Test-Driven Development with Oracles and Formal Specifications

By
Shadi G. Alawneh, B. Eng.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree of

Master of Engineering

Memorial University of Newfoundland

Master of Engineering (2010) Memorial University of Newfoundland
(Electrical and Computer Engineering) St. John's, NewFoundland

Title: Test-Driven Development with Oracles and Formal Specifications

Author: Shadi G. Alawneh, B. Eng. (Jordan University of Science and Technology)

Supervisor: Dr. Dennis K. Peters

Number of pages: xii, 104

Abstract

The current industry trend to using Test Driven Development (TDD) is a recognition of the high value of creating executable tests as part of the development process. In TDD, the test code is a formal documentation of the required behaviour of the component or system being developed, but this documentation is necessarily incomplete and often over-specific. An alternative approach to TDD is to develop the specification of the required behaviour in a formal notation as a part of the TDD process and to generate test oracles from that specification. In this thesis we present tools in support of this approach that allow formal specifications to be written in a readable manner that is tightly integrated with the code through an integrated development environment, and test oracles to be generated automatically. The generated test code integrates smoothly with test frameworks (e.g., JUnit) and so can be directly used in TDD. This approach has the advantage that the specifications can be complete and appropriately abstract but still support TDD.

Acknowledgements

I would like to express my sincere appreciation for the assistance and guidance of Dr. Dennis K. Peters in the preparation of this thesis.

Also, I gratefully acknowledge the financial assistance received from the Faculty of Engineering and Applied Science, Memorial University and the Natural Sciences and Engineering Research Council (NSERC).

I would specially like to thank the people in the CERL lab (Ala'a S. Al-habashna, Al-Abbass Al-Habashneh and Rabie Almatarneh). It has been a lot of coffee with them.

Last but not least my father (Ghazi) and mother (Moyasser), just for being you.

Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Description
TOG	Test Oracle Generator
TDD	Test Driven Development
XP	Extreme Programming
JML	Java Modeling Language
TTS	Table Tool System
FM	Formal Methods
ADT	Abstract Data Type

Chapter 1

Introduction

Test-Driven Development (TDD) is a methodology that uses tests to help developers make the right decisions at the right time. TDD is not about testing, it is about using tests to create software in a simple, incremental way. Not only does this improve the quality and design of the software, but it also simplifies the development process. The steps of TDD are illustrated in the UML activity diagram of Figure 1.1. TDD is one of the core practices of Extreme Programming (XP)[6, 21]. Two key principles of TDD are 1) that no implementation code is written without first having a test case that fails with the current implementation, and 2) that we stop writing the implementation as soon as all of the existing test cases pass. Although not all developers agree with all of the XP practices, the ideas of TDD have started to gain wide acceptance.

In TDD, the test code is a formal documentation that describes the required behaviour for the component or the system being developed for the particular test cases included. However, tests alone describe the properties of a program only in

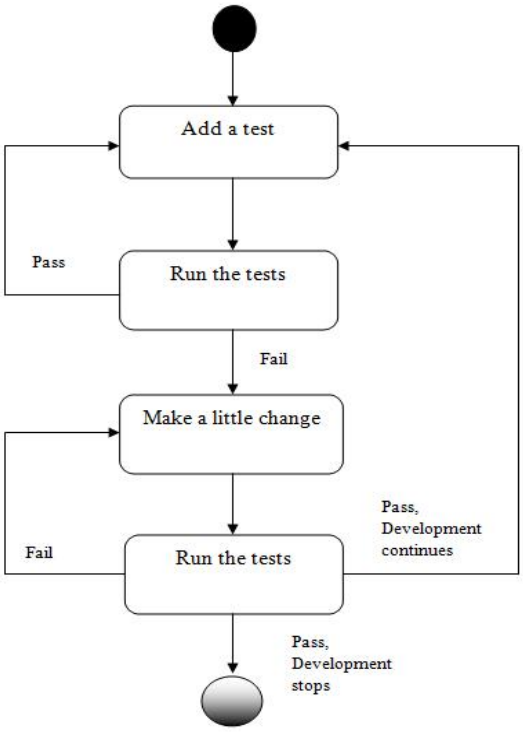

Figure 1.1: The Steps of Test-Driven Development (TDD)[2]

terms of examples and thus are not sufficient to completely describe the behaviour of a program. So, this documentation is unavoidably incomplete and often over-specific. To solve this problem we propose an alternative approach to TDD, which is to develop a formal specification of the required behaviour as a part of the TDD process and then generate test oracles from that specification. We thus propose a variation on the key TDD principles listed above: 1) No implementation code is written without first having a specification for the behaviour that is not satisfied by the current implementation, and 2) we stop writing the implementation as soon as the implementation satisfies the current specification. By generating oracles directly from the specification we are able to quickly and accurately check if the specification

is satisfied by the implementation for the selected test cases.

1.1 Purpose

In the context of test driven development, tests specify the behaviour of a program before the code that implements the program is actually written. In addition, they are used as a main source of documentation in XP projects, together with the program code.

An alternative approach to TDD is to develop a formal specification of the required behaviour as a part of the TDD process and then generate test oracles from that specification. If a program has been formally specified, it should be possible to use the specification as an oracle, so the expected output need not to be given by the user. This is particularly useful if the formal documentation is of a from that can be read and understood by both domain experts and programmers. Such documentation can be reviewed by the domain experts to ensure that the specified behaviour is correct and then used to communicate their intentions to the programmers. Generating an oracle from this documentation allows us to ensure that the documentation and prgram are consistent.

The purpose of this work is to develop tools in support of this approach that allow formal specifications to be written in a readable manner that is tightly integrated with the code through an integrated development environment, and test oracles to be generated automatically. One of the tools that we have developed is a Test Oracle Generator (TOG) tool that, given a relational program specification [33] using tabular

expressions [34], will produce a program that will act as an oracle. This oracle program will take as input an (input, output) pair from the program under test and will return ***true*** if the pair satisfies the relation described by the specification, or ***false*** if it does not.

1.2 Scope

In this thesis, we considered applying our approach for Test Driven Development(TDD) on methods and classes which are the basic components for any software application.

In our work, the kind of testing that we considered is the one composed of evaluating executable parts of the software system. Testing is one of the methods used to verify the software system, but in this work we didn't use the software verification since it has more wide meaning. We didn't discuss the selection of suitable tests for a component and how efficient those tests are. Interested readers are referred to the cited publication [46] for more details about these issues and a good survey of the related literature. Also, the kind of programs that we considered in this work is the terminating programs. For the non-terminating programs, some terminating sub programs (e.g. the body of an infinite loop) could be documented and tested using these methods.

Our methods are applicable for programs written in different kinds of programming languages but the tools that we have implemented to describe and explain these techniques only work for those written in 'Java'.

1.3 Types of Documents

The documentation is very important for computer systems. The goal of software documentation is to describe software systems and software processes. According to information in [22], consistent, correct and complete documentation of a software system is an important vehicle for the maintainer to gain an understanding of the system, to ease the learning and /or relearning processes, and to make the system more maintainable. Poor system documentation, on the other hand, is the primary reason for quick software system quality degradation and aging. Proper process documentation records the process, its stages and tasks, executing roles, their decisions and motivations, and the results of each individual process task.

With reference to the set of documents described in [36], in this work, we are focused on using module internal design documents [37] or module interface specifications to drive the development [42]. These two types of documents specify the behaviour of the module either in terms of the internal data structure and the effect of each access program on it, or in terms of the externally observable behaviour of the module.

1.4 Fillmore Software Project

The Fillmore Software Project [39, 40], is a collaborative project between researchers at Memorial University, McMaster University and the University of Limerick that was started in the Fall of 2006 and is aimed at building a suite of tools to provide

better support for software specifications or descriptions of software behaviour. The purpose of these tools is to improve the quality of the developed software.

This project attempts to develop a suite of tools for development, analysis and use of tabular software specifications. The set of tools that may be appropriate outcomes from this project is very large and includes powerful editors, document consistency checkers, verification systems, oracle generators, test case generators and model checkers. As a part of our work, we implemented the TOG part of the Fillmore Software Project.

1.5 Outline of This Thesis

Chapter 2 describes the related work. Chapter 3 describes the content and the format of the type of the program specification to be used for generating a test oracle. The design of the oracle itself and the design of the Test Oracle Generator are discussed in Chapter 4, and Chapter 5 discusses the Test Driven Development approach with oracles and formal specifications. Chapter 6 discusses the conclusions.

Chapter 2

Related Work

2.1 Test Driven Development

This section first describes TDD practice in detail, then details an empirical study of TDD that has been completed by researchers in Germany [26]. It also, describes some research that uses TDD.

In the TDD, before writing implementation code, the developer writes automated unit test cases for the new functionality they are about to implement. After writing test cases that generally will not even compile, the developers write implementation code to pass these test cases. The developer writes a few test cases, implements the code, writes a few test cases, implements the code, and so on. The work is kept within the developers intellectual control because he or she is continuously making small design and implementation decisions and increasing functionality at a relatively consistent rate. A new functionality is not considered properly implemented unless

these new unit test cases and every other unit test cases ever written for the code base run properly.

Based on [16], TDD is considered preferable over other approaches.

- In any process, there exists a gap between decision (design developed) and feedback (performance obtained by implementing that design). The favorable outcome of TDD can be ascribed to the lowering, if not eliminating, of that gap, as the granular test-then-code cycle gives constant feedback to the developer [7]. Consequently, bugs and their causes can be easily determined—the bug must lie in the code that was just written or in code with which the recently added code interacts. An often-cited tenet of Software Engineering, in concert with the Cost of Change [9], is that the longer a bug remains in a software system the more difficult and costly it is to remove. By using TDD, bugs are determined very quickly and the source of the bug is more easily determined. Therefore, it is this higher granularity of TDD that distinguishes the practice from other testing and development models.

- TDD gives programmers the ability to write code that can be tested automatically, such as having functions/methods returning a value which can be checked against expected results. Some benefits of automated testing include: (1) production of reliable systems, (2) improvement to the quality of the test effort, and (3) reduction of the test effort and minimization of the schedule.

- The TDD test cases create a thorough regression test bed. By continuously running these automated test cases, one can easily determine if a new change

breaks anything in the existing system. This test bed should also allow smooth integration of new functionality into the code base.

Lately, there are studies to analyze the efficiency of the TDD approach. Muller and Hagner [26] reported an experiment to compare TDD with traditional programming. The experiment is done with 19 graduate students, evaluated the efficiency of TDD in terms of (1) programming speed, (2) program reliability and (3) program understanding. In this experiment, the subjects were divided into two groups, TDD and control, with each group solving the same task. The task to be solved in this experiment is called "GraphBase". It consists of implementing the main class of a given graph library containing only the method declarations and method comments but not the method bodies; the students completed the body of the necessary methods. The programming was done in this way to give the researchers the ability to assess automated acceptance testing for their analysis.

The test cases that was specified by the TDD group was implemented while the code was written, but the control group students wrote automated test cases after completing the code. Subjects work for the two groups was divided into two phases, an implementation phase (IP), during which the subjects solved their assignment until they thought that their program would run correctly. This phase finished with their call for the acceptance-test. An acceptance-test phase (AP), during which the subjects had to fix the faults that caused the acceptance-test to fail. The researchers found no difference between the groups in overall development time. The TDD group had lower reliability after the IP phase and higher reliability after the AP phase.

However the TDD groups had statistically significant fewer errors when the code was reused. Based on these results the researchers concluded that writing programs in test-first manner neither leads to quicker development nor provides an increase in quality. However, the understandability of the program increases, measured in terms of proper reuse of existing interfaces.

Despite these results, this study is far from being a complete evaluation of test-first programming. The authors encourage other researchers to do the experiment again or to conduct a similar in order to extend the knowledge about test-first.

There are some researchers who have described tools that can be used to combine formal specifications with test driven development without loosing the agility of test driven development. In [5], Baumeister describes a tool that provides support to combine formal specifications with test driven development. This is done by using the tests, that drive the development of the code, also to drive the development of the formal specification. By generating runtime assertions from the specification it is possible to check for inconsistencies between code, specifications, and tests. Each of the three artifacts improves the quality of the other two, yielding better code quality and better program documentation in the form of a validated formal specification of the program. This method is exemplified by using the primes example with Java as the programming language, JUnit as the testing framework, and the Java Modeling Language (JML) [24] for the formulation of class invariants and pre- and postconditions for methods. They use JML since JML specifications are easily understood by programmers, and because it comes with a runtime assertion checker [11], which

allows them to check invariants and pre- and postconditions of methods at runtime.

Our work is different from the work above in that we use relations for the specifications, which characterize the acceptable set of outcomes for a given input. Also, we use test oracles that are generated automatically from the program specifications to determine if the software behaviour is correct or not for a given test input and output. By generating oracles directly from the specification we are able to quickly and accurately check if the specification is satisfied by the implementation for the selected test cases.

In [19], Herranz and Moreno-Navarro have studied how the technology of Formal Methods (FM) can interact with an agile process in general and with Extreme Programming (XP) in particular. They have presented how some XP practices can admit the integration of Formal Methods and declarative technology. In particular, unit testing, refactoring, and, in a more detailed way, incremental development have been studied from the prism of FM.

2.2 Oracle Generation

The research that has been done on improving the efficiency of software testing is divided into two categories: one is focused on the test case selection [17, 15, 27, 30], the other has concentrated on developing tools to help generate, maintain and track the testing documentation or run tests in simulated environments [10, 18, 31, 32]. All previous research areas are supportive to, but is different from the work that has done in this thesis.

Several researchers have developed tools that give the user the ability to determine if the results of a test are correct or not. In [31], Panzl explained three different kinds of automatic software test drivers that can be used to automate the verification of test results. In [18] Hamlet described another automatic testing system based on finite test-data sets, implemented by modifying a compiler. The disadvantages of these testing systems are: 1) The user should specify the expected result, which may be hard to acquire, and 2) The relational specifications, which may accept more than one acceptable result for a given input, can't be used because these systems only compare the expected and actual result.

The last disadvantage is partly solved by Chapman in [10]. This system describes the design and implementation of a program testing assistant which aids a programmer in the definition, execution, and modification of test cases during incremental program development. Moreover, it gives the programmer the ability to set the success criteria for a test case or use the default criterion equal, which checks for simple equality of a result and its correct value. Examples of other success criteria are set-equal, which checks two sets to see that they contain the same elements and isomorphic, which checks that arbitrary structures, possibly including pointer cycles, are topologically identical.

In[41] Peters and Parnas discuss the use of test oracles generated from program documentation. They describe an algorithm that can be used to generate a test oracle from program documentation, and present the results of using a tool based on it to help test part of a commercial network management application. The results demon-

strate that these methods can be effective at detecting errors and greatly increase the speed and accuracy of test evaluation when compared with manual evaluation. A design of test oracle generator they used allows using only C programming language in this prototype. If we need to choose among several programming languages we need to add several additional sub modules, one for each language.

In [38] Peters developed a prototype automated Test Oracle Generator (TOG) tool that, given a relational program specification using tabular expressions, will produce a program that will act as an oracle. This oracle program will takes input an input, output pair from the program under test and will return true if the pair satisfies the relation described by the specification, or false if it does not.

Other kind of systems, such as ANNA [25] and APP [44], give the user the ability to write code annotated with assertions that are evaluated while the code is executed. These assertions can be used as an oracle if they are completely specified and accurately placed to define the program specification.

In [45], Stocks and Carrington described a Test Template Framework (TTF) which is a structured strategy and a formal framework for Specification-based Testing (SBT) which is using the Z notation. In [43], Richardson et al. encourage the process of generating test oracles from formal specifications.

Other researchers have explained generating test oracles for abstract data types (ADTs) that are defined using algebraic specifications, e.g [3, 8, 14] or 'trace' specifications [47]. These kind of specification approaches discuss another kind of problem which is different from the specification approaches that is used in this work in that

they specify the desired properties of an ADT which is implemented by a group of programs, but the approaches that are used in this work are used specify the effect of a single program on some data structure.

Chapter 3

Methodology

3.1 Formal Software Specifications

Formal Specifications are documentation methods that use a mathematical description of software or hardware, which may be used to develop an implementation to drive automated testing. The emphasis is on what the system should do, not necessarily how the system should do it. Also, formal software specifications are expressed in a language whose vocabulary, syntax and semantics are formally defined. Examples of such languages (or notations) are VDM, Z, and B.

Formal specifications have several advantages over more traditional (informal) techniques:

I Since they are precisely defined, there is little room for misinterpretation of the intended meaning. This is in stark contrast to natural language and

other informal techniques, which leave lots of room for (mis)interpretation.

II Formal Specifications are mathematical entities, so they may be analyzed using mathematical methods and tools.

III They can be processed automatically, so we can use them as an exchange medium for software tools that depend on it.

IV They can be used as a guide for identifying appropriate test cases.

V They can be used to objectively determine if the behaviour of a system is acceptable or not.

For automated testing some form of formal specification of the required behaviour is essential. In a traditional automated testing process, this specification is in the form of the testing code, which will implement comparisons or tests to determine if the actual behaviour is acceptable. In this work we propose that the specification be expressed in a mathematical notation and that specification can be used to automatically generate testing code.

3.2 Program Specifications

A program specification in our work, describes the required behaviour of a program either in terms of the internal data structure and the effect of each access program on it, or in terms of the externally observable behaviour of the module. It consists of these components: constants, variables, auxiliary function and predicate definitions,

the program invocation, which gives the name and type of the program and lists all its actual argument program variables, and an expression that gives the semantics of the program. The following explains these in more detail.

3.2.1 Constants

A constant is a special kind of variable whose value cannot be altered during program execution. Many programming languages make an explicit syntactic distinction between constant and variable symbols. For example, in Java the following are constants: 10 and "Any Text".

3.2.2 Variables

In the specification, variables are strings of characters used to represent either the value of program variables in the initial state or final state of an execution, the value of expressions passed as arguments in auxiliary definitions, or as quantification indices. Variables which represent quantification indices are considered to represent a value only where they are bound.

All variables must have a type and should be defined in the documentation.

3.2.3 Auxiliary Function And Predicate Definitions

The definition of an auxiliary function consists of a name, a type, a list of argument variables and an expression that defines the semantics of the auxiliary function. Also, the definition of the auxiliary predicate is the same but the expression is a predicate

expression which is described in Section 3.2.4.

3.2.4 Predicate Expressions

A predicate expression is an expression that evaluates to **true** or **false** and consists of either quantified expression as described below, or a string of the form $G \wedge H$, $G \vee H$, $H \Rightarrow G$ or $\neg G$, where G and H represents predicate expressions.

3.2.5 Quantified Expressions

In our test oracle generator, quantification must be restricted to a finite set, which can be implemented as a java collection so that it can be automatically generated. This is done by permitting only the following forms of quantified expressions, where i is a variable, known as the index variable of the quantification, $G(i)$ is a collection and $H(i)$ is any predicate expression of a permitted form:

$$(\forall i : G(i).H(i))$$

$$(\exists i : G(i).H(i))$$

3.2.6 Tabular Expressions

The nature of computer system behaviour often is that the system must react to changes in its environment and behave differently under different circumstances. The result is that the mathematics describing this behaviour consists of a large number of conditions and cases that must be described. It has been recognized for some time

that tables can be used to help in the effective presentation of such mathematics [35, 1, 34, 20]. In our work we show such tabular representation of relations and functions as an significant factor in making the documentation more readable, and so we have specialized our tools to support them.

A complete discussion of tabular expressions is beyond the scope of this thesis, so interested readers are referred to the cited publications. In their most basic form, tabular expressions represent conditional expressions. For example, the function definition 3.1, could be represented by the tabular expression 3.2.

$$f(x,y) \stackrel{\text{df}}{=} \begin{cases} x+y & \text{if } x > 1 \wedge y < 0 \\ x-y & \text{if } x \leq 1 \wedge y < 0 \\ x & \text{if } x > 1 \wedge y = 0 \\ xy & \text{if } x \leq 1 \wedge y = 0 \\ y & \text{if } x > 1 \wedge y > 0 \\ x/y & \text{if } x \leq 1 \wedge y > 0 \end{cases} \tag{3.1}$$

$f(x,y) \stackrel{\text{df}}{=}$ (3.2)

	$x > 1$	$x \leq 1$
$y < 0$	$x + y$	$x - y$
$y = 0$	x	xy
$y > 0$	y	x/y

Although 3.1 and 3.2 are clearly a nonsensical example, they are representative of the kind of condintional expression that occurs often in documentation of software

based systems. We have found that the tabular form of the expressions is not only easier to read, but, perhaps more importantly, it is also easier to write correctly. Of particular importance is that they make it very clear what the cases are, and that all cases are considered.

Modern general purpose documentation tools, of course, have support for tables as part of the documents, but they are often not very good at dealing with tables as part of mathematical expressions. These tools also encourage authors to focus efforts on the wrong things: authors will work very hard to try to get the appearance of the table right, sometimes even to the detriment of readability(e.g., shortening variable names so that expressions fit in the columns).

3.2.7 Sample Program Specification

Figure 3.1, specifies a program '`ggcd`' which compares an integer value '`i`' with another integer value '`j`', returns the greatest common divisor of them if '$\texttt{i} > 0 \wedge \texttt{j} > 0$', otherwise returns 0. Additionally, it indicates if the two integers are positive by using the returned value, which is represented by a boolean variable '`result`'.

3.3 Tool Support

The tool support helped us to develop techniques and tools to facilitate the production of software design documentation that is 1) readable and understood by the users, 2) complete and accurate enough to allow analysis, both manually and mechanically and

Program Specification

Boolean
ggcd(**Integer** i, **Integer** j, **Integer** gcdvalue)

	i > 0 ∧ j > 0	i ≤ 0 ∨ j ≤ 0
gcdvalue =	$\max(\{x \in [0, \min(\mathtt{i}, \mathtt{j})] \mid \mathrm{cDiv}(\mathtt{i}, \mathtt{j}, x)\})$	0
result =	TRUE	FALSE

Auxiliary Predicate Definitions
Boolean cDiv(**Integer** a, **Integer** b, **Integer** x)
$\stackrel{\text{df}}{=}$ (a%x = 0) ∧ (b%x = 0)

Figure 3.1: Ggcd Program Specification

3) suitable for use as a specification from which to produce an acceptable program. We can't get these things with the general word processors.

3.3.1 OMDoc Document Model

As described in [23], the OMDoc (Open Mathematical Documents) format is a content markup scheme for (collections of) mathematical documents including articles, textbooks, interactive books, and courses. OMDoc also serves as the content language for the communication of mathematical software. OMDoc is an extension of the OpenMath and (content) MathML standards and concentrates on representing the meaning of mathematical formulae instead of their appearance. OpenMath and MathML are formats for individual mathematical expressions and OMDoc is a format for documents that include mathematics. The specifications in our work consist of program specifications, which, in OMDoc terms, are symbol definitions contained within theories. Also, each symbol has a type and possibly other information. Con-

sequently, this leads us to propose our specification model which consists of these OMDoc elements:

Theory : a theory is a self-contained part of a specification. It could, for example, represent a requirements specification, a module interface specification, a module internal design document or a single program function. A theory contains zero or more sections of each of the following kind.

Symbol : a symbol is a basic component of a specification: a variable, function, relation or constant. All symbols that are used in a specification must be defined somewhere, either by being declared to be a bound variable, defined in the specification itself, defined (globally) in an imported theory, or from a standard set (e.g., standard OpenMath content dictionary). A symbol has the following attributes:

Name : for referring to the symbol (required).

TTS Role : indicates how this symbol is used as part of a specification (optional).

Type : all symbols should have a type supplied.

Definition : a definition contains an expression that gives the semantics of a symbol.

Presentation : a presentation contains the format for a mathematical symbol. A presentation element has **for** attribute which identifies the symbol represented.

Each presentation contains one or more **use** elements. For more details see section 4.2.5.

Use : indicates how the symbol represented is in a specific language. A use element has the following attributes:

Format : specifies the name of the language this use element applies to. It could be a programming language, a text processing language such as latex or could identify some other tool.

Fixity : determines the placement of the symbol. This attribute can be one of the keywords **prefix**, **infix**, and **postfix**. For **prefix** it is placed in front of the arguments. For **infix** it is placed between the arguments. Finally, for **postfix** it is placed behind the arguments.

Separator : this specifies the separator in the argument list.

lbrack/rbrack : these two attributes handle the brackets to be used in presentation.

Code : is unparsed formal text and it is not needed in our documents but in some documents it is needed.

Text : is unparsed informal text and it is important for readability of the document.

Based on [4], any type of tabular expressions can be defined by providing:

A restriction : each type of tabular expression must satisfy a stated restriction. A restriction is a predicate that states the condition that a tabular expression

should meet, which might be on such properties as the number of grids, the index sets of grids, the type of elements in each grid and some properties of the grids. The restriction must be observed when the tables are constructed.

An evaluation term : a tabular expression represents a relation which may be a function. The evaluation term of a tabular expression has to be evaluated to determine the value of the tabular expression for a given assignment. The evaluation term is constructed using conventional and tabular expressions appear in the tabular expression as well as auxiliary functions.

A set of auxiliary function definitions : these functions are applied in defining the restriction and the evaluation term and will be used in evaluating or checking the table.

In OMDoc it is straightforward to add support for tabular expressions, simply by defining appropriate (OpenMath) symbols to denote them: we use a symbol for "table", which, following the model presented in [4], takes four argument expressions representing

1. The *evaluation term*, which expresses how the value of a tabular expression is defined in terms of the expressions in its grids. For (3.2) this expression would express that the value is that of the element grid, *T[0]*, which is indexed by indices of the true elements of each of the "header" grids, *T[1]* and *T[2]*, as follows: *T[0][select(T[1]),select(T[2])]*, where *select* is a function on a predicate grid that gives the index of the cell that is true.

2. The *static restriction*, which defines a condition that must be true of the grids, independent of the expressions in the grids, but possibly dependent on their types. This is used, for example, to assert the conditions on the number and size of the grids(i.e., the shape of the table). For (3.2) this would express that the index set of the central grid should be power set of the index sets of the header grids, and that the header grids must contain predicate expressions.

3. The *dynamic restriction*, which defines a condition that must be true of the grid expressions. This is used to assert constraints on the table to ensure that it has a well defined meaning. For (3.2) this would assert than the header grids, *T[1]* and *T[2]*, must be "proper" - only one cell expression should be true for any assignment.

4. A list of *grids*, which are indexed sets, represented by n-ary applications with symbol "grid" and taking pairs of cell index and cell contents as its arguments.

3.3.2 The Eclipse Framework

Eclipse is a software platform comprising extensible application frameworks, tools and a runtime library for software development and management. It is written primarily in Java to provide software developers and administrators an integrated development environment (IDE). "Eclipse employs plug-ins in order to provide all of its functionality on top of (and including) the runtime system, in contrast to some other applications where functionality is typically hard coded" [13]. Using this framework to develop our tool provides significant advantages over developing a stand-alone tool including its

widespread use in the user community, its facilities for tight integration of documents with other software artifacts, and provision of support for software development tasks.

3.3.3 Specification Editor

As part of our tools, we are developing a specification editor to support production of software documents, which is illustrated in Figure 3.2. This Editor provides a "multi-page editor" (which provides different views of the same source file) for ".tts" files, which are OMDoc files. One page of the editor is a structured view of the document, another one shows the raw XML representation, and another gives a detailed view of the document giving the user the ability to view and edit the mathematical expressions. The support libraries in Eclipse provide techniques to ensure that the views of the document are consistent. This editor is built using several open source libraries including the RIACA OpenMath Library.

This editor is seen as a primary means for the human users to interact with specification documents.

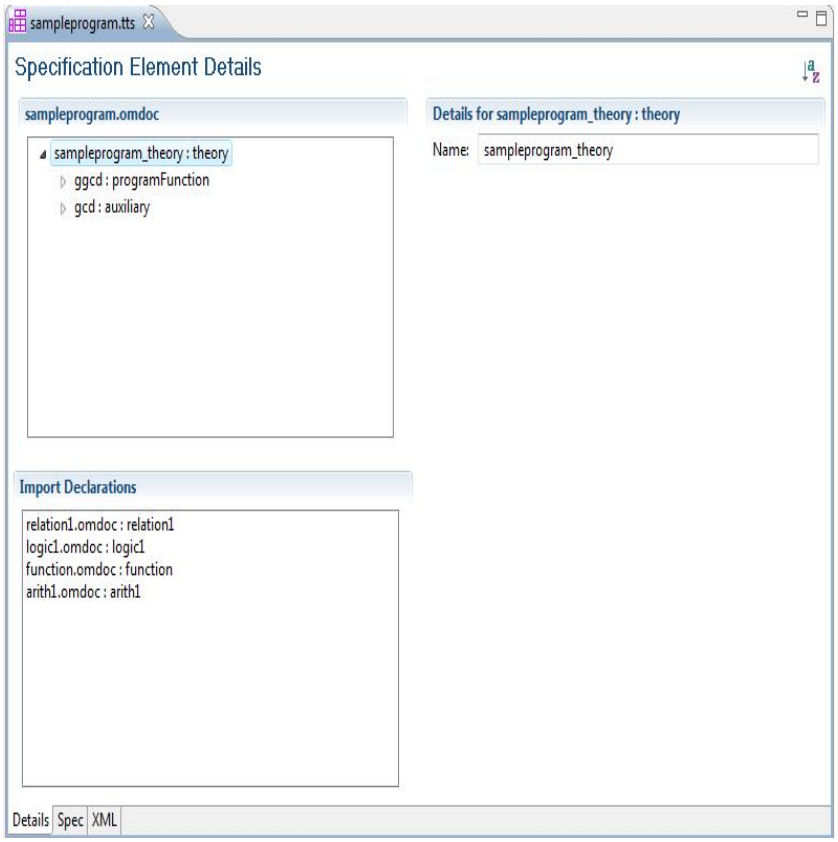

Figure 3.2: Screenshot of Editor

Chapter 4

Oracle Generation

This chapter describes the internal design of the oracle that will be the output of the Test Oracle Generator (TOG). The design is explained by using some examples from an oracle, which was produced for the sample '`ggcd`' program specification given in 3.2.7. This chapter also describes the requirements and design of the TOG. The work reported in this thesis is similar to the work in [41] but our approach for generating test oracles has the following characteristics that make it unique:

- We are using OMDoc as a standardized storage and communications format for our specifications, and so we can take advantage of other tools.
- The semantics of tabular expressions have been generalized to allow more precise definition of a broader range of tabular expression types.
- The test oracle generator is implemented using Java. This makes it easy to integrate with the Eclipse platform.

- The oracle generator has a 'graphical user interface' which is shown in Figure 3.2. This interface gives the user the ability to select any program specification and generate the oracle from it. This has the advantage of enabling the user to interact easily with the specifications.

- The generated test code integrates smoothly with test frameworks (e.g., JUnit) and hence, it can be directly used to test the behaviour of the program.

4.1 Oracle Design

4.1.1 Programming Language

The oracle is implemented using Java. This decision should not be seen as a significant feature of the design—if the intended application were different, the oracle design could be translated with some changes.

4.1.2 Internal Design Overview

The oracle can be viewed as a 'compiled' version of the specification in that it is generated by translating the 'source' specification into an executable form (Java code). The oracle can be executed without reference to the specification from which it was derived. So, it can be integrated smoothly with test frameworks (e.g., JUnit). This design has an advantage is that it reduces the time required for oracle execution by giving the user the ability to use optimization techniques.

An alternative approach to design of the oracle is to build it as an 'interpreter' which would represent the specification by data and evaluate it directly. This kind of design has an advantage that the oracle generation process is relatively simple and, since there is no generated code involved in the oracle, the oracle programs will be the same for any specification, only the data they use is dependent on the specification. A disadvantage for this design is that the oracle will need to interpret the semantics of the documentation during evaluation, and so would probably be comparatively slow to execute.

4.1.2.1 Expression Implementation

Any expression consists of one or more sub-expressions, the complexity of implementing this expression is managed by decomposing each expression into its sub-expressions and implementing each sub-expression individually. The oracle code thus consists of a set of internal functions and objects, each of which implements a sub-expression and may call other internal functions or object methods.

All programming languages in general, and Java in particular, provide support for basic logical and relational operators (i.e. $\wedge, \vee, \neg, >, <, =$ etc.), these operators can be used to implement some of the expressions. Also, it is possible to use these operators for implementing an entire expression as a single Java statement by translating it into a purely scalar, quantifier free expression (by expanding the quantification to a series of conjunctions or disjunctions) but the resulting Java statement would consist of many lines. While this would undoubtedly result in an oracle that executes relatively

quickly, since there would be none of the overhead associated with loops or function calls. It would, however, require significant effort on the part of the TOG to do the translation and would result in virtually incomprehensible oracle code. So, that is why the oracle is implemented using the Java logical and relational operators only where they directly represent the operators in the specification.

Another way to implement expressions is to use a class of Java objects. A specific expression is implemented by instantiating the suitable objects, which include references to their sub-expression objects. This helps to simplify the oracle generation process for expressions that have complex semantics such as tabular expressions. So, the TOG need only translate the expression into the suitable object constructor. In this work, we used the above two ways to implement the expressions.

The code to implement each type of expression is explained in the following sections below.

4.1.3 Scalar Expressions

Scalar(i.e. non-tabular) expressions can be translated into equivalent Java statements as described below.

4.1.3.1 Logical Operators

The logical operators can be directly translated to their Java eqivalent, as given in Table 4.2. (G and H are arbitrary predicate expressions.)

Table 4.1: Logical Operator Conversions

Logical Operator	Java Equivalent
$\neg G$	$!G$
$G \vee H$	$G\|\|H$
$G \wedge H$	$G\&\&H$

So, given the expression $(a > b \wedge a > 5)$ in the specifications, the corresponding Java code for that expression is:

$$(a > b)\&\&(a > 5)$$

4.1.3.2 Quantification

Quantifier expressions are implemented by using loops that call the suitable procedures to enumerate the elements of the set characterized as an integer interval and the boundaries for the interval given in the specifications. In our test oracle generator, quantification ($\forall$ — for all, and $\exists$ — there exists) must be restricted to a finite set, which can be implemented as a java collection so that it can be automatically generated from the specifications. In the example below the boundaries are (0,10). One distinction between the work reported in this thesis and that in [41] is that the previous work used Inductively Defined Predicate to specify the range for the quantification but we used a java collection.

The quantification "$(\forall i : \{0..10\}.p\_B[i] = p\_x)$", can be implemented as follows.

```
boolean result=true;
Integer_Interval bRange =new Integer_Interval(0,10);
```

```
Integer i=new Integer(0);
for(Iterator<Integer> it=bRange.iterator();it.hasNext()&&result;)
{
   i=it.next();
   result=((p_B[i]==p_x)&&result);
}
```

4.1.4 Tabular Expressions

Tabular expressions are implemented by instantiating an object of one of several classes of (Java) table objects which implement the various types of tabular expressions(normal, inverted and vector). These table classes contain all knowledge of the semantics of tabular expressions, so there is no need for this knowledge to be in the TOG. The expression in each cell of the table is implemented as Java class that extends a CellBase class and therefore contains a procedure, eval, which evaluates the expression in the cell.

Table objects have the following method, which is used to evaluate the table:
evaluateTable finds the index for the main cell that should be evaluated and returns the contents of that cell.

The expression "$\mathtt{i} > 0 \wedge \mathtt{j} > 0$", which is in the first cell of the column header of the ggcd tabular expression in Figure 3.1, is implemented as follows.

```
package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;
```

```
public class ggcd1_Grid_2_Cell_0 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_2_Cell_0(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Integer  i=(Integer)vars.getValue("i");
    Integer  j=(Integer)vars.getValue("j");

    return ((i>0)&&(j>0));

  }
}
```

The other cells in each table are implemented in a similar fashion. The oracle design for the `ggcd` tabular expression in Figure 3.1 is illustrated in Figure 4.1 and the design for the gcd tabular expression looks similar.

An alternative approach for implementing the tabular expressions that was considered is to convert the tabular expression into the equivalent scalar expression and implement the scalar expression as explained in the previous section. This approach has an disadvantage that the TOG would need to have the ability to do the translation.

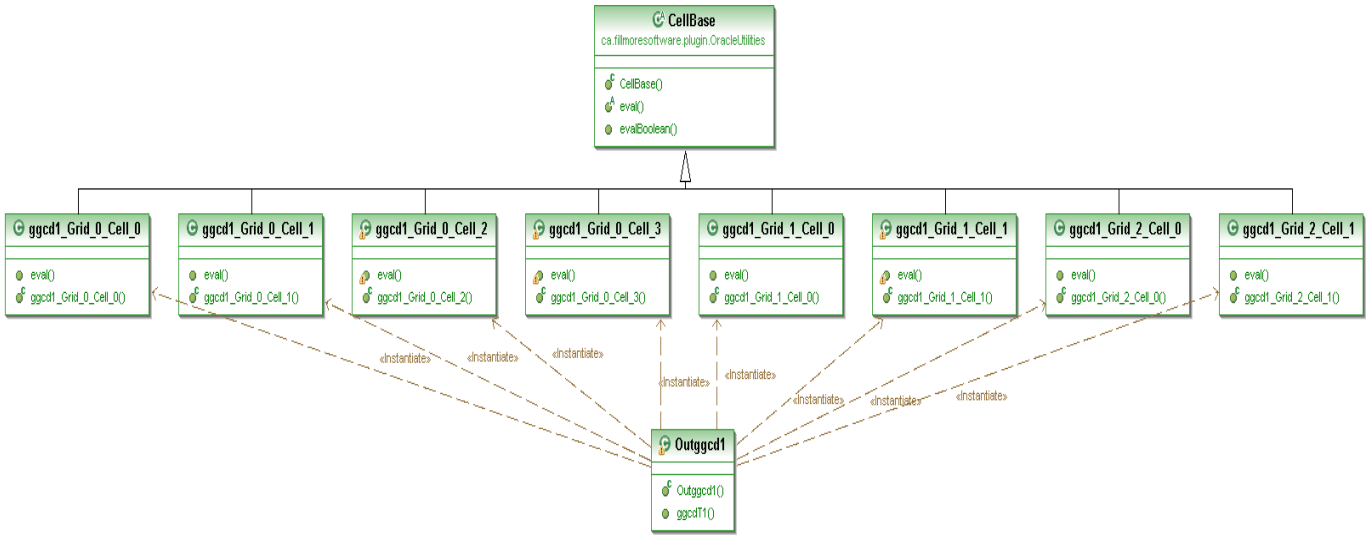

Figure 4.1: Oracle Design of ggcd Tabular Expression

4.1.5 Auxiliary Functions

An auxiliary function is implemented as a procedure, with the expression, implemented as described above, forming the body of the procedure. For example, consider the auxiliary function, which is used in the sample program specification in section 3.2.7 defined as follows:

Boolean cDiv(**Integer** a, **Integer** b, **Integer** x)

$\stackrel{\text{df}}{=} (\text{a}\%\text{x} = 0) \wedge (\text{b}\%\text{x} = 0)$

This is implemented by the following procedure:

```
package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;

public class AuxFunctions{

  static public Boolean cDiv(Integer a,Integer b,Integer
x){

    return (a % x == 0) && (b % x == 0);

  }



}
```

Suitable calls to this procedure are used in the code that implements expressions using the auxiliary function.

4.1.6 Compilation and Execution

The oracle in our approach consists of two kinds of code: that generated by the Test Oracle Generator (TOG), and the other kinds of classes, including Integer_Interval, InvertedTable, NormalTable and VectorTable, which are not generated by the TOG but are used by the TOG generated code. For more details about the above classes see section 4.2

The code below shows the implementation of the root class for the oracle (ggcdOracle.java) for the sample program specification that described in section 3.2.7. To see the whole generated classes from the example see appendix B

```
package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;
import static org.junit.Assert.*;

public class ggcdOracle{

  private VarMap vars;
  private Outggcd1 t0;

  public ggcdOracle(){

    vars=new VarMap();
```

```
    t0=new Outggcd1(vars);

  }

  private Boolean ggcdTOracle(Integer i,Integer j,
      Integer gcdvalue,Boolean result){

    Boolean resultOracle;

    vars.setValue("i",i);

    vars.setValue("j",j);

    vars.setValue("gcdvalue",gcdvalue);

    vars.setValue("result",result);

    resultOracle=t0.ggcdT1();

    return resultOracle;

  }
```

```
  public void assertggcdTOracle(Integer i,Integer j,
      Integer gcdvalue,Boolean result){

    assertTrue(ggcdTOracle(i,j,gcdvalue,result));

  }

}
```

Using the oracle involves implementing test code that calls the program under test and then calls the oracle procedures. In this work, the JUnit framework is used since it has a number of advantages. One important advantage of JUnit is that it is widely used, which will make it easier for others to understand the test cases and write new ones. In addition, it provides a graphical user interface (GUI) which makes it easier to write and test the program quickly and easily. JUnit shows test progress in a bar that is green if testing is going fine and it turns red when a test fails. This is makes it easy for the software developer to quickly identify failing test cases as they are found. The code below shows how to run the oracle generated from the sample program specification in 3.2.7 with JUnit:

```
package oracles;
import org.junit.Before;
import org.junit.Test;
```

```
public class OracleTest extends junit.framework.TestCase{

  ggcdOracle com;
  @Before
  public void setUp() throws Exception {
    com=new ggcdOracle();
  }

  @Test
  public void testCon(){

    Integer gc=GCD.gcd(25, 20);
    com.assertggcdTOracle(25, 20,gc , true);

  }

}
```

The previous code contains one test case to test that the program correctly finds the greatest common divisor of (25,20) which is 5. The greatest common divisor is computed by the static method GCD.gcd(int,int) meant to implement the specification. The user can add any number of test cases. The result for the previous code is shown in Figure 4.2.

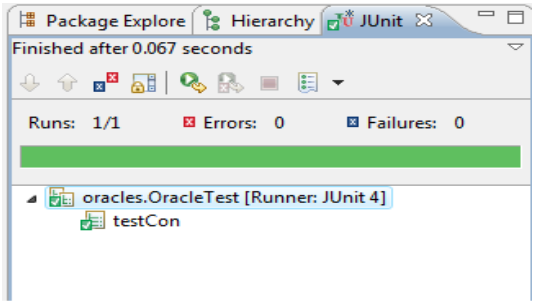

Figure 4.2: TestResult

4.2 Test Oracle Generator Design

4.2.1 Requirements

The requirements for the TOG are that using a specification written in the form discussed in chapter 3, it will output the executable test oracle code as described in section 4.1.

4.2.1.1 Assumptions

The oracle code generated by the TOG uses two kinds of object classes: Tabular expressions (Normal Table, Vector Table and Inverted Table) and Integer Interval implemented in NormalTable.java, VectorTable.java, InvertedTable.java and IntegerInterval.java. These table classes contain all knowledge of the semantics of tabular expressions and provide several methods (addHeaderCell, addMainCell, getMainCell, evaluateTable) which give the user the ability to create and evaluate the tabular expressions. The Integer Interval class is a java collection used to implement the finite set containing the integers in a specified range for the quantifications. These classes

are assumed to be correct.

4.2.1.2 User Interface

The Fillmore software specification editor leverages the eclipse plug-in architecture to create a software specification editor. A part of the Fillmore software project is to build a plug-in for eclipse to view and edit formal software specification documents. Eclipse is an open development platform that supports extension through a plug-in mechanism. The platform provides an advanced integrated development environment for software development, and a wide range of available plug-ins to support such tasks as testing, modeling and documentation. This plug-in is seen as a primary means for the users to interact with software specification documents. This plug-in is used as a user interface to the TOG, the plug-in is pictured in Figure 3.2. This interface gives the user ability to select any program specfication and generate the oracle from it. So, this is has the advantage that the user can interact easily with the specifications. In [41], they used a 'command line interface' for the oracle generator.

4.2.1.3 Input Format

The input to the TOG is in the form of a specification file which follows our specification model and contains information as described in Chapter 3. The file consists of a collection of theories and each of which consists of symbols and each symbol defines either a constant, variable, auxiliary function or program function.

4.2.1.4 Anticipated Changes

The items that are likely to change during the development of the TOG in the future:

- The format of the specification file. It is possible to add new elements to our specification file over time and change the existing elements.
- The programming language that used to implement the oracle. Currently, we are using Java to implement the oracle. It is possible in the future to use another language such as C++.
- The design of the oracle. For example: each cell in the tabular expressions is implemented as Java class. It is possible in the future to implement all cells in one class.
- The user interface that is used to interact with the specifications. We may add new features to the user interface such as giving the user the ability to view the tree representation of the mathematical expressions.

4.2.2 Package Design

The TOG is implemented as a set of packages, each of which contains a set of classes that encapsulate design decisions. Also, the packages can be divided into sub-packages which contain more specific design decisions. This approach has advantages that the design is easier to understand because of this separation of concerns, and it is easier to change the TOG since the decisions affected by the change are likely to be isolated.

To illustrate the system design, the class diagram is used. Figure 4.3 illustrates the package dependencies for the TOG.

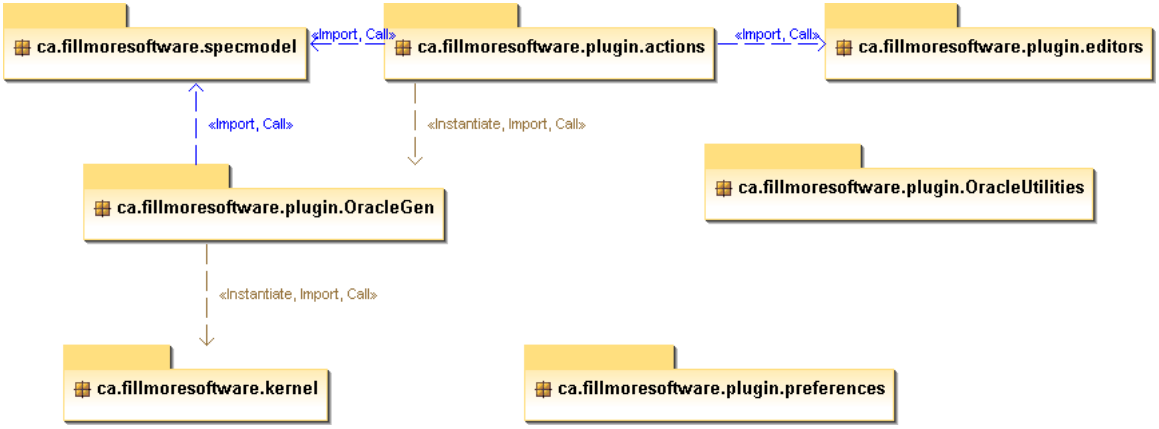

Figure 4.3: Packages Diagram

4.2.3 New Packages Added To Fillmore

The packages below are new packages written as part of this thesis work.

4.2.3.1 Oracle Generator Actions (ca.Fillmoresoftware.plugin.actions)

This package represents the main controlling package for the TOG. It contains the actions used to access the TOG (e.g. generate oracle and generate auxiliary function). It uses ca.Fillmoresoftware.plugin.editors to read the specification from the file, ca.Fillmoresoftware.plugin.specmodel to access the specification and ca.Fillmoresoftware.plugin.OracleGen to generate the oracles and auxiliary functions.

Figure 4.4 is the class diagram for the ca.Fillmoresoftware.plugin.actions showing the relationships between the classes. This package contains three classes (Gener-

ateAuxFunAction.java, GenerateOracleAction.java and OracleAction.java). The interface to these classes dictated by the eclipse plug-in interface.

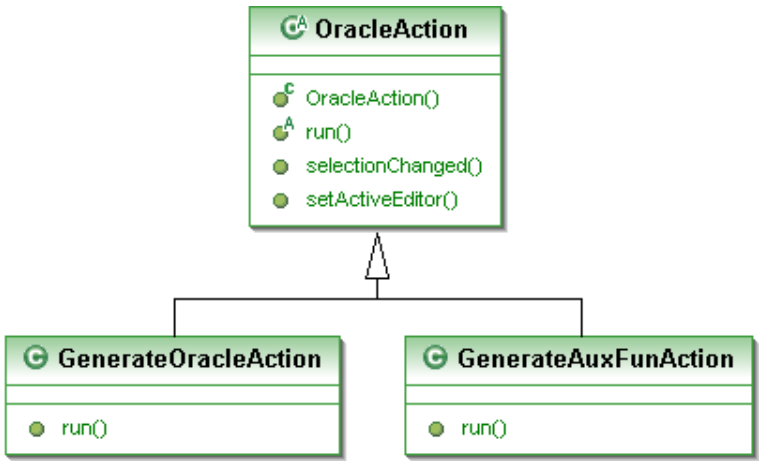

Figure 4.4: Actions Package Class Diagram

4.2.3.2 Oracle Generation (ca.Fillmoresoftware.plugin.OracleGen)

This package is responsible for converting the specification into the oracle implementation. It uses ca.Fillmoresoftware.plugin.specmodel and ca.Fillmoresoftware.plugin.kernel to access the tabular expressions. Figure 4.5 is the class diagram for the ca.Fillmoresoftware.plugin.OracleGen showing the relationships between the classes. This package contains eight classes (CodeFromOMobject.java, CodeFromOMA.java, CodeFromOMI.java, CodeFromOMS.java, CodeFromOMV.java, CodeFromTheory.java, CodeFromTabularExp.java and OracleModel.java). For more details about the responsibilities for the classes see appendix A.

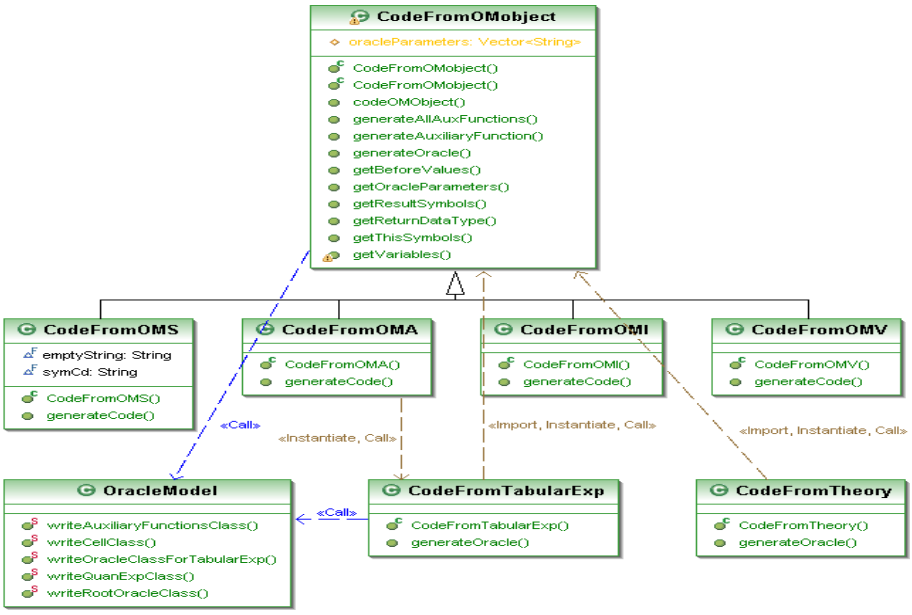

Figure 4.5: OracleGen Package Class Diagram

4.2.3.3 Oracle Utilities (ca.Fillmoresoftware.plugin.OracleUtilities)

This package provides classes that are necessary to run the test oracle. These classes are previously manually implemented and are used by the TOG generated code. These classes are CellBase.java, CellIndex.java, Integer_Interval.java, InvertedTable.java, NormalTable.java, TableGrid.java, VarMap.java and VectorTable.java. Figure 4.6 is the class diagram for the ca.Fillmoresoftware.plugin.OracleUtilities and shows the relationships between the classes.

4.2.4 Old Packages In Fillmore

These packages below came as part of Fillmore and are modified as part of this work.

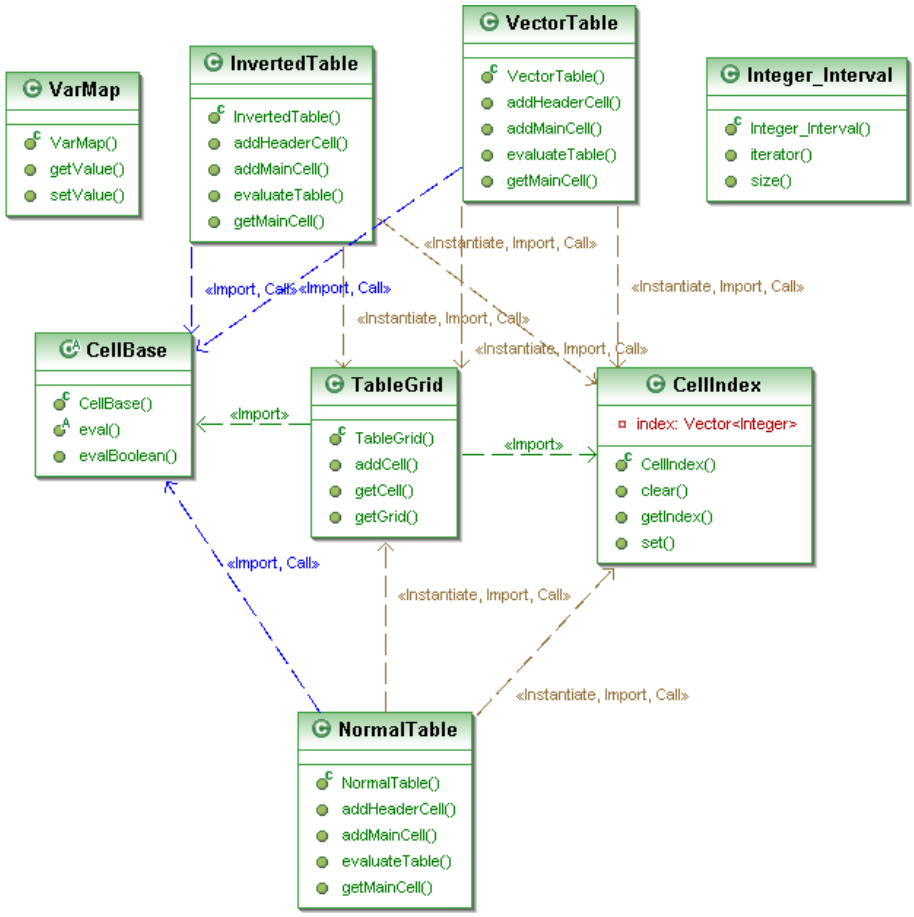

Figure 4.6: OracleUtilities Package Class Diagram

4.2.4.1 Specification Model (ca.Fillmoresoftware.plugin.specmodel)

This package used to construct our specification model that described in section 3.3.1 and provides classes that help us to access all parts of the specifications. These classes are ChangeNotifier.java, Definition.java, DOMXMLWriter.java, ElementTag.java, ISpecModelListener.java, MObject.java, OMDOMReader.java, Presentation.java, SpecModel.java, SpecModelElement.java, SpecModelErrorHandler.java, SpecModel-Parser.java, Symbol.java, Theory.java, TTSRole.java, Type.java and Use.java. This

package is modified to provide support for the “Presentation” and “Use” elements.

4.2.4.2 Kernel (ca.Fillmoresoftware.kernel)

This package used to construct our specifications for the tabular expressions and provides classes that help us to access all parts of the tabular expressions. These classes are EvalTerm.java, EvalTermFactory.java, GenRestFactor.java, Grid.java, Index.java, IndexFactory.java, InvertedEvalTerm.java, NormalEvalTerm.java, NormalGenRest.java, OMUtil.java, RectIndex.java, RectShape.java, RectShapeIterator.java, RectStructRest.java, Shape.java, ShapeFactory.java, StructRest.java, StructRestFactory.java, Symbol.java, Table.java, TableFactory.java and VectorEvalTerm.java. This package is modified to provide support for various kinds of tabular expressions (Normal, Vector and Inverted).

4.2.4.3 Editors (ca.Fillmoresoftware.plugin.editors)

This package used to implement “multi-page editor” to give the user ability to access and edit the specifications. This package consists of several classes (ElementDialog.java, SpecEditor.java, SpecEditorContributor.java, SpecElementLabelProvider.java, SpecErrorHandler.java, SpecOutlinePage.java and SpecTreeContentProvider.java). This package is modified to give the user the ability to view the names of the imported files (Content Dictionaries), which contain the java representation of the symbols.

4.2.4.4 Preferences (ca.Fillmoresoftware.plugin.preferences)

Eclipse also provides a Preferences APIs used to add plug-in specific preferences. This is a two step process:

- First the "org.eclipse.core.runtime.preferences" extension point is used to add a preference to initialize the plug-in. PreferenceInitializer class is contributed to initialize all the preferences when the plug-in is first initialized.
- Second the "org.eclipse.ui.preferencePages" extension point is used to add preference pages. It is important to note that the preference pages contributed must arrange themselves in a neat hierarchy to not interfere with other plug-ins. To accomplish this we add a base page name "Fillmore Preferences" and id "ca.Fillmoresoftware.plugin.preferences.FillmorePreferencePage"

All the preference pages must include the id mentioned in the second step as their category. Every preference page contributed through the extension point mechanism can include a category attribute. The category attribute basically includes the id path of the location of this preference page. For example the TOG preference page, which is contributed as a child to the Fillmore preference page, includes the id of the Fillmore preference page as it category attribute. The XML for this is shown below:

```
<extension
        point="org.eclipse.ui.preferencePages">
    <page
      class="ca.Fillmoresoftware.plugin.preferences.FillmorePreferencePage"
```

```
      id="ca.Fillmoresoftware.plugin.preferences.FillmorePreferencePage"
      name="Fillmore Preferences"/>
    <page
     category="ca.Fillmoresoftware.plugin.preferences.FillmorePreferencePage"
     class="ca.Fillmoresoftware.plugin.preferences.TestOraclePreferences"
     id="ca.Fillmoresoftware.plugin.preferences.testOracle"
     name="Test Oracle Generator"/>
   </extension>
   <extension
         point="org.eclipse.core.runtime.preferences">
      <initializer
 class="ca.Fillmoresoftware.plugin.preferences.PreferenceInitializer"/>
   </extension>
```

Also, this package includes these classes FillmorePreferencePage.java, PreferenceConstants.java, PreferenceInitializer.java and TestOraclePreferences.java which implement the preference pages. This package is modified to add preference pages for the Test Oracle Generator. These pages give the user the ability to specify: the path for the TOG output oracle code, the output package name and the imported libraries.

4.2.5 Symbols Representation

Functions and operators in OMDoc are encoded as "symbols", which are defined either in Content Dictionaries, for the standard functions and operators, or in the document itself, for functions that are particular to the given specification. The

OMDoc "presentation" element is used to define a representation of each symbol in Java so that the tool can translate expressions using these symbols into Java.

To be more general and cover most of symbols, we have used **presentation** and **use** elements for specifying the notation of symbols. OMDoc supplies a set of abbreviations that are sufficient for most presentation applications via the "use" elements that can occur as a children of "presentation" elements. Given the relevant information in the use elements, separate translation process generates the needed **Java Code** for the expression. The presentation element has the following attributes:

for specifies the name of symbol that is represented.

The use element has these set of attributes:

format specifies the name of the language that is used to represent the symbol.

lbrack/rbrack handle the brackets to be used in presentation for a symbol.

separator specifies the separator in the argument list of symbol.

fixity determines the placement of the symbol. This attribute can be one of the keywords **prefix**, **infix**, and **postfix**. For **prefix** it is placed in front of the arguments. For **infix** it is placed between the arguments. Finally, **postfix** it is placed behind the arguments.

4.2.5.1 Catagories of Symbols

Infix Symbols these symbols are placed between the arguments. For example: plus, minus, times, divide, eq, lt, gt, leq, geq, and, or, dot. A few examples of defining

symbols are necessary to illustrate the concept of defining the presentation for new symbols. So, these examples below illustrate how to represent the previous symbols.

Plus Symbol:

```
<presentation for="plus">
<use format="java" fixity="infix" lbrack="(" rbrack=")"> + </use>
</presentation>
```

If the children for this symbol were **a** and **b**. The Java Output Code will be: **(a+b)**

Minus Symbol:

```
<presentation for="minus">
<use format="java" fixity="infix" lbrack="(" rbrack=")"> - </use>
</presentation>
```

If the children for this symbol were **a** and **b**. The Java Output Code will be: **(a-b)**

All symbols in this category have the same values of the attributes in the "use" element but they are different in the value between the start and end tag of the "use" element. The table below shows the values of the "use" element for the rest of the previous symbols and the generated java code if the children of the symbols are **a** and **b**.

Table 4.2: Infix Symbols "Use" Values

Symbol	Value	Generated Code
Times	*	a*b
Divide	/	a/b
Equality	==	a==b
Less than	&lt;	a<b
Greater than	&gt;	a>b
Less than or equal	<=	a≤ b
Greater than or equal	>=	a≥ b
And	&&	a&&b
Or	‖	a‖b
Dot	.	a.b

Unary Symbols these symbols have one child and may be prefix or postfix. For example: not and predefined functions that have one child like: abs, sqrt and floor.

These examples below illustrate how to represent the previous symbols.

Not Symbol:

```
<presentation for="not">
<use format="java" fixity="prefix" lbrack="(" rbrack=")"> ! </use>
</presentation>
```

If the children for this symbol was **a**. The Java Output Code will be:

(!a)

All symbols in this category have the same values of the attributes in the "use" element but they are different in the value between the start and end tag of

the "use" element. Also, they are different in the value of fixity attribute. The symbols in the table below have no fixity. The table below shows the values of the "use" element for the rest of the previous symbols and the generated java code if the children of the symbols is **a**.

Table 4.3: Unary Symbols "Use" Values

Symbol	Value	Generated Code
Absolute	abs	abs(a)
Square Root	sqrt	sqrt(a)
Floor	floor	floor(a)

Function Symbols these symbols are functions that have more than one child. For example: any user defined function or predefined function.

This example below illustrate how to represent the previous symbols.

Power Function Symbol:

```
<presentation for="pow">
<use format="java" lbrack="(" rbrack=")" separator = ","> pow
</use>
</presentation>
```

If the children for this symbol were **a** and **b**. The Java Output Code will be: **(pow(a,b))**

Irregular Symbols these symbols use combined fixity. So, the fixity attribute is not defined. For example: array_get and dot symbols. These examples below illustrate how to represent the previous symbols.

Array_get Symbol:

```
<presentation for = "array_get">
<use format = "java" lbrack = "[" rbrack = "]"/>
</presentation>
```

If this symbol has two children and they were **A** and **i**. The Java Output Code will be:

(A[i])

If this symbol has three children and they were **A, i** and **j**. The Java Output Code will be:

(A[i][j])

Dot Symbol:

```
<presentation for = "bar">
<use format = "java" lbrack = "(" rbrack = ")" separator = ","> .
</use>
</presentation>
```

If this symbol has two children and they were **a** and **b**. The Java Output Code will be:

(a.bar(b))

If this symbol has more than two children and they were **a, b, c** and **d**. The Java Output Code will be:

(a.bar(b,c,d))

If there is no presentation or use for the symbol then it is assumed to be a function so a normal function call is generated. For example, if the symbol is **bar** and the children are **a** and **b**. Then the generated code will be:

bar(a,b)

4.2.6 Algorithem Overview

The alogrithm that we have used for generating test oracles is the same for the one which is used in [41] and consists of the following steps:

1. Initialization: open files, initialize data structures.
2. Read specification from file.
3. Create oracle program contexts.
4. Code Auxiliary Definitions: Create a Java function for each, code the expression.
5. Code the oracle.
6. Write and close files.
7. Free data structures.

4.2.6.1 Expression Coding

The mathmematical expressions used in the specfications or in auxiliary definitions are translated into code in the following manner: The expression syntax tree is tra-

versed using a depth-first traversal and each sub-expression is implemented in turn as described in section 4.1.2.1. The code that gives the value of each sub-expression is written into a buffer which is used to construct the code for the 'parent' expression. This process continues until the root expression has been implemented and the resulting code is used as the body of the procedure in the oracle.

Chapter 5

Test Driven Development With Oracles

This chapter describes our new approach for TDD. It also describes examples which show how to apply this approach.

5.1 Test Driven Development with Oracles

This section introduces an alternative approach to TDD that is to develop the specification of the required behaviour in a formal notation as a part of the TDD process and to generate test oracles from that specification.

The process looks like this:

- Write the specification for some required behaviour.
- Generate the test oracle from the specification and select test inputs.

- Run the program under test in the test framework (e.g., JUnit) using the test oracle to verify if it passes or fails.
- If the test fails, write code until this test passes.
- If the test passes and the specification is not completed yet, add to or refine the specification and redo the process again.
- keep doing this process until the specification is complete.

The completeness in our work is determined by the designer. Using these tools to do analysis of the test cases (e.g., coverage of the specification) is beyond this work. So, this is could be done in the future.

The steps of TDD approach are illustrated in the flowchart in Figure 5.1.

TDD approach is applicable for methods and classes. This approach focuses on deriving test oracles from the module internal design document [37] for methods and module interface specification [42] for classes.

5.1.1 Test Driven Development For Methods

The illustration of TDD provided in [12, 29], in which a program is developed to convert decimal numbers into their roman numeral equivalent, serves as a good, although somewhat simplistic, illustration of this method.

The following example shows the whole process for specification supported TDD. According to the TDD approach, the first step is to write a specification for some required behaviour. So, starting with this specification:

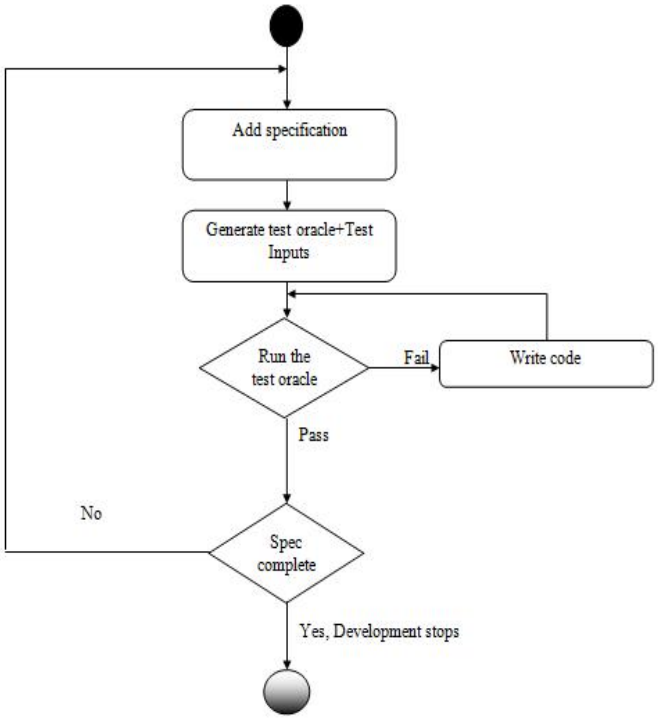

Figure 5.1: The Steps of TDD Approach

String `dToR`(**Integer** `i`)

$\stackrel{\text{df}}{=}$

	$\texttt{i} \geq 1 \wedge \texttt{i} < 4$
`result` =	subDToR(`i`)

String subDToR(**Integer** i)

$\stackrel{df}{=}$

i = 3	"III"
i = 2	"II"
i = 1	"I"

The above specification consists of two parts: the first part is the definition for `dToR(i)` function which is the program function, the second part is the definition for subDToR(i) function which is an auxiliary function. In program function definitions, we use the convention that `result` represents the value returned by the function. The required behaviour that is represented by this specification is to support the conversion of numbers (1–3) into their corresponding roman numerals (I, II, III).

After writing the specifications, generate the test oracle from it and run the test oracle to make sure that the program behaviour is consistent with the required behaviour. Following the TDD approach, the test cases should initially fail since the program isn't yet implemented. Then implement enough of the program to make the cases pass.

The previous specification only specifies a behaviour for numbers in the range 1–3, so if a test case outside that range is used then the test oracle will give an error that says "NoSuchElementException". Figure 5.2 shows that error.

The pattern used in the previous specification (i.e., explicitly specifying the corresponding roman numeral representation for each decimal number) is clearly not practical for a very broad range of inputs. The previous specification can be re-written, as follows (where "+" on **String**s is used to represent concatenation):

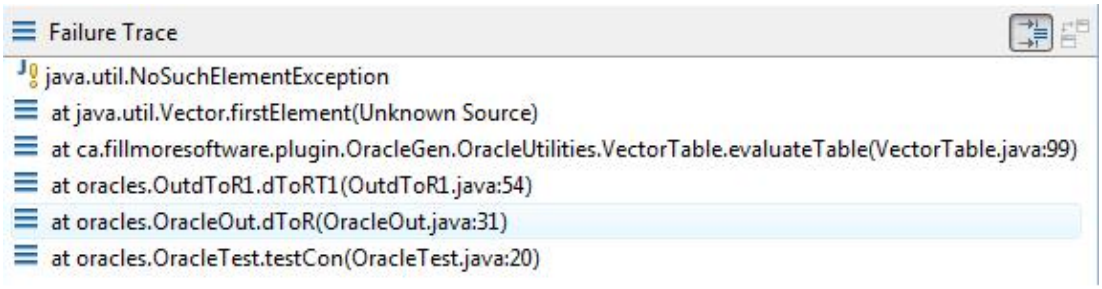

Figure 5.2: NoSuchElementException

String `dToR`(**Integer** `i`)

$\stackrel{df}{=}$

	$\texttt{i} \geq 1 \wedge \texttt{i} < 4$
`result` =	subDToR(`i`)

String subDToR(**Integer** i)

$\stackrel{df}{=}$

$\texttt{i} > 0 \wedge \texttt{i} < 4$	"I" + subDToR(i − 1)
$\texttt{i} = 0$	""

Then the domain of the previous specification can be broaden as follows:

String `dToR`(**Integer** `i`)

$\stackrel{df}{=}$

	$\texttt{i} \geq 1 \wedge \texttt{i} < 5$	$\texttt{i} \geq 5 \vee \texttt{i} < 1$
`result` =	subDToR(`i`)	"`NA`"

String subDToR(**Integer** i)

$\stackrel{df}{=}$

i = 4	"IV"
i > 0 ∧ i < 4	"I" + subDToR(i − 1)
i = 0	""

The previous specification defines the conversion of numbers from (1–4) into their corresponding roman numerals (I, II, III, IV) and handles the error where subDToR is not defined by specifying the behaviour for those inputs. After refining the initial specification, do the same steps as we did in the previous one. Again refine the implementation until the behaviour is consistent with the specification, then continue to revise the specification, as follows.

String `dToR`(**Integer** `i`)

$\stackrel{df}{=}$

	`i` ≥ 1 ∧ `i` < 9	`i` ≥ 9 ∨ `i` < 1
`result` =	subDToR(`i`)	"`NA`"

String subDToR(**Integer** i)

$\stackrel{df}{=}$

i ≥ 5 ∧ i < 9	"V" + subDToR(i − 5)
i = 4	"IV"
i > 0 ∧ i < 4	"I" + subDToR(i − 1)
i = 0	""

The specification defines behaviour for the conversion of numbers from (1–8) into their corresponding roman numerals (I, II, III, IV, V, VI, VII, VIII). We do the same

steps as before and after that, we continue to revise the specification, as follows.

String `dToR`(**Integer** `i`)

$\stackrel{\text{df}}{=}$

	$\texttt{i} \geq 1 \wedge \texttt{i} < 10$	$\texttt{i} \geq 10 \vee \texttt{i} < 1$
`result` =	subDToR(`i`)	"`NA`"

String subDToR(**Integer** i)

$\stackrel{\text{df}}{=}$

$\text{i} = 9$	"IX"
$\text{i} \geq 5 \wedge \text{i} < 9$	"V" + subDToR(i − 5)
$\text{i} = 4$	"IV"
$\text{i} > 0 \wedge \text{i} < 4$	"I" + subDToR(i − 1)
$\text{i} = 0$	""

Now, the specification defines the conversion of numbers from (1–9) into their corresponding roman numerals (I, II, III, IV, V, VI, VII, VIII, IX). So, in every step we revise the specification to describe new behaviour and the specification is represented in a formal way. Also, if the tests fail after we revise the specification we have to write some code to satisfy the specification, and after that we continue to revise the specification.

We keep doing this process until the specification is complete and the code behaviour is consistent with the required behaviour that is described by the specification. After we have done several steps using TDD approach to develop the specification and code together, the complete specification is as follows.

String `dToR`(**Integer** `i`)

$\stackrel{\text{df}}{=}$

	`i` $\geq 1 \wedge$ `i` ≤ 3999	`i` $> 3999 \vee$ `i` < 1
`result` =	subDToR(`i`)	"`NA`"

String subDToR(**Integer** i)

$\stackrel{\text{df}}{=}$

$i \geq 1000$	"M" + subDToR(i − 1000)
$i \geq 900 \wedge i < 1000$	"CM" + subDToR(i − 900)
$i \geq 500 \wedge i < 900$	"D" + subDToR(i − 500)
$i \geq 400 \wedge i < 500$	"CD" + subDToR(i − 400)
$i \geq 100 \wedge i < 400$	"C" + subDToR(i − 100)
$i \geq 90 \wedge i < 100$	"XC" + subDToR(i − 90)
$i \geq 50 \wedge i < 90$	"L" + subDToR(i − 50)
$i \geq 40 \wedge i < 50$	"XL" + subDToR(i − 40)
$i \geq 10 \wedge i < 40$	"X" + subDToR(i − 10)
$i = 9$	"IX"
$i \geq 5 \wedge i < 9$	"V" + subDToR(i − 5)
$i = 4$	"IV"
$i > 0 \wedge i < 4$	"I" + subDToR(i − 1)
$i = 0$	""

Now, we have a complete specification that describes the whole required behaviour for the program, and presumably the working implementation developed along with

it using TDD. So, using this TDD approach results in a complete specification, implementation and suite of test cases for the program.

5.1.2 Test Driven Development For Classes

We now consider applying our approach to modules or classes that have an internal data structure and methods for accessing or modifying the values of that data structure. As an illustrative example we use the bounded stack as developed in [28].

As before, the first step in our approach is to specify some required behaviour, in this case for creation of an empty stack:

Data Stucture

Integer `s[ ]`

Integer `maxSize`

Integer `length`

Program Functions

Stack `stack`(**Integer** `x`)

$\stackrel{\text{df}}{=}$ (`result.isEmpty`() $\wedge$ `result.maxDepth`() = `x`)

Boolean `isEmpty`()

$\stackrel{\text{df}}{=}$ `result` = (`length` = 0)

Integer `maxDepth`()

$\stackrel{\text{df}}{=}$ `result` = `maxSize`

The specification consists of the data structure description, the definition for `stack(x)` function, which is the program function specifying the behaviour of the constructor and two program functions specifying the behaviour of the methods `isEmpty()` and `maxDepth()`.

After we write the specification, we generate the test oracle from it and write the test code to call it (e.g., using JUnit). The test case will, of course, fail, so we should implement the constructor and methods so that the test cases pass and we have a program that is consistent with the specified behaviour.

We then modify the specification for `push` to cover the case where the stack is initially empty, and add two more methods:

void `push`(**Integer** `x`)

$\stackrel{\text{df}}{=}$

	`p_this.size()` $= 0$
`this.size()` $=$	`p_this.size()` $+ 1$
`this`|	`this.lastElement()` $=$ `x`

Integer `size`()

$\stackrel{\text{df}}{=}$ `result` $=$ `length`

Integer `lastElement`()

$\stackrel{\text{df}}{=}$ `result` $=$ `s`[`length` $- 1$]

Here we use the naming convention of prepending "`p_`" to a program variable name (e.g., `p_this`) to represent the value of the program variable (e.g., `this`) in the state immediately before the function was executed. The new behaviour defined by the

specification is to push an object on an empty stack. After the push the stack should contain that element and the size for the stack after is increased by one. Again we generate the test oracle and implement a test case, which will initially fail. The stack code is then developed until the test case passes, and so it implements the specified behaviour.

As we see the previous specification only defines pushing on an empty stack, which is clearly not complete. We need to modify the specification to define behaviour for pushing on a non-full stack:

void `push`(**Integer** `x`)

$\stackrel{\text{df}}{=}$

	$\texttt{p\_this.size}() \geq 0 \wedge \texttt{p\_this.size}() < \texttt{this.maxDepth}()$
$\texttt{this.size}() =$	$\texttt{p\_this.size}() + 1$
$\texttt{this}\vert$	$\forall i : [0, \texttt{p\_this.size}() - 1].($ $\texttt{this.elementAt}(i) = \texttt{p\_this.elementAt}(i)) \wedge$ $(\texttt{this.lastElement}() = \texttt{x})$

Integer `elementAt`(**Integer** `i`)

$\stackrel{\text{df}}{=} \texttt{result} = \texttt{s}[\texttt{i}]$

Again we generate the test oracle and implement test cases, this time to push a few elements onto the stack. After modifying the implementation to make it pass the tests, we then modify the specification to cover the case where the stack is full:

void push(**Integer** x)

$\stackrel{df}{=}$

	$\texttt{p\_this.size}() \geq 0 \wedge \texttt{p\_this.size}() < \texttt{this.maxDepth}()$	$\texttt{p\_this.size}() = \texttt{this.maxDepth}()$
$\texttt{this.size}() =$	$\texttt{p\_this.size}() + 1$	$\texttt{p\_this.size}()$
$\texttt{this}\|$	$\forall i : [0, \texttt{p\_this.size}() - 1].$ $\left(\begin{array}{l}\texttt{this.elementAt}(i) = \\ \texttt{p\_this.elementAt}(i)\end{array}\right) \wedge$ $(\texttt{this.lastElement}() = \texttt{x})$	$\texttt{this} = \texttt{p\_this}$

The new behaviour supported by this specification is to attempt to push an object on a full stack. The requirement is that the stack after the call returns is the same size and contains the same elements as the stack before the call. The new test case should check this behaviour by attempting to push on a full stack. Continuing the development we add the specification for pop on a non-empty stack:

Integer pop()

$\stackrel{df}{=}$

	$\texttt{p\_this.size}() \geq 1$
$\texttt{this.size}() =$	$\texttt{p\_this.size}() - 1$
$\texttt{this}\|$	$\forall i : [0, \texttt{this.size}() - 1]. \left(\begin{array}{l}\texttt{this.elementAt}(i) = \\ \texttt{p\_this.elementAt}(i)\end{array}\right) \wedge$ $(\texttt{result} = \texttt{p\_this.lastElement}())$

Continuing in this manner, we eventually reach the full specification of the bounded stack, as below, and we have at the same time developed a full imple-

mentation and a full suite of test cases.

Data Stucture

Integer s[ ]

Integer maxSize

Integer length

Program Functions

Stack stack(**Integer** x)

$\stackrel{\text{df}}{=}$ (result.isEmpty() $\wedge$ result.maxDepth() = x)

void push(**Integer** x)

$\stackrel{\text{df}}{=}$

	p_this.size() $\geq 0 \wedge$ p_this.size() $<$ this.maxDepth()	p_this.size() = this.maxDepth()
this.size() =	p_this.size() + 1	p_this.size()
this|	$\forall i : [0, \text{p\_this.size}() - 1].$ $\left(\begin{array}{l}\text{this.elementAt}(i) = \\ \text{p\_this.elementAt}(i)\end{array}\right) \wedge$ (this.lastElement() = x)	this = p_this

Integer pop()

$\stackrel{\text{df}}{=}$

	$\texttt{p\_this.size}() \geq 1$
$\texttt{this.size}() =$	$\texttt{p\_this.size}() - 1$
$\texttt{this}\vert$	$\forall i : [0, \texttt{this.size}() - 1]. \begin{pmatrix} \texttt{this.elementAt}(i) = \\ \texttt{p\_this.elementAt}(i) \end{pmatrix} \wedge$ $(\texttt{result} = \texttt{p\_this.lastElement}())$

Integer top()

$\stackrel{\text{df}}{=} \texttt{result} = \texttt{this.lastElement}()$

Boolean isEmpty()

$\stackrel{\text{df}}{=} \texttt{result} = (\texttt{length} = 0)$

Integer maxDepth()

$\stackrel{\text{df}}{=} \texttt{result} = \texttt{maxSize}$

Integer size()

$\stackrel{\text{df}}{=} \texttt{result} = \texttt{length}$

Integer lastElement()

$\stackrel{\text{df}}{=} \texttt{result} = \texttt{s}[\texttt{length} - 1]$

Integer elementAt(**Integer** i)

$\stackrel{\text{df}}{=} \texttt{result} = \texttt{s}[\texttt{i}]$

Chapter 6

Future Work and Conclusion

6.1 Future Work

Clearly a next step in this research and tool development will be to support test case generation from the specification as well, which will further reduce the amount of 'manual' test code development effort.

Also, applying the techniques to real problems in a real-world development environment will undoubtedly provide some insight and help to refine the techniques. Other possible improvements in the tool set (e.g., better visual editing etc.) could be done in the future development of these tools. In addition to that using these tools to do analysis of the test cases (e.g., coverage of the specification).

6.2 Conclusions

In test driven development, tests are used to specify the behaviour of the program, and the tests are additionally used as documentation of the program. However, tests are not sufficient to completely define the behaviour of a program because they only define the program behaviour by example and do not state general properties. So, the latter can be achieved by using our TDD approach, which uses a formal specification to specify the behaviour of the program and supports testing directly against that specification by generating oracles. The outcome of this technique is that, at the end of the development period, the developer has produced not only a working implementation, but also a complete specification and a full set of test cases.

Appendix A

Class Responsibility Collaborator (CRC)

The UML diagrams for the packages in the system are described in chapter 4.

A.1 Class Responsibility Collaborator (CRC) Tables

Table A.1: GenerateAuxFunAction Class Responsibility Collaborator (CRC)

GenerateAuxFunAction	
Generates the code for auxiliary functions	SpecModel SpecModelElement Symbol Definition CodeFromOMobject

Table A.2: GenerateOracleAction Class Responsibility Collaborator (CRC)

GenerateOracleAction	
Generates the code for oracles	SpecModel SpecModelElement Symbol Definition CodeFromOMobject

Table A.3: OracleAction Class Responsibility Collaborator (CRC)

GenerateOracleAction	
Abstract base class for all test oracle actions	SpecEditor

Table A.4: CodeFromOMobject Class Responsibility Collaborator (CRC)

CodeFromOMobject	
Generates the code from the open math objects Generates the context for the test oracle and auxiliary functions	SpecModel Table CodeFromTabularExp Definition

Table A.5: CodeFromOMS Class Responsibility Collaborator (CRC)

CodeFromOMS	
Generates the code from the open math symbol object	SpecModel Definition

Table A.6: CodeFromOMA Class Responsibility Collaborator (CRC)

CodeFromOMA	
Generates the code from the open math application object	SpecModel Definition CodeFromTabularExp

Table A.7: CodeFromOMI Class Responsibility Collaborator (CRC)

CodeFromOMI	
Generates the code from the open math integer object	

Table A.8: CodeFromOMV Class Responsibility Collaborator (CRC)

CodeFromOMV	
Generates the code from the open math variable object	

Table A.9: CodeFromTabularExp Class Responsibility Collaborator (CRC)

CodeFromOMobject	
Generates the code from the tabular expressions	SpecModel Table CodeFromOMobject Definition StructRest EvalTerm

Table A.10: CodeFromTheory Class Responsibility Collaborator (CRC)

CodeFromTheory	
Generates the code from the theory	CodeFromOMobject SpecModel Theory

Table A.11: OracleModel Class Responsibility Collaborator (CRC)

OracleModel	
Write all required files for the oracle	

Table A.12: CellBase Class Responsibility Collaborator (CRC)

CellBase	
Represents the Cell Base that used to implement the tabular expressions	

Table A.13: CellIndex Class Responsibility Collaborator (CRC)

CellIndex	
Represents the index for the cell	

Table A.14: Integer_Interval Class Responsibility Collaborator (CRC)

Integer_Interval	
Represents the interval for quantifiers expressions	

Table A.15: InvertedTable Class Responsibility Collaborator (CRC)

InvertedTable	
Implements the inverted table and encapsulates all semantics knowledge about the inverted table	TableGrid

Table A.16: NormalTable Class Responsibility Collaborator (CRC)

NormalTable	
Implements the normal table and encapsulates all semantics knowledge about the normal table	TableGrid

Table A.17: VectorTable Class Responsibility Collaborator (CRC)

VectorTable	
Implements the vector table and encapsulates all semantics knowledge about the vector table	TableGrid

Table A.18: TableGrid Class Responsibility Collaborator (CRC)

TableGrid	
Represents table grid that used to implement tabular expressions	CellIndex CellBase

Table A.19: VarMap Class Responsibility Collaborator (CRC)

VarMap	
Represents the values for variables that used in the specification	

Table A.20: SpecModel Class Responsibility Collaborator (CRC)

SpecModel	
Top level for the specification model The model provides an abstract API for accessing the content of a software specification	SpecModelElement ErrorHandler ChangeNotifier

Table A.21: SpecModelElement Class Responsibility Collaborator (CRC)

SpecModelElement	
Base class for all elements in a specification	SpecModel ElementTag

Table A.22: SpecModelErrorHandler Class Responsibility Collaborator (CRC)

SpecModelErrorHandler	
Handle the errors	ErrorHandler

Table A.23: SpecModelParser Class Responsibility Collaborator (CRC)

SpecModelParser	
A parser for specification models It knows the details about how to validate the specification files against the Relax NG schema To avoid unnecessary re-reading of the DTD and schema files and to conserve memory this is a singleton class	DocumentBuilder SpecModelErrorHandler

Table A.24: ISpecModelListener Class Responsibility Collaborator (CRC)

ISpecModelListener
Interface for listeners for changes to the specification model Classes that want to be notified of changes to the specification model should implement this interface and register themselves via { link SpecModel#addListener(ISpecModelListener)}

Table A.25: ChangeNotifier Class Responsibility Collaborator (CRC)

ChangeNotifier	
Manage change notification to ISpecModelListeners	ISpecModelListener ChangeNotification

Table A.26: DOMXMLWriter Class Responsibility Collaborator (CRC)

DOMXMLWriter
Convert DOM to XML This class is based almost entirely on XMLtoTree

Table A.27: OMDOMReader Class Responsibility Collaborator (CRC)

OMDOMReader
An OpenMath DOM reader

Table A.28: ElementTag Class Responsibility Collaborator (CRC)

ElementTag
The possible kinds of elements in a specification (Definition, Symbol, Theory, Type, MObject, Presentation, Use)

Table A.29: Theory Class Responsibility Collaborator (CRC)

Theory	
Represent an omdoc theory	Symbol Presentation

Table A.30: Symbol Class Responsibility Collaborator (CRC)

Symbol	
Represents a symbol declaration and definition which is the main building block of a specification	TTSRole definition SpecModelElement

Table A.31: TTSRole Class Responsibility Collaborator (CRC)

TTSRole	
The possible values for the tts:role attribute These classify a definition by the role the defined symbol plays in a specification	

Table A.32: Type Class Responsibility Collaborator (CRC)

Type	
A representation of an omdoc element as specialized for software specifications	MObject

Table A.33: Definition Class Responsibility Collaborator (CRC)

Definition	
A representation of an omdoc element as specialized for software specifications and contains an open math expression that defines the semantic meaning for the symbol	MObject

Table A.34: Presentation Class Responsibility Collaborator (CRC)

Presentation	
Represents symbol presentation	Use

Table A.35: Use Class Responsibility Collaborator (CRC)

Use	
Represents the format for symbol	

Table A.36: MObject Class Responsibility Collaborator (CRC)

MObject	
A representation of an omdoc math object	

Table A.37: Table Class Responsibility Collaborator (CRC)

Table	
A representation for the tabular expression	Grid
A tabular expression consists of :	EvalTerm
An evaluation term	GenRest
A structural restriction expression, the value	StructRest
of which must be independent of the value of the	
expressions in the table	
A general restriction expression, the value of which	
may depend on the value of the expressions in the table	
A sequence of grids, each of which is an indexed set of expressions	

Table A.38: TableFactory Class Responsibility Collaborator (CRC)

TableFactory	
Construct a rectangular table(normal, vector and inverted)	Table

Table A.39: EvalTerm Class Responsibility Collaborator (CRC)

EvalTerm	
Interface for the evaluation term	

Table A.40: EvalTermFactory Class Responsibility Collaborator (CRC)

EvalTermFactory	
Constructs the evaluation term for the tabular expressions	EvalTerm

Table A.41: GenRest Class Responsibility Collaborator (CRC)

GenRest	
Interface for the general restriction	

Table A.42: GenRestFactory Class Responsibility Collaborator (CRC)

GenRestFactory	
Constructs the general restriction term for the tabular expressions	

Table A.43: Grid Class Responsibility Collaborator (CRC)

Grid	
Represents a grid which has a shape (index set) and corresponding expressions represented by OMObject	Shape OMObject Index

Table A.44: Index Class Responsibility Collaborator (CRC)

Index	
Interface for the cell index	

Table A.45: IndexFactory Class Responsibility Collaborator (CRC)

IndexFactory	
A factory class for generating shapes	

Table A.46: InvertedEvalTerm Class Responsibility Collaborator (CRC)

InvertedEvalTerm	
Represents the inverted table evaluation term	

Table A.47: NormalEvalTerm Class Responsibility Collaborator (CRC)

NormalEvalTerm	
Represents the normal table evaluation term	

Table A.48: VectorEvalTerm Class Responsibility Collaborator (CRC)

VectorEvalTerm	
Represents the vector table evaluation term	

Table A.49: NormalGenRest Class Responsibility Collaborator (CRC)

NormalGenRest
Represents general restriction for the normal table

Table A.50: OMUtil Class Responsibility Collaborator (CRC)

OMUtil
Provides open math utlities

Table A.51: RectIndex Class Responsibility Collaborator (CRC)

RectIndex
Selects a particular cell within a grid

Table A.52: RectShape Class Responsibility Collaborator (CRC)

RectShape
Describes the index set for a rectangular grid

Table A.53: RectShapeIterator Class Responsibility Collaborator (CRC)

RectShapeIterator	
An Iterator to iterate over a RectShape RectShapeIterator's can be used to iterate over any grid that has a shape of type RectShape	RectShape RectIndex

Table A.54: RectStructRest Class Responsibility Collaborator (CRC)

RectStructRest
Represents the rectangular structure restriction for the tabular expressions

Table A.55: Shape Class Responsibility Collaborator (CRC)

Shape
An Interface for shape objects A Shape describes the index set for a grid

Table A.56: ShapeFactory Class Responsibility Collaborator (CRC)

ShapeFactory	
A factory class for generating shapes	

Table A.57: StructRest Class Responsibility Collaborator (CRC)

StructRest	
Interface for the structure restriction	

Table A.58: StructRestFactory Class Responsibility Collaborator (CRC)

StructRestFactory	
Constructs the structure restriction for the tabular expressions	

Table A.59: ElementDialog Class Responsibility Collaborator (CRC)

ElementDialog	
Specify parts of the specifications	SpecModel

Table A.60: ISpecModelSelectable Class Responsibility Collaborator (CRC)

ISpecModelSelectable	
This interface must be implemented by pages of the Spec Editor which need to be notified of changes to the selected element	

Table A.61: SpecEditor Class Responsibility Collaborator (CRC)

SpecEditor	
A multipage editor with the following pages: XML Editor	XMLEditor SFormEditor SpecModel SpecModelElement SpecOutlinePage

Table A.62: SpecEditorContributor Class Responsibility Collaborator (CRC)

SpecEditorContributor	
Manages the *installation/deinstallation* of global actions for multi-page editors Responsible for the redirection of global actions to the active editor Multi-page contributor replaces the contributors for the individual editors in the multi-page editor	

Table A.63: SpecElementLabelProvider Class Responsibility Collaborator (CRC)

SpecElementLabelProvider	
Provides labels for the specification elements	

Table A.64: SpecErrorHandler Class Responsibility Collaborator (CRC)

SpecErrorHandler	
Handle the errors	

Table A.65: SpecOutlinePage Class Responsibility Collaborator (CRC)

SpecOutlinePage	
Constructs the outline specifications	SpecEditor

Table A.66: SpecTreeContentProvider Class Responsibility Collaborator (CRC)

SpecTreeContentProvider	
Adaptor for the SpecModel to ITreeContentProvider	SpecModel

Table A.67: FillmorePreferencePage Class Responsibility Collaborator (CRC)

FillmorePreferencePage	
Blank Preference page to properly organize all the preference page for the plugin	

Table A.68: PreferenceConstants Class Responsibility Collaborator (CRC)

PreferenceConstants	
Constant definitions for plug-in preferences	

Table A.69: PreferenceInitializer Class Responsibility Collaborator (CRC)

PreferenceInitializer	
Used to initialize default preference values	

Table A.70: TestOraclePreferences Class Responsibility Collaborator (CRC)

TestOraclePreferences	
Test oracle preferences page	StringFieldEditor

Appendix B

The Generated Oracle Code

B.1 The Generated Oracle Code From The Sample Example

This section shows the classes generated from the sample '`ggcd`' program specification given in 3.2.7

```
package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;
import static org.junit.Assert.*;

public class ggcdOracle{

  private VarMap vars;
```

```
private Outggcd1 t0;

public ggcdOracle(){

  vars=new VarMap();

  t0=new Outggcd1(vars);

}

private Boolean ggcdTOracle(Integer i,Integer j,
    Integer gcdvalue,Boolean result){

  Boolean resultOracle;

  vars.setValue("i",i);

  vars.setValue("j",j);

  vars.setValue("gcdvalue",gcdvalue);

  vars.setValue("result",result);
```

```
    resultOracle=t0.ggcdT1();

    return resultOracle;

  }

  public void assertggcdTOracle(Integer i,Integer j,
      Integer gcdvalue,Boolean result){

    assertTrue(ggcdTOracle(i,j,gcdvalue,result));

  }
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class Outggcd1{

  private VarMap vars;
  private VectorTable nTable;
```

```
public Outggcd1(VarMap vars){

  this.vars=vars;

  nTable=new VectorTable(3);

  CellIndex inHeader1[]=new CellIndex[2];

  CellIndex inHeader2[]=new CellIndex[2];

  for(int k0=0;k0<2;k0++)
    inHeader1[k0]=new CellIndex(1);

  for(int k1=0;k1<2;k1++)
    inHeader2[k1]=new CellIndex(1);

  CellIndex inMain[]=new CellIndex[4];

  for(int j=0;j<4;j++)
    inMain[j]=new CellIndex(2);

  inHeader1[0].set(0,0);
```

```
nTable.addHeaderCell(0,inHeader1[0],new
ggcd1_Grid_1_Cell_0(vars));
inHeader1[1].set(0,1);
nTable.addHeaderCell(0,inHeader1[1],new
ggcd1_Grid_1_Cell_1(vars));
inHeader2[0].set(0,0);
nTable.addHeaderCell(1,inHeader2[0],new
ggcd1_Grid_2_Cell_0(vars));
inHeader2[1].set(0,1);
nTable.addHeaderCell(1,inHeader2[1],new
ggcd1_Grid_2_Cell_1(vars));

int index=0;

for(int l0=0;l0<2;l0++)
  for(int l1=0;l1<2;l1++)
  {
    inMain[index].set(0,l0);
    inMain[index].set(1,l1);
    index++;
  }
nTable.addMainCell(inMain[0],new ggcd1_Grid_0_Cell_0(vars));
```

```
    nTable.addMainCell(inMain[1],new ggcd1_Grid_0_Cell_1(vars));
    nTable.addMainCell(inMain[2],new ggcd1_Grid_0_Cell_2(vars));
    nTable.addMainCell(inMain[3],new ggcd1_Grid_0_Cell_3(vars));

  }

  public Boolean ggcdT1(){

    Boolean result=nTable.evaluateTable();

    return result;

  }
}

package oracles;
import ca.Fillmoresoftware.plugin.OracleUtilities.*;

public class AuxFunctions{

  static public Boolean cDiv(Integer  a,Integer  b,Integer
x){
```

```
    return (a % x == 0) && (b % x == 0);

  }
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_0_Cell_0 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_0_Cell_0(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Integer  i=(Integer)vars.getValue("i");
    Integer  j=(Integer)vars.getValue("j");
    Integer  x=(Integer)vars.getValue("x");
```

```
    Integer  gcdvalue=(Integer)vars.getValue("gcdvalue");

    return (gcdvalue==AuxFunctions.cDiv(i,j,x));

  }
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_0_Cell_1 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_0_Cell_1(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Integer  gcdvalue=(Integer)vars.getValue("gcdvalue");
```

```
    return (gcdvalue==0);

  }
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_0_Cell_2 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_0_Cell_2(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Boolean  result=(Boolean)vars.getValue("result");
```

```
    return (result==true);

  }
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_0_Cell_3 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_0_Cell_3(VarMap vars){

    this.vars=vars;
  }

  public Object eval(){

    Boolean  result=(Boolean)vars.getValue("result");

    return (result==false);
```

```
  }
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_1_Cell_0 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_1_Cell_0(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Integer  gcdvalue=(Integer)vars.getValue("gcdvalue");

    return gcdvalue;

  }
```

```
}

package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_1_Cell_1 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_1_Cell_1(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Boolean  result=(Boolean)vars.getValue("result");

    return result;
  }
}
```

```
package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_2_Cell_0 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_2_Cell_0(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Integer  i=(Integer)vars.getValue("i");
    Integer  j=(Integer)vars.getValue("j");

    return ((i>0)&&(j>0));

  }
}
```

```
package oracles;
import java.util.*;
import ca.fillmoresoftware.plugin.OracleUtilities.*;
public class ggcd1_Grid_2_Cell_1 extends CellBase{

  private VarMap vars;

  public ggcd1_Grid_2_Cell_1(VarMap vars){

    this.vars=vars;

  }

  public Object eval(){

    Integer  i=(Integer)vars.getValue("i");
    Integer  j=(Integer)vars.getValue("j");

    return ((i<=0)||(j<=0));
  }
}
```

Bibliography

[1] R. F. Abraham. Evaluating generalized tabular expressions in software documentation. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Feb. 1997.

[2] S. Ambler. *Agile Database Techniques:Effective Strategies for the Agile Software Developer*. Wiley Publishing, United States of America, 2003.

[3] S. Antoy and D. Hamlet. Self-checking against formal specifications. In W. W. Koczkodaj, P. E. Lauer, and A. A. Toptsis, editors, *Proc. Int'l Conf. Computing and Information (ICCI)*, pages 355–360. IEEE Computer Society Press, May 1992.

[4] A. Balaban, D. Bane, Y. Jin, and D. Parnas. Mathematical model of tabular expressions. SQRL Document, 2006.

[5] H. Baumeister. Combining formal specifications with test driven development. pages 1–12. Springer Berlin/Heidelberg, 2004.

[6] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

[7] K. Beck. *Test-Driven Development by Example*. Addison-Wesley, 2003.

[8] G. Bernot, M. Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal*, 6:387–405, June 1990.

[9] B. W. Boehm. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

[10] D. Chapman. A program testing assistant. *Communications ACM*, 25(9):625–634, Sept. 1982.

[11] Y. Cheon and G. T. Leavens. A runtime assertion checker for the java modeling language (JML). In H. R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice (SERP02)*, pages 322–328. CSREA Press,Las Vegas, 2002.

[12] C. Ching. A brief introduction to test driven development using microsoft excel and vba. http://www.clarkeching.com/2006/04/test_driven_dev.html.

[13] E. Clayberg and D. Rubel. *Eclipse Plug-ins*. Addison-Wesley, 2008.

[14] J. Gannon, P. McMullin, and R. Hamlet. Data-abstraction implementation, specification, and testing. *ACM Trans. Programming Languages and Systems*, 3(3):211–223, July 1981.

[15] D. Gelperin and B. Hetzel. The growth of software testing. *Communications ACM*, 31(6):687–695, June 1988.

[16] B. George and L. Williams. An initial investigation of test driven development in industry. Proceedings of the 2003 ACM symposium on Applied computing, Melbourne, Florida.

[17] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Engineering*, 1(2):156–173, June 1975.

[18] R. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Software Engineering*, SE-3(4):279–290, July 1977.

[19] A. Herranz and J. J. Moreno-Navarro. Formal extreme (and extremely formal) programming. In *Extreme Programming and Agile Processes in Software Engineering, 4th International Conference, XP 2003, Genova, Italy, May 2003*, volume 2675 of LNCS, pages 88–98. Springer, 2003.

[20] R. Janicki. On a formal semantics of tabular expressions. CRL Report 355, Communications Research Laboratory, Hamilton, Ontario, Canada, Oct. 1997.

[21] R. Jeffries, A. Anderson, and C. Hendrickson. *Extreme Programming Installed*. Addison-Wesley, 2001.

[22] M. KAJKO-MATTSSON. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55, January 2005.

[23] M. Kohlhase. *OMDoc: An Open Markup Format for Mathematical Documents (Version 1.2)*. Number 4180 in Lecture Notes in Artificial Intelligence. Springer Verlag, 2006.

[24] G. T. Leavens, A. L. Baker, and C. Ruby. JML: a notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[25] D. Luckham, F. von Henke, B. Krieg-Brückner, and O. Owe. *ANNA A Language for Annotating Ada Programs Reference Manual*. Number 260 in Lecture Notes in Computer Science. Springer-Verlag, 1987.

[26] M. Muller and O. Hagner. Experiment about test-first programming. *IEEE Software*, October 2002.

[27] G. J. Myers. *The Art of Software Testing*. John Wiley & sons, 1979.

[28] J. Newkirk and A. Vorontsov. *Test-Driven Development in Microsoft .NET*. Microsoft Press, 2004.

[29] D. Nicolette and K. Scotland. Manager's introduction to test-driven development. Agile Conference, 2008. http://www.infoq.com/presentations/TDD-Managers-Nicolette-Scotland.

[30] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications ACM*, 31(6):676–686, June 1988.

[31] D. Panzl. Automatic software test drivers. *Computer*, pages 44–50, Apr. 1978.

[32] D. J. Panzl. A language for specifying software tests. In S. P. Ghosh and L. Y. Liu, editors, *Proc. National Computer Conf.*, pages 609–619. AFIPS, June 1978.

[33] D. L. Parnas. A generalized control structure and its formal definition. *Communications ACM*, 26(8):572–581, Aug. 1983.

[34] D. L. Parnas. Tabular representation of relations. CRL Report 260, Communications Research Laboratory, Hamilton, Ontario, Canada, Nov. 1992.

[35] D. L. Parnas. Inspection of safety critical software using function tables. In *Proc. IFIP Congress*, volume I, pages 270–277. North Holland, Aug. 1994.

[36] D. L. Parnas and J. Madey. Functional documentation for computer systems. *Science of Computer Programming*, 25(1):41–61, Oct. 1995.

[37] D. L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Trans. Software Engineering*, 20(12):948–976, Dec. 1994.

[38] D. K. Peters. Generating a test oracle from program documentation. M. Eng. thesis, McMaster University, Dept. of Electrical and Computer Engineering, Hamilton, ON, Apr. 1995.

[39] D. K. Peters, M. Lawford, and B. T. y Widemann. An IDE for software development using tabular expressions. In B. Spencer, M.-A. Storey, and D. Stewart, editors, *Proc. Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 248–251, Ontario, Canada, Oct. 2007.

[40] D. K. Peters, M. Lawford, and B. T. y Widemann. Software specification using tabular expressions and omdoc. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Proc. Calculemus/MKM 2007 Work in Progress*, number 07-06 in RISC-Linz Report Series, pages 61–75, Johannes Kepler University, A-4040 Linz, Austria, June 2007. Research Institute for Symbolic Computation.

[41] D. K. Peters and D. L. Parnas. Using test oracles generated from program documentation. *IEEE Trans. Software Engineering*, 24(3):161–173, Mar. 1998.

[42] C. Quinn, S. Vilkomir, D. Parnas, and S. Kostic. Specification of software component requirements using the trace function method. In *Int'l Conf. on Software Engineering Advances*, page 50, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[43] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 105–118, May 1992.

[44] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Software Engineering*, 21(1):19–31, Jan. 1995.

[45] P. A. Stocks and D. A. Carrington. Test templates: A specification-based testing framework. In E. Straub, editor, *Proc. Int'l Conf. Software Eng. (ICSE)*, pages 405–414, May 1993.

[46] Q. M. Tan, A. Petrenko, and G. v. Bochmann. A test generation tool for specificaiton in the form of state machines. Technical Report 1016, Department d'IRO, Université de Montréal, 1996.

[47] Y. Wang. *Specifying and Simulating the Externally Observable Behavior of Modules*. PhD thesis, Dept. of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1994.