, *publicField1´* refers to the value of *publicField1* after the test method is run while the identifier *publicField1*, without the prime, refers to the value before the method is run. In the generated code, the value of the field is stored before the method under test is run and the stored value is compared to the value of the field after the method is run.

```
/*
    POST:
        //publicField1' refers to the value after the method is run
        //publicField1 refers to the value before the method is run
        publicField1' == (publicField1 + 1)
 */
public void increment()
{
    //...
}
```

(a) Example method with old and prime notation

```
@Test
public void incrementTest_3() throws Exception
{
    int publicField1 = (int)(java.lang.Integer)getFieldValue(testObj, "publicField1");


    String methodId = "increment";

    testObj.increment();

    Assert.assertTrue(testObj.publicField1 == publicField1 + 1);
    setUp();
}
```

(b) Example test method using old vs. prime notation

Figure 5.8: Example method with old and prime notations and SPEST generated test

```
/*
    POST:
        privateField1' > privateField1
 */
public void accessRestrictions()
{
    //...
}
```

(a) Example method with private field references

```
@Test
public void accessRestrictionsTest_4() throws Exception
{
    int privateField1 = (int)(java.lang.Integer)getFieldValue(testObj, "privateField1");


    String methodId = "accessRestrictions";

    testObj.accessRestrictions();

    Assert.assertTrue(javaTestUtility.getFieldValue(testObj, "privateField1") > privateField1);
    setUp();
}
```

(b) Example test method with private fields referenced

Figure 5.9: Example method with private fields referenced and SPEST generated test

Figure 5.9a shows a Java method with a postcondition describing a field that can not be directly accessed from the object because of Java's access control and the private modifier on the field. Figure 5.9b shows the SPEST generated test method from the *accessRestrictions* method in figure 5.9a. In this example, *privateField1* refers to a private field. In the generated code, Java's reflection is used to bypass the access restrictions to allow the value of the private field to be compared.

Figure 5.10a shows a Java method with the return keyword contained within a postcondition. Figure 5.10b shows the SPEST generated test method from the *returnKeyword* method in figure 5.10a. The SPEST generated test declares a variable of the same type as the return and stores the return value when the method under test is called. The return keyword in the assertion is replaced with the variable name to avoid a keyword conflict with the source language.

Figure 5.11a shows a Java method using the conditional syntax within a postcondition. Figure 5.11b shows the SPEST generated test method from the *conditionalStatements* method in figure 5.11a. The SPEST generated test maintains a user's conditional logic flow when adding the assertions to the unit tests.

```
/*
    POST:
        return == publicField1;
 */
public int returnKeyword()
{
    return publicField1;
}
```

(a) Example method using “return” keyword in a postcondition

```
@Test
public void returnKeywordTest_5() throws Exception
{
    int publicField1 = (int)(java.lang.Integer)getFieldValue(testObj, "publicField1");


    Integer ret;
    String methodId = "returnKeyword";

    ret = testObj.returnKeyword();

    Assert.assertTrue(ret == publicField1);
    setUp();
}
```

(b) Example test method referencing the return value of the method

Figure 5.10: Example method with the return value of the method referenced and SPEST generated test

```
/*
    POST:
        if(a < b)
        (
            a < 0
        )
        else
        (
            a > 0
        )

 */
public void conditionalStatements(int a, int b)
{
    //...
}
```

(a) Example method using if-then-else conditional in a postcondition

```
@Test
public void conditionalStatementsTest_8() throws Exception
{
    int testComboIndex;

    String methodId = "conditionalStatements_int_int";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "a",
            java.lang.Integer.class);
    List<java.lang.Integer> testPoints_1 = javaTestUtility.getSamplePrimitives(testObj, methodId, "b",
            java.lang.Integer.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size(), testPoints_1.size());

    Class[] parameterClasses = {int.class,int.class};
    int param_0;
    int param_1;

    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        param_1 = testPoints_1.get(combinations[testComboIndex][1]);

        testObj.conditionalStatements(param_0, param_1);

        if(param_0 < param_1)
        {
            Assert.assertTrue(param_0 < 0);
        }
        else
        {
            Assert.assertTrue(param_0 > 0);
        }

        setUp();
    }
}
```

(b) Example test method with conditional assertions

Figure 5.11: Example method with conditional assertions and SPEST generated test

Figure 5.12a shows a Java method using the bidirectional conditional syntax within a postcondition. Figure 5.12b shows the SPEST generated test method from the *bidirectionalConditional* method in figure 5.12a. The SPEST generated test translates the bidirectional conditionals into two smaller conditionals which test both the assertion and its negation.

```java
/*
    POST:
        (a == b iff b == a)
 */
public void bidirectionalConditional(int a, int b)
{
    //...
}
```

(a) Example method using if and only if conditional in a postcondition

```java
@Test
public void bidirectionalConditionalTest_9() throws Exception
{
    int testComboIndex;

    String methodId = "bidirectionalConditional_int_int";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "a",
            java.lang.Integer.class);
    List<java.lang.Integer> testPoints_1 = javaTestUtility.getSamplePrimitives(testObj, methodId, "b",
            java.lang.Integer.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size(), testPoints_1.size());

    Class[] parameterClasses = {int.class,int.class};
    int param_0;
    int param_1;
    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        param_1 = testPoints_1.get(combinations[testComboIndex][1]);

        testObj.bidirectionalConditional(param_0, param_1);

        if(param_0 == param_1)
        {
            Assert.assertTrue(param_1 == param_0);
        }
        if(!(param_1 == param_0))
        {
            Assert.assertTrue(!(param_0 == param_1));
        }

        setUp();
    }
}
```

(b) Example test method with split-up bidirectional conditional assertions

Figure 5.12: Example method with bidirectional conditional and SPEST generated test

Figure 5.13a shows a Java method using the universal quantifier *forall* syntax within a postcondition. Figure 5.13b shows the SPEST generated test method from the *forallSorting* method in figure 5.13a. The SPEST generated test verifies the users assertions for every value in the range of values.

Figure 5.14a shows a Java method using the universal quantifier *exists* syntax within a postcondition. Figure 5.14b shows the SPEST generated test method from the *existsSorting* method in figure 5.14a. The SPEST generated test verifies the users assertions for every value in the range of values.

```
/*
      //Verify that the elements in the field "list" are sorted by natural ordering
      POST:
          forall(int index; (index > -1) && (index < list.size() - 1);
              list.get(index) < list.get(index + 1)
              )
*/
public void forallSorting()
{
    //...
}
```

(a) Example method using the universal quantifier forall in a postcondition

```
@Test
public void forallSortingTest_10() throws Exception
{
    java.util.List<java.lang.Integer> list = cloner.deepClone(getFieldValue(testObj, "list", java.util.List.class));


    String methodId = "forallSorting";
    Class[] parameterClasses = {};
    List<java.lang.Integer> indexs_0 = javaTestUtility.getUniversalValues(testObj, methodId, 0);
    boolean forall_6 = true;

    testObj.forallSorting();

    for(int index : indexs_0)
    {
        forall_6 = forall_6 && (list.get(index) < list.get(index + 1));
    }
    Assert.assertTrue(forall_6);

    setUp();
}
```

(b) Example method testing assertions over a range

Figure 5.13: Example method with forall and SPEST generated test

```
/*
    //Verify that the elements in the field "list" are NOT sorted by natural ordering
    POST:
        !(exists(int index; (index > -1) && (index < list.size() - 1);
            list.get(index) > list.get(index + 1)
            )
         )
 */
public void existsSorting()
{
    //...
}
```

(a) Example method using the universal quantifier exists in a postcondition

```
@Test
public void existsSortingTest_11() throws Exception
{
    java.util.List<java.lang.Integer> list = cloner.deepClone(getFieldValue(testObj, "list", java.util.List.class));


    String methodId = "existsUnsorting";
    Class[] parameterClasses = {};
    List<java.lang.Integer> indexs_0 = javaTestUtility.getUniversalValues(testObj, methodId, 0);
    boolean exists_7 = false;

    testObj.existsUnsorting();

    for(int index : indexs_0)
    {
        exists_7 = exists_7 || (list.get(index) > list.get(index + 1));
    }
    Assert.assertTrue(!(exists_7));

    setUp();
}
/*End generated tests*/
```

(b) Example method testing assertions over a range

Figure 5.14: Example method with exists and SPEST generated test

Figure 5.15a shows a Java method with a variety of parameter types to be generated in the unit tests. Figure 5.15b shows the SPEST generated test method from the *primitiveParameterGeneration* method in figure 5.15a. The SPEST generated test is responsible for generating a variety of values for each variable type and using pairwise combinations to test the source method. Table 5.2a shows the values of the primitive fields, table 5.2b shows a summary of the values of the one dimensional int array, and table 5.2c shows a summary of the two dimensional int arrays that were used to test the source method.

```java
/*
    POST:
        true
 */
public void primitiveParameterGeneration(int integer, double doub, String str, int[] singleArray,
                                         int[][] nestedArrays)
{
    //...
}
```

(a) Example method with primitive parameters

```java
@Test
public void primitiveParameterGenerationTest_6() throws Exception
{
    int testComboIndex;

    String methodId = "primitiveParameterGeneration_int_double_java.lang.String_[I_[[I";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "integer",
            java.lang.Integer.class);
    List<java.lang.Double> testPoints_1 = javaTestUtility.getSamplePrimitives(testObj, methodId, "doub",
            java.lang.Double.class);
    List<java.lang.String> testPoints_2 = javaTestUtility.getSampleObjects(testObj, methodId, "str",
            java.lang.String.class);
    List<int[]> testPoints_3 = javaTestUtility.getSampleObjects(testObj, methodId, "singleArray", int[].class);
    List<int[][]> testPoints_4 = javaTestUtility.getSampleObjects(testObj, methodId, "nestedArrays", int[][].class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size(), testPoints_1.size(),
            testPoints_2.size(), testPoints_3.size(), testPoints_4.size());

    int param_0;
    double param_1;
    java.lang.String param_2;
    int[] param_3;
    int[][] param_4;
    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        param_1 = testPoints_1.get(combinations[testComboIndex][1]);
        param_2 = testPoints_2.get(combinations[testComboIndex][2]);
        param_3 = testPoints_3.get(combinations[testComboIndex][3]);
        param_4 = testPoints_4.get(combinations[testComboIndex][4]);

        testObj.primitiveParameterGeneration(param_0, param_1, param_2, param_3, param_4);

        Assert.assertTrue(true);
        setUp();
    }
}
```

(b) Example test method with primitive parameter generation

Figure 5.15: Example method with primitive parameters and SPEST generated test

Identifier:	Type	Value 1	Value 2	Value 3	Value 4	Value 5
integer	Integer	-2147483648	-1895150792	-1	803433912	2147483647
doub	Double	-9.2233E18	0.0	6.5190E18	9.1977E18	9.2233E18
str	String	".ch"	"0000000009"	"00001122"	"000111english"	"000495"

(a) Primitive parameter values for the SPEST generated method *primitveParameterGenerationTest*

Value Index	Number of Elements	Value
1	0	{}
2	1	{-2147483648}
3	8	{2147483648, -1, 2147483647, 1061358870, 2047401661, -383127425, -1994955876, -2147483648}
4	64	{-2147483648, -1, -2015498915, -88857804, 2147483647, 914471808, -1495853356, -1715130815, . . .}
5	512	{-2147483648, -1, 1801597794, 2147483647, 382596815, -674427959, -832294302, 813993101, . . .}
6	4096	{-2147483648, -1, 2147483647, -1965888813, 373880394, 1171047886, -1842515410, 1151272091, . . .}

(b) int[] parameter values for the SPEST generated method *primitveParameterGenerationTest*

Value Index	Dimensions	Value
1	0 x 0	{}
2	1 x 1	{{-2147483648}}
3	2 x 2	{{2147483648, -1}, {2147483647, 940588298}}
4	8 x 8	{{2147483648, -1, 795703405, 2147483647, 467423069, 847867402, 1446586685, -718984542}, . . .}
5	22 x 22	{{-2147483648, -1, 2147483647, -177841822, -1441246691, 875574326, -2118676048, -914239098 . . . }, . . .}
6	64 x 64	{{-2147483648, -271700018, -1, 1613021220, 883004577, 2147483647, -1549663327, -1368420758 . . .}, . . .}

(c) int[][] parameter values for the SPEST generated method *primitveParameterGenerationTest*

Figure 5.16a shows a Java method with a variety of complex parameter types to be generated in the unit tests. Figure 5.16b shows the SPEST generated test method from the *complexObjects* method in figure 5.16a. The SPEST generated test is responsible for generating a variety of values for each variable type and using pairwise combinations to test the source method. Table 5.3a shows the values of the fields in the *Translation* parameter, table 5.3b shows a summary of the values assigned to the *Collection* parameter, and table 5.3c shows a summary of the values assigned to the *Map* that were used to test the source method.

```java
/*
    POST:
        //The interfaces Collection and Map are resolved to instantiable classes
        translations.recursiveField == null
        && publicField1' > doubles.size()
        && map.containsKey("Key_To_Success")
 */
public void complexObjects(Translations translations, Collection<Double> doubles, Map<Integer, Boolean> map)
{
    //...
}
```

(a) Example method with complex object parameters

```java
@Test
public void complexObjectsTest_7() throws Exception
{
    int testComboIndex;

    String methodId = "complexObjects_demo.Translations_java.util.Collection_java.util.Map";
    List<demo.Translations> testPoints_0 = javaTestUtility.getSampleObjects(testObj, methodId, "translations",
            demo.Translations.class);
    List<java.util.Collection> testPoints_1 = javaTestUtility.getSampleObjects(testObj, methodId, "doubles",
            java.util.Collection.class, java.lang.Double.class);
    List<java.util.Map> testPoints_2 = javaTestUtility.getSampleObjects(testObj, methodId, "map",
            java.util.Map.class, java.lang.Integer.class, java.lang.Boolean.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size(),
            testPoints_1.size(), testPoints_2.size());

    demo.Translations param_0;
    java.util.Collection<java.lang.Double> param_1;
    java.util.Map<java.lang.Integer,java.lang.Boolean> param_2;

    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        param_1 = testPoints_1.get(combinations[testComboIndex][1]);
        param_2 = testPoints_2.get(combinations[testComboIndex][2]);

        testObj.complexObjects(param_0, param_1, param_2);

        Assert.assertTrue(param_0.recursiveField == null);
        Assert.assertTrue(testObj.publicField1 > param_1.size());
        Assert.assertTrue(param_2.containsKey("Key_To_Success"));

        setUp();
    }
}
```

(b) Example test method with complex object parameter generation

Figure 5.16: Example method with primitive parameters and SPEST generated test

Identifier:	Type	Value 1	Value 2	Value 3	Value 4	Value 5
publicField1	int	-2147483648	-1	2147483647	2147483647	1151142463
publicField2	Double	-2147483648	-1	2147483647	2147483647	1165280315
recursiveField	Translations	object reference	object reference	object reference	object reference	object reference
privateField1	int	-2147483648	-1	2147483647	2147483647	-1522510788

(a) Translation parameter values for SPEST generated complexObjectTest

Value Index	Number of Elements	Value in the Collection
1	0	
2	1	-9.223372036854776E18
3	8	-9.223E18, 0.0, 9.2233E18, 9.2233E18, -8.6565E18, 7.1460E18, -2.7698E18, -2.9974E18
4	64	-9.2233E18, 0.0, 9.2233E18, -7.7257E18, 2.0269E18, 5.39307E18, -3.8164, -4.1342E18. . .
5	512	-9.2233E18, 0.0, 9.2233E18, 8.3064E18, -5.4999E18, -8.9274E18, -6.5830E18, -7.2955E18. . .
6	4096	-9.2233E18, 4.6083E18, 9.2233E18, -4.7937E18, 4.3043E18, 8.1423E18, -2.0471E18, 7.8022E18 . . .

(b) Collection parameter values for SPEST generated complexObjectTest

Value Index	Number of Elements	Key → Values
1	0	
2	1	2147483648 → false
3	2	-2147483648 → false, -1 → true
4	8	-2147483648 → false, -1 → true, 2147483647 → true, -1324622472 → true, 2036353400 → true . . .
5	22	-2147483648 → false, -1 → true, 2147483647 → true, 598260777 → true, 592176364 → true . . .
6	64	-2147483648 → false, -1 → true, 2147483647 → true, -566264002 → true, 430549315 → true . . .

(c) Map parameter values for SPEST generated complexObjectTest

Chapter 6

Experimental Validation & Results

Two experiments were designed to validate SPEST's ability to reduce effort spent on testing, the readability of the generated tests, and the quality of SPEST generated tests. This chapter describes the experiments used to validate SPEST's contributions, the results collected from the experiments, and analyzes the significance of the results. The experimental subjects were students in Software Engineering courses taught at Cal Poly university during the time period of September 2014 through December 2015.

6.1 Reducing effort spent on testing with SPEST

This section describes the experimental setup used to compare SPEST's understandability to JML and SPEST's ability to reduce the effort required by developers to write unit tests as well as the results from the experiment.

6.1.1 Experimental Setup

To assess the effort of generating tests through SPEST, SPEST generated tests were compared against hand written tests that used JML as a specification language. SPEST was not compared against the state of the art black-box test generation tool for Java, JMLUnitNG, because the tool was unable to be run on the students' projects due to the version of Java used. In this experiment, hand written unit tests that used JML as a specification language were compared to tests generated entirely by SPEST. Since the quality of specification generated tests is based on the quality of specifications, student-written Java classes that met both of the following criteria were used:

1. the specifications were well specified, based on human inspection

2. the specifications were well tested, based on code coverage

Tests were written by students who were a mixture of Software Engineering students and Computer Science students in their third, fourth, and fifth years of school. The SPEST generated files were generated from translations of the JML specifications to SPEST specifications using the conversion procedure in appendix B.

To quantitatively measure the effort required, the *test efficiency ratio* was used to compare the two sets of tests. The test efficiency ratio was defined as the ratio of line coverage to the number of lines of code written. The line coverage of a Java file is the ratio of the number of lines of source code that were executed while running unit tests to the total number of executable lines of code in the source code. For SPEST the number of lines was calculated as the number of lines containing a SPEST keyword or a boolean expression within a SPEST comment. For the Hand Written tests, the number of lines was counted from the number of lines in the test file. The test files used to count lines were formatted to remove all blank lines and comments. The line coverage was calculated using the line coverage metrics tool built into IntelliJ version 14. While the test efficiency ratio is a non-standard metric, it was found to be the most meaningful measurement for comparing data from the limited input set.

To ensure the tests SPEST generated were meaningful, the number of false negatives produced in SPEST were recorded for each test file generated. A false negative test result was defined as a unit test that failed due to error within the SPEST framework and not due to an error within a developer's source code or a developer's assertions. Generated assertions may fail due to lack of preconditions to describe the structures of objects or false postconditions that do not properly describe the behavior of the functionality under test. Faults that were the result of incomplete or incorrect specifications were not included in this data sample. The percent of false negative test results represents the number of unit tests that were false negatives out of the total number of unit tests within a generated test class. If a test was unable to be run due to a compilation error, then all of its methods were considered to have resulted in false negatives. The number of false negatives was calculated by running the tests generated by SPEST and manually verifying and counting each failed assertion.

6.1.2 Results and Analysis

The experiment was conducted to answer the question does SPEST outperform the Java Modeling Language in effort required to write/generate tests from the specifications. This was measured quantitively

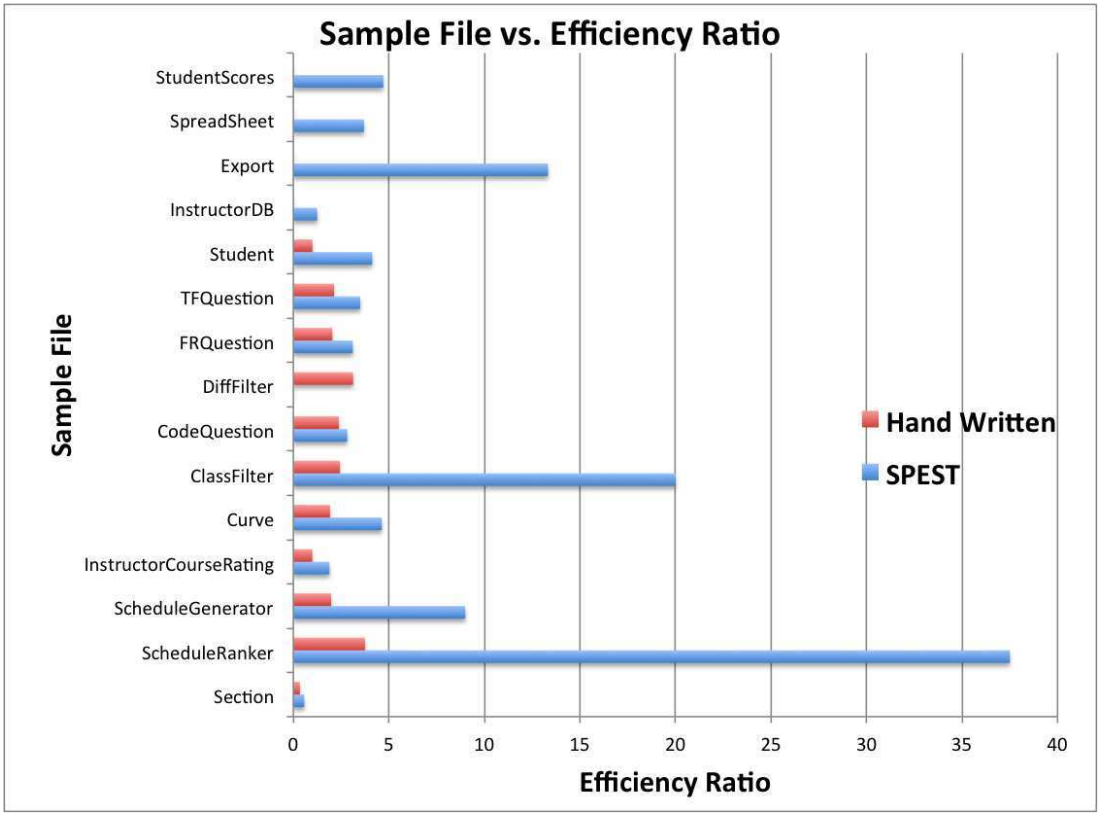


Figure 6.1: Sample File vs. Efficiency Ratio

as a function of test metrics and manual verification of the results of running the tests. There was also a qualitative assessment of SPEST in the form of a survey of students' opinions of how the tool helped them understand their programs.

Figure 6.1 shows the relationship between the number of lines written to generate a given percentage of line coverage. A higher ratio indicates that a developer can write less code and still get large amounts of coverage while a lower ratio shows that a developer must write more code to get large amounts of coverage. The results vary from file to file with SPEST having a higher test efficiency than the hand written tests in all but one test case. In the test case where SPEST did not outperform the hand written tests, SPEST was unable to compile the original file and so it was also unable to generate any tests.

The results of the Efficiency Ratio vs. File in figure 6.1 show that in the average case SPEST is as efficient or much more efficient than the hand written tests. In the case where SPEST failed to outperform the hand written tests, the file DiffFilter.java, the SPEST parser was unable to interpret the

file and so no tests were generated.

The efficiency metric was highly dependent on the number of methods within each test case. In the cases where SPEST was much more efficient, the files had very few methods. This is likely because SPEST was able to create a lot of template code and still maintain high line coverage for the small number of tests. In larger files that have more methods, the hand written tests become nearly as efficient as the SPEST generated tests because the amount of code needed to setup the unit tests represents a smaller portion of the file. These results show that SPEST can help developers write less code to test their systems which reduces the potential for human error and is very effective at reducing the amount of effort needed, especially early in development.

In all cases where SPEST's efficiency was as good or better than the hand written tests, the tests generated by SPEST tests are robust to changes within the source code. The tests SPEST generated require less effort to maintain than the hand written tests since the effort needed to refactor is reduced by the tests dynamically generating the parameters to the methods under test and the member variables within those objects. In the cases where SPEST was able to successfully generate tests, the SPEST tests are the better choice for regression tests since they are less likely to break when code is refactored. In the cases that the hand written tests were nearly as efficient as the SPEST generated tests, the hand written tests are better at targeting individual bugs and providing more precise feedback about known edge cases.

Figure 6.2 shows the number of errors that occurred while running the unit tests produced by SPEST. Errors were the result of internal faults in the tool and not the result of the specifications used to generate the test or the tests themselves. 25"%" of the samples contained one or more false negatives.

The results in Figure 6.2 are less conclusive. In the majority of the tests, there were no false negatives. In the few files there were false negatives, the false negatives made generated tests unusable. The leading causes of the false negatives in order of commonness were an infinite loop in the tool, a null pointer exception while the tool was generating objects to test with, and a compilation error preventing the tests from being run. These false negatives contaminated the developer's results by failing unit tests; however, the false negatives are clearly the fault of the SPEST tool as indicated by the warnings and exceptions. Since the tests either resulted in no false negatives or resulted in warnings and errors that clearly indicated SPEST was the source of the error, the false negatives are not a prohibitive barrier to the use of the tool, but do show one aspect where the hand written tests are more effective. The hand

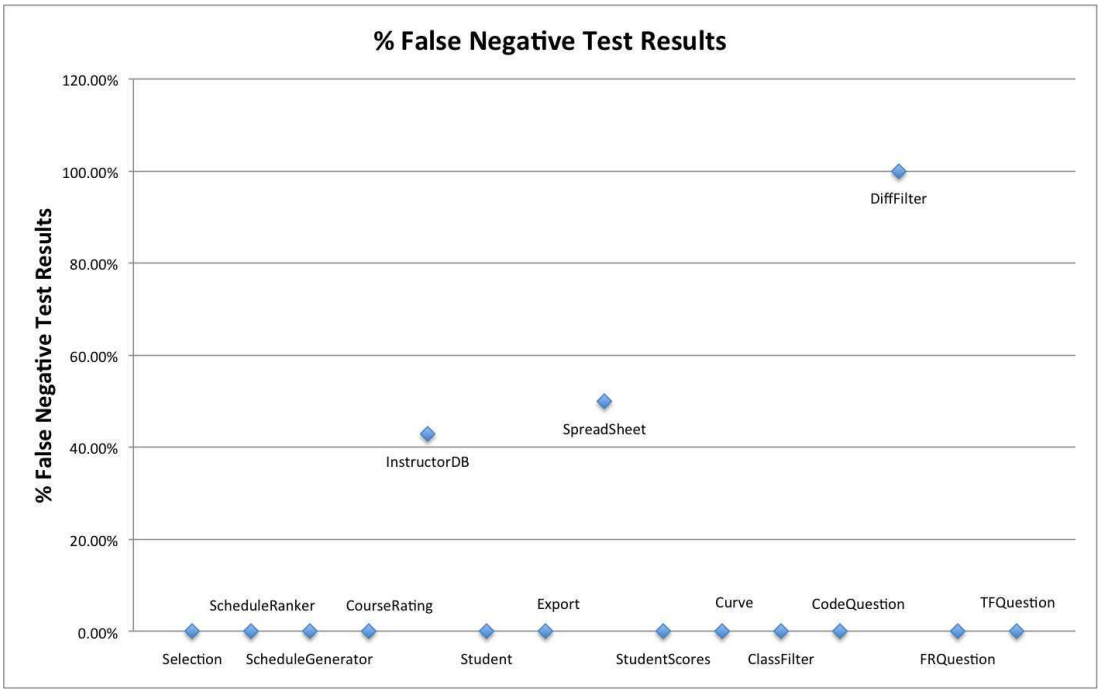


Figure 6.2: "%" False Negative Test Results

written tests are more consistent and offer the developer more control since they can correct any false negatives within their own tests and are not subject to bugs within a third party tool.

Figure 6.3 shows the survey questions asked of students at the end of the two-quarter sequences for JML and SPEST. The letter *L* was substituted with *JML* or *SPEST*. The free form responses can be seen in appendix C.1 and appendix C.2.

Table 6.1 shows the results of asking students about their experience using both JML and SPEST as a specification language. The results show that there was no significant difference between students

Question:	**1**	**2**	**3**	**4**	**5**	**6**	**7**
JML: Mean	2.68	2.72	2.52	2.84	2.96	2.92	2.64
SPEST: Mean	2.45	2.77	2.91	2.77	2.86	2.59	2.77
JML: Std Dev	1.22	1.4	1.29	1.43	1.17	1.15	1.19
SPEST: Std Dev	0.96	1.11	1.15	1.07	1.17	0.85	0.92
Significant?	No	No	No	No	No	No	No

Table 6.1: Statistical summary of survey results

1. Using L for formal specification helped me understand the programs I developed, better than if L had not been used.

2. The L specifications written by my teammates helped me understand the intended behavior of the code.

3. When using L, I had to write additional support code that I would not have written if L had not been used.

4. Using L helped with the development of unit tests for the programs I wrote.

5. As a language, L was easy to understand.

6. The L compilation tools were easy to use.

7. The L documentation was useful.

8. Describe your experience and any problems you encountered wile using L. (Response here is optional free-form text.)

Figure 6.3: Survey of the usefulness and understandability of SPEST

using SPEST and students using JML.

The survey results shown in figure 6.1 did not meet expectations but are still promising. The students' free response answers, seen in appendix C.2, reveal two notable problems with SPEST: lack of documentation and the tool was not fully implemented. JML has been around for years, with dozens of developers working on the language and associated tools. During JML's development a combination of community questions and documentation were built up and are available as resources to students. SPEST is still in its beta stages of development, so while there is basic documentation on the tool, the documentation is still being developed and there is not a community asking and answering questions yet. The free responses for JML, seen in figure C.1 were less conclusive but generally revealed that students felt using the tool did not provide any concrete value. Despite the problems with SPEST, there was still no statistically significant difference when comparing responses to JML. Furthermore, SPEST successfully addressed one of the complaints that students had with JML by providing concrete value in the form of unit tests to students.

6.2 Readability of Generated Tests

This section describes the experimental setup used to compare SPEST's readability to highly readable hand written tests and JMLUnit generated tests as well as the results from the experiment.

6.2.1 Experimental Setup

To assess the readability of SPEST generated tests, SPEST generated tests were compared against hand written tests from well established testing developers as well tests generated by JMLUnit. The hand written tests were hand selected to correspond to areas of interest within SPEST's test generation process. When multiple sample tests were found for a given area of interest, the number of references to the source as well as the number of publications from the author were used to select the best candidate. The tests were collected from publications, educational websites, and from documentation from tools with similar capabilities. SPEST was not compared against the state of the art black-box test generation tool for Java, JMLUnitNG, because the tool had no meaningful examples of generated test code nor could any tests be created using the tool. Because readability is highly subjective, 134 students were asked to compare the readability of code from the different sources using the following scale:

1. A is far more readable
2. A is slightly more readable
3. A & B are equally readable
4. B is slightly more readable
5. B is far more readable

A and B in the scale refer to a segment of code from either SPEST or from an alternative source. Students were not told which of the sources were generated using SPEST and which were from alternative sources. Additionally, students were not provided with a definition for readability but were instead instructed to use their own judgement about what it means based on their programming experience. Appendix D shows the instructions provided to students as well as the segments of code used within the survey.

To ensure that the order the segments of code were presented in did not affect students' opinions of readability, the order in which the sources were presented was swapped intermittently throughout the survey and two different surveys were given to ensure that the order the questions were presented in did not influence students' decisions.

To gain a greater understanding of what students found more and less readable, the survey was designed to include questions that covered the following topics:

1. Testing all possibilities
2. Testing using pairwise parameter selection
3. Testing boundary cases
4. Testing with generic data
5. Testing a private method
6. Java Modeling Language Unit generated code

To understand how much of the variability was the result of students' opinions and not the result of the different sources of code, two of the six questions compared two samples of code from the same

Comparison Type	Average	Interpretation
Control 1	-0.45	Baseline variability in students responses to readability questions.
Control 2	-0.32	Baseline variability in students responses to readability questions.
Exhaustive	-1.21	The alternative source is slightly to far more readable than SPEST's code.
Pairwise	-1.01	The alternative source is slightly more readable than SPEST's code.
Reflection	-0.56	The alternative source is equally to slightly more readable to SPEST's code.
JML	0.65	SPEST's code is equally to slightly more readable than the alternative code.

Table 6.2: Summary of code readability survey results

source. The two control questions compared non SPEST generated code to non SPEST generated code as well as SPEST generated code to SPEST generated code.

This experiment was designed under the advice of Andrew Schaffner from the Cal Poly statistics department. Consultations with Mr. Schaffner led to the decisions to use two control questions, a Likert scale, and the organization of the different versions of the survey[1].

6.2.2 Results and Analysis

Table 6.2 shows a summary of the results of the code readability survey. The first two rows came from the same source and ideally would have had an average of 0. The first two rows also show the average variability amongst students responses to code readability questions. A positive average indicates that SPEST was more readable than the alternative source and a negative response indicates that the alternative source was more readable than SPEST. The magnitude of the average indicates how much more readable one source was compared to the other.

The results shown in table 6.2 identify several different areas that lead to different test generation paths. Ideally SPEST would be as readable or more readable than the handwritten tests; however, the

results show that SPEST still needs to improve in several areas.

The largest observed gap in readability occurred when comparing SPEST generated exhaustive tests to hand written exhaustive tests. The main difference between the two sets of source code was that the generated tests had to rely on branch logic and loops to iterate over all the data while the hand written test avoided any logical structures and enumerated all possibilities by hand. The next largest difference in averages was between the sources for pairwise test generation. Two primary differences between the two samples of code in this case was that SPEST generated tests had several additional variable declarations and used generic names, such as param_x, for each of the variables while the hand written tests were as minimal as possible. The average difference for test code that involved reflexive access to a private method showed only a small difference in the readability. Despite the small average difference, the samples of code for this area were very different; the hand written code was far more verbose and included additional comments that explained each of the exceptions that could be thrown while the SPEST generated code hid the exceptions. These results show different aspects of readability within code that students prefer and suggest a path forward for improving the readability of SPEST generated code.

While SPEST is still less readable than hand written tests, the results show SPEST is more readable than its predecessor JMLUnit. While the average difference between SPEST and JMLUnit generated tests is less than the average difference between SPEST and hand written tests, this is still a promising result. Due to constraints on the format the survey was presented, JMLUnit generated tests may have received inflated readability scores because only a subsection of one file was shown to users despite the JMLUnit generating several test files for every source file.

Chapter 7

Conclusions

The focus of this thesis has been a test generation tool to facilitate faster development of unit tests that are able to adapt as a user's specifications change. Demonstrations of the tool's functionality were presented, as was a detailed review of the tool's design and implementation.

7.1 Summary of Contributions

The specific contributions of this thesis are:

1. The design and implementation of a specification language and black-box test generation tool.
2. The design and implementation of a dynamic object generator that is capable of respecting users specifications.
3. Demonstration of how the execution of SPEST can transform users specifications into executable unit tests.
4. A thorough discussion of how to create artificial limits when generating sets of data have no inherent boundary, in particular for creating limits on the sets of quantified values for unbounded quantifier expressions, thereby allowing unbounded quantifiers to be finitely executable.

Chapter 8

Future Work

The following sections describe potential future work, which could involve creating a SPEST plugin for major IDE, improving the object generation algorithm used to create the parameters for the methods under test, preserving test parameters for regression testing, reducing the number of false negatives produced when running SPEST generated tests, and expanding SPEST's syntax and generation capabilities to include additional tools in the source language. Additional research will also be necessary to assess the best development efforts for the tool.

8.1 SPEST as a Plugin

SPEST is currently designed to be used as a stand alone tool. Incorporating the tool into major IDEs would reduce the amount of information users need to provide and reduce the learning curve associated with using the tool. The plugins would be used to add a validation button and a generation button directly into the IDE so that developers could run these services as easily as they compile their source code. Integration with the IDE would also allow for direct syntax highlighting to show users what compilation errors they have with their specifications.

8.2 Advancing the Object Generation Tool

Currently, the object generation algorithm is able to successfully generate objects in its current state, but does not test to see if the objects constructed meet all of the preconditions associated with it. This will require that objects be generated and filtered in a repeating process until there are enough generated objects that meet a user's preconditions or there are no more possible combinations of member variables. Additionally, when the object generation pulls objects that have already been created from the cache,

there is no guarantee of variance among the objects. To ensure variance, objects must be "scored" based on the variance amongst their fields so that the most representative data set will be selected. The scoring mechanism must be able to reflexively compare the fields of two objects of the same type.

8.3 Persisting Test Information

Because test parameters are retrieved from the user accessible object cache, there is no guarantee that the same inputs will be used between multiple execution of the unit tests when the cache changes. As a result of potential variability the inputs that failed when tests were run should be preserved. In order to preserve the failing test inputs for regression purposes, SPEST must perform two additional steps. First, when a SPEST unit test fails, the state of all of the parameters and the method that failed must be stored to an external file. The data can be persisted as JSON, XML, or any other data storage format. Second, when test file containing SPEST tests is generated, an additional test method must be created that will load the files containing information about the failing tests and the test methods that should be invoked.

8.4 Improving Stability

Reducing the number of false negatives will require further testing of the tool to ensure that internal errors are handled gracefully and do not contaminate a developer's tests. Many of the tools internal problems stem from a lack of sample data to test with. Finding or creating a large sample of code containing specifications will allow better testing and help the tool to become more robust. In addition to the development efforts, more research needs to be conducted to determine the best use for the tool and ultimately shape the way the tool is developed. Future studies should further this research by assessing alternative means for generating data as well as alternative means to bounding unbounded data sets. A large amount of raw data was gathered from the Fall 2015 307 class and can be used as a data set for future studies.

8.5 Expanding Capabilities

While SPEST currently offers developers most of the tools that they could need to create meaningful unit tests, SPEST currently lacks support for exceptional behavior. Exceptions are part of most modern development languages and ensuring that they are thrown when the correct inputs are passed in and that

their behavior matches expectations is an essential part of testing. To support this feature in SPEST, three sets of changes will need to be implemented: updating the syntax checker to identify exceptional syntax, updating the type checker to verify that objects used as exceptions are of the appropriate type, and updating the test generator to include additional information to check for specific exceptions being thrown.

BIBLIOGRAPHY

[1] Andrew Schaffner private communication. “private communication”, December 2015.

[2] Daniel M. Zimmerman private communication. “private communication”, May 2015.

[3] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass—java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[4] Hamid A. Basit, Damith C. Rajapakse, and Stan Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, pages 109–114, 2005.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[6] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[7] Cédric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Pearson Education, 2007.

[8] Harsh Bhasin and Esha Khanna. Black box testing based on requirement analysis and design specifications. *International Journal of Computer Applications*, 87(18), 2014.

[9] Juei Chang, Debra J. Richardson Richardson, and Sriram Sankar. Structural specification-based testing with ADL. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA ’96, pages 62–70, New York, NY, USA, 1996. ACM.

[10] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 231–255. Springer, 2002.

[11] David R. Cok. OpenJML: JML for java 7 by extending OpenJDK. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 472–479, Berlin, Heidelberg, 2011. Springer-Verlag.

[12] Apache Commons. Java expression language (JEXL). http://commons.apache.org/proper/commons-jexl/, December 2011.

[13] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *Proceedings of the IEEE 25th International Symposium on Software Reliability Engineering*, FoVeOOS'10, pages 201–211, Berlin, Heidelberg, November 2014. Springer-Verlag.

[14] Irwin S. Dunietz, Collin L. Mallows, Anthony Iannino, Willa Kay Ehrlich, and B.D. Szablak. Applying design of experiments to software testing. In *Proceedings of the 1997 International Conference on Software Engineering*, pages 205–215, May 1997.

[15] Robert W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967.

[16] Apache Software Foundation. Introduction to the standard directory layout. http://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html.

[17] John V. Guttag, James J. Horning, and Jeannette M. Wing. The larch family of specification languages. *Software, IEEE*, 2(5):24–36, Sept 1985.

[18] Harri Hakonen, Sami Hyrynsalmi, and Antero Järvi. Reducing the number of unit tests with design by contract. In *Proceedings of the 12th International Conference on Computer Systems and Technologies*, CompSysTech '11, pages 161–166, New York, NY, USA, 2011. ACM.

[19] John Hatcliff, Gary T. Leavens, Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Computing Surveys*, 44(3):16:1–16:58, June 2012.

[20] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009.

[21] David S. Janzen. Software architecture improvement through test-driven development. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 240–241, New York, NY, USA, 2005. ACM.

[22] Gabriele Kahlout. Implementing patterns with annotations. In *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, page 6. ACM, 2011.

[23] Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective java library to support design by contract. In *Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 175–196. Springer, 1999.

[24] Reto Kramer. iContract-the Java ™ design by contract ™ tool. In *Proceedings of the 1998 Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE, 1998.

[25] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a java modeling language. In *Proceedings of the 1998 conference on Object-oriented programming, systems, languages, and applications*, pages 404–420. Citeseer, 1998.

[26] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, Daniel M. Zimmerman, et al. JML reference manual. http://www.eecs.ucf.edu/ leavens/JML/refman/jmlrefman.pdf, 2008.

[27] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 417–420, New York, NY, USA, 2007. ACM.

[28] Tim Mackinnon, Steve Freeman, and Philip Craig. Extreme programming examined. In Giancarlo Succi and Michele Marchesi, editors, *Extreme programming examined*, chapter Endo-testing: Unit Testing with Mock Objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[29] Eliane Martins, Cecilia M.F. Rubira, and Nelson G.M. Leme. Jaca: a reflective fault injection tool based on patterns. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 483–487, 2002.

[30] Bertrand Meyer. On formalism in specifications. *Software, IEEE*, 2(1):6–26, Jan 1985.

[31] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall; 1 edition, 1991.

[32] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[33] Bertrand Meyer. *Agile!: The Good, the Hype and the Ugly*. Springer Science & Business Media, 2014.

[34] Steven S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[35] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003.

[36] Oracle. Java se development kit 8 downloads. http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html.

[37] Oracle. javac - java programming language compiler. http://docs.oracle.com/javase/7/docs/technotes/tools/solaris/javac.html.

[38] Oracle. Oracle java se support roadmap. http://www.oracle.com/technetwork/java/eol-135779.html, May 2015.

[39] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Proceedings of the 36th International Conference on Software Engineering on Future of Software Engineering*, FOSE 2014, pages 117–132, New York, NY, USA, 2014. ACM.

[40] Jonathan S. Ostroff, David Makalsky, and Richard F. Paige. Agile specification-driven development. In Jutta Eckstein and Hubert Baumeister, editors, *Proceedings of the 5th International Conference on Extreme Programming and Agile Processes in Software Engineering*, volume 3092 of *Lecture Notes in Computer Science*, pages 104–112. Springer Berlin Heidelberg, 2004.

[41] Petros Papadopoulos and Neil Walkinshaw. Black-box test generation from inferred models. In *Proceedings of the Fourth International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, RAISE '15, pages 19–24, Piscataway, NJ, USA, 2015. IEEE Press.

[42] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[43] Isabelle Perseil. Towards a specific software development process for high integrity systems. *SIGSOFT Softw. Eng. Notes*, 36(1):1–8, January 2011.

[44] Per Runeson. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, July 2006.

[45] Jean Scholtz and Susan Wiedenbeck. Learning a new programming language: a model of the planning process. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume ii, pages 3–12 vol.2, Jan 1991.

[46] J. Michael Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge University Press, New York, NY, USA, 1988.

[47] Eiffel Studio. Debugging limitations. https://docs.eiffel.com/book/eiffelstudio/debugging-limitations.

[48] Adnan Čaušević, Sasikumar Punnekkat, and Daniel Sundmark. Quality of testing in test driven development. In *Proceedings of the Eighth International Conference on the Quality of Information and Communications Technology*, pages 266–271, Sept 2012.

[49] Yurong Wang, Suzette Person, Sebastian Elbaum, and Matthew B. Dwyer. A framework to advise tests using tests. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 440–443, New York, NY, USA, 2014. ACM.

[50] A. Wassyng. Though this be madness, yet there is method in it? (keynote). In *Proceedings of the First Formal Methods in Software Engineering Workshop on Formal Methods in Software Engineering*, pages 1–7, May 2013.

[51] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *Software Engineering, IEEE Transactions on*, 20(5):353–363, May 1994.

[52] Daniel M. Zimmerman and Rinkesh Nagmoti. JMLUnit: The next generation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, FoVeOOS'10, pages 183–197, Berlin, Heidelberg, 2011. Springer-Verlag.

APPENDICES

Appendix A

ANTLR Grammar

This chapter includes the tokens, lexer, and parser grammar rules used to generate the syntax for SPEST in ANTLR 3.

A.1 Syntax

Figure A.1 shows a list of the tokens, the most basic building blocks of the grammar, used to build the syntax checker for SPEST.

```
BYTE           =  'byte';
SHORT          =  'short';
CHAR           =  'char';
INT            =  'int';
DOUB           =  'double';
FLOT          =   'float';
BOOL           =  'boolean';
LONGDECL       =  'long';
FUN            =  'fun';
VOID           =  'void';
TRUE           =  'true';
FALSE          =  'false';
RETURN         =  'return';
THROWS         =  'throws';
FORALL         =  'forall';
EXISTS         =  'exists';
NEW            =  'new';
PUBLIC         = 'public';
PROTECTED      = 'protected';
PRIVATE        = 'private';
STATIC         = 'static';
ABSTRACT       = 'abstract';
IF             = 'if';
IFF            = 'iff';
ELSE           = 'else';
PROGRAM;
FUNCTION;
TYPES;
TYPE;
DECLS;
FUNCS;
DECL;
DECLLIST;
PARAMS;
RETTYPE;
BLOCK;
STMTS;
INVOKE;
ARGS;
ARGTYPES;
NEG;
PRECOND;
ARRAY;
FIELDS;
```

Figure A.1: Tokens used to define SPEST's syntax

Figures A.2 and A.3 show a list of the lexer rules used to build the syntax checker for SPEST. These rules represent more complex tokens that can be defined from other tokens as well as regular expressions.

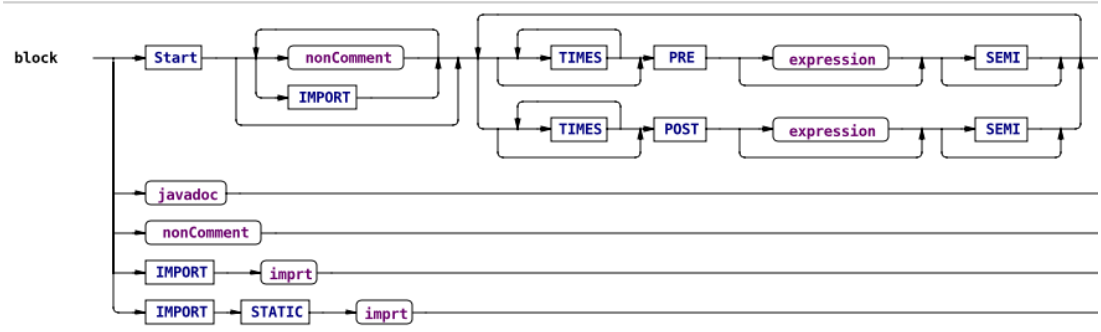
```
LBRACE    :  '{'    ;
RBRACE    :  '}'    ;
LBRACKET  :  '['    ;
RBRACKET  :  ']'    ;
SEMI      :  ';'    ;
COLON     :  ':'    ;
COMMA     :  ','    ;
QUOTE     :  '"'    ;
LPAREN    :  '('    ;
RPAREN    :  ')'    ;
DOT       :  '.'    ;
AND       :  '&&'   ;
OR        :  '||'   ;
EQ        :  '=='   ;
LT        :  '<'    ;
GT        :  '>'    ;
NE        :  '!='   ;
LE        :  '<='   ;
GE        :  '>='   ;
PLUS      :  '+'    ;
MINUS     :  '-'    ;
TIMES     :  '*'    ;
DIVIDE    :  '/'    ;
NOT       :  '!'    ;
ASSIGN    :  '='    ;
CARET     : '^'     ;
IFF       : 'iff'   ;
IF        : 'if'    ;
ELSE      : 'else'  ;
ESCAPE    : '\\'    ;
BITWISEAND: '&';
BITWISEOR: '|';
ANNOTATION: '@'{skip();};
DOLLARSIGN: '$';
NUMBERSIGN: '#';
QUESTION : '?';
MOD       : '%';
PRIME     : '\'' | '\`';
NULL      : 'null' | 'NULL';
IMPORT    : 'import';
UNDERSCORE: '_';
```

Figure A.2: Lexer rules used to define SPEST's syntax

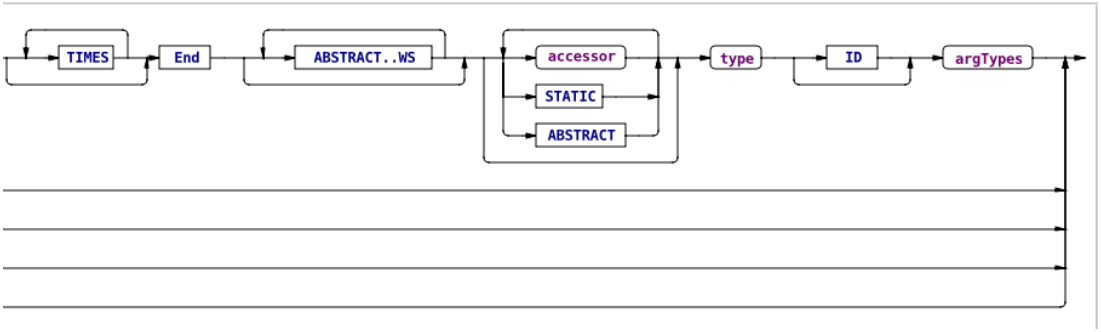
```
ID      : ('a'..'z' | UNDERSCORE)('a'..'z' | 'A'..'Z' | '0'..'9' | UNDERSCORE | '-')* PRIME?;

CHARACTER : PRIME ('a'..'z' | UNDERSCORE | 'A'..'Z' | '0'..'9'
           | WS | LBRACE | RBRACE | LBRACKET | RBRACKET | SEMI | COLON | COMMA  | LPAREN | RPAREN | DOT | LT | GT | PLUS | MINUS |
           TIMES | DIVIDE | NOT | ASSIGN | CARET | BITWISEAND | BITWISEOR | ANNOTATION | DOLLARSIGN | NUMBERSIGN | QUESTION | MOD
           | ('\\')('\'') | ('\\') (~('\''))*)
          PRIME;
INTEGER :  ('0'..'9')+ ;
LONG    :  '0l'|'0L'| ('1'..'9') ('0'..'9')* 'l' | ('1'..'9') ('0'..'9')* 'L';
DOUBLE  :  '0' | ('0'..'9')+ '.' ('0'..'9')*;
FLOAT   :  '0f'| '0F' | ('0'..'9')+ '.' ('0'..'9')* 'f' | ('0'..'9')+ '.' ('0'..'9')* 'F';
STRING  : '\"' (('a'..'z' | 'A'..'Z' | '0'..'9') | ('\'') | ('\\') | WS | LBRACE | RBRACE | LBRACKET | RBRACKET | SEMI | COLON | COMMA
          | LPAREN | RPAREN | DOT | LT | GT | PLUS | MINUS |  TIMES | DIVIDE | NOT | ASSIGN | CARET | BITWISEAND | BITWISEOR
          | ANNOTATION | DOLLARSIGN | NUMBERSIGN | QUESTION | MOD | UNDERSCORE)* '\"';
OBJ     : ('A'..'Z')('a'..'z' | 'A'..'Z' | '0'..'9' | UNDERSCORE | '-')* ;
INLINECOMMMENT: '//' ~('\r' | '\n')*  { skip();};
WS      :  (  ' '
           |  '\t'
           |  '\f'
           |  '\r'
           |  '\n'
           )+
            { skip(); }
        ;
IGNORE  : PRIME('a'..'z' | 'A'..'Z' | '0'..'9' | ' ' | UNDERSCORE | '-')+ {skip();}
        ;
```

Figure A.3: Lexer rules used to define SPEST's syntax

Figures A.4a, A.4b, A.5, A.6, A.7, A.8, A.9, A.10, and A.11 show graphical representations of the key parsing rules used to build the syntax checker for SPEST and build the abstract syntax tree.



(a) SPEST comment block syntax



(b) SPEST comment block syntax (Continued)

Figure A.4: Parser rules describing comments that contain preconditions and postconditions.

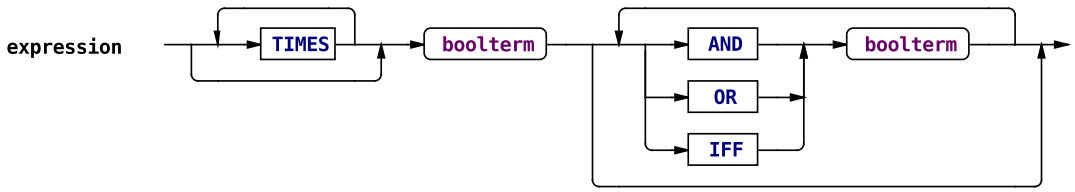


Figure A.5: Parser rules describing an expression

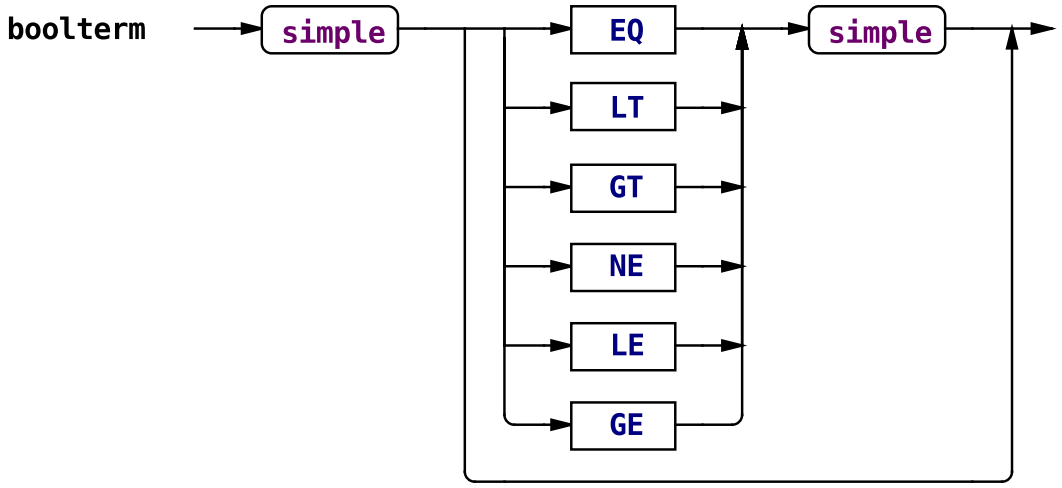


Figure A.6: Parser rules describing logical joining of boolean expressions

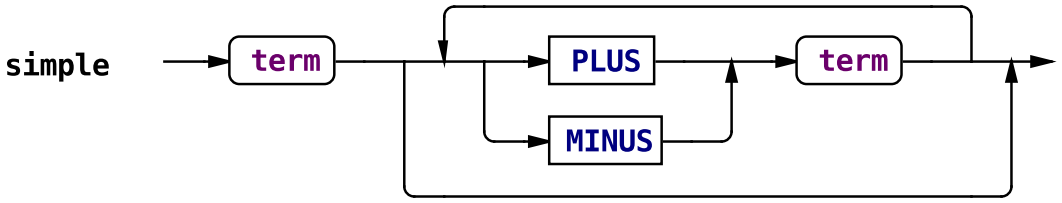


Figure A.7: Parser rules describing low precedence arithmetic operators

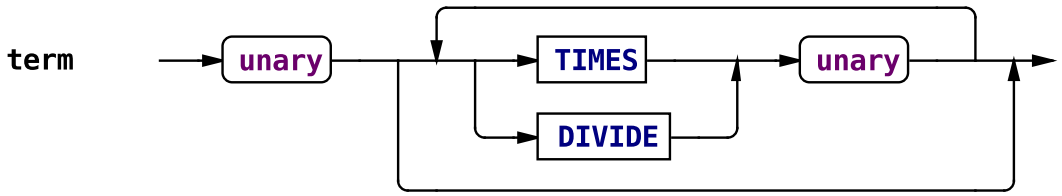


Figure A.8: Parser rules describing high precedence arithmetic operators

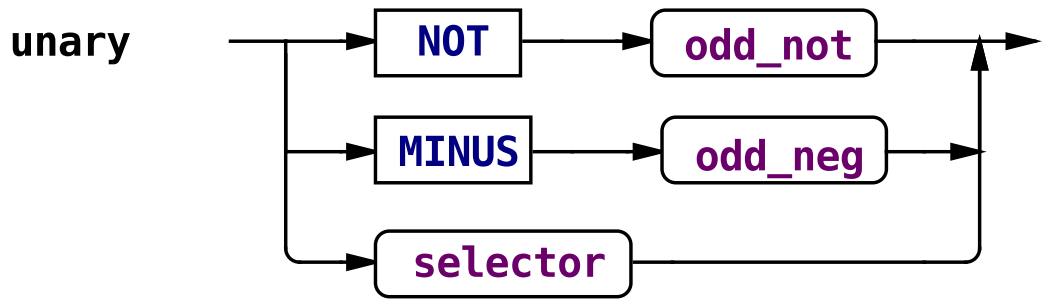


Figure A.9: Parser rules describing unary operators

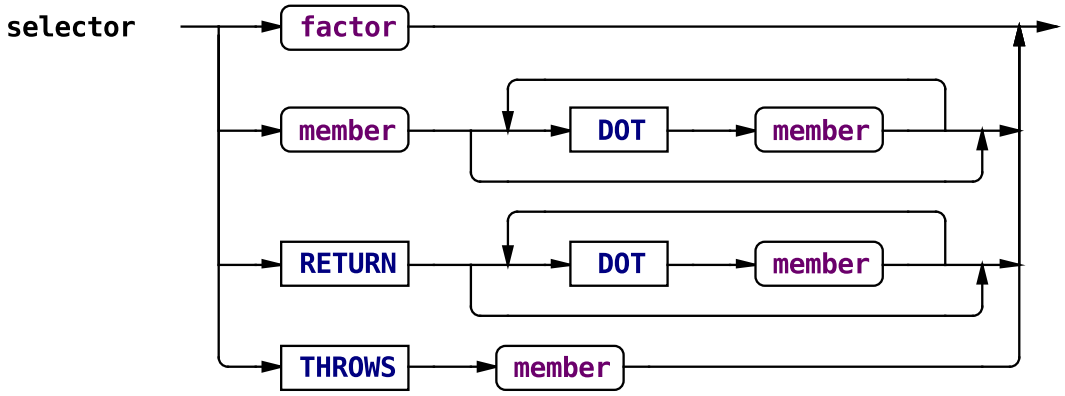


Figure A.10: Parser rules describing selection rules for identifying member variables or primitive data

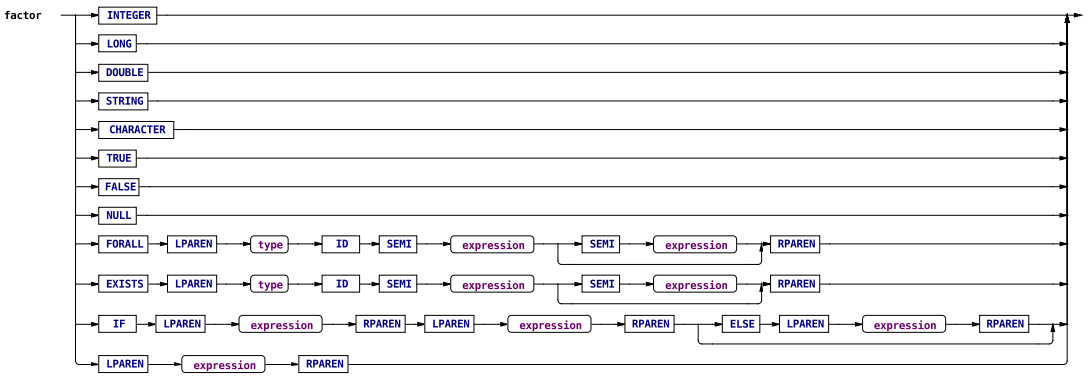


Figure A.11: Parser rules describing primitive data and subexpressions

Appendix B

JML to SPEST Conversion

The following instructions explain the steps taken to convert JML comments to SPEST comments. For each JML comment follow the steps below to create SPEST comments:

1. Replace the /*@ and @*/ surrounding the jml comment with /** and **/
2. Replace “requires” with “PRE:”
3. Replace “ensures” with “POST:”
4. Replace “/result” with “return”
5. Replace “/sum Type name; [boolean expression]; boolean expression;” with true a. /sum is currently unsupported in SPEST
6. Replace “forall Type t; genericList.contains(t); assertions with t” with “forall(Integer index; index ¡ list.size; index++; assertions”
7. Replace “exists Type t; genericList.contains(t); assertions with t” with “exists(Integer index; index ¡ list.size; index++; assertions”
8. Replace “\forall body” with “forall(body)”
9. Replace “\exists body” with “exists(body)”
10. Remove foralls and exists that have no assertions
11. Replace “exists” with “!forall”
12. Replace “x ==> y” with “if(x) (y)”
13. Replace “x <== y” with “if(y) (x)”

14. Replace “forall body” in Preconditions with true;

15. Replace “exists body” in Preconditions with true;

16. If a SPEST comment is above a method with an uppercase first letter, make the first letter lowercase.

17. If a package name starts with a capital letter and the name clashes with a Java Object name, replace the first letter of the name with its lowercase equivalent.

18. If a package name matches an objects name, then rename the package name to start with a lower case letter and the Object name to an uppercase letter.

19. Remove “;” from the end of each boolean expression

20. Remove “@Override” from methods that have SPEST comments

21. Replace expressions containing “instanceof” with true

Appendix C

Understandability Survey

Figures C.1 and C.2 show students responses to the survey questions in figure 6.3.

If I'm using Java to develop I like JML OK. But if I'm gonna develop in some other language like Python, it'd be cool to have a "python flavored" version of JML. Think you could come up with something like that?
annoying and redundant - i just wanted to write the damn code already
it was aight
I don't believe JML was a necessary component in learning how to program in a big team environment. However, it was interesting to know.
Some very simple functions didn't quite need JML and it created a hastle when having to write JML for every function.
It was simple to use and helped with understanding the code.
I can see the benefits of robustly using JML, but in our case, I felt like the requirements of using JML was more troublesome than helpful. Writing JML was sometimes harder than writing the actual implementation. Also, adding JML on top of our javadocs made our files extremely long and hard to read.
It was sometimes difficult to understand examples, since there aren't too many available.
JML is an old way of writing elegant, working code. Nowadays, we have effective debuggers that notify us of where errors occur.
JML was pretty easy to use. Getting used to the syntax at first was a little challenging, but that's usually a given for any new tools.
We found that we were just rewriting easy to read code. Also with Javadocs already being used it there wasn't a need for more explanation.
JML feels redundant to me. I would end up re-writing other test code.
Helpful to use before implementation phase. Less bugs if done this way.
I'm not sure how useful the language was, I mainly found it useful when I wrote tests but other than that I found some of the syntax a little annoying.

Figure C.1: JML survey free response answers

It was a bit hard to understand some of the errors.
Most of my members, including me weren't a fan of Spest. Spest was always put on the backburner to everything else.
Slightly difficult to use and didn't find documentation. However, seems promising and useful.
I was not a fan of SPEST. It was slightly difficult to use and there wasn't any documentation.
more difficult to implement than it needed to be
Not good.
Did not work with python but writing it for java allowed us to clearly understand which model behavior we would need to implement for cases outside of normal usage.
I had trouble using the spest compilation tool and ended up writing the unit tests without the help of spest. Was good for helping understand team members code.
Felt like it wasn't too helpful
Wasn't fully implemented when we started which made it hard for us to use. If it was a finished product, it could have been useful.
I think it would have been a great system had it been working. Future classes will benefit greatly from a fully functioning SPEST test generator.
Using spest helped me set up unit tests which actually revealed a number of design flaws within some of my classes. The compilation tools for spest were difficult to use. I don't think we ever were able to get it successfully set up.
I had issues when I didn't know it used the .class files because for eclipse they are separated from the .java files. I feel it wasn't as useful as it will be when it was completed because it felt like we were just describing general behavior of how the function will perform without and gratification besides getting it to compile since it didn't generate tests.
I didn't use spest in 308 (I am one of the 309 newcomers) so I didn't use it too much. I think the main use for me was that it forced me to formalize and think about the expected behaviour of my functions. I was bummed we couldn't get too much going as far as generated tests, but overall there were no bad experiences with spest.
The lower scores I gave were mostly due to the fact that I could never get SPEST working correctly no matter how much help I got, which was frustrating. I gave up using it after awhile because I didn't think I would ever get it working correctly.
I don't get SPEST, I couldn't figure it out and I couldn't use it.

Figure C.2: SPEST survey free response answers

Appendix D

Code Readability Survey

The following figures show the instructions and code samples used to analyze how participants rank SPEST generated test code against alternative sources in terms of readability.

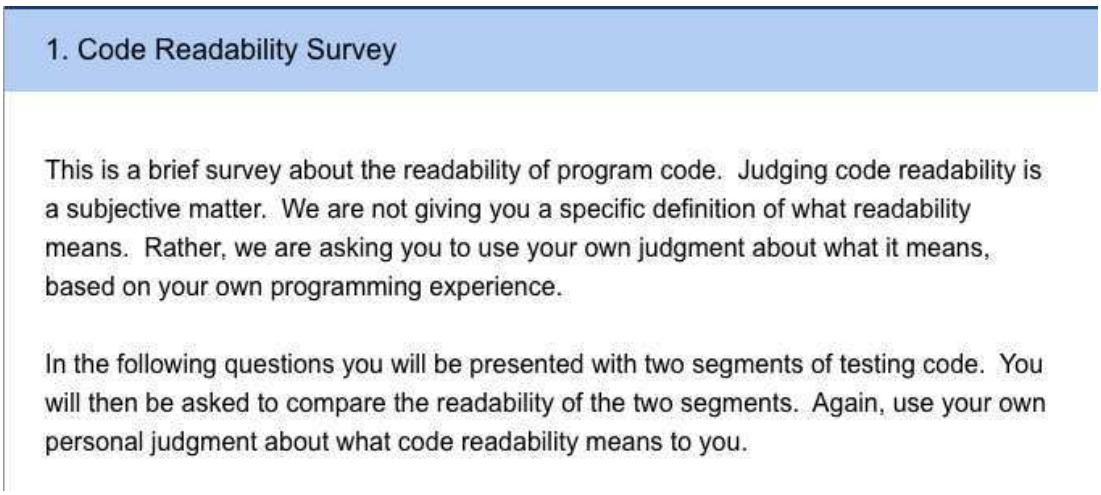


Figure D.1: Code eadability survey instructions

2. Testing all possibilities

A)

```
@Test
public void testExhaustive()
{
    assertEquals("wrong grade", 'B', Grade.getLetterGrade(100));
    assertEquals("wrong grade", 'B', Grade.getLetterGrade(99));
    assertEquals("wrong grade", 'B', Grade.getLetterGrade(98));
    ...
    ...
    ...
    assertEquals("wrong grade", 'A', Grade.getLetterGrade(2));
    assertEquals("wrong grade", 'A', Grade.getLetterGrade(1));
    assertEquals("wrong grade", 'A', Grade.getLetterGrade(0));
}
```

B)

```
@Test
public void getLetterGradeTest_0() throws Exception
{
    int testComboIndex;

    java.lang.String ret;
    String methodId = "getLetterGrade_int";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "val", java.lang.Integer.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size());

    int param_0;
    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        ret = testObj.getLetterGrade(param_0);

        if(param_0 >= 90)
        {
            Assert.assertTrue(ret == "A");
        }
        if(param_0 >= 80 && param_0 < 90)
        {
            Assert.assertTrue(ret == "B");
        }
        if(param_0 >= 70 && param_0 < 80)
        {
            Assert.assertTrue(ret == "C");
        }
        if(param_0 >= 60 && param_0 < 70)
        {
            Assert.assertTrue(ret == "D");
        }
        else
        {
           Assert.assertTrue(ret == "F");
        }

        setUp();
    }
}
```

Figure D.2: Readability sample code for question 1

3. Testing using Pairwise parameter selection

A)

```
@Test
public void placeOrderTest_0() throws Exception
{
    int testComboIndex;

    String methodId = "placeOrder_java.lang.String_Product_Coupon_Payment";
    List<java.lang.String> testPoints_0 = javaTestUtility.getSampleObjects(testObj, methodId, "user", java.lang.String.class);
    List<Product> testPoints_1 = javaTestUtility.getSampleObjects(testObj, methodId, "product", Product.class);
    List<Coupon> testPoints_2 = javaTestUtility.getSampleObjects(testObj, methodId, "coupon", Coupon.class);
    List<Payment> testPoints_3 = javaTestUtility.getSampleObjects(testObj, methodId, "payment", Payment.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size(), testPoints_1.size(), testPoints_2.size(),
testPoints_3.size());

    java.lang.String param_0;
    Product param_1;
    Coupon param_2;
    Payment param_3;

    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        param_1 = testPoints_1.get(combinations[testComboIndex][1]);
        param_2 = testPoints_2.get(combinations[testComboIndex][2]);
        param_3 = testPoints_3.get(combinations[testComboIndex][3]);

        testObj.placeOrder(param_0, param_1, param_2, param_3);
        Assert.assertTrue(testObj.orderWasSuccessful());
        setUp();
    }
}
```

B)

```
@Test
public void userCanPurchaseProductWithCouponAndPayment()
{
   Object[][] pairwiseSets = valuesToTestForEachParameter();

   for (int i = 0; i < pairwiseSets.length; i++)
   {
      final Order order = new Order();
      order.setUser(pairwiseSets[i][0]);
      order.addProduct(pairwiseSets[i][1]);
      order.setCoupon(pairwiseSets[i][2]);
      order.addPayment(pairwiseSets[i][3]);
      final ECommerceWebSite webSite = new ECommerceWebSite();
      webSite.placeOrder(order);
      assertTrue(webSite.orderWasSuccessful());
   }
}
```

Figure D.3: Readability sample code for question 2

4. Testing Boundary Cases

A)

```
@Test
public void testBoundaries()
{
    assertEquals("wrong grade", 'A', Grade.getLetterGrade(75));
    assertEquals("wrong grade", 'A', Grade.getLetterGrade(100));
    assertEquals("wrong grade", 'B', Grade.getLetterGrade(60));
    assertEquals("wrong grade", 'B', Grade.getLetterGrade(74));
    assertEquals("wrong grade", 'C', Grade.getLetterGrade(50));
    assertEquals("wrong grade", 'C', Grade.getLetterGrade(59));
    assertEquals("wrong grade", 'F', Grade.getLetterGrade(0));
    assertEquals("wrong grade", 'F', Grade.getLetterGrade(49));
}
```

B)

```
@Test
public void testGiantTriangle()
{
    System.out.println("testGiantTriangle");
    Triangle instance = new Triangle("3000000", "4000000", "3000000");
    String expResult = "I feel your triangle is too big\n";
    String result = instance.determineTriangleType();
    assertEquals(expResult, result);
}
```

Figure D.4: Readability sample code for question 3

5. Testing with Generic Data

A)

```
@Test
public void addToListTest_0() throws Exception
{
    java.util.List<java.lang.Integer> list = cloner.deepClone(getFieldValue(testObj, "list", java.util.List.class));

    int testComboIndex;

    String methodId = "addToList_java.util.List";
    List<java.util.List> testPoints_0 = javaTestUtility.getSampleObjects(testObj, methodId, "integers", java.util.List.class,
java.lang.Integer.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size());

    java.util.List<java.lang.Integer> param_0;

    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);

        testObj.addToList(param_0);
        Assert.assertTrue(list.size() > 0);
        setUp();
    }
}
```

B)

```
@Test
public void addToMapTest_1() throws Exception
{
    java.util.HashMap<java.lang.Integer,java.lang.String> map = cloner.deepClone(getFieldValue(testObj, "map",
java.util.HashMap.class));

    int testComboIndex;

    String methodId = "addToMap_java.util.HashMap";
    List<java.util.HashMap> testPoints_0 = javaTestUtility.getSampleObjects(testObj, methodId, "idMapping",
java.util.HashMap.class, java.lang.Integer.class, java.lang.String.class);
     int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size());

     java.util.HashMap<java.lang.Integer,java.lang.String> param_0;
     for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
     {
         param_0 = testPoints_0.get(combinations[testComboIndex][0]);

         testObj.addToMap(param_0);
         Assert.assertTrue(map.keySet().size() > 0);
         setUp();
     }
}
```

Figure D.5: Readability sample code for question 4

6. Testing a Private Method

A)

```
@Test
public void methodTest_0() throws Exception
{
    int testComboIndex;

    String methodId = "method_int_double";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "x", java.lang.Integer.class);
    List<java.lang.Double> testPoints_1 = javaTestUtility.getSamplePrimitives(testObj, methodId, "y", java.lang.Double.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size(), testPoints_1.size());

    int param_0;
    double param_1;

    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);
        param_1 = testPoints_1.get(combinations[testComboIndex][1]);

        Object[] paramValues = {param_0, param_1};
        Class[] paramClasses = {int.class, double.class};

        javaTestUtility.getMethodValue(testObj, "method", paramValues, paramClasses);
        Assert.assertTrue(true);
        setUp();
    }
}
```

B)

```
@Test
public void invokePrivateMethod(Class targetClass, String methodName, Class[] argClasses, Object[] argObjects) throws
InvocationTargetException
{
    try
    {
        Method method = targetClass.getDeclaredMethod(methodName, argClasses);
        method.setAccessible(true);
        method.invoke(null, argObjects);
    }
    catch (NoSuchMethodException e)
    {
        // Should happen only rarely, because most times the
        // specified method should exist. If it does happen, just let
        // the test fail so the programmer can fix the problem.
        throw new TestFailedException(e);
    }
    catch (SecurityException e)
    {
        // Should happen only rarely, because the setAccessible(true)
        // should be allowed in when running unit tests. If it does
        // happen, just let the test fail so the programmer can fix
        // the problem.
        throw new TestFailedException(e);
    }
    catch (IllegalAccessException e)
    {
        // Should never happen, because setting accessible flag to
        // true. If setting accessible fails, should throw a security
        // exception at that point and never get to the invoke. But
        // just in case, wrap it in a TestFailedException and let a
        // human figure it out.
        throw new TestFailedException(e);
    }
    catch (IllegalArgumentException e)
    {
        // Should happen only rarely, because usually the right
        // number and types of arguments will be passed. If it does
        // happen, just let the test fail so the programmer can fix
        // the problem.
        throw new TestFailedException(e);
    }
}
```

Figure D.6: Readability sample code for question 5

7. Advanced testing example

A)

```
@Test
public void testAddKgs()
{
    this.init receivers("addKgs");

    for(int i = 0; i < receivers.length; i++)
    {
        this.init_vint("addKgs");
        final int[] kgs = vint;

        for(int j = 0; j < kgs.length; j++)
        {
            if(receivers[i] == null)
            {
                /* ... tell framework test case was meaningless ... */
            }
            else
            {
                try
                {
                    receivers[i].addKgs(kgs[j]);
                }
                catch(JMLEntryPreconditionError e)
                {
                    /* ... tell framework test case was meaningless ... */
                   continue;
                }
                catch(JMLAssertionError e)
                {
                    String msg = /* a String showing the test case */;
                    fail(msg + NEW_LINE + e.getMessage());
                }
                catch(java.lang.Throwable e)
                {
                    continue;
                }
            }
        }
    }
}
```

B)

```
@Test
public void addKgsTest_0() throws Exception
{
    int testComboIndex;

    String methodId = "addKgs_int";
    List<java.lang.Integer> testPoints_0 = javaTestUtility.getSamplePrimitives(testObj, methodId, "numKgs",
java.lang.Integer.class);
    int[][] combinations = CombinationSupport.getCombinations(testPoints_0.size());

    int param_0;

    for(testComboIndex = 0; testComboIndex < combinations.length; testComboIndex++)
    {
        param_0 = testPoints_0.get(combinations[testComboIndex][0]);

        testObj.addKgs(param_0);
        Assert.assertTrue(true);
        setUp();
    }
}
```

Figure D.7: Readability sample code for question 6