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

July 29, 2016

Contents

1 Introduction **2**

2 Background **2**

3 Testing **2**
3.1 Introduction 2
3.2 Approach 2
3.3 Tests conducted 2

4 Testing Strategy **3**
4.1 Base Vistor Implentation Test 3

5 Execution Strategy **4**
5.1 Entry and Exit Criteria 4

6 Test Environment **5**

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.

2 Background

The client needs a way to test potential new employees without running the risk of tests being leaked and without having to manually create complicated in-house tests. Through this system the client can test potential new employees via a custom programming language, an esoteric language, while keeping the tests interesting through the use of gamification principles. Testees' performance and use of help can also help determine the ideal employment position.

3 Testing

3.1 Introduction

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.

3.2 Approach

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.

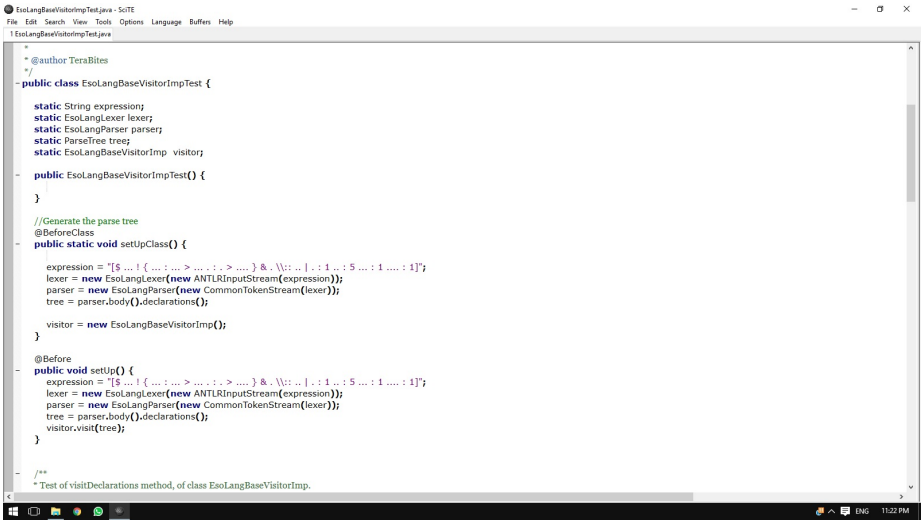
3.3 Tests conducted

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.

4 Testing Strategy

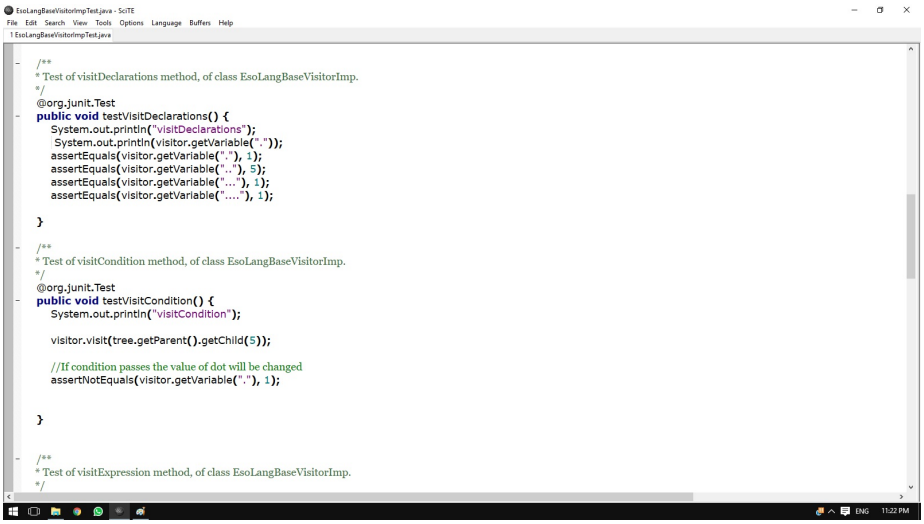
4.1 Base Vistor Implentation Test



```
/**
 * @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 @BeforeClass prepare the test environment for the next test methods by executing and restoring the states prerior to running another test.

4.2

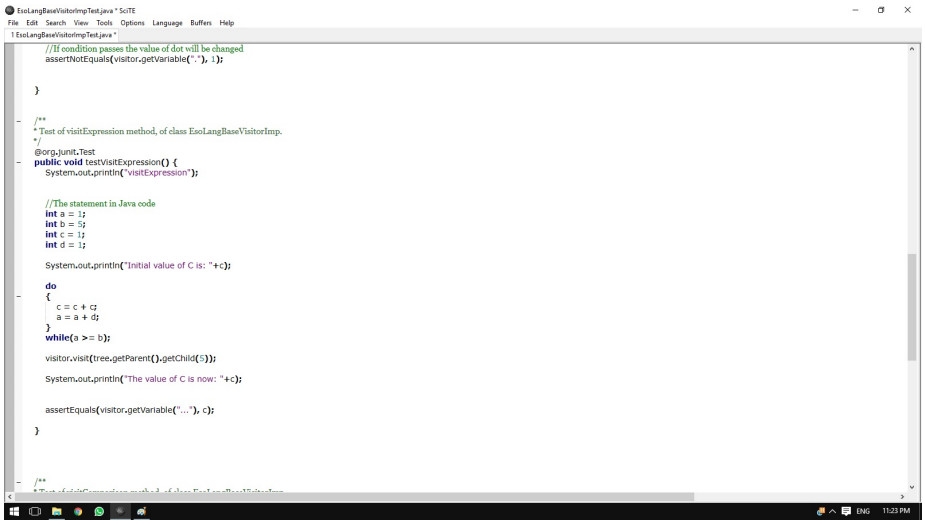


```
    /**
     * Test of visitDeclarations method, of class EsoLangBaseVisitorImp.
     */
    @org.junit.Test
    public void testVisitDeclarations() {
        System.out.println("visitDeclarations");
         System.out.println(visitor.getVariable("."));
        assertEquals(visitor.getVariable("."),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

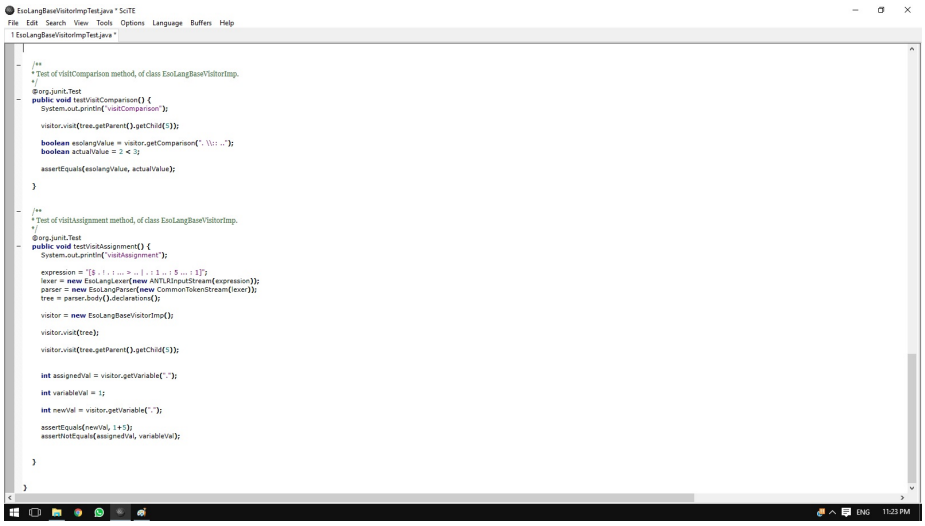
and if conditional statements are functioning as expected. To test if the methods are functioning as expected, we compare values from the methods and the values that we expect to find via the assert() method.

4.3



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



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

5 Execution Strategy

5.1 Entry and Exit Criteria

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.

6 Test Environment

The tests are independent of the operating system and occur under Maven. The environment setup of the tests is the inclusion and import of the used frameworks. The tests are jUnit tests and are running under jUnit 4.12. Maven sets up the environment by providing all the needed dependencies which are included in the POM file.