A Test-Driven Approach for Developing Software Languages

Omar Badreddin, Andrew Forward and Timothy C. Lethbridge
School of Electrical Engineering and Computer Science (EECS), University of Ottawa, Ottawa, Canada

Keywords: Test Driven Development, Model Oriented Programming Language, UML.

Abstract: Test-Driven Development (TDD) is the practice of attempting to use the software you intend to write, before you write it. The premise is straightforward, but the specifics of applying it in different domains can be complex. In this paper, we provide aTDD approach for language development. The essence is to apply TDD at each of four levels of language processing, hence we call our approach Multi-Level TDD, or MLTDD. MLTDD can be applied to programming languages, preprocessors, domain specific languages, and transformation engines. MLTDD was used to build Umple, a model-oriented programming language available for Java, Ruby, and PHP. We present two case studies where this approach was implemented to develop two other domain specific languages.

1 INTRODUCTION

Test Driven Development (TDD) (Beck, 2002), and its similarly-named practices (Test First Development, Test Driven Design, and Behaviour Driven Development (BDD)) provide a trusted strategy for software development regardless of the development lifecycle. The benefits of TDD are well documented in (Gupta & Jalote, 2007). The focus in this paper is not to convince the reader of the value of TDD, but rather to demonstrate a TDD approach geared for software language development.

The main contribution of this paper is what we call Multi-Level TDD (MLTDD), which means applying TDD to each level of the language processing pipeline.

MLTDD has been successfully used to develop Umple (Badreddin, 2010), (Lethbridge et al, 2013), (Lethbridge et al, 2010), a general purpose model-oriented programming language. It has also been used to create Appstats, (Forward, 2012) a DSL for managing usage statistics and Osl, a proprietary DSL for describing network topologies. The examples shown in the paper focus primarily on Umple, publicly available at (Lethbridge et al, 2012). MLTDD has been extensively tested in Umple development and has helped to keep Umple's overall quality high, and to facilitate its evolution.

The paper is organized as follows. We first introduce the components of a language processor and describe how TDD can be streamlined for each component. We explain how this manages defects and reduces regression by presenting a hypothetical bug scenario. We then explain how this approach was applied in practice.

2 DESIGN A NEW LANGUAGE BY TESTING: MLTDD

The number of domain specific and general-purpose software languages is increasing. There are several reasons for creating such languages (Gronback, 2009). An approach to building new languages is shown in Figure 1. The exact architecture of each language processor may differ, but this workflow provides a typical view. Variations of this architecture were used in developing Umple, Appstats, and Osl.

The four common steps in processing a language are given below. Each step can be encapsulated in a component in the language processing architecture:

- **Parse the Source** – Processing occurs according to a grammar that finds matches for the language constructs. The language text (or textual representation of a visual language) is consumed by the parser, which translates it into an an abstract syntax tree (AST). The AST remains a syntactic view of the system.

Badreddin O., Forward A. and Lethbridge T..
A Test-Driven Approach for Developing Software Languages.
DOI: 10.5220/0004699502250234
In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2014), pages 225-234
ISBN: 978-989-758-007-9

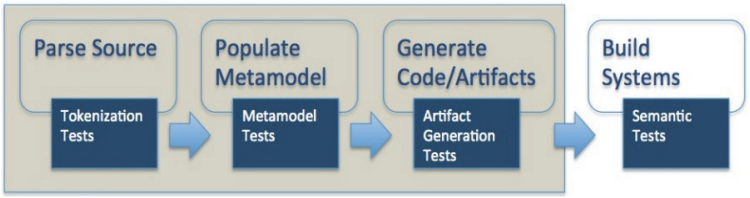

Figure 1: Building a new language.

- **Populate the Metamodel** – The AST is analyzed and an instance of the language's metamodel is constructed. The metamodel includes additional semantic details.
- **Generate Artifacts** – The metamodel instance is used to generate work products of the language, which may include code generation (e.g. for preprocessor languages like CoffeeScript, Sass, or Umple), or runtime code for virtual machines (e.g. Java object-code, or ASP.Net CLI). Alternatively, runtime interpretation of the metamodel instance may occur (e.g. Ruby, Bash, or SQL).
- **Build Systems** – Now that we have a new language, we build real systems (the purpose for creating the language). For general purpose programming languages this may include using the new language to build itself (i.e. using the Umple language to create the Umple compiler).

For simple languages, generating language artifacts could be performed directly against the AST, skipping the second step . But in practice it is worthwhile to introduce an intermediate stage (the metamodel) to decouple the semantics of the language from its specific syntax.

More complex languages may include modules such as a debugger, integrated-development tools and meta-language extension mechanisms. The focus of this paper remains on the four steps above and the components that implement these steps. These form a general architecture for software language development.

The following sections investigate each of the four components and demonstrates how TDD is applied, using Umple as a case study.

2.1 Parser Testing

The primary intent of the parser is to correctly interpret the source code according to the language's syntax. Driving parser design through tests requires explicit test cases to make these assertions: a) Asserting the parser properly tokenizes the input, and b) Asserting the parserproperly populates the metamodel instance

A template for testing parser in JUnit is:

```
@Test
 public void someSyntaxToVerify() {
  // Step 1: Load the source code
  // Step 2: Parse file and assert its success
  // Step 3: Assert correct tokenization
  // Step 4: Clean up }
```

A simplified excerpt from the Umple codebase is shown below that demonstrates the proper parse of a simple Umple class.

```
@Test
public void emptyClass() {
 String input = "class Student{}";
 String expectedOutput =
  "[classDefinition][name:Student]";
 UmpleModel model = new
 UmpleModel(new UmpleFile("empty.ump"));
 UmpleParser parser = new UmpleParser(model);
 boolean answer = parser.parse(
  "program", input).getWasSuccess();
 Assert.assertEquals(true, answer,
  "Unable to parse Umple code");
 answer = parser.analyze(false).getWasSuccess();
 Assert.assertEquals(true, answer,
  "Unable to analyze Umple code");
 Assert.assertEquals(expectedOutput,
  parser.toString()); }
```

This tests the empty class case; a class with no content except possibly white spaces and new line characters.

The parser testing must also handle invalid input, or negative cases. The extent of negative test cases depends on the proficiency of the target audience, the complexity of the language and the desired debugger feature set. A template for testing invalid input is shown below:

```
@Test
public void someSyntaxErrorToVerify()
{
  // Step 1: Load the source code
  // Step 2: Parse file and assert its failure
  // Step 3: Assert correct error in formation
  // Step 4: Clean up }
```

2.2 Metamodel Testing

The objective of metamodel testing is to ensure that the compiler is able to maintain a valid internal representation of the input language. It can be argued that parser testing (above) and artifacts generation testing (discussed on the next section) are sufficient. However, from experience we find that testing the metamodel instance is crucial step to help maintain the language as it evolves, and facilitates debugging by isolating issues into either problems with syntax analysis (testing the parser) or language metamodel semantics (testing the metamodel).

Metamodel tests use the following standard unit testing pattern.

```
@Test
public void someSpecification() {
  // Step 1: SetUp
  // Step 2: Execute
  // Step 3: Verify
  // Step 4: TearDown }
```

It is important to document not only how the system behaves under normal conditions, but also how it behaves in abnormal scenarios where, for example, preconditions are not satisfied.

Here is a sample test case for the Multiplicity metamodel class. Below we see that setting the range on a Multiplicity properly sets both the upper and lower bound.

```
@Test
public void setRange_ExplicitBounds() {
 Multiplicity m = new Multiplicity();
 m.setRange("1","2");
 Assert.assertEquals(1,m.getLowerBound());
 Assert.assertEquals(2,m.getUpperBound());
}
```

Some may question the value of such simplistic tests. It should be noted that the test is merely an example to demonstrate the structure of a metamodel test. But, more importantly, the spirit of following test-driven design (as well other driven approaches) is the concept of evolving design through tests. By following a test-driven approach, the tests (and the ability to run them over and over again in an automated fashion) is a welcome side effect, but the true power of the approach is in the initial design whereby you first exploit the common uses of your metamodel and only then do you concern yourself with the implementation.

2.3 Artifact Generation Testing

The end result of any language compiler is a set of generated artifacts. Take Java for example, the Java interpreter generates byte code. Even for visual languages such as UML, the generated artifacts can be either high level programming language code (such as Java or C++), or XML based artifacts (such as XMI) that are used for saving and interchanging models. The objective of this category of tests is to ensure that the compiler is able to generate artifacts that match what is expected.

Many of the defects of the language processor are likely to be discovered against these types of tests. While parser and metamodel tests (are related code) are important to continued success of your project, the desired output of your language is really the generated artifacts (e.g. byte code).

Umple generates a wide range of artifacts. Given an Umple source, the compiler generates a number of high level programming languages code (Java, PhP, Ruby, and C++). The compiler also generates XML based artifacts, such as Ecore, and Papyrus XMI, as well as other artifacts such as SQL and TextUML. We explain here the testing platform to support this wide range of artefact types.

The code generator typically takes a populated metamodel instance as input. The output is one or more target artifacts or a transformation into another modeling syntax. In addition to overall setup and tear down, the high-level approach to testing code generation is shown below.

```
@Test
public void verifyGeneratedCode() {
  // Step 1: Prepare Metamodel
  // Step 2: Run Code Generator
  // Step 3: Verify results }
```

To prepare the metamodel instance, there are two approaches: populating the model by direct calls to the available API of the metamodel (or perhaps using a mock object facility; or, parsing source code using the existing infrastructure.

The first approach (to populate the metamodel instance directly using the API) has several drawbacks: The setup code can become quite cumbersome and complex, which could make the test code less readable and maintainable. And, it can also be error prone, as the testing developer must properly populate the metamodel prior to testing. The primary benefit of this approach is the isolation of the code generator's behaviour from that of the code that parsed the source and generated the

metamodel instance. Issues related to the parsing phase (translating code into a populated metamodel instance) would not interfere with testing the code generator.

For example, the following code creates a Student class with three attributes (id, name, and program).

```
UmpleModel m = new UmpleModel(null);
UmpleClass student =  m.addUmpleClass("Student");
student.addAttribute(
  new Attribute("id","Integer", null,null,true));
student.addAttribute(
  new Attribute("program","String", null,"SEG",false));
```

The second approach where the test uses raw source code that is then parsed into the Metamodel instance (which is then used as input to the code generators) has two primary benefits. First, it is easier to express a system in its own syntax as opposed to building it using a metamodel's API. Second, you provide an important integration between the external inputs and outputs of your language. The obvious drawback is that these tests are no longer pure unit tests, and that failing tests in this component could be resulting from the parser or the code generator.

Here is the same example from above written using the source language (Umple) syntax directly.

```
class Student {
  Integer id;
  name;
  program = "SEG"; }
```

Regardless of the approach, the metamodel instance must be populated before the code generation can be tested. Instead of crafting a new means to populate that model, we favour the more pragmatic approach of simply reusing the existing (and tested) parsing approach as described in the previous section.

By parsing the model code directly, an added benefit is that you can create a generic TemplateTest to manage the test artefacts (i.e. input model code, expected output system code); leaving the testing mostly boilerplate-code free.

The outline of such a template class is shown below.
The method signatures will vary slightly depending on the type of code generator that is being created; but the overall structure remains intact.

With the infrastructure shown above in place, adding new code generation tests is straightforward, as the template encapsulates the distracting elements of the test setup. A sample code generator test is shown below.

```
public class TemplateTest {
@Before
public void setUp()
{
  // configure paths to Umple data files
  // this can be configured to support
  // multiple languages }
@After
public void tearDown() {
  // clean up any temporary or generated files }
public void assertTemplate (
  String modelFile,
  String expectedGeneratedFile) {
  // Parse / tokenize modelFile
  // Create an instance of meta model
  // Generate code for the underlying system
  // Compare the actual generated code
  // with the expectedGeneratedFile } }
```

```
@Test
public void Association() {
  assertUmpleTemplateFor("AttributeTest.ump",
  languagePath + "/ AttributeTest."
    + languagePath +".txt","Student"); }
```

The test above requires a model file (AttributeTest.ump), as well as a source code file based on the selected language. In Umple, we currently support Java, PHP, Ruby, and C++. Using the test case above, the same model file can be reused to test against all the supported languages. This infrastructure can easily be extended to add testing for other generated artifacts.

2.4 Testing of End-user Systems

The previous sections described testing the language compiler / code generator itself in what can be termed white-box testing. So far, we only asserted that the system outputs what we expect, but not necessarily what the target platform requires. This class of testing only applies if the language compiler generates either executable artifacts (i.e, bytecode), or artifacts that themselves can generate executable artifacts (i.e, Java or C++), or artifacts that serves a tangible purpose (i.e, an XMI artifact that is used for model versioning or interchanging).

The objective of this category of tests is to ensure the appropriate behaviour of the resulting system. This is a powerful concept because it enables the language developer to assert the semantics of the generated system (not just its syntax).

The semantics of Umple's modeling components are quite rich so it is important to provide adequate testing of generated systems to ensure that the

semantics of an Umple model is upheld in the underlying base language (i.e. Java, PHP or Ruby). This level of testing ensures the appropriate behaviour of the generated Umple executable artefacts, which is essential to support our industrial case studies.

Let us consider a simple example of testing the semantics of a class attribute.

```
class Student { name; }
```

The specifications for an attribute as defined above include the following properties and behaviours: the attribute is included as a constructor argument, the attribute can also be modified and retrieved. Based on the above description of an attribute, we could write the following tests (the tests are written using JUnit4 syntax).

```
@Test
public void attributeBehaviour() {
 Student s = new Student("james");
 Assert.assertEquals("james",s.getName());
 s.setName("henry");
 Assert.assertEquals("henry",s.getName()); }
```

This test can be equally expressed in PHP using PHPUnit (an xUnit testing framework for PHP applications) as shown below.

```
public function test_attributeBehaviour() {
  $s = new Student("james");
   $this->assertEqual("james",
   $s->getName());
   $s->setName("henry");
   $this->assertEqual("henry",
   $s->getName()); }
```

By capturing the properties and behaviour of systems built with Umple, we are able to build up an extensive library of executable specifications which more concretely demonstrate the realized behaviour of the system, as opposed to its documented behaviour (and as is common knowledge amongst most software practitioners, it is common for documentation to quickly get stale and out of sync).

Tests under this class are not limited to simple systems. In the case of Umple for example, the whole of Umple compiler and development environment are implemented using Umple itself. In that sense, Umple provides itself is a very large system test case (as each new version of Umple is re-tested against itself to ensure it continues to abide by it's own semantics).

Tests in this category provide both an excellent test-bed for experimenting with code generation techniques (i.e. testing the effects of changing the code generation on the resulting system), as well provided added confidence generated systems (not just your own language platform) work as expected. Meanwhile, test failures in this category potentially identify issues in any of the previous test categories and will most likely require additional exploratory tests to uncover the true cause of the semantic error.

3 MANAGING DEFECTS AND MINIMIZING REGRESSION

Despite having over 2,500 automated Umple tests that span all facets of the toolset from parsing to code generation, any compiler inevitably will contain defects. In this section, we discuss how the testing infrastructure described above allows for better defect management by representing bugs as failing tests, effectively diminishing the time and effort required to perform regression testing.

When a defect is uncovered, it might be one of the following:

1. Defects in the way in which the language is tokenized into an abstract syntax tree
2. Incorrect population of the metamodel instance from the tokenized language.
3. Inappropriate behaviour of the metamodel classes.
4. Syntax errors in the generated artefacts.
5. Semantic errors (i.e. incorrect behaviour) in the generated base code
6. Execution errors in the generated systems.

In addition to the defect scenarios above, there is always the possibility of usability defects. Dealing with this type of defect is outside the scope of our work, and instead, we focus our attention on well-defined, repeatable issues.

As defects are uncovered, it is not always apparent which category of defects has been uncovered. Where the root cause of a defect is unknown, it is recommended to resort to bottom-up defect resolution. We start by verifying the tokenization, then the metamodel, followed by the generated artifact, and finally, the system level testing.

3.1 Step 1: Identify the Problematic Input Language Code and Expose It with a failing Test

The first step is to identify the component

responsible for the problematic output. This process may involve a few iterations to isolate the exact symptoms causing the issue, but that is not always necessary.

For illustration purposes, our sample defect is that un-typed Umple attributes are not reflected in the generated code. The following code shows the potentially problematic Umple code.

```
class Student { id; }
```

The test case would perform an end-to-end high level test that properly documents the identified issue with a failing test.

3.2 Step 2: Verify the Tokenization of the Problematic Umple Code

With our high-level failing test in place, we now analyze each step of the process to identify the root cause of the problem. We start with the Umple parser.

The process for verifying the parser is already available. We simply add an additional test using the problematic Umple code as the input and we verify the output.

To continue with the example above, we first make sure that un-typed attributes are properly parsed and tokenized. The expected result *[class][name:Student][attribute][name:id]* represents a toString view of the tokenization sequence used to assert equality in a human readable form.

```
@Test
public void untypedAttributes() {
  assertParse("untyped.ump",
    "[class][name:Student]"+
    "[attribute][name:id]"); }
```

If this test fails, we resolve it and re-run our test from Step 1. If that test succeeds, then it is likely that the problem is now properly resolved and the debugging process is complete. If not, then we move on to the next step.

3.3 Step 3: Verify the Instance of the Umple Metamodel

Once the Umple source has been shown to parse correctly (but that the observed issue persists), we then validate that the instance of the Umple metamodel is consistent with the Umple input. Here, we are ensuring that the metamodel was properly populated following the parser tokenization process.

To test the metamodel, we enhance the test identified in Step 2 as follows.

```
@Test
public void untypedAttributes() {
  assertParse("untyped.ump", "[class][name:Student]"+
     "[attribute][name:id]");

  UmpleClass aClass = model.getUmpleClass(
    "Student");
  Assert.assertEquals("Student", aClass.getName());
  Attribute attr = aClass.getAttribute("id");
  Assert.assertEquals("id", attr.getName());
  Assert.assertEquals("String",  attr.getType()); }
```

Here, we assert that the Student class is created, and that it has an attribute of type String with the name id. If this test fails, we follow the same procedure starting from the previous step: re-test our high-level test and proceed to the next step only if that test still fails.

3.4 Step 4: Validate the Proper Behaviour of the Metamodel

Once the Umple code appears to be parsed correctly, and the metamodel is properly populated, we then investigate if there is any special behaviour that is performed by the metamodel instance that may not be handled properly.

For example, an Attribute has an operation isPrimitive which checks for the Umple predefined types, and perhaps this operation is not functioning as expected. Below is a test case demonstrating the expected behaviour.

```
@Test
public void isPrimitive() {
  Attribute av;
  av = new Attribute("a",null,null,null,false);

  Assert.assertEquals(true, av.isPrimitive());
  av.setType("String");

  Assert.assertEquals(true, av.isPrimitive());
  av.setType("Address");

  Assert.assertEquals(false, av.isPrimitive()); }
```

Dealing with this type of testing is difficult to categorize, and each scenario will need to be analyzed individually. If the behaviour of the metamodel appears to be working correctly (but our high-level test still fails), we continue to the next step.

3.5 Step 5: Compare the Expected versus Actual Generated Code

Next, we analyze the expected code versus actual generated code. Here, we are testing that the syntactic translation of the Umple metamodel instance into the generated base language is correct.

The example test case would resemble the following code.

```
@Test
public void untypedAttributes() {
 assertUmpleTemplateFor("attribute.ump",
  "attribute.java.txt", "Student"); }
```

Where the "attribute.ump" would be the problematic Umple code and the "attribute.java.txt" would contain the desired Java code to be generated from the model.

3.6 Step 6: Test the Behaviour of the Generated Code

If all other tests are passing successfully, the final aspect to testing the Umple system is that the generated code conforms to the semantics of the model. It might be the case where we are producing what is believed to be the correct code, when in fact the generated code does not behave as the intended by the model.

In our on-going example presented above, we write unit tests against a sample application that contains a Student with an id attribute.

```
@Test
public void constructor() {
  Student s = new Student("x");
  Assert.assertEquals("x",s.getId()); }

@Test
public void setAndGetStringAttribute() {
  Student s = new Student("x");
  s.setId("y");
  Assert.assertEquals("y",s.getId()); }
```

The steps outlined above provide a high-level approach to deal with issues as they arise. The most important first step is to create a failing test that exhibits the invalid behaviour of the system. The granularity of this test is not that important, as we have developed a systematic approach to verify each step of the Umple compiler to help determine the root cause of the issue.

By adopting a test-driven approach, the regression test suite continues to grow to deal with new issues and at the same time mitigate the risk of regressing on existing functionality. In the next section, we look at how the current infrastructure supports future potential enhancements.

4 APPLYING MLTDD TO A LANGUAGE WRITTEN IN ITSELF

As the core features of the language become available, a new feature can be described as a defect; in other words, something is missing that should not be missing. The approach of adding new features should follow a similar path as described in the previous sections whenever possible. In this section, we describe how to manage new features by means of 'dog-feeding' examples.

By following the steps described above, it is possible to specify the desired behaviour of a new language feature without that feature being available yet. And if the language is written in itself, it is required to implement most behaviour using the existing language constructs.

In the example below, we demonstrate this process using Umple. The example illustrates adding an OCL-like constraint syntax to be specified against attributes. The semantic test below assumes the OCL constraint that age >= 18 for any Student.

Before implementing the change, we first write a test in the base language demonstrating the desired functionality. This test would need to be translated into each language supported. Below is a sample test written for Java.

```
@Test
public void cannotSetTo17() {
 Student s = new Student(18);
 Assert.assertEquals(18,m.getAge());
 Assert.assertEquals(false,s.setAge(17));
 Assert.assertEquals(18,m.getAge()); }
```

Next, we use Umple itself to implement the feature using existing constructs.

```
class Student {
  Integer age; before setAge {
    if (aAge < 18) { return false; }
  }
  after getAge {
    if (age < 18) {
     throw new RuntimeException("Age must be >= 18")
    }
  } }
```

Once the behaviour is validated with sufficient (and

passing) tests within our base languages, we then enhance the parser and metamodel with the new language constructs.

The potential Umple syntax might look like the following.

```
class Student {
  Integer x;
  // this is the potential invariant syntax
  [x >= 18] }
```

Next, we migrate the custom code written in the behaviour tests into the code generation process to validate the generated syntax. Following that, we deploy a new version of Umple itself and update the original behaviour tests to use the new language constructs (as opposed to having to write the behaviour by hand, as was required before the feature was available). These tests themselves remain relatively unchanged; we simply update the tests to use the new language constructs.

In pure TDD methodology, the process is not just about testing; but rather about *designing* the system in a modular fashion maintaining low coupling and well-defined interfaces. The process is also about capturing the intention of the software (i.e. automated tests) that can be easily verified (i.e. re-running the test suite) effectively enhancing reusability.

For example, the act of manually testing and modifying (i.e. debugging) an application until it works benefits only the developer performing the task. It cannot be replicated easily, as the debugging steps are not documented and are lost once the debugging exercise is complete. Conversely, by capturing the testing process through automation, all developers can benefit as knowledge is gained about the true behaviour of the system that can be easily re-run and re-verified.

In the case of building a new programming language (or in the case of Umple, extending existing base languages), we first need to be concerned with testing the tooling itself. But, because the outputs of such systems are systems too, they can also be tested (i.e. semantic testing of systems generated using the new language).

In addition, because Umple is implemented in itself, we are able to capture the debugging effort of new code generation behaviour in automated tests, and then modify the underlying Umple language to replicate that behaviour natively, as shown with the OCL constraint example. In summary, we enhance the Umple language so that we can refactor Umple (which is written in Umple) to make use of the enhanced language elements; 'eating our own dog food', so-to-speak.

5 CASE STUDIES

To further explore and validate our MLTDD approach, we applied it to two industrial projects, Appstats and OSL. These were undertaken in a software company specializing in the development of online process solutions.

5.1 Case Study 1: Appstats

Appstats (Forward, 2012) is a small open-source logging and statistics library that provides a "counting" framework with features such as built-in caching, delayed processing, as well as the creation of ad-hoc and scheduled reporting. The language provides a simple, yet effective, DSL. The source code is about 2 KLOCs with 97% coverage from over 650 tests. The basic structure of an appstats query is

 # <action> <timeframe> <host/server> <context filter> <group filter>

Actions are user defined and could include things like # logins, # objects created, # exceptions. Date ranges support several formats such as "between Mar, 2010 and Mar, 2011", "today", "last year|month|week|day", etc. The host/server simply allows the user to grab statistics from a particular server such as testing, versus staging versus production. Finally, all data logged by appstats is tagged with, and is searchable / groupable by any number of contexts (as defined by the user - not appstats), examples include tracking the logged in user, the address searched, number of results found, duration of request etc.

By instrumenting an application appstats logs, you enable very powerful and efficient queries with a simple DSL. In addition, developers can write 3rd party plug-ins to augment the "logging" statistics with other data sources to facilitate a uniform API between the raw data and any application reporting functionality.

The approach used to test Appstats was the same as that used by Umple. Below, we discuss the unique characteristics of testing the statistics collection mechanism of Appstats. Just like any other type of test, you need to emulate the situation you are testing, and then verify that the right things has occurred. For Appstats statistics, there are a few items that require configuration, otherwise it is no different than testing other aspects of the system.

The examples below are written in Ruby using the RSpec testing framework.

1) Reset the logged and simulate the current time.

This will ensure the log starts empty, and that your logs will occur at a specific (and testable) time. You will also want to remove the log file after each test to ensure all tests start with a clean slate.

```
before(:each) do
  Appstats::Logger.reset
  Appstats::Logger.filename_template =
   "test_appstats_lookup_controller"
  Time.stub!(:now).and_return(
    Time.parse('2010-09-21 23:15:20 UTC'))
end
after(:each)do
  File.delete(Appstats::Logger.filename)
If File.exists?(Appstats::Logger.filename)
end
```

2) Write the test that should "gather" some type of statistic.

```
get :lookup, :address =>
  "123 Victoria Ave, Ottawa, K1P 1P2",
  :radius => "0.05"
    Appstats::Logger.raw_read.should ==
  Appstats::Logger.entry_to_s(
  "buyer-address-lookup",
  :accuracy => "4",
  :area => "Ottawa area",
  :total_results => 123 )
  end  end
```

Each project requires unique testing, and the above demonstrates the use of the appstats API. Such testing is not (yet) required in Umple as Umple exposes itself only as a language (and related tools), whereas Appstats is both a language and an API.

5.2 Another Case Study: OSL

Another language that has benefited from the TDD approach described in this paper is OSL. OSL is a proprietary language developed by CENX Inc. to manage the inventory, ordering and monitoring of Ethernet back-haul networks. OSL is comprised of a mark-up language, using YAML as its base as well as a runtime environment which is its own set of commands. The description language provides the building blocks to describe network topologies whereas the runtime language helps to compile telecom planning spreadsheets into actual network assets. OSL has also been used to help large telecom companies properly complete circuit orders.

OSLs use of dynamic language constructs such as multiple inheritance, object mix-ins and duck-typing have allowed network engineers to build various network topologies between various organizations with little duplication. To date, OSL is about 7 KLOCs of production code and are 96% covered by 1100 automated tests, structured using MLTDD.

6 RELATED WORK

Since agile methodologies emphasize the delivery of working executable code in a repeatable manner, the notion of automatically testing these deliverables is appealing. Since the emergence of such methodologies, one can notice the increase of work on the value of TDD. The literature is rich with work reporting and assessing the benefits of test driven development methodologies. One study in IBM reported that adoption of TDD has reduced the number of defects by 50% (Maximilien & Williams, 2003). Our work goes beyond arguing for or against TDD, but rather studies how TDD can be applied to a specific application domain (Forward & Lethbridge, 2008), software language development.

Steel (Steel & Lawley, 2004) reports on a study of the methodology of TDD of a model transformation engine. Similar to our work, steel's study emphasizes the importance and value of TDD, emphasises the need of structuring tests. Steel also concludes that there is a need for maintaining tests in a structured format (i.e, in XML) rather than programmatically as we have done in Umple.

One core difference in approach between our work and Steel's is the way tests are structured. We advocate structuring tests around application tiers or components (i.e, parsing, metamodel instance creation, code generation and system creation). Steel, on the other hand, advocates structuring tests around features. Our view is that at the high level, tests must be streamlined and organized by level. Within each level, tests can be grouped by feature and sub-features. The benefit of the high-level grouping by levels is evident in the following:

- Bugs and features are organized and reported against levels (parser, etc.).
- A bug report against a feature often spans a number of levels. Further work is therefore needed to isolate the concerned level.
- As a language grows, features overlap. If tests were grouped by feature it is not always clear which tests belongs to which feature.

7 CONCLUSIONS

This paper has presented a methodology we call

multi-level TDD (MLTDD) for applying Test Driven Development to software languages. Motivated by the success we achieved with applying MLTDD to Umple, as well applying it to other projects including Appstats and OSL, we strongly believe that MLTDD can and should easily be applied to all general purpose programming languages, domain languages, model-to-model transformations and model-to-code transformations.

REFERENCES

Badreddin, O. "Umple: A Model-Oriented Programming Language," *in Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, 2010. pp. 337-338.

Beck, K. *Test Driven Development: By Example. Boston, MA, USA*: Addison-Wesley Longman Publishing Co., Inc, 2002.

Forward, A. and Lethbridge, T. C. *"A Taxonomy of Software Types to Facilitate Search and Evidence-Based Software Engineering,"* in CASCON '08: Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research, 2008. pp. 179-191.

Forward, A, 2012. *Appstats. Accessed 2013*. https://rubygems.org/gems/appstats.

Gronback, R. C. "*Eclipse Modeling Project: A Domain-Speci c Language (DSL)Toolkit*". 2009. Addison-Wesley Longman.

Gupta, A. and Jalote, P. "*An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development*". 2007. Empirical Software Engineering and Measurement, 2007.ESEM 2007.First International Symposium on, pp. 285-294.

Lethbridge, T. C., Forward, A. and Badreddin, O. " *Umple Language Online*.", accessed 2013, http:// try.umple.org.

Lethbridge, T. C., Forward, A. and Badreddin, O. "*Umple Google Code Project*". 2012. Available: code.umple.org.

Lethbridge, T. C., Forward, A. and Badreddin, O. "*Umplification: Refactoring to Incrementally Add Abstraction to a Program*," in Working Conference on Reverse Engineering, 2010. pp. 220-224.

Maximilien, E. M. and Williams, L. "*Assessing Test-Driven Development at IBM*," in Software Engineering, 2003. *Proceedings. 25th International Conference on, 2003*. pp. 564-569.

Steel, J and Lawley, M. "*Model-Based Test Driven Development of the Tefkat Model-Transformation Engine"*. 2004. *15th International Symposium on Software Reliability Engineering*, pp. 151-160.