TEST FIRST MODEL-DRIVEN DEVELOPMENT

by

Bartlett A. Shappee

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2012

APPROVED:

_____

Professor Gary Pollice, Major Thesis Advisor

_____

Professor Kathi Fisler, Thesis Reader

_____

Professor Craig Willis, Head Of Department

Abstract

Test Driven Development (TDD), Model-Driven Development (MDD), and Test Case Generation with their associated practices and tools each in their own right promise to deliver robust higher quality code more economically then other approaches. These process are not mutually exclusive but are not typically used together. This thesis develops a combined approach using complimentary aspects of each of the above three process. Test cases are described, generated, and then injected back into the model, which is then used to produce the test and production code.

We have enhanced a model-driven tool to support the approach, adding a test case generator, capable of understanding augmented MDD software model and utilizing the constraints captured in our test-centric language to generate model-level test cases back into the model. Our results show that, with a reduction in overall effort one can produce a tested model-based system in which its test and implementation for multiple platforms such as C and Java, using one of multiple test xUnit frameworks.

Acknowledgements

I would like to thank Professor Gary Pollice who has helped guide since the start of masters work and helped me learn to scope the problem making this thesis possible.

I would also like to thank Professor Kathi Fisler for her support and role as thesis reader.

Additionally I would like to thank Austin Noto-Moniz and Sean Maguire for their supporting the evaluation of this thesis.

Contents

Figures

Tables

Chapter 1 Introduction

The software engineering community continues to develop new practices and development strategies around improving productivity and quality. These approaches intend to mitigate problems inherent in software development, such as increasing code complexity, unintended gaps between requirements and implementation, multiple target platforms, and human error. Experts in the community present a wide number of practice focused and tool aided methodologies as solutions, each with the promise of simplifying software development.

This research investigates if generating test cases from requirements based constraints prior to feature implementation is effective, given a platform-independent model foundation for development. If effective, the advantages of such a tool-aided methodology will manifest themselves in the following ways:

1. The software created during development more accurately reflects the requirements.
2. The quality of test development is improved with an overall reduction in required effort.
3. Time to market on multiple platforms is reduced with reuse of test and application development efforts.

1.1. The Problem

Modeling the major portion of software testing has received support [1][2], but there has not been a successful paradigm in which generated model-based systems take full advantage of test-driven development utilizing test case generation, instead of the common coverage based generation approaches post implementation artifacts such as system models and source code. Other approaches tend to suffer from the following issues:

- Model-based testing of hand-coded systems requires continuing synchronization between the model and the implemented system, introducing maintenance challenges.
- Most test case generation tools do not incorporate the test first nature of TDD, instead relying on an existing code base.
- Independent model-based test generation requires significant customization to integrate with existing source bases.
- Testing existing code or models derived from existing code can create disconnects between requirements and tests.

The goal of this work is to determine if it is effective to generate test cases based on applying Requirements-Driven Constraints prior to implementation in a MDD Environment.

1.2. Test Driven Development

One of these practice-based methodologies, Test Driven Development (TDD) intendeds to improve quality while increasing code confidence and team

productivity. The approach involves the development of appropriate test case(s) prior to the implementation of each part of a feature. TDD aims to help improve a number software architecture “ilities" including quality, simplicity, and testability.

1.2.1. Red-Green-Refactor

The driving philosophy of TDD is “Red-Green-Refactor" which is an iterative development micro-cycle in which the tests and behavior are implemented[3]. First, newly written test cases test the unwritten code and as such fail (Red). Once the developer implements the feature in the simplest manner, the corresponding tests should pass (Green). Before starting with the next feature developers refactor their code for clarity and simplicity, with confidence provided by the ability to rerun the previously implemented test (Refactor).

1.2.2. Benefits of TDD

Testability

Testability of the system is improved through the act of writing tests first, requiring developers to think about how the code can be tested and design to that, no longer leaving testing as an afterthought[3][4]. This, in turn, leads to a common reduction in effort required to test components developed using TDD.

Simplicity

During the implementation of a feature, the test-driven developer performs the simplest activity to result in turning all test Green. This approach to development has the effect of simplifying code design [3] and achieving the results that come with that including improved readability and understandability.

Quality

Quality improves through the development of a regression test base rooted in the requirements. Building the regression tests from the beginning allows for the continual verification of the impact due to changes, giving developers and product owners a greater confidence that when changes are made, there are no negative side effects.[5][6]

1.2.3. Barriers to Adoption

The primary barrier to adoption is the perception of testing as tedious or non-value adding by developers and project managers. In combination with the now increased cost of refactoring, test-driven developers may tend to revert to an implementation first focus and testing again becomes a secondary activity[1]. Test-centric methodologies strive to mitigate this barrier, as Grenning describes how a quick turnaround time of the TDD micro-cycle is critical to its adoption[7]. We pose the question of whether a test-driven approach augmented by test case generation, can become a successful part of the application specification and design phase improving adoption.

1.3. Model-Driven Development

Model-Driven Development (MDD) is a maturing software development practice of documenting and producing software systems as models. MDD is built upon Model-Driven Architecture (MDA) as outlined by the Object Model Group (OMG) utilizes their modeling language, the Unified Modeling Language (UML) to provide a high-level framework for which to document software architectures. By raising the level of abstraction away from the implementation details of a system, modelers are able to easily capture and document the architectural patterns of the modeled systems in regards to providing a solution the defined problem space[8]. MDD transforms the Platform Independent UML models using a set of mappings into the corresponding system implementation (source code).

To produce these executable systems MDD extends MDA, augmenting the architecture models with abstract behavioral representations or action languages, such as the UML Action Language (UAL)[9] and Platform-Independent Action Language (PAL)[10]. Benefiting from a high level of software abstraction, MDD models when combined with efficient code generation tools, promise to reduce software complexity by allowing the architectural and behavioral aspects of an application to be directly mapped to the implementation. The advantage is that unlike with MDA models, which require manual effort to synchronize the models and implementation, MDD automates the process.

Additionally by utilizing this abstract representation, software testers can define their testing parameters and test cases, to reflect the expected behavior and documented interfaces of the system instead of the current or expected

implementation. This reduces the influence of a number of implementation specific details, such as storage structures on test development.

1.3.1. Platform Independence

Software development groups building complex and high performance shared components for multiple product lines face substantial difficulties, including varying programming languages across products and time. While it is possible to share common architectures and design patterns across development groups, there is no realization of true productivity gains until we reduce parallel development, debugging, and testing of mirrored components for separate platforms to a single development activity.

We can capture this activity in the development of Platform-Independent (PI) UML models to deliver multiple products for various platforms (such as having both C and Java variations) from a single modeled source. For further productivity improvements, one must incorporate the test base into these PI models, allowing the generation of the base for the target platform.

1.3.2. Development through Code Transformation

The ability to generate both the application code and test code from the same augmented UML models provides several key benefits. First, the test infrastructure can properly access the application elements independent of the generated language or coding patterns utilized. Eliminating or minimizing the

maintenance requirements placed on the test base during refactoring or re-work/structuring activities.

The second is the ability to instrument or augment the generated application code with various test aids, including tools such as test spies (see glossary) or mock objects (see glossary). Test aids such as spies and mocks allow testers additional options for validating a components behavior based upon internal state and execution flow. The ability to generate such tools provides another avenue for testing effort reduction.

The last benefit of generating test cases is that while xUnit (see glossary) and mocking frameworks can share a common approach/style, the model level testing capabilities provide a generic interface allowing generation to various test frameworks. This interface is important to abstract away the slight nuance that come with individual test frameworks on multiple platforms. This allows us to keep testing and development, platform independent.

1.4. Assumptions

Based upon the author's experience in industry developing embedded systems and for the simplification of the TFMDD tool and scoping, we make several assumptions in this work. These assumptions include:

- To aid in the generation of off-nominal test cases, the generator treats Boolean return parameters as pass/fail values based upon expected as defined in the constraints expected. For an off-nominal case, which

violates the preconditions, the non-expected (fail) value will be the expected output of running the test.

Chapter 2 Model-Based Testing Approaches

This chapter presents an overview of several different model-based testing approaches highlighting aspects taken into account in developing the TFMDD approach presented at the end.

2.1. Sequence Diagrams Based Test Generation

Javed et al. present a tool supported methodology providing an approach to translating sequence diagrams into system level tests through a series of vertical and horizontal transformations using a variety of toolkits[11]. The authors define two terms to describe different transformation; a horizontal transformation is when a PIM is transformed into another altered PIM and a vertical transformation is when PIM is transformed into a PSM[11]. The rationale behind the approach is that utilizing sequence diagrams produced during system analysis, “can initiate software-testing activities in an early stage of software development process.” [11] Choosing this type of development artifact allows the approach to produce results quickly and usable throughout the complete process.

The methodology presented by Javed et al. bases test validation on two different criteria; xUnit style validation and comparison of execution tracing captured. The xUnit test generator combines provided test data, including inputs and expected outputs for method invocations with the extracted sequence information to

produce test cases. Additionally, the generated xUnit test cases facilitate driving the execution of the system producing the execution logs required for the system level validation against previously captured logs.

The authors claim that a major advantage to their approach, as compared to others based upon sequence diagrams is the ability to quickly redeploy their tool to various xUnit platforms, citing a 85% code reuse between their JUnit and SUnit (for Smalltalk) test generators[11].

2.2. Constraint Based Test Generation

Aichernig and Salas present an approach to constraint based test generation drawing on the principles of Mutation Testing (see Glossary) to test a system for nominal behavior and attempt to detect possible faults[12]. The authors claim that when provided a program specification, "[t]esting can show the absence of faults, if we have a knowledge of what can go wrong."[12] The authors present a version of their tool, which analyzes OCL constraints to solve the Constraint Satisfaction Problem (see Glossary) to support their claim.

Their work presents several interesting aspects around generating constraints from OCL specifications. In addition to handling the nominal cases, their tool attempts to mutate the specification to create test cases against an alternate incorrect specification. The generation of off-nominal test cases provides a foundation for the authors' goal of detecting possible faults through Fault Based Testing (see Glossary). The version of the system created by mutating constraints, allows the tool to generate test cases that would specifically exercise the

potential fault exposed by the mutation and determine its presence in the code. Their solution provides an interesting approach for developing off-nominal test cases in addition to the straightforward nominal cases.

Aichernig and Salas also remind the reader that “a faulty design may be correctly refined into an implementation” [12] implying the need for a careful and strong refinement of system constraints from requirements.

2.3. Use Case Based Test Generation

Nebut et al. present an approach utilizing model use cases and scenarios to generate test scenarios from which test cases can be generated[13]. Similar to the goals of this project, their approach focuses on the development of these use cases during requirements analysis, resulting in a tight relationship between the requirements and generated test cases. Additionally, Nebut et al. consider that “[t]o allow seamless industrial acceptance, the contract language must be simple, so that it can be easily used during the requirements analysis.”[13]

Use cases provide the authors approach with a foundation for capturing free-language (natural language) requirements in a standardized approach that is semi-machine readable format and easy to develop. In addition to keeping the process simple, use cases are utilized in well-established development life cycles unlike formal methods[13]. The authors point out the need for converting requirements into more machine-readable formats to simplify the test generation process. Additionally, their test generation technology (the UCTS tool[13]) attempts to further simplify the test generation problem by allowing the modeled

use cases to be augmented with contracts (similar to constraints) containing pre and post conditions.

The contract language allows for the usage of predicates in a definition in an effort to allow a richer semantic meeting in the contracts. The semantic definition of each predicate exists in an external artifact known to the UTCS tool. The contract language utilizes these predicates to become more flexible and in essence develop a Domain Specific Language (see Glossary) or "vocabulary"[13] for defining contracts. However, the authors state that the growth of the predicate base becomes hard to manage requiring tool support.

The tools presented in [13] also provided the ability for simulating modeled use case execution and validation in addition to JUnit test case generation.

2.4. Specification Based Model Validation

Gogolla et al. present the tool USE (UML Specification Environment) for providing UML validation through specifications derived from requirements[14]. USE utilizes UML models augment with OCL based constraints and provides a java based toolkit to support modelers. USE accomplishes this in two ways; it checks the consistency of provided pre and post conditions, and invariants. It has the ability to validate produced snapshots against the defined constraints.

USE evaluates a provided UML model to determine whether the consistency of invariants in regards to being contradictory, independent, or implied[14]. Contradictory constraints produce a situation in which no system state is possible that satisfies one constraint without invalidating the other one. USE provides the

ability to determine the ability to produce a system state satisfying all invariants but the one under test, showing an invariant to be independent of all others.

Snapshot generation produces all the various plausible system states based upon the test procedure provided. Documented in a proprietary language, each procedure contains executable semantics and containing nested test data sets. By parsing a test procedure the USE engine has the ability to simulate every valid path through the procedure, using each applicable entry in the data set to report the resulting system state as a snapshot[14]. These combinatorial-based snapshots provide the validation basis equivalent to execution of unit tests. After simulation, USE validates these snapshots against the specified OCL constraints (valid criteria) reporting if any invalid states were found[14].

Overall, the consistency checking of USE is a feature applicable to any full-featured constraint based model validator. The snapshot generation provides an interesting opportunity for validating a constrained model through simulation with requiring the production of executable code; however, the authors do not present a path forward for an application applicable to testing executable systems.

2.5. TFMDD

The work presented in this thesis draws on constraint solving with a focus on translating requirements into testable constraints through the development and translation of user stories and generating both nominal and off nominal cases as determined through constraint solving. TFMDD relies heavily on pre and post

conditions defined in a customized constraint language similar to the works mentioned here. Similar to several of the works, TFMDD focuses on execution based validation instead of simulation, through the translation of MDD models into both executable system and test implementations.

However, TFMDD differs from the previously mentioned works by focusing on the holistic combination and integration of TDD and test generation technologies into MDD. The two key differentiating aspects of this work to the others described are the combination of test generation into TDD and the generation of test cases back into the original MDD model. By combining a TDD approach with the use of test case generation we attempt to achieve software quality improvements[3] with a further reduction in required effort, easing the barriers of adoption to TDD. Without the need to transform the constrained PIM into another PIM containing test specific information or transformation into a system specific test execution framework we believe that the final transformation to implementation from one complete model produces a richer test set.

Additionally, this work presents the practice of nested variations inside modeled constraints as a method for producing combinations of constraints in an approach seemingly unique to this work.

Chapter 3

Your First Test Case using TFMDD

The following example takes us through the development of a feature in the School Management System, adding a Student to the system.

Before any of these steps, an example project provides the initial development environment in Rational Software Architect™ (RSA) with PathMATE including the corresponding project, system models, and testing tools configured for code generation and testing, for more information see Appendix A.

3.1. Analyze the Task and Add Model Element to Constrain

As this is the first feature to implement in the system we need to add several model elements to house the test constraints, generated test cases, and future implementation.

1. Identify a Domain (see Glossary) based on the most likely subject matter addressed by this feature (Remembering that this Domain can be restructured/organized later as a refactoring effort).

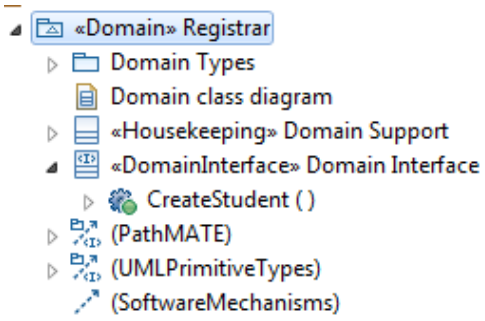


Figure 1-Modeled

For example, we add a "Registrar" Domain to the system as seen in Figure 1.

2. A Domain Service (see Glossary) needs to be added as the starting point for defining the constraints and new functionality.

   For example, we add the Domain Service CreateStudent() .

3. In RSA, the next step is to add a Body Constraint UML Element, which stores the test constraints as seen in Figure 2.

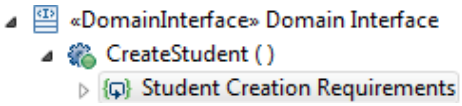


Figure 2-Modeled Constraint

4. Then roughly identify a number of constraints present in the requirements. For this example, some of the requirements are:
   - A return value indicates TRUE on success
   - The provided student id must be valid (Positive non-zero 7 digit number)
   - The provided id must be unique to the new student

3.2. First Test Case – Return Value Validation

Starting with the simplest activity the first test case we walk through constraining that the return value is true when all preconditions are satisfied.

3.2.1. Add Test Constraints to the Service (RED)

The next step is to model the requirement as a set of constraints on the model behavior. These constraints serve a dual purpose, one of which is to help the developer analyze the problem space in regards to the requirements and the second is for test case generation.

1. In the Body Constraint element, developers will define the constraints based upon the requirement under development using the tool's test language.

   Following TDD guidelines, in this example we will first specify that the provided return value is TRUE for successful execution as seen in Figure 3.

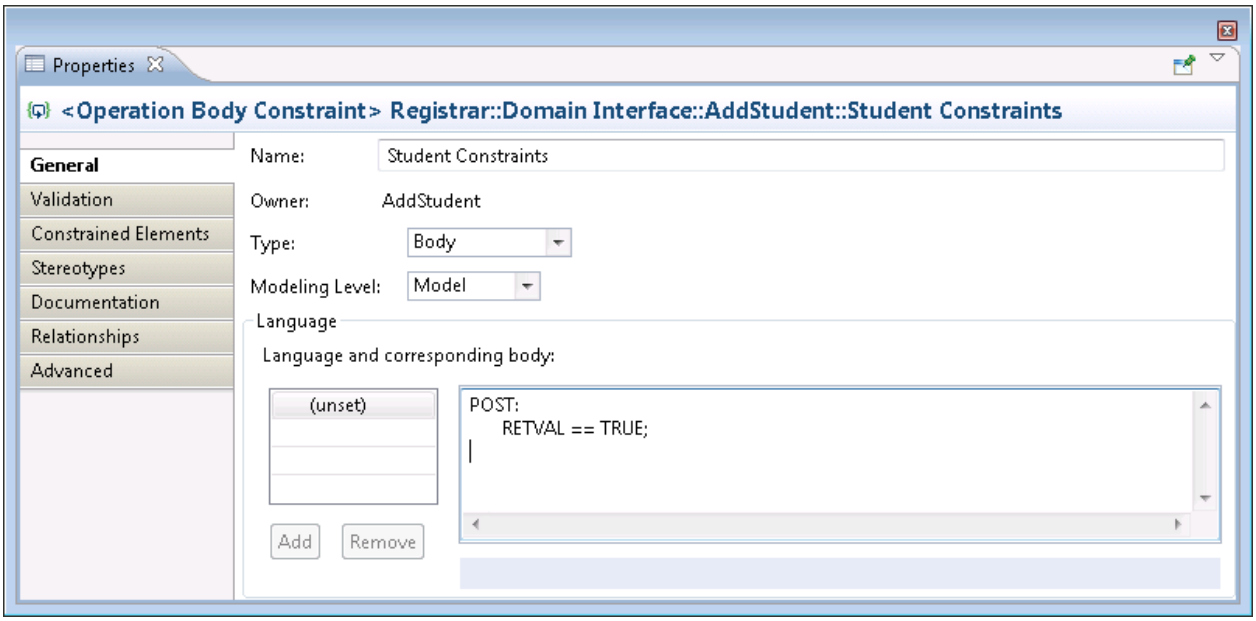


Figure 3 Constrain that a Student's Name is not empty

3.2.2. Generate Test Cases (RED)

After constraining the new system behavior, the user can generate the test Cases using the TFMDD tool. The generated tests are automatically added "Unit Test" package within the domain and each domain service will have its own Domain Interface defined in the new package to organize and store the related tests.

1. Use the TFMDD tool to generate positive and negative test cases from the defined constraints.

   For this example without any defined preconditions, we will only see one positive test generated. The generated test case is available in Appendix B.

3.2.3. Add Model Elements Identified by the Constraints (RED – Model Structure)

Developing testing constraints, helps identify necessary parameters of the Domain Service in addition to classes and other model elements. In our example, we further highlight this fact with an attempted model transformation. The test cases fail to transform because of the missing required return status.

1. Add the identified model elements to address the transformation errors.

   For this example, we add the Boolean return parameter to the CreateStudent Domain Service.
2. Additionally, when adding return parameters the implementation body of the service will need to return a stubbed value.

   Given that is the case with this example; we add a standard failure stub returning false into the CreateStudent Domain Service.

```
RETURN FALSE;
```

3.2.4. Run the Test Cases (RED - Implementation)

After implementing the model structure, the next step is to transform the system and test cases into code. We compile and execute the generated code, resulting in the test cases failing.

1. Run the generated code using the JUnit Test Runner and observe the results, which given TDD practice should include several failures.
   For this example, we see that returning FALSE as a placeholder resulted in a failing test.

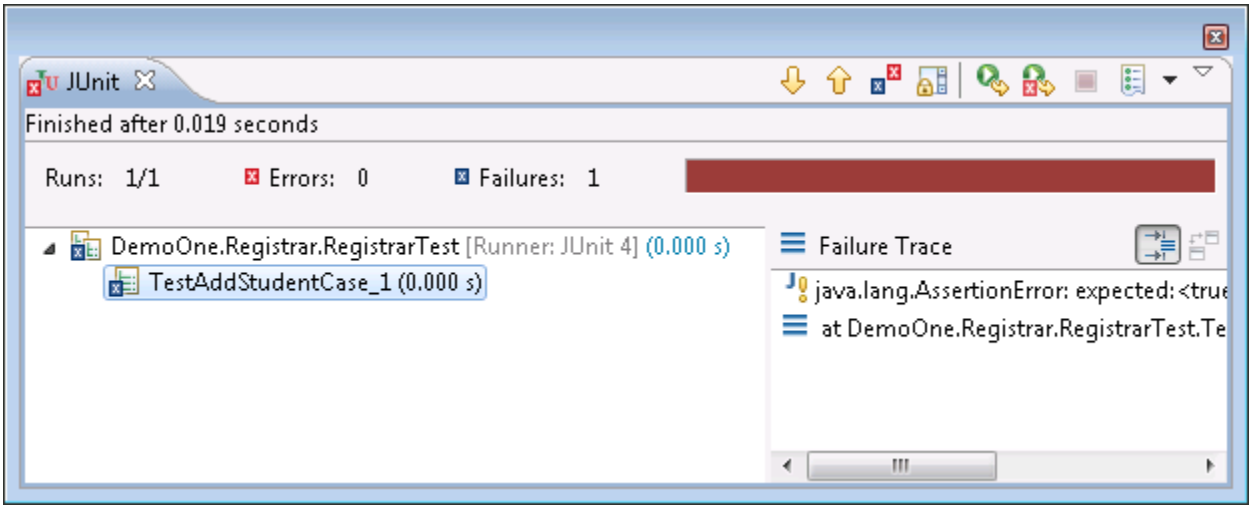


Figure 4 TFMDD Example Part 1 RED

3.2.5. Implement the Behavior (GREEN)

The second step of the TDD micro-cycle is Green, where we implement the simplest version of the intended behavior in order to have all test cases pass. For our example, this involves implementing the CreateStudent Domain Service in the

simplest manner, which requires changing the method to return TRUE for success. The following syntax is an example of how this could look.

```
RETURN TRUE;
```

Now we rerun the tests to ensure that the implementation correctly satisfies the current constraints.

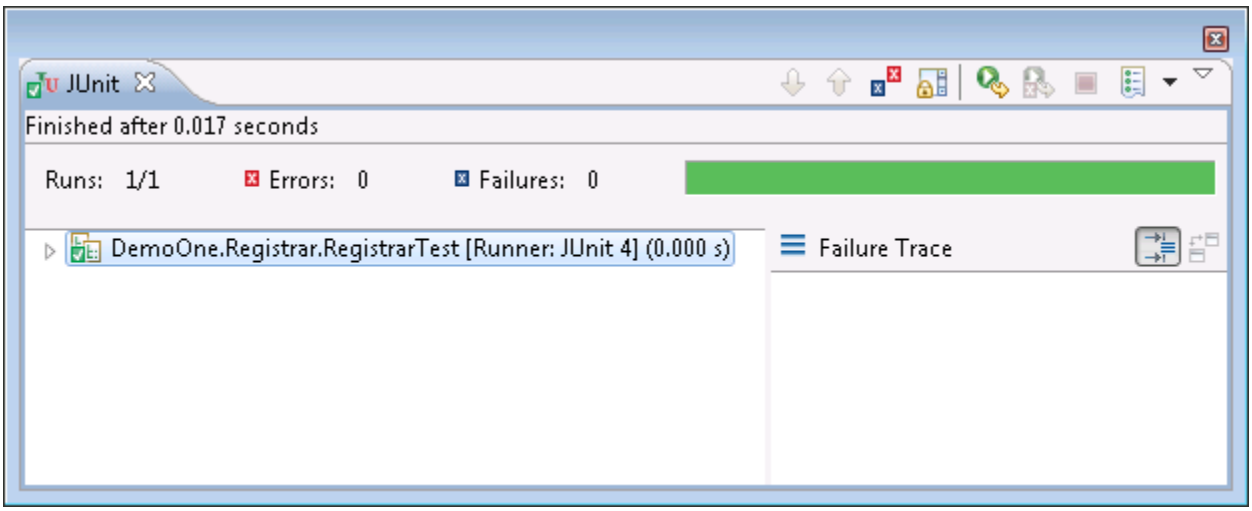


Figure 5 TFMDD Example Part 1 GREEN

3.2.6. Refactor the Model (REFACTOR)

Given that this is the first test case/feature, no refactoring is required. However, this is the point where we would refactor the model going forward.

3.3. Second Test Case – Input Parameter Validation

The second test case we will walk through adds a precondition regarding the supplied parameter values, as the next constraint is that the provided student id is valid. The constraints on AddStudent now read:

```
PRE:
   PARAM (id) > 0;
   PARAM (id) <10000000;
POST:
   RETVAL == TRUE;
```

Adding preconditions to this example expands the functionality covered by existing test cases as well as generating off-nominal test cases, validated by checking that the return value is not TRUE.

Red

With this second constraint, we go through similar steps to the first, fixing transformation errors before compiling, executing the test, and continuing on to implementation.

1. To solve the transformation error, we add the student id integer input parameter to the AddStudent Domain Service.
2. Next, running the generated JUnit test cases shows that while the nominal-test case still passes the off-nominal cases for out of bound ids fails to catch a failure. The generated test cases are in Appendix B.

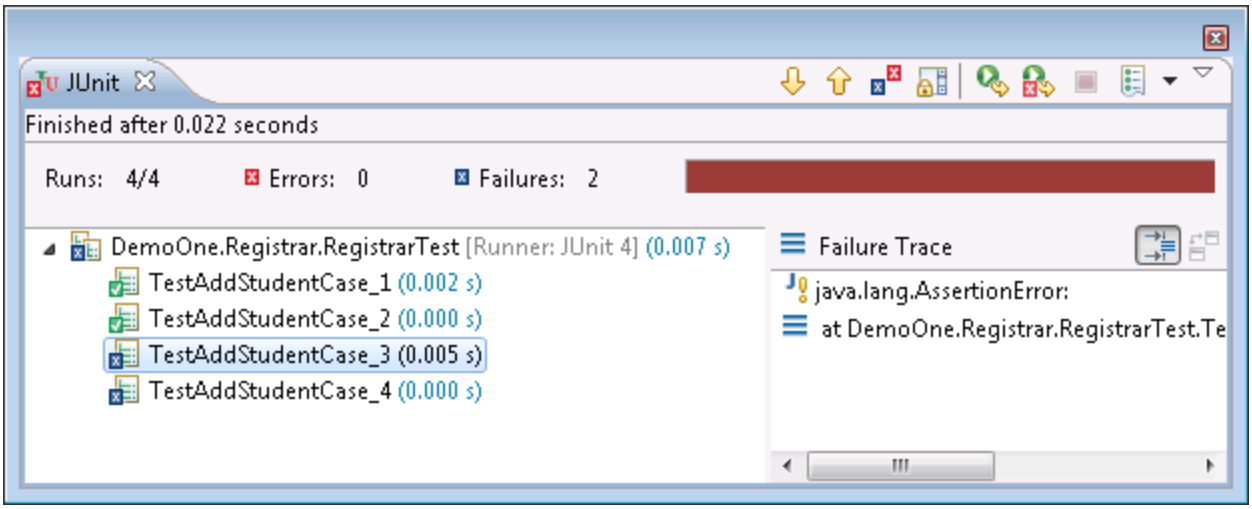


Figure 6 TFMDD Example Part 2 RED

Green

Now we update the implementation to check the incoming parameter value and report error cases. The following syntax is an example of this:

```
Boolean ret = FALSE;

IF(student_id > 0 && student_id < 10000000)
{
   ret = TRUE;
}

RETURN ret;
```

Now we rerun the tests to ensure that the implementation correctly satisfies the current constraints.

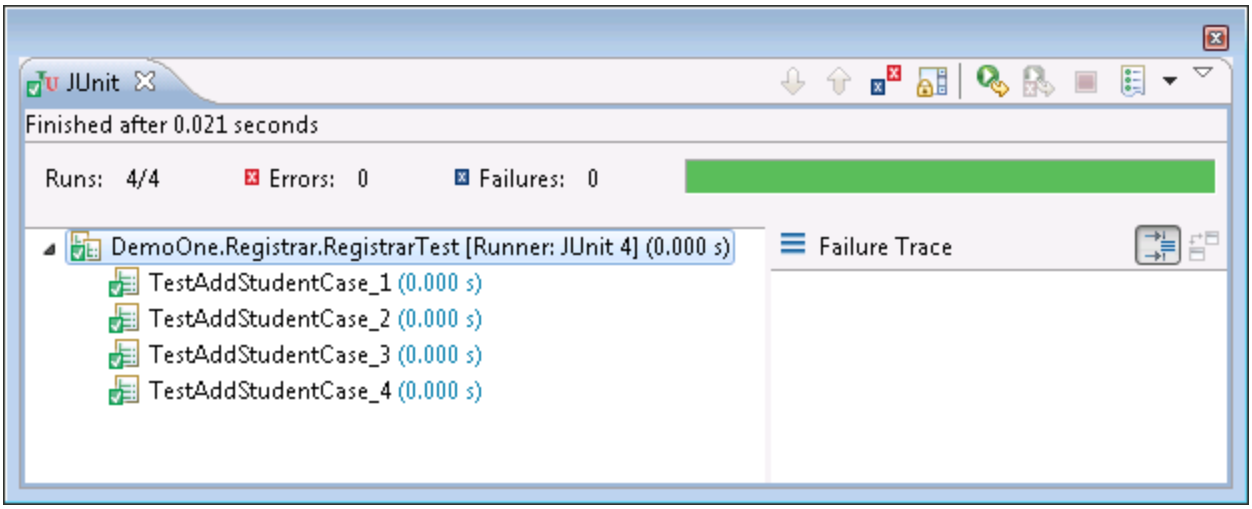


Figure 7 TFMDD Example Part 2 GREEN

3.4. Third Test Case – Model State Validation

The third test case introduces requirements about pre-existing system state required for proper execution. The next constraint is to have unique student ids, which results in the condition that previously supplied student ids have been stored and used in parameter validation. We add the constraint that a student with the provided id does not already exists in the system.

PRE:

PARAM (id) > 0;

PARAM (id) <10000000;

NOT EXISTS Student WHERE (id == PARAM (id))

POST:

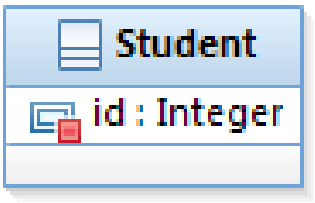
RETVAL == TRUE;

Adding state-based pre or post conditions expands the complexity of the generated test case to either modify the pre-existing state or validate the state prior and post execution.

Red

The stateful nature of the recently added constraint causes the transformation process to fail due to a reference of an unidentified test class in the test case action language.

1. To solve the transformation errors, add "Student" class with an id attribute to the system.



2. Next, running the generated JUnit test cases shows that while the previously executing test cases still pass, the new off-nominal cases around uniqueness will fail to catch a failure. The generated test cases are in Appendix B.

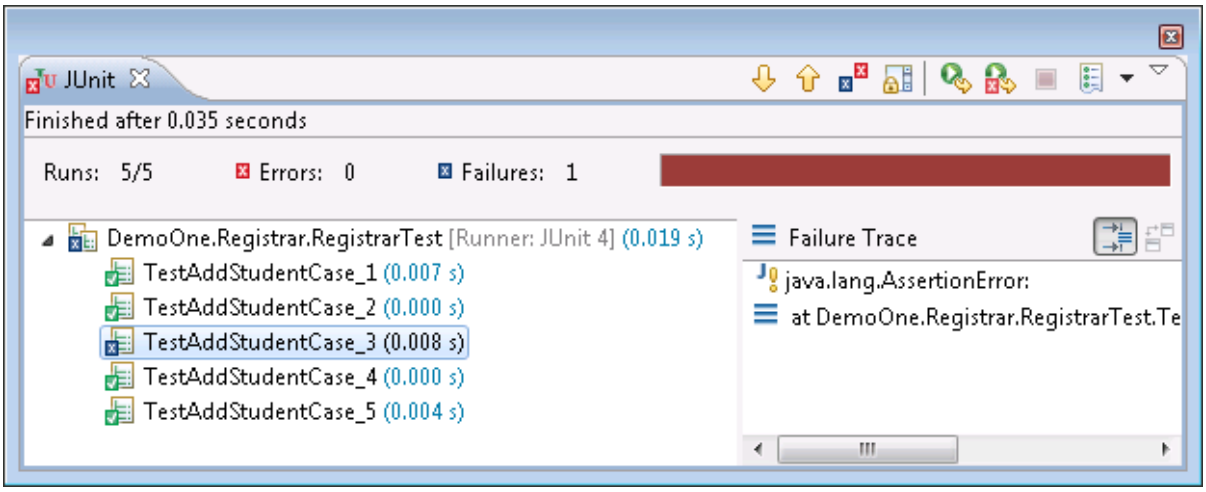


Figure 8 TFMDD Example Part 3 RED

Green

Continuing, updates to the implementation add checks for an existing student with the supplied id and if one is found, return an error condition. The following syntax is an example of this.

```
Boolean ret = FALSE;

IF(student_id > 0 && student_id < 10000000)
{
   Ref<Student> student = FIND CLASS Student WHERE (id == student_id);
   IF(student == NULL)
   {
      ret = TRUE;
   }
}

RETURN ret;
```

Now we rerun the tests to ensure that the implementation correctly satisfies the current constraints.

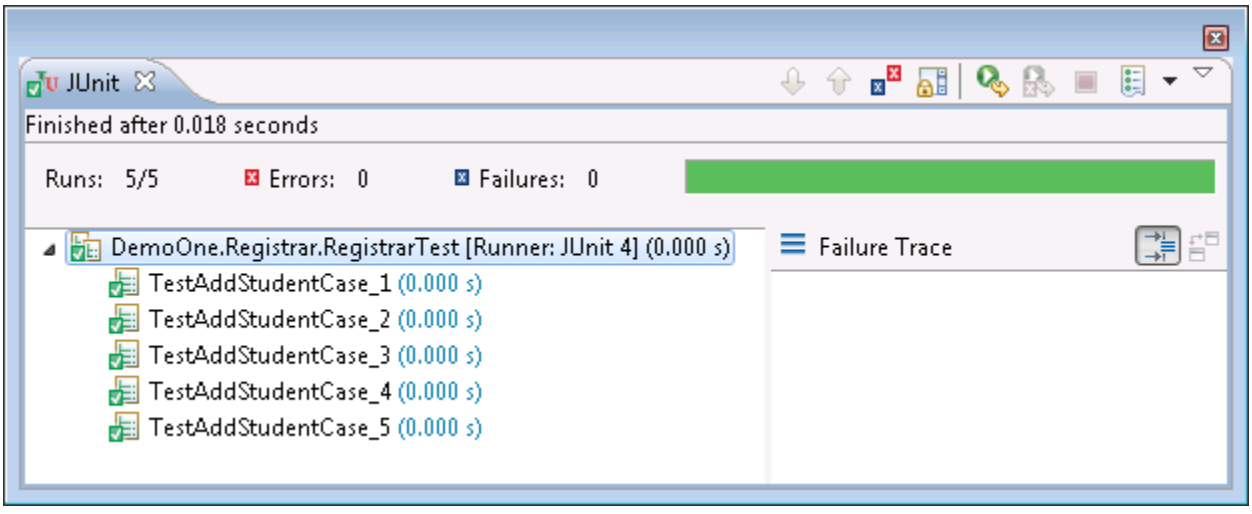


Figure 9 TFMDD Example Part 3 GREEN

Refactor

While the introduction of this constraint early in the system development does not require any refactoring, the introduction of a class can lend itself to refactoring efforts moving forward. For example, pushing the majority of the domain service logic down into the class, if deemed necessary is possible without affecting the generated test cases' validity.

Chapter 4 TFMDD Supporting Tools

Testing the concepts presented in this thesis requires the development of a prototype test case generator. The goal of this tool is to analyze the constraints added to a platform independent UML model, then produce and inject unit test cases back into the model as presented in Figure 10. In this regard, the test-driven process resides completely within the model. The supporting tools include a test-centric constraint language and test generation engine containing a corresponding parser and integration into a modeled environment.

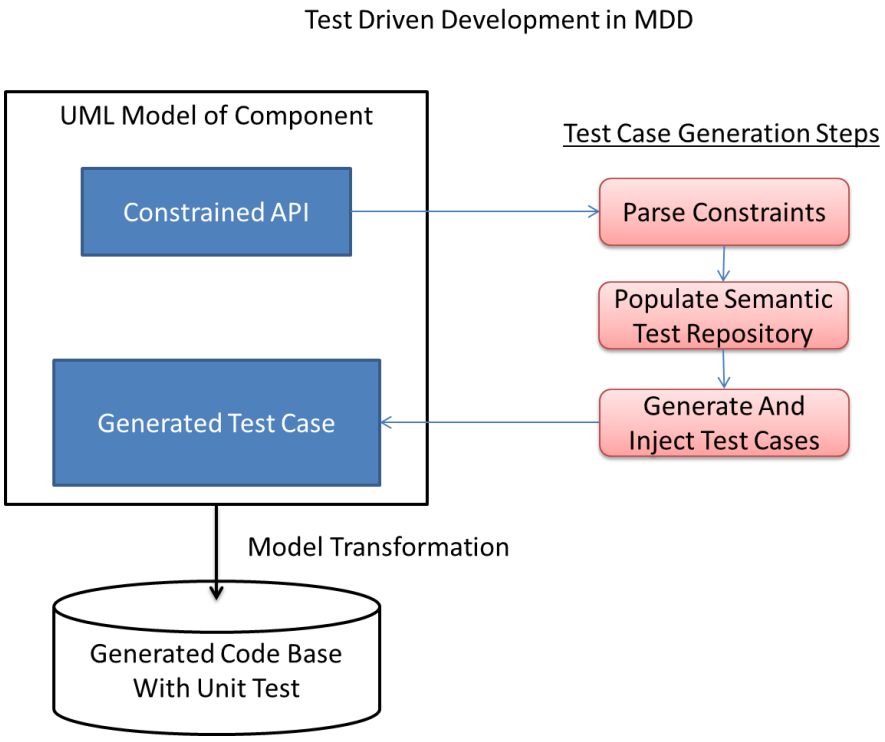


Figure 10 TFMDD Development Overview

4.1. Constraint Language

The constraint language developed for this thesis loosely draws on OCL for its general structure, but was developed to map constraints to directly to Model Structure and behavioral specification that could be used to generate test cases.

4.1.1. Overview

The language is broken into two sets of keywords. First are the organizational aspects to the language, the pieces that identify scope. Next is the set that specifies the behavioral requirements of the system functionality being constrained. The next section describes the keywords currently supported in the constraint grammar.

4.1.2. Key Words

This section presents the organization of keywords available in the constraint language:

Constraint Organization

- **VARNT –** A Variant used in conjunction with {} to scope multi-level nested constraints.
- **PRE –** Used to designate the beginning of a precondition definition.
- **POST –** Used to designate the beginning of a post condition definition.

Behavioral Specification

- **[NOT] EXISTS –** Used to specify the existence or absence of a class instance.
- **PARAM –** Used to define valid input and output value sets.
- **Standard Comparison Operators –** Including >, >=, <, <=, ==, and != used for the definition of a condition through the comparison two elements.
- **WHERE –** The WHERE specification is used in conjunction with the EXISTS to specify which class instance is referenced.
- **RETVAL –** Designates the value returned from the invocation of the constrained operation.

Additionally optional annotations designated by the symbol @ provide assistance to the test case generator. The following sections will outline which annotations are available to modify different keywords and statements.

4.1.2.1. [NOT] EXISTS

The EXISTS keyword is used define whether class instances exist. NOT is an additional modifier available for this keyword to designate when an instance should not exist. The class references by this keyword should exist within the same-modeled component as the constrained service.

Precondition: When used in a precondition, the EXISTS constraints define the required system state prior to execution. In testing these constraints provided the information necessary to generate the existing state of the system, necessary to complete nominal and off-nominal testing.

Post condition: The post condition usage of EXISTS constrains how the state of the system is modified, by the execution of the operation given all preconditions are met.

Example: In the following example, a post condition of a cleanup operation is that no instances of the Registration class exist.

POST:

```
NOT EXISTS Registration ();
```

Available Annotations

When there exists a precondition that a class must exists, an annotation can be provided to specify the creation syntax (in PathMATE Action Language) to use when creating an instance in the appropriate test cases. The following example specifies the create statement to use when it is necessary to create an instance of the Registration class in associated tests:

```
@CREATE Registration (id = 10, callback = EMPTY_SERVICE_HANDLE);
EXISTS Registration;
```

4.1.2.2. VARNT

Variants (VARNT) constrain behavioral requirements based upon the combination preconditions that are satisfied. Each path through the constraints results in a different combination of expected preconditions and corresponding execution behavior.

Example: In the following example precondition, *A* must hold true and if the parameter value is zero or greater *X* and *Y* will occur, otherwise *X* and *Z* will occur.

PRE: *A*

POST: *X*

VARNT {

    PRE: PARAM (value) >=0;

    POST: *Y*

}

VARNT {

    PRE: PARAM(value) <0;

    POST: *Z*

}

4.1.2.3. PARAM

The PARAM keyword references the parameter value when defining operation constraints. When used in conjunction with a standard comparison operator and another value, PARAM constrains input and output values. A string parameter to the keyword specifies the name of the constrained parameter. Additionally, other constraints use PARAM within their definitions to access the parameter values supplied to a function.

Precondition: As a precondition, PARAM defines the acceptable inputs in an operation.

Post condition: Only output parameters can be constrained in the post conditions, specifying the expected values returned from execution.

Example: The flowing example constrains the id parameter to be a nonzero positive number.

PRE:

PARAM (id) > 0;

4.1.2.4. PRE

PRE designates the beginning of defined preconditions.

Example: In the following example constraint *A* is defined as precondition of the

PRE:

Constrain *A*

4.1.2.5. POST

POST designates the beginning of defined post conditions.

Example: In the following example constraint, *A* must be the result of executing the function.

POST:

Constrain *A*

4.1.2.6. RETVAL

RETVAL is a special keyword referring to any value returned from the constrained parameter. When a return parameter is of type Boolean and is constrained, generated error case use the expected the opposite value to be returned.

Precondition: Not Available.

Post condition: Using RETVAL with standard comparison operators constrains the acceptable output of services.

Example: The flowing defines success criteria.

POST:

RETVAL == TRUE;

4.1.2.7. Standard Comparison Operators

The standard array of comparisons are available: >, >=, <, <=, !=, and ==. When used outside of constraint clause in conjunction with PARAM statement, the use of comparison operators signifies a constraint on those parameters.

4.1.2.8. WHERE

The where clause is used in conjunction with an EXIST statement to specify whether a specific Class instance should or should not exist as either a pre or post condition.

Example: After the service executes there exists an instance of the Student class where its id is equal to the student_id parameter.

POST:

```
EXISTS Student WHERE (id == PARAM(student_id);
```

4.2. The Engine

The following section describes the TFMDD Engine including its integration into a UML Modeler, constraint parser, and test generation capability.

4.2.1. Modeler Integration

This project plugs into the Eclipse Framework and utilizes the Rational Software Architect (RSA) tool as a UML Editor.

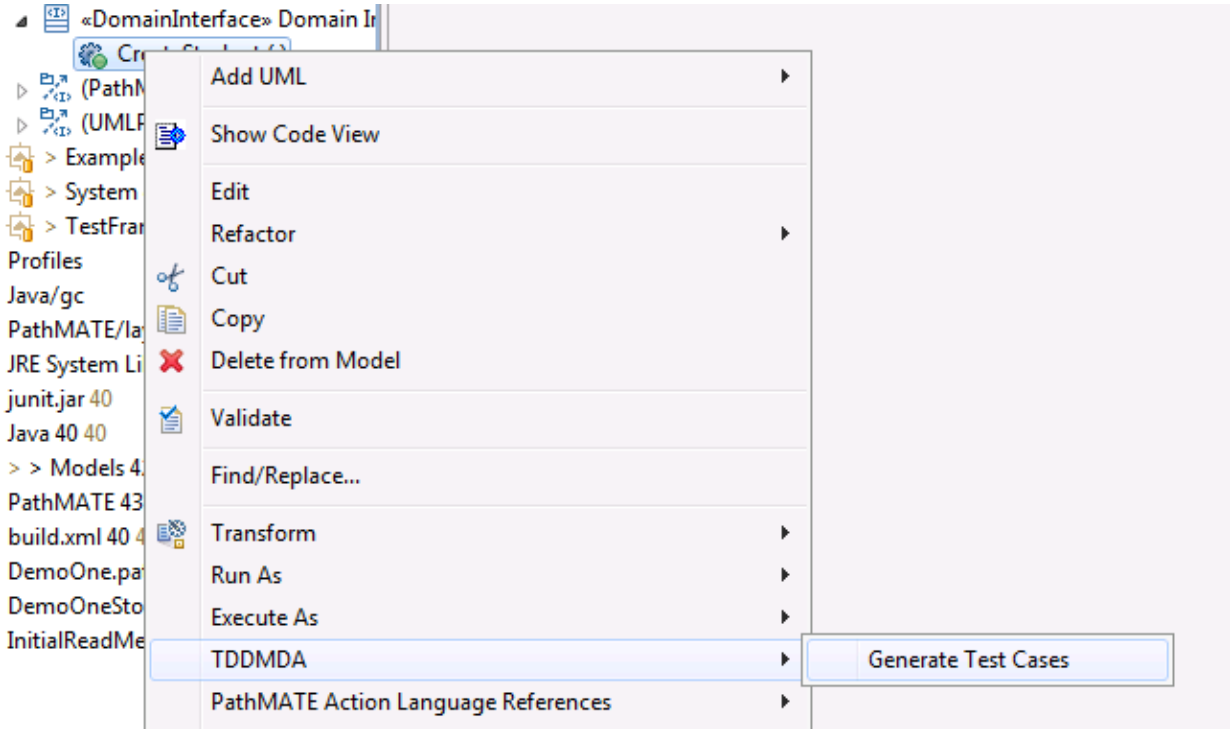


Figure 11 Menu Option Provided by the Integration into the RSA (Eclipse) Tool

4.2.1.1. Eclipse Technology

The Eclipse Framework provides a rich plug and play architecture, allowing for easy extension[15]. The plug-ins are written in Java, allowing for integration of the developed constraint parser and test case generator of which both are generated into Java.

4.2.1.2. Rational Software Architect

We decided that any modeling tool must fulfill two requirements to support the prototype development. First, the modeling tool must provide the ability to specify constraints with-in the model. Secondly, the information must be

accessible from an external tool through an API available to plugins executing in the same Eclipse environment. The RSA tool met both of these requirements.

4.2.2. Parser

After extracting the modeled constraints, the next step is for the engine to pass the constraints into a parser. This requires the engine to have a Constraint Parser which we develop using the ANTLR toolkit. The resulting parser provides a multi-pass process, in which it syntactically verifies the constraints and converts them into a more usable intermediate structure, and by invoking, the appropriate APIs populating the Semantic Test Repository.

4.2.2.1. ANTLR Technology

ANTLR (ANother Neat Tool for Language Recognition) is a tool that provides parser generation from a set of grammar and rules files[16]. This project takes advantage of the ANTLR's Java generation feature to produce our parser. ANTLR's ability to embed java code into almost any aspect of the grammar files and subsequently the generated parse provides the mechanism for populating the test repository from the parser.

4.2.3. Semantic Test Repository/Generation

This project utilizes MDD within the RSA with PathMATE environment to develop the Semantic Test Repository (STR). The modeled system publishes enough domain services so that the test information may be populated during constraint

parsing, storing the primary aspects of the constraints in equivalent model structures. The classes located in two semantic groups have the appropriate support services to generate test cases as described in the following.

- Parameter Constraints – Constraints used to generate the input values to drive the test down nominal and off-nominal path and used to validate the output from those test.
- State Based Constraints – Constraints used to set up the state of the system for nominal and error path testing and used to validate the correct changes did or did not occur to the system state.

Test case generation from the STR works recursively, traversing the instantiated constraint information with in the TestData domain, invoking a chain of instance-based operations that chain together to form a test case. Each class understands how to create the appropriate action language necessary for normal and error path test cases and precondition setup or post condition validation where applicable.

The test case generation starts at the Operation class, which collects all test cases returned from invoking generateTestCases on each associated TestCases class. Each TestCases instance maps to a variant identified during parsing the constraints and is responsible for constructing well-formed test cases, with the four pieces identified early; setup, execution, validation, and cleanup, for example:

```
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;
```

```
//Setup Initial State
Ref<Student> _pre_Student = FIND CLASS Student WHERE (id== _student_id);
TestFramework:CheckEqual(NULL, _pre_Student);

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

Figure 12 Example Test Case Generated From Constraints

Additionally, the TestCase instances are also responsible for generating off-nominal test cases. The algorithm for accomplishing this is to generate one test case for each precondition in which it is not satisfied, prior to invoking the service under test.

Upon completion of test generation, the engine injects each element of the resulting group as an individual test case back into the model. For organizational purposes, each domain service is mapped to one test container in the model were all test cases are stored. One additional note is that the engine does not keep previously generated test cases synchronized with updated constraints; each run it deletes and recreates the corresponding test container each time tests.

4.2.3.1. PathMATE Action Language

As seen in Figure 12, the engine generates test cases in the PathMATE Action language. We utilize PAL for test generation because it is a higher-level language utilizing a simpler syntax without as many nuances of the modern general-purpose languages. The additional benefit is that when implementing the model during the TDD process, the same PathMATE Action Language is used to

document behavior. Keeping the implementation and test cases in the same model level language allows us to use the same transformation technology to produce both the system implementation and the test cases at the same time.

4.2.3.2. PathMATE Test Framework Domain

As seen in Figure 12, the PathMATE Test Framework domain provides an API for the modeled action language and generated test cases, to utilize the integrated xUnit style testing frameworks.

4.2.3.3. JUnit Integration

In order to develop using JUnit integration, it is necessary to add several JUnit specific code-generation templates to the PathMATE Framework for Java and the CheckNotEqual service to the TestFramework domain interface. The JUnit extension to PathMATE facilitated the need to keep all development efforts in Java for comparison to the student-developed systems (see Chapter 5) and the additional test hook simplifies test case generation around instance existence and NULL checks.

Chapter 5 Empirical Results

To analyze the effectiveness of TFMDD we utilize multiple methods to develop the School Management System (SMS) introduced in Chapter 2. The provided requirements focus on the establishing a set of API routines allowing the system state to be built up, for example the creation of information such as students and course offerings. Each requirement presents a slightly different testing challenge, from simple object instantiation to complex relationships with different error path opportunities.

We conduct the evaluation based on a comparison between three implementations of the SMS system, two of which were developed following standard TDD and the third developed using TFMDD practices and tools developed for this thesis.

5.1. Evaluation Criteria

The evaluation criterion compares the developed School Management systems focusing on the desired characteristics including quality, productivity, and complexity. Code Quality measures of how correct the code is as it relates to requirements and absence of defects. Productivity is a measurement of the time required to plan, test, develop, and refactor the system. Code Complexity measures the code in terms of its Cyclomatic complexity.

All development begins from a documented set of user stories provided to serve as a backlog of tasks for developers, as seen in Appendix D. The provided background information attempts to place developers from either approach at an equivalent starting point.

5.1.1. The Standard TDD Systems

Two undergraduate students who recently completed a course in Software Engineering with a focus on TDD developed the first two systems (TDD1 and TDD2) in Java using JUnit as a testing harness.

Developed independently these two systems present a number of different characteristics. The largest difference comes in the form of error reporting approaches in which TDD1 chose to use an exception scheme, which allows for the precise validation of off-nominal paths, whereas TDD2 places the burden of validation on the caller, eliminating a majority of the development of off-nominal test cases. With that, the design presented in TDD1 is more in line with the requirements, which require for example that a provided student id must be unique. As tested TDD2 functions properly, showing that test coverage does not ensure requirement coverage.

5.1.2. The TFMDD Developed System

For comparison, we develop the TFMDD system (TFMDD1) using RSA with PathMATE and TFMDD Tool generating the implementation into Java with JUnit test cases. Like the other two systems, it was developed as a single Domain (in

the case of TDD1 and TDD2 a single top-level package) with a central access point to the functionality through a Domain Interface. Additionally, like the two TDD systems, TFMDD1 took its own path for error reporting in that each Domain Service returned a fail/pass Boolean result which provided a simple method of validation (as specific error codes or exceptions were not required) in turn improving testability through test case generation.

5.2. Tools

The following tools are utilized in collecting the data during and after the development cycles, presented later in this chapter.

5.2.1. Software Process Dashboard

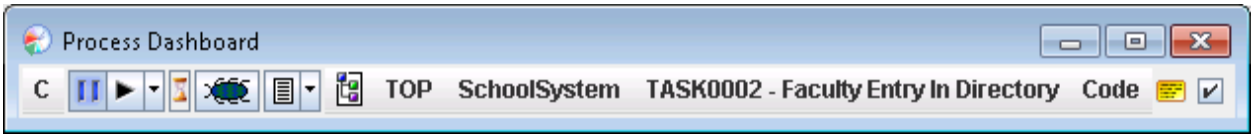


Figure 13 Software Process Dashboard

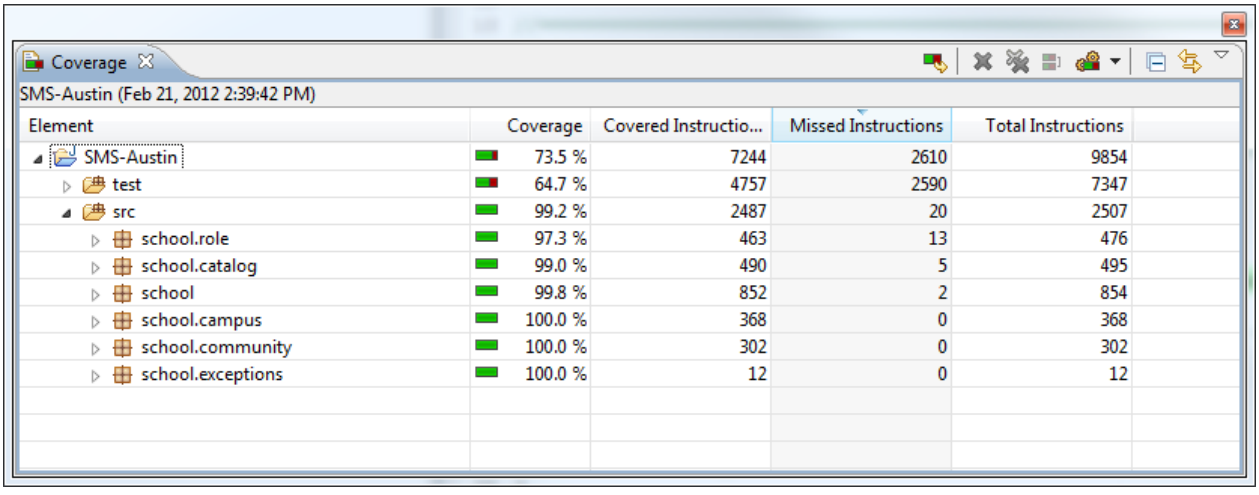
Software Process Dashboard provides a simple interface for capturing time spent in developing each feature of the system[17]. Using a template, each feature of the SMS has a corresponding an entry for which to log time. The template activity entitled “TDD” contains four phases of activity:

- Planning: This phase determines the next feature to implement and captures some initial thoughts of an approach to handling development.

- Test: Part of the micro-cycle, this phase includes all time spent developing test cases and constraints.
- Code: Part of the micro-cycle, this phase accounts for all time spent implementing the system behavior.
- Refactor: Part of the micro-cycle, this category is used to capture refactoring efforts associated with feature development.

5.2.2. EclEmma

Emma is a Java Code Coverage tool which instruments the compiled byte code, so that execution statistics can be captured during test execution[18]. Emma provides the ability to capture branch, line, method, and instruction coverage statistics and produce txt, csv, xml, and html reports. Using EclEmma, an Eclipse integration which utilizes Emma we collect the code coverage metrics for the three systems and generate the corresponding reports [19]. Another reason we choose to use EclEmma, is its ability to provide Cyclomatic Complexity statistics.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
SMS-Austin	73.5 %	7244	2610	9854
test	64.7 %	4757	2590	7347
src	99.2 %	2487	20	2507
school.role	97.3 %	463	13	476
school.catalog	99.0 %	490	5	495
school	99.8 %	852	2	854
school.campus	100.0 %	368	0	368
school.community	100.0 %	302	0	302
school.exceptions	100.0 %	12	0	12

Figure 14 EclEmma Coverage View

5.3. Results

We present the results of examining the collected data in regards to efficiency, simplicity, and quality in the following sections.

5.3.1. Development Effort

The captured development effort required for each of the three systems determines the overall efficiency of each approach. Presented in Table 1 are the total development times collected by each developer during different phases of development, in minutes.

Project	Totals	Planning	Testing	Coding	Refactoring
TDD 1	1254	10	696	446	102
TDD 2	504	109	121	177	97
TFMDD1	603	80	229	253	41

Table 1 Empirical Data – Development Effort (in minutes)

5.3.2. Code Complexity

We measure code complexity against the following metrics:

- The Source Lines of Code (SLOC, see Glossary) count, a measure of the size of a system, which is in turn an indicator of complexity, when a small component is extra-large.
- Cyclomatic Complexity (see Glossary) represents the potential opportunity for error. Additionally the Cyclomatic complexity also has a loose relation

to the number of required test cases to cover each path and the efficiency of the developed test cases (in regards to minimal required test base).

- Cyclomatic Density is the ratio of SLOC to Cyclomatic Complexity used to determine the ratio of how many lines of code result in a different execution, the higher the density the more complex every portion of a component is.

Project	SLOC	Cyclomatic Complexity	Cyclomatic Density	Classes	Test Cases
TDD 1	558	299	0.53	26	309
TDD 2	365	185	0.51	13	37
TFMDD1	1265	582	0.46	15	113

Table 2 Empirical Data - Code Complexity

5.3.3. Code Coverage

Code coverage statistics are measures at the instruction, line, and branch levels. The coverage characteristic varies with each level of measurement:

- Line and Instruction Coverage are as they sounds, the measure of exactly how many instructions or lines (which can contain multiple instructions) present in an application are covered, the most basic coverage measurements. These measurements correlate to covering the complexity present in a system as it relates to SLOC.
- Branch Coverage, which in turn relates to Cyclomatic Complexity, measures all paths through each decision point in the code.

Project	Branch	Line	Instruction
TDD 1	96%	99%	99%
TDD 2	60%	95%	96%
TFMDD1[1]	68%	88%	90%

Table 3 Empirical Data - Code Coverage

[1] As a note, given the nature of generated code the TFMDD1 code coverage statistics apply only to the generated code related to the domain under test (DirectoryServices) the PathMATE portability layer and related framework pieces are not included in these statistics.

Chapter 6 Evaluation

6.1. Overview

The goal of this thesis is to determine if a test generation approach in a modeled environment, such as TFMDD is efficient. We examine efficiency in regards to both the development effort and the overall quality of the developed software artifact. This chapter presents an analysis of the findings of our three-system comparison as they relate to the efficiency of TFMDD.

6.1.1. TDD1 Analysis

The TDD1 System closely matched the requirements and it is apparent that the developer developed the test cases in close relation to them. Without an explicit requirement on how to report error conditions, the use of exceptions did the job and provided an easy avenue for validation. Additionally with refactoring TDD1 has a modular design without significant duplication of code.

With 99% code coverage and utilizing a respectively large test base, there is a perceived code quality carrying a high level of confidence.

6.1.2. TDD2 Analysis

In contrast, the TDD2 System only provides a partial match to the requirements, by providing only the high-level functionality without the restrictions found in the requirements. This problem, the "gap" between requirements and implementation, is one of the primary drivers behind this thesis. Rather than requiring the often-painful mapping of requirements to their implementation counterparts, the generated tests provide evidence of this mapping through their successful execution. However, without the link to requirements standard TDD can suffer from the same disconnect as regular development using coverage based test case generation, having a fully working and covered system that is incorrect, such as TDD2.

Once we inject the additional correct test cases into TDD2, the differences between requirements and implementation becomes apparent. While error reporting is not required and thus its absence acceptable, its absence appears to have contributed to a system implementation that allows changes to the system state in a manner that violates the requirements. For example in TDD2:

- The system allows adding two students with the same unique id.(ADMIN 1)
- A faculty member is incorrectly limited to three course offerings instead of three course offerings a semester.(ADMIN 18)

While the system had a high 95% line based code coverage, the 60% branch coverage highlights the lack of error path testing.

6.1.3. TFMDD1 Analysis

Similar to TDD1, TFMDD1 maps closely to the provided requirements, given their translation into system constraints. This helped to improve the code coverage of the application logic; however, to improve results the PathMATE java templates need to extend the test generation templates to cover errors paths in the infrastructure of the generated code, which modeled action language cannot easily exercise.

6.2. The Effectiveness of TDD and MDD with Test Generation

The following section inspects TFMDD for its effectiveness as compared to standard TDD.

6.2.1. Simplicity

By comparing cyclomatic density, we found that both approaches produced systems with a similar level of complexity with equivalent APIs. While the class count of TDD1 was higher, these additional classes support the exception-driven error handling. These additional classes serve as a possible indicator that without the specific requirement to report errors the development of this feature went against the practice of doing the simplest thing first. TFMDD on the other hand plays to the strengths of doing the simplest thing first, without having default support for features such as throwing exceptions, PI-MDD models are initially built upon the base primitives with the introduction of complex types, only when

required. This influences the design to keep things simple as seen in TFMDD1, using a Boolean to report errors.

6.2.2. Quality

In general, the code quality of the delivered code between all three approaches was acceptable as is expected with a test-driven approach. Although TDD2 did not have sufficient error branch coverage and requirements satisfaction, the developed component performed as coded and without error. Additionally, TFMDD has an advantage of the ability to generate complex reports from the models to facilitate code reviews and additional Quality controls where required (For an example report see the attached PathMATE provided sample).

6.2.3. Testability

Both approaches show the ability to produce testable systems. The construction of all three systems allows state changes to be easily validated using white box testing (see Glossary). TDD1 and TFMDD1 added simple forms of error reporting which facilitated the easy development of off-nominal test cases.

6.2.4. Code Size

In regards to code size, TFMDD1 illustrates one downside to automatic code generation; the fact that code generators are usually not as space efficient as hand written code. However with an open generation framework, such as the

one provided by the PathMATE tool we have the ability to improve this in two ways:

- The software developer has the ability to apply markings (see Glossary) to the model, influencing the code generator to take into account different implementation details that could result in a smaller code base.
- Modify the code generation templates introducing optimizations to generate the code with a smaller footprint.

6.2.5. Requirements Compliance

In regards to translating requirements into system implementation, TFMDD shows that the emphasis constraint development first from the requirements ensures a more correct translation, while TDD without a strict process around requirements mapping presents an increased opportunity for incorrect specification refinement.

Without seeing a significant difference in time spent developing constraints as compared to developing test cases, there appears to be no negative effect on efficiency with the utilization of the test case generator.

Chapter 7 Conclusions

In this thesis, we compare the effectiveness of a standard test-driven process against a tool supported test generation based approach to MDD. In support of this research and testing this thesis, we developed the TFMDD process including the supporting test centric constraint language and corresponding test case generation tool. These developed tools facilitate the generation of test cases into the PI Model prior to implementation. We presented the ability to construct a system using TFMDD and test case generation in a manner compliant with TDD practices. In the development of this system, we grew confidence in TFMDD by iteratively building up each constraint and demonstrating the capability of the TFMDD approach to continually generate additional test cases and as such, result in continual validation of the resulting newly modeled behavior.

Through the analysis of three independent system implementations, we found that it is approximately as effective to generate test cases from requirements based constraints prior to feature implementation developing the System in a platform-independent model as developing the system using traditional TDD.

Additionally, TFMDD provides the following benefits:

- The apparent reduction in the required refactoring effort, probably due to the Modeled nature of the software system, in which the number of

artifacts that have to be updated as part of a refactoring effort are minimized, and reduced to a small number of drag and drop between elements in the model outline.

- The PI nature of both the implementation and the test cases provides the ability to generate the system and corresponding test cases to different deployments (such as implementation language or tasking/process assignments). While providing the opportunity for significant re-use gains the efficiency measurement of the TFMDD approach, does not consider this capability. For example if a team, who developed their component in Java received a request to deliver a C version, would be required to undertake a major re-work effort. This benefit presents itself with first TDD example, as there is no simple conversion of the delivery to C as it relies on an exception mechanism, which would be no longer available. However, the system developed using TFMDD could produce a C version of the component and its corresponding test base with the push of a button.

7.1. Future Work

As many authors have pointed out the TDD micro-cycle works due to an almost instantaneous turnaround time[3] [4][7], whereas the current implementation of TFMDD requires two additional phases as compared to standard TDD test case generation and code transformation. Additionally, in the RSA environment we were required to perform a refresh after each transformation to trigger an instant recompile prior to execution of the new test cases and behavior, causing an additional time cost. While the test case generation can be isolated to only the

constraint under development, a one-click option would be optimal to trigger all four steps, including any necessary refresh and resulting in the test results being displayed to the developer, enhancing the TFMDD micro-cycle and overall productivity.

While there are opportunities to enrich the developed test-centric constraint language, two keywords jumped out during development of the TFMDD system. There is a need for a COUNT keyword to constrain instance counts in both pre and post conditions. The other expansion would be to add MAY EXIST functionality which can be used in preconditions to facilitate richer test case generation with simpler constraints reducing the number of variant definitions.

The TFMDD tool has the ability to integrate a number of features present in existing tools to further industry adoption. One primary shortcoming is the lack of support for automatic and integrated state-machine testing. Another feature key to industry adoption is the ability for mapping requirements directly to the constraints. This would allow the tool to automate the connection between requirements and code, as all the pieces touched by the resulting test cases may be affected by changing requirements. Currently there is no support for mocked objects, which could be facilitated utilizing the constraints of the other models.

Project Setup

Appendix A

Setup the Environment and Project

In order to begin developing a system there are a number of required setup steps:

1. Create an Eclipse project to house the system.
2. Create a PathMATE System Model – to contain the modeled information
3. Add the Appropriate PathMATE Testing Support Domains and Testing Profile to the Model
4. Add PathMATE Java mechanisms to the project.
5. Setup the Java Deployment
6. Add Java/gc and mechanisms folder to the build path.
7. Add JUnit 4 to the classpath

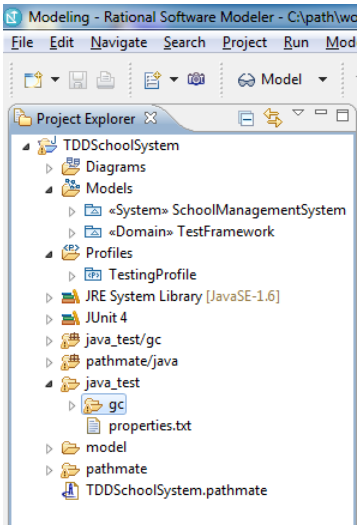


Figure 15 Eclipse Project Explorer

Generated Test Cases

Appendix B: Supplement

Constraint 1 – Generated Test Cases

TestAddStudentCase_1

```
//Setup Parameters
Boolean _RETVAL;

//Setup Initial State

//ExecuteTest
RETVAL = Registrar:AddStudent();

//Validate
TestFramework:CheckEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

Constraint 2 – Generated Test Cases

TestAddStudentCase_1

```
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Setup Initial State

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_2

```
//Setup Parameters
```

```
Boolean _RETVAL;
Integer _student_id =  9999999.0;

//Setup Initial State

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_3

```
//Negative Test Case for Parameter
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Parameter Setup For Error Case
_student_id =  0.0;

//Setup Initial State

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckNotEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_4

```
//Negative Test Case for Parameter
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Parameter Setup For Error Case
_student_id =  1.0E7;

//Setup Initial State

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckNotEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

Constraint 3 – Generated Test Cases

TestAddStudentCase_1

```
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Setup Initial State
Ref<Student> _pre_Student = FIND CLASS Student WHERE (id== _student_id);
TestFramework:CheckEqual(NULL, _pre_Student);

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_2

```
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  9999999.0;

//Setup Initial State
Ref<Student> _pre_Student = FIND CLASS Student WHERE (id== _student_id);
TestFramework:CheckEqual(NULL, _pre_Student);

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_3

```
//Negative Test Case for StateStudent
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Setup Initial State
Ref<Student> _pre_Student = CREATE Student(id=_student_id); //State Setup

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);
```

```
//Validate
//Cleanup Negative State
IF(_pre_Student != NULL){
        DELETE _pre_Student;
}
TestFramework:CheckNotEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_4

```
//Negative Test Case for Parameter
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Parameter Setup For Error Case
_student_id =  0.0;

//Setup Initial State
Ref<Student> _pre_Student = FIND CLASS Student WHERE (id== _student_id);
TestFramework:CheckEqual(NULL, _pre_Student);

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckNotEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

TestAddStudentCase_5

```
//Negative Test Case for Parameter
//Setup Parameters
Boolean _RETVAL;
Integer _student_id =  1.0;

//Parameter Setup For Error Case
_student_id =  1.0E7;

//Setup Initial State
Ref<Student> _pre_Student = FIND CLASS Student WHERE (id== _student_id);
TestFramework:CheckEqual(NULL, _pre_Student);

//ExecuteTest
RETVAL = Registrar:AddStudent( _student_id);

//Validate
TestFramework:CheckNotEqual(TRUE,RETVAL); //Validate the returned result
//Cleanup
```

Appendix C
Constraint Grammar

The following is the constraint grammar developed for this thesis (this as been adapted from its ANTLR form):

```
//------ Constraint Header Section ------//
constraints: (PRE precondition | POST post condition)*  variant*
variant: VARNT LBRACE invariant_body RBRACE
variant_body: (PRE precondition)? (POST post condition)? variant*

precondition: preconstraint*
post condition: postconstraint*

preconstraint: (countConstraint | annotations? exsistenceConstraint | dataInput |
parameterConstraint | associativeClassConstraint) SEMICOLON

postconstraint: (returnConstraint | exsistenceConstraint |
classAttributeConstraint | dataInput   | associativeClassConstraint) SEMICOLON

parameters: Identifier (LBRACE annotations* constraints RBRACE)?;
annotations: Annotation;

//------ Constraint Types Section ------//
countConstraint: COUNT^ classLookup
exsistenceConstraint: NOTW? EXISTS classReference
returnConstraint: rightComparison? RETVAL leftComparison?
parameterConstraint: parameterReference leftComparison?

associativeClassConstraint: Identifier DOT AssociationIdentifier DOT Identifier
leftComparison
```

classAttributeConstraint: Identifier DOT Identifier leftComparison

//------ Constraint Helper Section ------//
classReference: classLookup (ARROW associationReference)?
classLookup: elementIdentifiction attributeConstraint?
associationReference: AssociationIdentifier attributeConstraint?

attributeConstraint: WHERE LPAR (attributeEqualityExpression (COMMA attributeEqualityExpression)*)? RPAR

elementIdentifiction: Identifier
parameterReference: PARAM LPAR (Identifier) RPAR
constraintVariable: Identifier Identifier

rightComparison: comparisonExpression comparisonOperator
leftComparison: comparisonOperator comparisonExpression

//------ Expresion Section ------//
attributeEqualityExpression: Identifier (EQ|NEQ|LT|GT|LEQ|GEQ) (expression | RETVAL)

comparisonExpression: expression;
expression: logicalOrExpression
logicalOrExpression: logicalAndExpression (COR logicalAndExpression)*
logicalAndExpression: equalityExpression (CAND equalityExpression)*
equalityExpression: relationalExpression ((EQ|NEQ) relationalExpression)?
relationalExpression: additiveExpression ((LT|GT|LEQ|GEQ) additiveExpression)?

additiveExpression: multiplicativeExpression ((PLUS|MINUS) multiplicativeExpression)*

multiplicativeExpression: exponentialExpression ((STAR|SLASH|MOD|DIV)^ exponentialExpression)*

exponentialExpression: unaryExpression (EXP exponentialExpression)*

unaryExpression: (MINUS unaryExpression) | (NOT unaryExpression) | primaryExpression

primaryExpression: LPAR expression RPAR |Integer | StringLiteral |FloatConstant | TRUE | FALSE | CharacterLiteral|Identifier |parameterReference

comparisonOperator: EQ | NEQ | LT | LEQ | GT | GEQ

School Management System

Appendix D.

Introduction

This document describes the requirements of the School Management System.

Instructions

This example system is to be developed using test-driven development techniques, in that each task will be taken individually and be developed following the RED-GREEN-REFACTOR approach.

To facilitate the collection of metrics we will be using the Process Dashboard tool. In which each feature will be added to the list as they are undertaken. During development all time spent should be recorded in the appropriate phase of each task.

Assumptions

- For this iteration of the requirements there is no user role validation.
- The user interface will be developed independently.

Backlog

<u>Directory Management</u>

This section describes task related to the management of human resources, students, employees, and faculty.

- As an administrator I would like to be able to create a Student in the System with a unique student id and name. (ADMIN1)
    - Name (Single Non Empty String)
    - Unique Faculty ID (Int - Provided)
- As an administrator I would like to be able to create a Faculty in the System. (ADMIN2)
    - Name (Single Non Empty String)
    - Unique Faculty ID (Int - Provided)
    - Department (String)
- All members of the community should be created with background information including. (ADMIN3)
    - Address (Single Non Empty String)
    - Birth-date (3 Ints representing month, day, and year)
- As an administrator when I add Faculty member I must supply an extension. (ADMIN4)
- As an administrator I can add an Employee. (ADMIN5)
    - A name(single non empty string)
    - A Unique Employee ID (int - Faculty are also employees)
    - A Employment Organization (such as Administration, Facilities, Athletic Department)

Facility Management & Scheduling

This section focuses on managing building resource management and scheduling.

- As an administrator I would like to be able to define campus buildings. (ADMIN6)
    - An unique Name
    - An unique Abbreviation (i.e. Fuller -> FUL or Olin -> OLIN)
- As an administrator I would like to be able to assign classrooms to campus buildings with max student sizes. (ADMIN7)
    - An unique Name
    - An unique Abbreviation (i.e. Fuller -> FUL or Olin -> OLIN)
- As an administrator I would like to be able to assign meeting to campus buildings with max attendance sizes. (ADMIN8)
- As an administrator I would like to be able to assign offices buildings to campus which can be open or utilized. (ADMIN9)
- As an administrator I would like to be able to assign a campus member to an office. (ADMIN10)
- As an administrator I would like to be able to assign a building to a primary department. (ADMIN11)
- As an administrator I would like to be able to assign a building to allow two secondary departments. (ADMIN12)
- As user I would like to like to be able to request an available timeslot for class room. (ADMIN13)

<u>Registrar</u>

The registrar requirements allow for the definition of degrees, courses, course offerings and there relation to students, professors, and facility resources.

- As an administrator or department head I would like to be able to create catalog entries, including departments and courses. (ADMIN14)
- As an administrator or department head I would like to be able disable or re-enable catalog entries, including departments and courses. (ADMIN15)
- As an administrator or department head I would like to be able to create catalog entries, including departments and courses. (ADMIN16)
- As an administrator or department head I would like to be able to create course offerings. (ADMIN17)
- As an administrator I would like to be able to define degrees. (ADMIN18)
- As an administrator or department head I would like to be able to define requirement bins for degrees in my department and the courses that go in them. (ADMIN19)
- As a student I would like to register for an available course offering. (STUDENT2)
- As a student I would like to sign up for the waitlist on a full course offering. (STUDENT3)
- As a student I would like the ability to unregister from a course offering, freeing the spot for a waitlisted student. (STUDENT4)
- As a student I would like to be automatically moved from the waitlist to the class as spots become available. (STUDENT5)

- As a professor I would like to be able to assign a course offering. (PROF1)
- As a professor I would like to be able to accept individuals from the wait-list, provided that the assigned classroom can support the increased occupancy. (PROF2)
- As a user when I create a course offering I would like the system to assign it a room. (USER2)
- As a user I would like to get list of all active courses with in a department (USER3)

Glossary

This thesis uses the following terminology throughout; a number of these terms derive from Platform Independent Modeling methodology.

Constraint Satisfaction Problem

The problem of generating test cases from formal specification[12].

Cyclomatic Complexity

The minimum number of linear paths that in combination, will exercise all possible paths through a method.[20] Each path represents another potential opportunity for error. Additionally the Cyclomatic complexity also has a loose relation to the number of test cases required to cover each path and the efficiency of the developed test cases (in regards to minimal required test base).

Domain

A subject matter domain is a logical grouping of software elements based upon common subject matter base opposed to the more traditional functional grouping and is similar to the concept of a software component.

Domain Service

A domain service is the common and exposed interaction point with to a Domain's internals. This form of data hiding is analogous to a black-box component API routine.

Domain Specific Language (DSL)

A DSL is a language developed to address a specific problem, allowing developers to express the problem space in a simpler format without using a general-purpose language. DSLs can include items such as the syntax of configurations files, shell languages, makefiles, XML schemas, and more.

Fault Based Testing

Fault based testing is the development of unit test cases around exposing potential faults in the system, by driving the system in various manners which would expose the potential fault. A test execution in which the fault does not occur stands for validation of its absence.

Marking

A marking, also known as property can be applied to a modeled system during the transformation process as a way of coloring the system with different implementation aspects. This coloring in turn effects how the generation of code. For example, when the MaxIndex marking is applied an array with a fixed size is utilized in-place of the standard linked list to store references to class instances.

Mock Objects

A test double that is substitutable for a real object to verify function calls, call order, parameter validation. Additionally, mocks provide the ability to script return values.[4]

Mutation Testing

A testing strategy where the insertion of small variations (mutants) into a program should be exposed by the subsequent execution of an existing test suite. If the mutant is not detected additional test cases or revisions to the test suite may be required.[1]

Source Lines of Code (SLOC)

SLOC is a measure of lines of code in a developed application. For this paper, the term SLOC will refer to the logical (or actionable) lines, those containing instructions of Java code in a developed system.

Test Spy

A test spy is a testing tool that provides visibility into data members and state information that would otherwise be unavailable.

White Box Testing

A form of testing in which the internal state of the unit under test can be used for validation.

xUnit

xUnit is a style of assertion based testing.

Bibliography

[1] R. Binder, *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 2000.

[2] Z. Dai, "Model-driven testing with UML 2.0," *Computer Science at Kent*, 2004.

[3] K. Beck, *Test-driven development: by example*. Addison-Wesley, 2003.

[4] J. W. Grenning, *Test Driven Development for Embedded C*. Pragmatic Bookshelf, 2011, p. 310.

[5] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The art of software testing*. John Wiley and Sons, 2004, p. 234.

[6] D. Hamlet, "Connecting test coverage to software dependability," *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pp. 158-165.

[7] J. W. Grenning, *Test Driven Development for Embedded C*. Pragmatic Bookshelf, 2011, p. 310.

[8] M. D. A. G. Version, A. Kennedy, K. Carter, and W. F. X.-change Technologies, "MDA Guide Version 1.0.1," *OMG*, no. June, 2003.

[9] "Concrete Syntax for a UML Action Language," *International Business*, no. August. 2010.

[10] P. Solutions, "Platform Independent Action Language Version 2.2 Reference Manual," *Pathfinder Solutions LLC*, 2004.

[11] A. Z. Javed, P. A. Strooper, and G. N. Watson, "Automated Generation of Test Cases Using Model-Driven Architecture," *Second International Workshop on Automation of Software Test (AST '07)*, p. 3, May 2007.

[12] B. K. Aichernig and P. a. Pari Salas, “Test Case Generation by OCL Mutation and Constraint Solving,” *Fifth International Conference on Quality Software (QSIC’05)*, pp. 64-71.

[13] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, “Automatic test generation: a use case driven approach,” *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140-155, Mar. 2006.

[14] M. Gogolla, F. Büttner, and M. Richters, “USE : A UML-Based Specification Environment for Validating UML and OCL,” *Science of Computer Programming*, vol. 69, no. 1-3, 2007.

[15] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2003, p. 416.

[16] “ANTLR Parser Generator.” [Online]. Available: http://www.antlr.org/. [Accessed: 21-Feb-2012].

[17] “The Software Process Dashboard | The Software Process Dashboard Initiative.” [Online]. Available: http://www.processdash.com/. [Accessed: 21-Feb-2012].

[18] V. Roubtsov, “EMMA: a free Java code coverage tool,” 2006. [Online]. Available: http://emma.sourceforge.net/. [Accessed: 2012].

[19] “EclEmma - Java Code Coverage for Eclipse.” [Online]. Available: http://www.eclemma.org/. [Accessed: 21-Feb-2012].

[20] T. McCabe, “Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,” *National Bureau of Standards*, no. Special Publication SP 500-99. National Bureau of Standards, 1982.