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	D Mulugisi	C Nel	LE Tom
13229908	13071603	14029368	13325095

September 28, 2016

Contents

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.

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: Language Features

4.1.1 Condition 1:

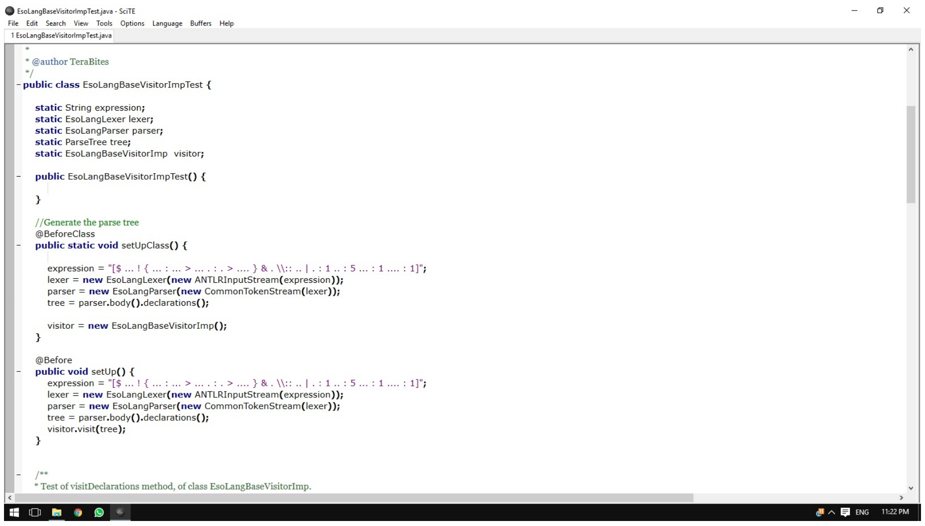
4.1.1.1 Objective :

4.1.1.2 Input :

4.1.1.3 Outcome :

4.1.2 Condition 2:

4.1.3 Condition 3:



EsoLangBaseVisitorImpTest.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 EsoLangBaseVisitorImpTest.java

```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 .. : 5 ... : 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 .. : 5 ... : 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.
```

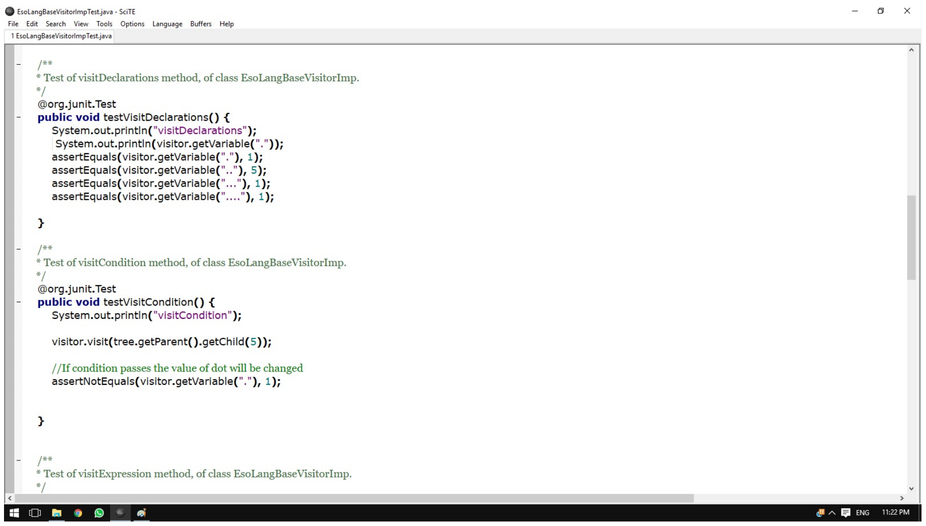
The two above above methods annotated with the @Before and the @Before-Class prepare the test environment for the next test methods by executing and restoring the states prerior to running another test.

4.2 Test Case 2: Testing Declaration and Conditions

4.2.1 Condition 1:

4.2.2 Condition 2:

4.2.3 Condition 3:


EsoLangBaseVisitorImpTest.java - SciTE
File Edit Search View Tools Options Language Buffers Help
1 EsoLangBaseVisitorImpTest.java
ENG
11:22 PM

```
/**
* 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
    assertNotEquals(visitor.getVariable("."), 1);


}


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

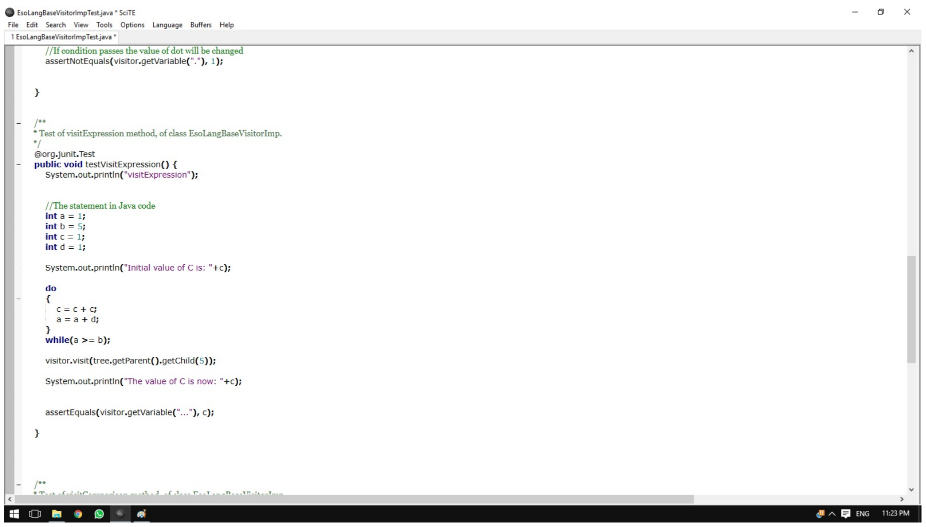
Here we test to see if the assignments under declarations and working properly and if conditional statements are functioning as expected. To test if the methods are functioning as expected, we compare values from the methods and the values that we expect to find via the assert() method.

4.3 Test Case 3: Expressions Testing

4.3.1 Condition 1:

4.3.2 Condition 2:

4.3.3 Condition 3:


EsoLangBaseVisitorImpTest.java * SciTE
File Edit Search View Tools Options Language Buffers Help
1 EsoLangBaseVisitorImpTest.java *

```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 + c;
            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);

    }



    /**
```

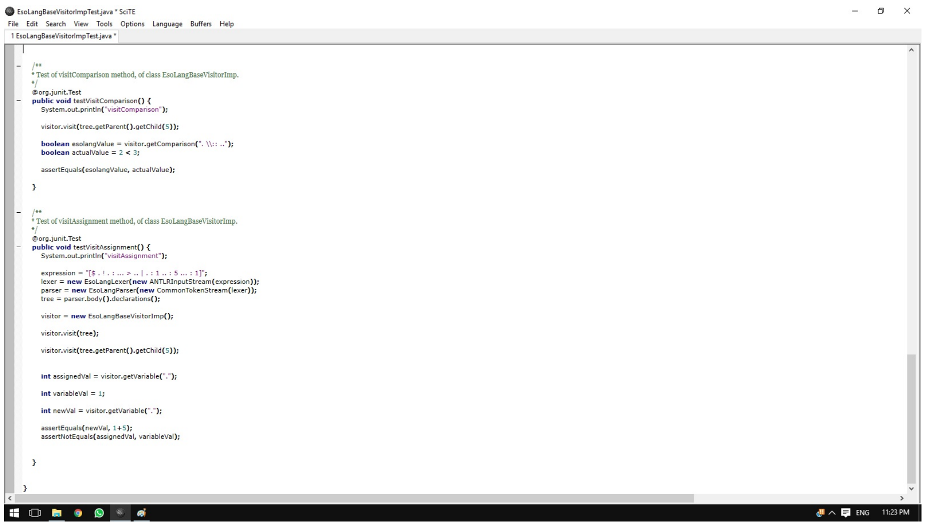
An expressions which is ran in the esolang is also ran in java code and the end results of the two same but differently executed methods are compared for equality through the assert methods.

4.4 Test Case 4: Assignment and Comparison Testing

4.4.1 Condition 1:

4.4.2 Condition 2:

4.4.3 Condition 3:


EsoLangBaseVisitorImpTest.java * SciTE
File Edit Search View Tools Options Language Buffers Help
1 EsoLangBaseVisitorImpTest.java *
ENG
11:23 PM

```java

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

        visitor.visit(tree.getParent().getChild(5));

        boolean esolangValue = visitor.getComparison(". \\:: ..");
        boolean actualValue = 2 < 3;

        assertEquals(esolangValue, actualValue);

    }


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

        expression = "[$ . ! . : ... > .. | . : 1 .. : 5 ... : 1]";
        lexer = new EsoLangLexer(new ANTLRInputStream(expression));
        parser = new EsoLangParser(new CommonTokenStream(lexer));
        tree = parser.body().declarations();

        visitor = new EsoLangBaseVisitorImp();

        visitor.visit(tree);

        visitor.visit(tree.getParent().getChild(5));


        int assignedVal = visitor.getVariable(".");

        int variableVal = 1;

        int newVal = visitor.getVariable(".");

        assertEquals(newVal, 1+5);
        assertNotEquals(assignedVal, variableVal);


    }


}
```

We test assignment and comparison by comparing expected values with the values we get from the esoteric language.

5 Execution Strategy

6 Item Pass/Fail Criteria

7 Detailed Test Results

7.1 Overview of Test Results

7.2 Functional Requirements Test Results

7.2.1 Authentication Module

7.2.2 User Management Module

7.2.3 Challenge Management Module

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 executions 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.