



UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

TESTING DOCUMENT

PROJECT NAME: ARCANE ARCADE

CLIENT: TONY VD LINDEN

TEAM NAME: TERABITES

TEAM MEMBERS

NG Maluleke
13229908

D Mulugisi
13071603

C Nel
14029368

LE Tom
13325095

September 28, 2016

Contents

1	Introduction	2
1.1	Purpose	2
1.2	Scope	2
1.3	Test Environment	2
1.4	Assumptions and Dependencies	3
2	Test Items	4
3	Functional Features to be Tested	4
4	Test Cases	4
4.1	Test Case 1: Syntax Testing	4
4.1.1	Condition 1: The back-end server must be running . . .	4
4.2	Test Case 2: Testing Declaration and Conditions	5
4.2.1	Condition 1: The back-end server must be running . . .	5
4.3	Test Case 3: Expressions Testing	6
4.3.1	Condition 1: The back-end server must be running	6
5	Pass/Fail Criteria	7
6	Detailed Test Results	8
6.1	Overview of Test Results	8
6.2	Functional Requirements Test Results	8
6.2.1	Test Case 1: Syntax	8
6.2.2	Test Case 2: Declarations and Conditions	8
6.2.3	Test Case 3: Expression Testing	8
7	Other	9
8	Conclusions and Recommendations	9

1 Introduction

The project is called *Arcane Arcade*, which references the esoteric language users will have to use, as well as the gamification approach to try and make it as fun as possible.

1.1 Purpose

This documentation is the testing documentation for Arcane Arcade project. It outlines the entire testing plan and its documentation process. The documentation firstly establishes the scope, thereafter the testing environment is discussed including all the relevant assumptions and dependencies made during the testing process.

Unit testing is a crucial part of a continual software development approach and is necessary to ensure that all components properly work before integrating them into the system. The system is used by employers to determine the level of skills and classification of potential employees and based on those skills they placed on appropriate positions within the company.

1.2 Scope

1.3 Test Environment

- Programming Languages
Java SE standard and ANTLR4 were used for backend development of the system and angular js and HTML was used on the frontend of the system.
- Testing Frameworks
The tests are JUnit tests and are running under JUnit 4.12. Junit was chosen because it can be used separately or integrated with build tools like Maven and Ant and third party extensions.
- Coding Environment
Netbeans and IntelliJ IDEs that we chose for the project development. Maven sets up the environment by providing all the needed dependencies which are included in the POM file.
- Operating System
The tests are independent of the operating system and occur under Maven.
- Internet Browsers
The web-interface was tested to execute accurately on GOOGLE Chrome, Mozilla Firefox and Internet Explorer web browser.

1.4 Assumptions and Dependencies

Most of the unit testing is to test the compiler component of the system. The compiler consists of a lexer, parser, and visitor. The lexer and parser are automatically generated by ANTLR4 using a specified grammar file which specifies the lexer and syntax rules for the language. The visitor is manually implemented and is in charge of walking the parse tree in the correct order and executing the appropriate commands on each node.

Testing is done through JUnit test files and Maven automatically running the tests when deploying the project or running the *test* command through Maven.

Unit Test Plan

2 Test Items

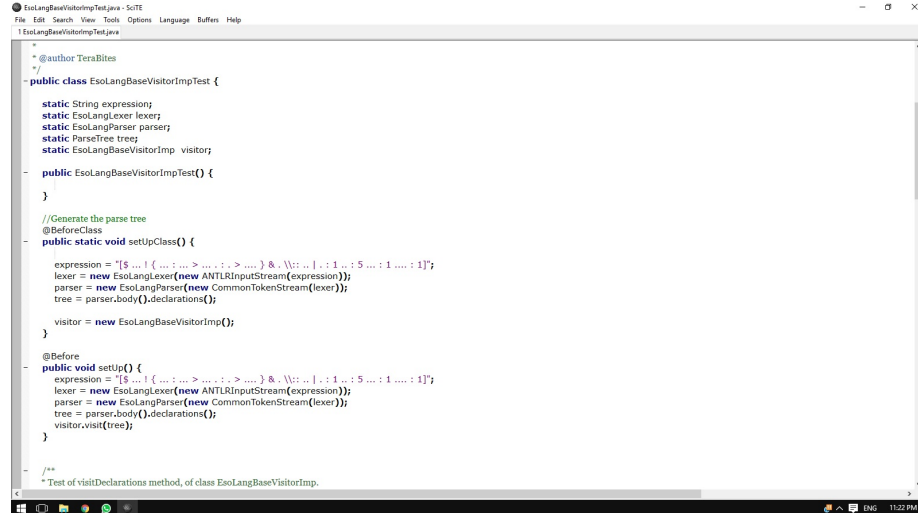
The lexer and parser are tested together by feeding a list with a syntatically correct and wrong statements in the implemented language. The lexer then tokenizes the input whereafter the parser parses the token stream and builds a parse tree. It is then tested that the parser reports the correct symantic errors.

The visitor is tested by having it run through a list of statements. The visitor is implemented in such a way that it holds a list of values that need to be printed. The test then simply looks at this list in the visitor object and compares it with a list of expected results.

3 Functional Features to be Tested

4 Test Cases

4.1 Test Case 1: Syntax Testing



```

EsoLangBaseVisitorTest.java - SCITE
File Edit Search View Tools Options Language Buffers Help
1:EsoLangBaseVisitorTest.java
*
* @author TeraBites
*/
public class EsoLangBaseVisitorImpTest {

    static String expression;
    static EsoLangLexer lexer;
    static EsoLangParser parser;
    static ParseTree tree;
    static EsoLangBaseVisitorImp visitor;

    public EsoLangBaseVisitorImpTest() {

    }

    //Generate the parse tree
    @BeforeClass
    public static void setUpClass() {

        expression = "[ $ ... ! { ... : ... > ... : : > ... } & . \\ : : . : 1 ... : $ ... : 1 ... : 1 ]";
        lexer = new EsoLangLexer(new ANTLRInputStream(expression));
        parser = new EsoLangParser(new CommonTokenStream(lexer));
        tree = parser.body().declarations();

        visitor = new EsoLangBaseVisitorImp();
    }

    @Before
    public void setUp() {
        expression = "[ $ ... ! { ... : ... > ... : : > ... } & . \\ : : . : 1 ... : $ ... : 1 ... : 1 ]";
        lexer = new EsoLangLexer(new ANTLRInputStream(expression));
        parser = new EsoLangParser(new CommonTokenStream(lexer));
        tree = parser.body().declarations();
        visitor.visit(tree);
    }

    /**
     * Test of visitDeclarations method, of class EsoLangBaseVisitorImp.

```

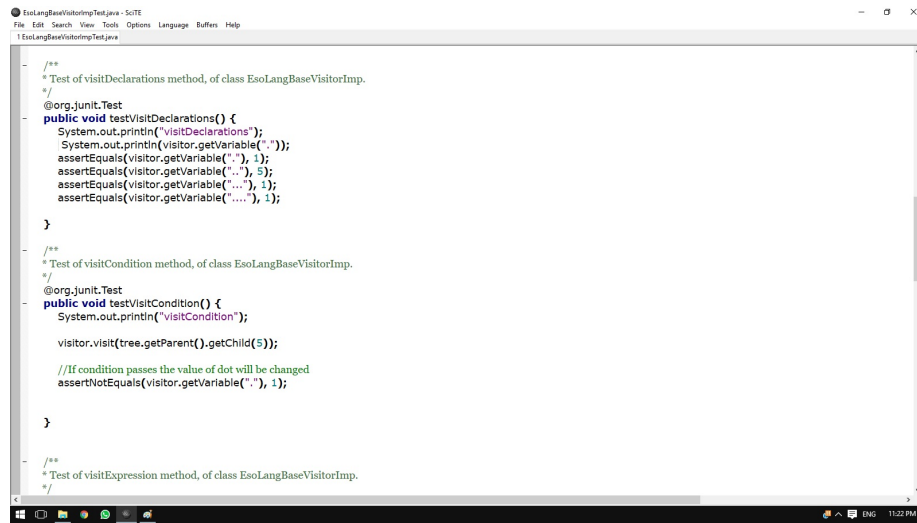
4.1.1 Condition 1: The back-end server must be running

4.1.1.1 Objective : The main objective of this test is to validate whether the syntax of the user input is correct.

4.1.1.2 Input : For every use case, both valid and invalid entries will be constructed to ensure that exceptions are raised if pre-conditions are not met. Further all return values are checked to ensure that the returned object is correct according to the stated functional requirements. In this case a string of user esolang input expression will be our input.

4.1.1.3 Outcome : All use cases should fulfill their stated service contract by returning the appropriate response object which was outlined on the functional requirements.

4.2 Test Case 2: Testing Declaration and Conditions



```

EsoLangBaseVisitingTest.java - ScITE
File Edit Search View Tools Options Language Buffers Help
1: EsoLangBaseVisitingTest.java

- /**
 * Test of visitDeclarations method, of class EsoLangBaseVisitorImp.
 */
- @org.junit.Test
- public void testVisitDeclarations() {
-     System.out.println("visitDeclarations");
-     System.out.println(visitor.getVariable("."));
-     assertEquals(visitor.getVariable("."), 1);
-     assertEquals(visitor.getVariable("..."), 5);
-     assertEquals(visitor.getVariable("..."), 1);
-     assertEquals(visitor.getVariable("...."), 1);
- }

- /**
 * Test of visitCondition method, of class EsoLangBaseVisitorImp.
 */
- @org.junit.Test
- public void testVisitCondition() {
-     System.out.println("visitCondition");
-     visitor.visit(tree.getParent().getChild(5));
-     //If condition passes the value of dot will be changed
-     assertEquals(visitor.getVariable("."), 1);
- }

- /**
 * Test of visitExpression method, of class EsoLangBaseVisitorImp.
 */

```

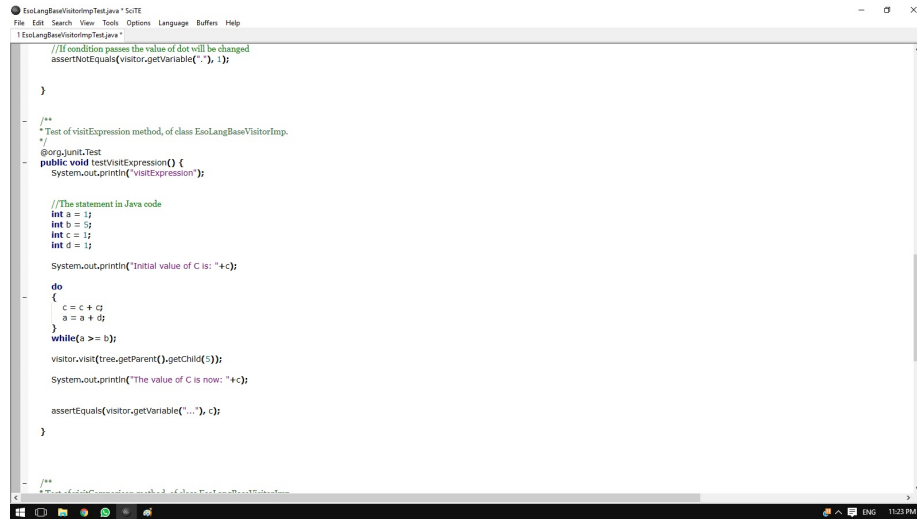
4.2.1 Condition 1: The back-end server must be running

4.2.1.1 Objective : The purpose of this test is to validate that all the declared variables are initialized with valid values

4.2.1.2 Input : Both valid and invalid entries will be constructed to ensure that exceptions are raised if pre-conditions are not met. Further all return values are checked to ensure that the returned object is correct according to the stated functional requirements. In this case a string of user esolang input expression will be our input.

4.2.1.3 Outcome : On failure this test will raise a declaration failed exception while on a pass the test return the appropriate response object which was outlined on the functional requirements

4.3 Test Case 3: Expressions Testing



```
1 EsoLangBaseVisitorTest.java
// If condition passes the value of dot will be changed
assertNotEquals(visitor.getVariable("."), 1);
}

/**
 * Test of visitExpression method, of class EsoLangBaseVisitorImp.
 */
@org.junit.Test
public void testVisitExpression() {
    System.out.println("visitExpression");

    // The statement in Java code
    int a = 1;
    int b = 5;
    int c = 1;
    int d = 1;

    System.out.println("Initial value of C is: " + c);
    do
    {
        c = c + d;
        a = a + d;
    }
    while(a >= b);

    visitor.visit(tree.getParent().getChild(5));
    System.out.println("The value of C is now: " + c);
    assertEquals(visitor.getVariable("."), c);
}

/**
 *
 */
}
```

4.3.1 Condition 1: The back-end server must be running

4.3.1.1 Objective : The purpose of this test is to validate whether the expression contains valid tokens. An expressions which is ran in the esolang is also ran in java code and the end results of the two same but differently executed methods are compared for equality through the assert methods.

4.3.1.2 Input : Both valid and invalid entries will be constructed to ensure that exceptions are raised if pre-conditions are not met.

4.3.1.3 Outcome : Upon recognizing an invalid token the test raises an exception and if no invalid token is found, the test returns the appropriate response object which was outlined on the functional requirements.

5 Pass/Fail Criteria

For every use case tested, the following criteria must be met for the test to be considered a **Pass**.

- All pre-conditions should be fulfilled. If not, an appropriate exception must be raised as to alert the client.
- All post-conditions must hold upon returning from the function call. If a post condition doesn't hold, the test will fail.
- Any stated functionality specific to the use case as set out in the functional requirements document must be validated within the unit test to ensure stated functionality is implemented.

Unit Test Report

6 Detailed Test Results

6.1 Overview of Test Results

All the test cases passed. The tests were supposed to pass to prove that everything is working properly as expected. The tests were from the jUnit framework.

We selected jUnit because of simplicity to use and how it stays updated. jUnit was also easy to include into maven so that all tests are executed automatically every time the project is built. Executing the tests is then greatly simplified and changes can be tracked by observing changes in the test results.

6.2 Functional Requirements Test Results

The tests we have created for this module are contained within the file `EsoLang-BaseVisitorImpTest.java` from TeraBites.

6.2.1 Test Case 1: Syntax

- The first proper/wrong context passed the test, there were no exceptions.
- An error was displayed to indicated that the syntax is not correct

6.2.1.1 Result: Pass.

6.2.2 Test Case 2: Declarations and Conditions

- The values were assigned to the variables
- Certain statements did not execute because the conditions were unmet

6.2.2.1 Result: Pass.

6.2.3 Test Case 3: Expression Testing

- Mathematical operations can be done
- Language features are recognized and excuted

6.2.3.1 Result: Pass.

7 Other

8 Conclusions and Recommendations

The body of our tests re-use the same variables which can have a negative effect on our testing as values get altered before the test so to combat this we used junit `@BeforeClass` and `@After` annotations to prepare the environment for accurate testing and to make sure that each execution begins with the appropriate criteria.

Exit criteria are observed and compared immediately before all test functions exit. Here the effects of the of the statements are compared and asserted as the expected values.