DOMAIN SPECIFIC LANGUAGE FOR UNIT TESTING

Andreea CHIPER[1], Denis SMOCVIN[1], Cezar GUZUN[1], Vladimir LUCHIAOV[1], Anatolie TELUG[1]

[1]*Department of Software Engineering and Automation, group FAF-212, Faculty of Computers, Informatics and Microelectronics, Technical University of Moldova, Chisinau, Moldova*

*Corresponding author: Andreea Chiper, andreea.chiper@isa.utm.md

Scientific coordinator: Vasile DRUMEA, university assistant, DISA

Abstract. *The purpose of this article is to present a project based on the development of a domain specific language for unit testing. Unit testing is the process of testing individual pieces of code. Even while it is incredibly useful, it has a few drawbacks that can be solved by developing a domain-specific language. It will be related about the importance of a domain specific language in unit testing and how to create one so that users can get the most out of it.*

Key words*: domain specific language, bugs, developers, module, parser, grammar, suite.*

Introduction

A domain specific language (DSL) is a collection of functions named after intuitively obvious behaviors in a certain environment. A customized language that focuses on a certain issue domain. It defines the problem at a greater level of abstraction than general purpose languages. It simply presents another angle on the same issue.

Unit testing is a testing technique that isolates individual modules to find, analyze, and remedy any faults [1]. The goal is to ensure that each unit of software code operates as planned. The developers test an application throughout the development named code phase. Unit tests isolate and check the validity of a section of code. A unit is a single function, method, procedure, module, or object.

By decreasing errors, enhancing architectural conformance, and increasing maintainability, domain specific languages can improve product quality. It can help specialists write more expressive and legible test cases, reduce repetitive code, enforce domain-specific rules, and improve communication and cooperation. It streamlines the writing, management, and execution of tests, resulting in higher quality, consistency, and automation.

Domain analysis of unit testing

Testing is a wide and important domain in application development. Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation [1].

Unit testing is important because software developers often try to save time by performing minimal unit testing; however, this is a myth because insufficient unit testing leads to high-cost defect fixing during system testing, integration testing, and even beta testing after the application is built [2]. Proper unit testing during early development saves time and money in the long run.

The following are the primary reasons for performing unit testing in software engineering:

- Unit tests aid in the early detection of errors and the reduction of development costs.
- It allows developers to quickly comprehend and improve the testing code base.
- Unit tests help with code reuse.

As any other type of product, when developing a language, it has to be taken into consideration the potential users that will be interested in the service.

As it may be believed, only testers are going to use it because their job involves testing the developed software product and unit tests are an often used technique for doing tests.

Sometimes a team doesn't have a tester, which means that one software developer or a group of developers will take on the responsibility to perform automated tests on the developed software [3].

Similar to software developers, library developers, when working on a library its individual units should be tested to ensure that the many users it will have will not have any problems, errors or crashes while using the library.

Although it is not usually the case, cybersecurity specialists may write unit tests to validate the behavior of security-critical components in the code, such as encryption and authentication modules.

Unit testing challenges that a DSL could solve

Unit testing is the process of evaluating individual software modules or components to ensure they work as expected. A domain specific language can help unit testing in a variety of ways, as specified below.

Improved readability: a DSL for unit testing can offer a shorter, clearer syntax, making it simpler to design, comprehend, and manage tests. This can improve the accuracy and reliability of unit tests and make it simpler to find and address problems [4].

Abstraction: a DSL can provide a level of abstraction that makes testing complicated or challenging to understand code simpler.

Customization: it is simpler to develop tests that are pertinent and useful for a given context when a DSL has been adapted to the particular requirements of a project or team. This can enhance the efficacy and quality of unit tests and point out areas where the code should be better managed.

Improved teamwork: a DSL can make it simpler for members of the team to work together on unit testing.

Improved automation: by offering a clear and uniform syntax that automated testing tools can quickly parse, a DSL can make it simpler to automate unit testing. This makes it simpler to detect faults and regressions early in the development cycle and can speed up, reliably, and scale unit tests.

Basic data structures in DSL

A domain-specific language for unit testing often has a number of data structures that aid in the definition and organization of test cases. The following are some of the most commonly used data structures in unit testing DSLs:

A test case is a unit of testing that represents a single set of input values and expected output for a particular function or feature that is being tested. In general, test cases are stated in terms of input data, expected output, and any additional context or requirements that must be met.

A test suite is a collection of test cases that are organized according to their purpose, function, or area of concentration. Test suites allow you to organize tests into logical groups and manage huge numbers of test cases more easily.

A statement that outlines an expected condition or behavior for a certain test case is known as an assertion. Assertions are used to ensure that the actual results of a test correspond to the expected results and are frequently defined using a specific syntax or function within the language.

A fixture is a set of preconditions that must be met in order to execute a certain test case. Fixtures are used to build up the environment or context in which a test case will be conducted and can contain things like the initialization of variables, database connections, or other resources.

Mock object is a simulated object that is used to test a specific feature or behavior by replacing a genuine item. Mock objects are often used to isolate specific components of an application or system, and their methods, properties, and intended behavior are described within the DSL.

Grammar overview

Domain specific languages can be used to simplify the process of writing unit tests by providing a custom syntax that is tailored to the needs of the software being tested. It will be introduced a grammar for a DSL that can be used for unit testing, exploring its syntax and capabilities, and demonstrating how it can be used to write effective and efficient unit tests [5].

Firstly, it will be introduced with a bunch of examples that show how the DSL will work for some base cases. Then it will be presented the grammar that is used to parse the presented code.

In Figure 1 it can be seen some examples that are already possible to be parsed in the DSL

```
Suite mySuite

Test foo -skip -repeat 5 times
When param1=1
Then result should be 1

Test foo2
When param2=True, param3="John"
Then result should be empty

Test foo3 -skip
When no parameters
Then result should be 1

Execution order: foo1, foo3, foo2
```

Figure 1: DSL code examples

First, notice the **Suite** keyword. A suite defines a collection of functions. This DSL is for the Python programming language. It will search for the file that contains the name of the suite. It will then search for functions in this file.

Test *functionName* [flags]

Notice that Test is bolded, meaning it is a keyword. The following is an identifier of a function. The interpreter will search for the function in the file specified in the suite definition. This function will be tested in this test. Next, optionally, flags can be specified. Using the flags for skipping a test (*-test*), for repeating a test multiple times (*-repeat N times*).

After specifying a test's name and flags, there is the **When** [params]. This is the line that specifies the parameters of the function that will be called. Parameters are specified as *paramName=paramValue.* Notice that if a function doesn't require any parameters at all, then you can just type *no parameters* and the function will be called without any parameters.

Then result should be [value] - specifies the expected return value of the function call. Also, no return value (void function) still works. It is required to type *empty* to mention that no value should be returned.

The keyword *empty* when specifying the result value is interchangeable with *Empty, void, Void, none and None.* Similarly, the *no parameters* is interchangeable with *None, none, Void and void.*

Lastly, it can be specified the execution order of your tests. This is done by typing *Execution order:* [functionName1], [functionName2] and so on.

The following is a formal definition of the grammar in Backus-Naur Form (BNF):

V_T = {
<letter>,
<digit>,
<STRING>
<BOOL>,
<EmptyParameters>,
<EmptyResult>
},

V_N = {
<ID>, <INT>, <FLOAT>, <NUMBER>,
<Suite>, <Test>, <Result>, <Parameter>,
<Datatype>, <Flag>, <Repeat>, <Order>
}

<letter> ::= "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
<digit> ::= "0" | "1" | ... | "9"
<ID> ::= <letter> (<letter> | <digit>)*
<STRING> ::= "" (~"")* ""
<INT> ::= <digit>+
<FLOAT> ::= <digit>+ "." <digit>+
<NUMBER> ::= <INT> | <FLOAT>
<BOOL> ::= "True" | "False"
<Suite> ::= "Suite" <ID> <Test>* <Order>?
<Test> ::= "Test" <ID> <Flag>* "When" <Parameter>+ <Result>
<Result> ::= "Then" ("result" "should" "be")? (<Datatype> | <EmptyResult>)
<Parameter> ::= (<ID> "=" <Datatype>) | <EmptyParameters>
<Datatype> ::= <STRING> | <INT> | <FLOAT> | <NUMBER> | <BOOL>
<EmptyParameters> ::= "no parameters" | "None" | "none" | "void" | "Void"
<EmptyResult> ::= "empty" | "Empty" | "void" | "Void" | "none" | "None"
<Flag> ::= "-" ("skip" | "Skip" | <Repeat>)?
<Repeat> ::= ("repeat" | "Repeat") <INT> ("times" | "Times")
<Order> ::= "Execution order:" <ID> ("," <ID>)*

The language is defined as follows:
L(G) = (V_N, V_T, P, S), where
S = { <Suite> },
P = {
<letter> => "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
<digit> => "0" | "1" | ... | "9"
<ID> => <letter> (<letter> | <digit>)*
<STRING> => "" (~"")* ""
<INT> => <digit>+
<FLOAT> => <digit>+ "." <digit>+
<NUMBER> => <INT> | <FLOAT>
<BOOL> => "True" | "False"
<Suite> => "Suite" <ID> <Test>* <Order>?
<Test> => "Test" <ID> <Flag>* "When" <Parameter>+ <Result>
<Result> => "Then" ("result" "should" "be")? (<Datatype> | <EmptyResult>)
<Parameter> => (<ID> "=" <Datatype>) | <EmptyParameters>
<Datatype> => <STRING> | <INT> | <FLOAT> | <NUMBER> | <BOOL>
<EmptyParameters> => "no parameters" | "None" | "none" | "void" | "Void"
<EmptyResult> => "empty" | "Empty" | "void" | "Void" | "none" | "None"
<Flag> => "-" ("skip" | "Skip" | <Repeat>)?
<Repeat> => ("repeat" | "Repeat") <INT> ("times" | "Times")
<Order> => "Execution order:" <ID> ("," <ID>)*
}

In Figure 2 it is represented the meta-model for the language as generated by TextX.

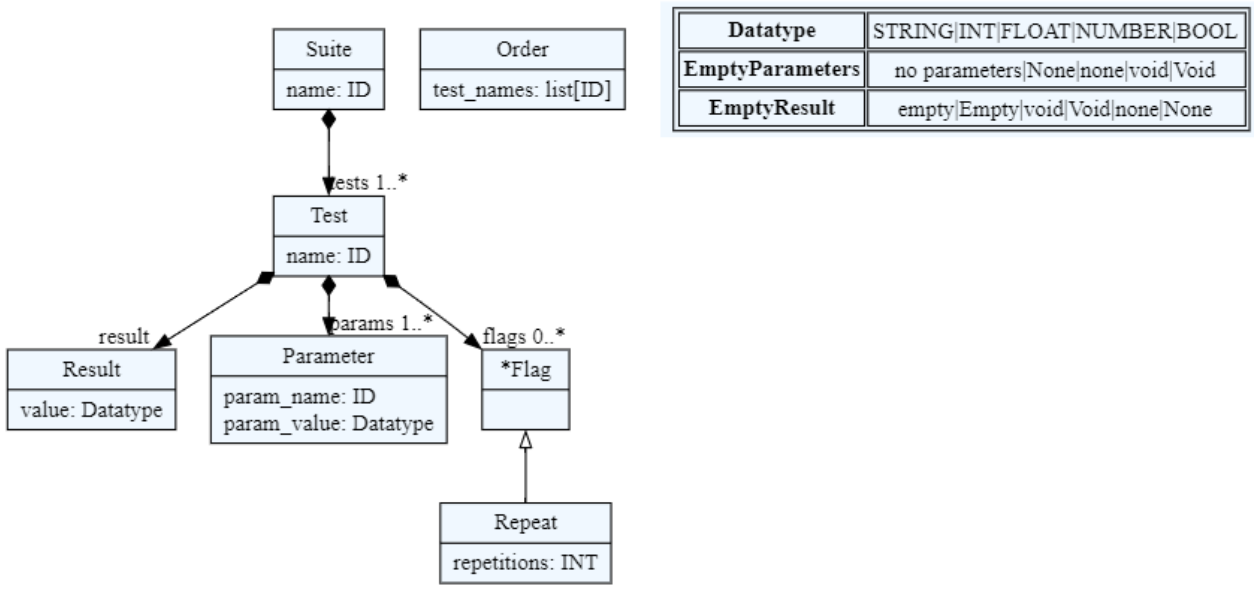

Figure 2: Language meta-model

An example parse tree would be useful.

Suite mySuite

Test foo -skip -repeat 5 times
When param1=1, param2=True
Then result should be 42

The obtained parse tree from the above piece of code is constructed in Figure 3.

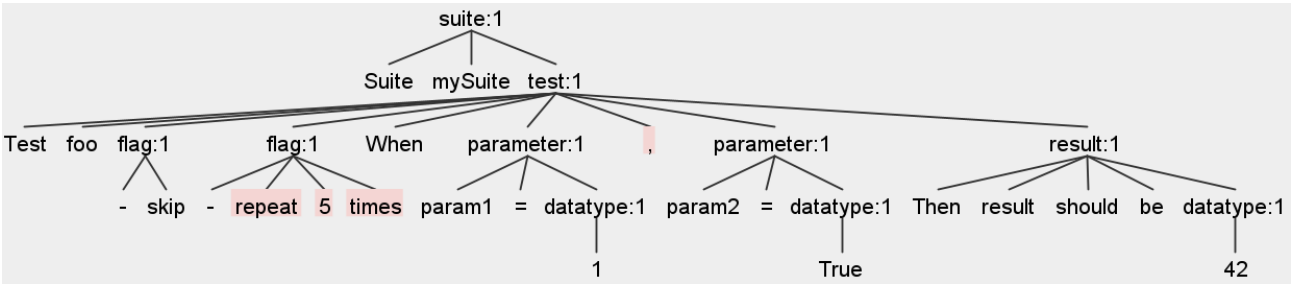

Figure 3: Parse Tree for example program

Conclusions

All things considered, a domain specific language for unit testing may simplify the development, management, and execution of tests while also improving the quality, consistency, and automation of unit tests. This allows for the rapid and effective identification and resolution of problems. It is a great tool that could be used by developers and testers to make their work easier and save their time.

Users create and change data by writing code that defines test cases and related structures. The DSL allows users to express their test cases in a succinct and legible manner, as well as a framework for executing and reporting on the results of those tests.

References:

1. Unit Testing, TechTarget [online]. [accessed 20.02.2023] Available: https://www.techtarget.com/searchsoftwarequality/definition/unit-testing

2. Domain overview, GURU99 [online]. [accessed 22.02.2023] Available: https://www.guru99.com/unit-testing-guide.html
3. Who performs unit testing?, METHODPOET [online]. [accessed 22.02.2023] Available: https://methodpoet.com/who-performs-unit-testing/
4. Unit Testing Problems, Mozaic Works [online]. [accessed 22.02.2023] Available: https://mozaicworks.com/blog/5-common-unit-testing
5. Security unit tests, DaDario [online]. [accessed 28.02.2023] Available: https://dadario.com.br/security-unit-tests-are-important/