



**UNIVERSITY
OF STUDY
OF BERGAMO**

DEPARTMENT OF ENGINEERING

Computer Engineering

Linguaggi e Compilatori: from SCA-XML to Java

Maccarana Yamuna - matr.1014766

Maroni Gabriele - matr.1001749

Bordogna Giampietro - matr.1017372

**ACADEMIC
YEAR
2013/2014**



Contents

1	Introduction	1
2	Aims	3
3	Background and Case Study	5
4	Configuration	11
5	Semantic definition	13
6	Environment	19
7	Handler	21
8	Writer	23
9	Using the tool	29
10	Conclusions	31

List of Figures

3.1	XML file defining a robot composite	6
3.2	Graphic view of a robot composite with GPS and Navigator components	7
3.3	Example of a Java class implementing a XML component	8
3.4	Example of a Java output class	9
4.1	Project environment	12
7.1	Screen of a correct Parsing	22
8.1	Screen of an Exception thrown because of an incorrect ID completion	24
8.2	Screen of an Exception thrown because of an incorrect targetNames- pace completion	25
8.3	First part of the write method in JavaWriter	25
8.4	createIntestation() method	25
8.5	createPackage() method	26
8.6	createClass() method	26
8.7	createFields() method	27
8.8	createConstructor() method	27
8.9	Creating a new file in the write() method	28
9.1	main method of the ParserTester class	30

9.2	Final view of a correct configuration and run of the tool	30
-----	---	----

List of Tables

3.1	Associations between XML and Java	8
5.1	Start	13
5.2	Header	14
5.3	Body	14
5.4	Methods	14
5.5	Tokens	17
5.6	Methods called by myScanner.g	17
6.1	Fields in Environment.java	20
7.1	Effects of the methods in the Handler class on the Environment fields	21

Chapter 1

Introduction

This project has been developed as completion to the course of Linguaggi e Compilatori (Language and Compiler) with professor Psaila, at the University of Bergamo, placed in Dalmine. The subjects of this project fall within the scope of the course's field, in particular it focuses on the specific aspects of the arguments treated in the course.

Through the development of this project, important matters such as predefined grammar structures recognition, syntactic analysis (parsing), semantic structure definition, different grammar reconstruction and management of exceptions, most and more of the programme of the course of Linguaggi e Compilatori has been treated.

With this manual the user will be able to understand each step of the project and, moreover, they will be capable to extend or modify this tool for a specific need.

Chapter 2

Aims

The aim of this project is to provide a certain number of syntactic rules to allow, from a XML file as input containing a code defining a Service Component Architecture (SCA), to recognize its structure, to memorize relative variables and then to convert the input file in a Java file.

This project has been conceived for a robotic environment. There are several applications that could take advantage from this project, such as, for example, the Task Component Architecture (TCA) [1], an extension of the Service Component Architecture which easily manages the movement of a robot that, knowing its position, speed and direction in the frame of reference (asking it to a GPS component), receives instructions about the path to follow and the environment (maps) (receiving information from a Navigator component).

Through the use of the tool provided by this project, the automatic generation of a Java class allows to avoid the use of SCA and its XML file (named "composite") without losing any functionality.

Chapter 3

Background and Case Study

Extensible Markup Language (XML) [2] is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. A markup language is a modern system for annotating a document in a way that is syntactically distinguishable from the text. Through these languages it is possible to define a set of specification to describe a text's representation mechanisms (structural, semantic, etc.). The great advantage is that, using standard conventions, each file is reusable on different work stations and it can be implemented using different languages.

The Service Component Architecture (SCA) [3] is a set of specifications intended for the development of applications using a Service Oriented Architecture (SOA) [4], which defines how computing entities interact to perform work for each other. Moreover, SCA is an XML-based metadata model that describes the relationships and the deployment of services independently from SOA platforms and middleware Application Programming Interfaces (APIs) (as Java, C++, etc.). The specification of a SCA component is defined as an XML-based model in terms of provided services, required services (called references), and properties. For this project's purposes, the specification of SCA components is defined by Java interfaces and their implemen-

tation is provided by Java classes.

The structure of the XML files is standard. The first part of the xml is the header, and it will be the same for all of these kind of files. The following lines are intended for the definition of the composite, which is the container of several components. Then, there are the lines dedicated to the components. After the component name definition there is a line to define the type of the implementation: this project is based on a java implementation. Next, the properties are listed. In this case study, the ID property is required, and the other ones are not compulsory. Last, there is a part to define the services, the references and their interfaces.

A simple example of a correct implementation of a standard XML file can be seen in listing 3.1.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
3     name="robotEnvironment"
4     targetNamespace="http://robotEnvironment">
5
6 <component name="RobotComponent">
7     <implementation.java class="resources.robotEnvironment.Robot"/>
8     <property name="ID">TestRobot001</property>
9     <property name="nickname">Discovery</property>
10    <property name="weight">20</property>
11    <property name="height">50</property>
12    <property name="price">499.99</property>
13    <service name="move">
14        <interface.java interface="resources.robotEnvironment.RobotInterface"/>
15    </service>
16    <reference name="updatePosition">
17        <interface.java interface="resources.robotEnvironment.GPSInterface"/>
18    </reference>
19 </component>
20
21 </composite>
```

Figure 3.1: XML file defining a robot composite

There is no need to be careful with white spaces and new paragraphs, as the semantic rules properly manage them.

A graphic view of the case study can be found in fig. 3.2. In this figure the

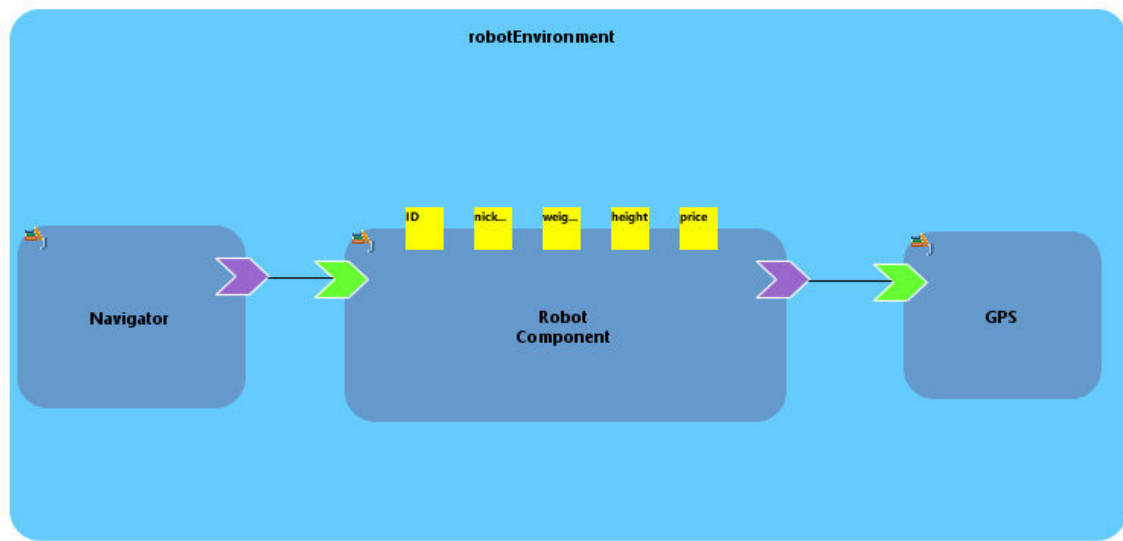


Figure 3.2: Graphic view of a robot composite with GPS and Navigator components

robot component is defined as shown in listing 3.1: it is possible to see the type of component (RobotComponent), its implementation (Java implementation - see the small "J" symbol in the top-left of the component), its properties (ID, nickName, weight, height and price) and its service and reference. The composite is named as "robotEnvironment". Two other components are listed in the XML file as Navigator and GPS, with their reference (Navigator) and service (GPS), their implementation (Java) and no properties. Next, it will be explained how to use Java to implement the XML files.

Java [5] is a computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that code which runs on one platform does not need to be recompiled to run on another.

The Java file needed to implement the XML components is a Java interface, that has to be implemented by a standard Java class containing properties as fields (and

several methods such as getter and setter methods) (see listing 3.3).

```
public abstract class Robot implements RobotInterface {

    //----Properties-----

    @Property(required=true) //see SCA documentation
    private String ID;
    @Property(required=false) //see SCA documentation
    protected String nickname;
    @Property(required=false) //see SCA documentation
    protected double weight;
    @Property(required=false) //see SCA documentation
    protected double height;
    @Property(required=false) //see SCA documentation
    protected double price;
}
```

Figure 3.3: Example of a Java class implementing a XML component

The Java output file to obtain from the XML file concerns the use of a simple grammar to define a Java class and its fields.

A simple example of a correct output Java file can be seen in listing 3.4.

To do so, basic associations between XML and Java structures are needed and they can be found in tab.3.1.

XML:	Java:
targetNamespace= "Path"	package Path
implementation.java class= "ClassPath"	extends ClassPath
<property name= "ID" >ClassName	public class ClassName
<property name= "FieldName" >Value	private String FieldName FieldName=Value
<service name= "ServiceName" >ClassName	implements ServiceName
<reference name= "RefName" >ClassName	implements RefName

Table 3.1: Associations between XML and Java

```
1 //-----
2 // <auto-generated>
3 // This code was generated by a tool.
4 //     @author Yamuna Maccarana.
5 //     @author Gabriele Maroni.
6 //     @author Giampietro Bordogna.
7 // This file can be modify, changes will not cause incorrect behaviors of the tool.
8 // ATTENTION: Changes could be lost if the code is regenerated.
9 // </auto-generated>
10 //-----
11
12 package out_robotEnvironment;
13
14 public abstract class TestRobot001 extends resources.robotEnvironment.Robot
15     implements resources.robotEnvironment.RobotInterface,
16     resources.robotEnvironment.GPSInterface {
17
18     private String nickname;
19     private String weight;
20     private String height;
21     private String price;
22
23     public TestRobot001() {
24         nickname = "Discovery";
25         weight = "20";
26         height = "50";
27         price = "499.99";
28     }
29 }
```

Figure 3.4: Example of a Java output class

Chapter 4

Configuration

To configure the environment in Eclipse it is necessary to import the provided project and configure the build path adding the ANTLR jars if there is the need to modify myScanner.g. Any ANTLR version from 3 to 4 should be fine. In case of doubt, please use the antlr-3.4-complete.jar.

The final result of a correct configuration can be seen in fig. 4.1.

The resources.robotEnvironment package contains the case study's Java classes and interfaces needed to implement the XML composite files. Specific information can be found in the code.

The inputResources folder contains the input file that will be analyzed by the tool (in this project robot.composite is the input file used). Later (in chapter 9) it will be explained how to choose the input file to analyze.

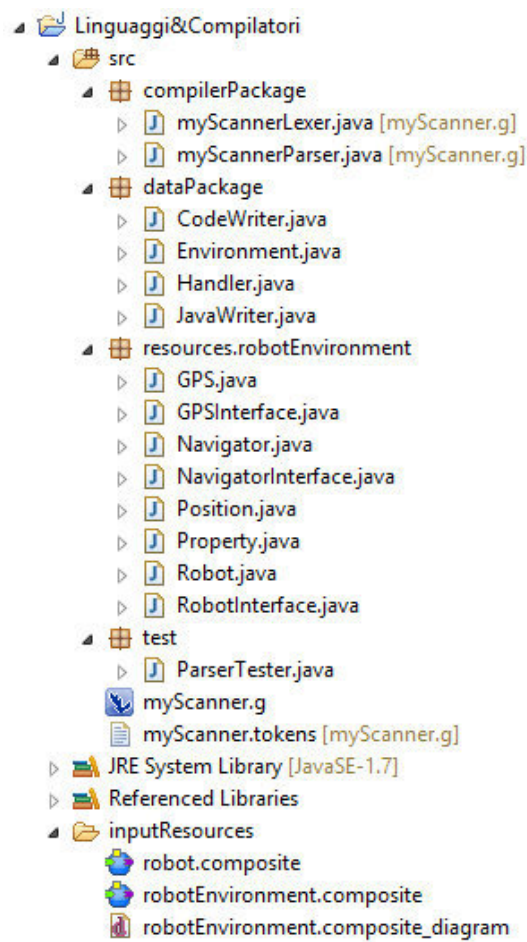


Figure 4.1: Project environment

Chapter 5

Semantic definition

Semantic rules and definitions are contained in `myScanner.g` file. The XML file is analized and split into four major parts, as described in tab. 5.1.

start:	
header	Contains the standard first part of the XML file, defining the type of XML, the composite, the component and its implementation (Java in this project)
body	Contains the definition of the properties
methods	Contains services and references
endings	Contains the standard endings of the XML file

Table 5.1: Start

Three of the four parts, in turn, are composed of other several parts, as described in tab. 5.2, 5.3, 5.4. The endings part contains just a few semantic rules to manage the ending of the XML file.

All these rules can be found in `myScanner.g`, in which, for this project, `k` is set to 1, because a LL(1) grammar is needed, and `language` is set to Java.

Moreover, a certain number of specific tokens has been used. The used ones can be found in tab. 5.5, but, in `myScanner.g` file, there are other tokens that may be useful in case of different applications (such as a token to manage comments recognition: see code for more details).

header:	
intestazioneXml?	Contains the standard first part of the XML file
intestazioneComposite	Contains the definition of the composite (unused in Java)
packagedef	Contains the definition of the package
typeofcomponent	Contains the definition of the type of component (unused in Java)
implementationdef	Contains the definition of the implementation

Table 5.2: Header

body:	
propertyID	Contains the name of the Java Class
propDef*	Contains the fields of the Java Class

Table 5.3: Body

methods:	
service	Contains the implementation of the Java Class
reference	Contains the implementation of the Java Class

Table 5.4: Methods

Each under-part of the four parts is composed of a serie of tokens (see code for more details). The most important thing to know is that each of this under-part calls a method from the handler (that is defined in *@members* as an instance of the class *Handler*, which is in the *dataPackage* package). These methods are respectively called as shown in tab 5.6.

The *Handler* class is explained in chapter 7.

For whom it may concern, for the not required properties there is no error check, as they are not compulsory: if wrong spelled or not present, the result will be seen in the output Java file, where some fields will remain initialized to null. Other variables' errors are managed by Java *IOExceptions* (see chapter 8) and some methods implemented in the *Handler* class (see chapter 7).

Tokens	
INTESTAZIONE	Contains the first standard line of the XML file
INTESTAZIONE_COMPOSITE	Contains the standard line to insert a composite
COMPOSITEDEF	Contains the tag to define the start of a composite
ENDCOMPOSITE	Contains the tag to define the end of a composite
COMPONENTDEF	Contains the tag to define the start of a component
ENDCOMPONENT	Contains the tag to define the end of a component
PROPERTYDEF	Contains the tag to define the start of a property
ENDPROPERTY	Contains the tag to define the end of a property
IMPLEMENTATIONDEF	Contains the tag to define the implementation in XML (extension in the Java Class)
ENDTAG	Contains the end tag (e.g. to define the end of a implementation)
SERVICEDEF	Contains the tag to define the start of a service
ENDSERVICE	Contains the tag to define the end of a service
REFERENCEDEF	Contains the tag to define the start of a reference
ENDREFERENCE	Contains the tag to define the end of a reference
INTERFACEIMPL	Contains the tag to define a implementation
NAME	Contains the name tag
TARGETNAMESPACE	Contains the targetName tag
CLASS	Contains the class tag
INTERFACEDEF	Contains the interface definition tag
IDPROP	Contains the ID property definition tag
NICKNAMEPROP	Contains the nickName property definition tag
WEIGHTPROP	Contains the weight property definition tag
HEIGHTPROP	Contains the height property definition tag
...continues in the next page	

Tokens	
PRICEPROP	Contains the price property definition tag
ID	Contains an alphanumeric string
NUM	Contains numbers (int and float)
WS	Contains white spaces
GT	Contains the greater tag
LT	Contains the lesser tag
QUOTE	Contains the quote tag
DOT	Contains the dot tag
RET	Contains spaces and new paragraph tag
CODE: QUOTE (ID — CODE) * QUOTE ;	Contains the code block

Table 5.5: Tokens

Methods:	Called by:
handler.start()	intestazioneXml
handler.setCompositeName(CODE)	intestazioneComposite
handler.setPackage(CODE)	packagedef
handler.setTypeOfComponent(CODE)	typeofcomponent
handler.addExtension(CODE)	implementationdef
handler.setIDProperty(ID)	propertyID
handler.setNickNameProperty(ID)	propertyNickName
handler.setWeightProperty(NUM)	propertyWeight
handler.setHeightProperty(NUM)	propertyHeight
handler.setPriceProperty(NUM)	propertyPrice
handler.addImplementation(CODE)	service reference
handler.stop()	endings

Table 5.6: Methods called by myScanner.g

Chapter 6

Environment

Before defining the Handler it is necessary to define the environment and the characteristics that a Java output class needs to have. To do so, in `dataPackage` package there is a file named `Environment.java` containing a class presenting several fields. These fields are the ones representing all the characteristics needed to define a robot component as it is shown in the case study. In particular, the fields are shown in tab 6.1.

The Robot Name is univocal and is the name of the Java class as well.

The Environment constructor initializes each fields to null. In chapter 7 it will be explained how each one of these fields is set to a specific value depending on the XML file's variables.

Fields:	Description:
protected String compositeName;	Composite Name (not used in the Java class)
protected String packageName;	Package name
protected String typeOfComponent;	Type of Component (not used in the Java class)
protected ArrayList<String> extensions;	Extensions
protected String robotName;	Robot Name
protected String nickName;	Robot NickName
protected String weight;	Weight
protected String height;	Height
protected String price;	Price
protected ArrayList<String> implementations;	Implementation
Error Management:	
public ArrayList<String> errorList;	Error List
public ArrayList<String> warningList;	Warning List
public String translation;	Translation String

Table 6.1: Fields in Environment.java

Chapter 7

Handler

In this chapter it is explained how each fields of the Environment class is modified and which method is called to get to the next step of the project.

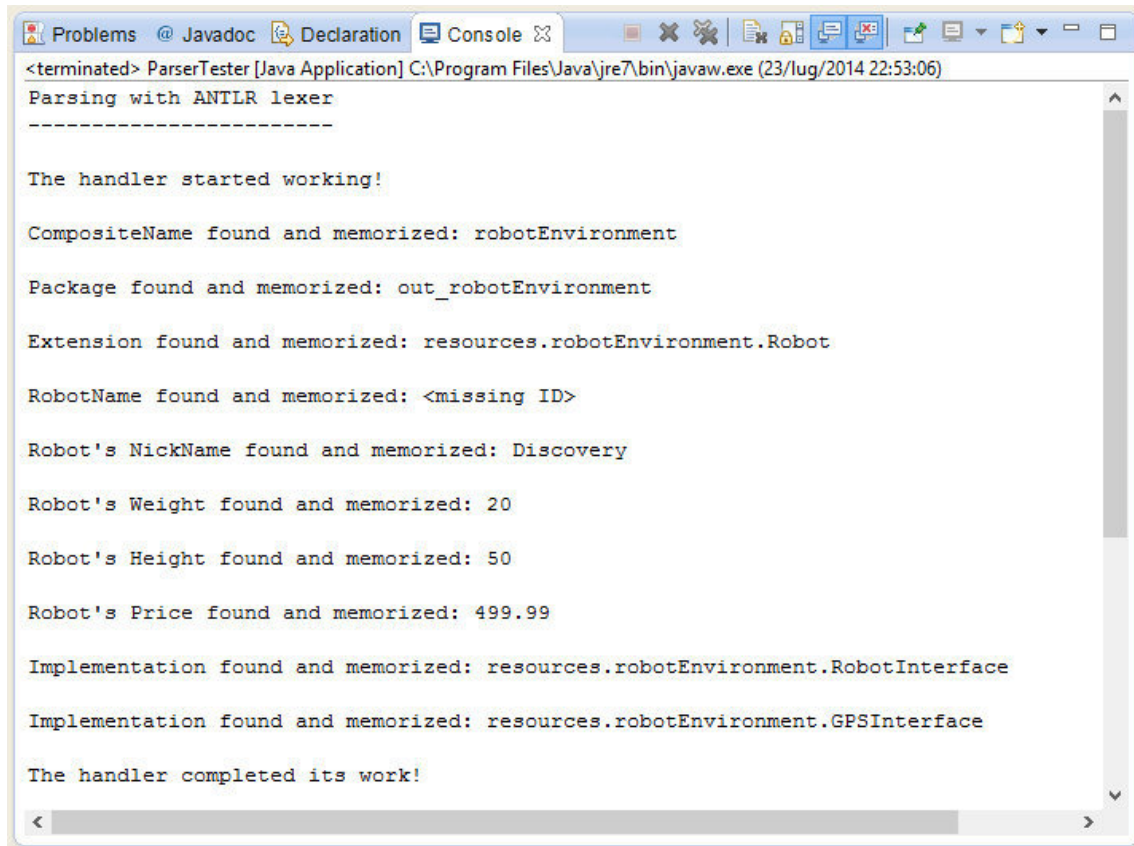
It is recommended to give another look to tab. 5.6, where it is shown which Hanlder method is called by which syntactic rule. In tab. 7.1 are shown the effects that these methods have on the Environment's fields. The object *env* is the instance of the class Environment used all through the project.

Methods:	Effects:
handler.start()	println("The handler started working!")
handler.setCompositeName(CODE)	env.compositeName=CODE
handler.setPackage(CODE)	env.packageName=CODE
handler.setTypeOfComponent(CODE)	env.typeOfComponent=CODE
handler.addExtension(CODE)	env.extensions.add(CODE)
handler.setIDProperty(ID)	env.robotName=ID
handler.setNickNameProperty(ID)	env.nickName=ID
handler.setWeightProperty(NUM)	env.weight=NUM
handler.setHeightProperty(NUM)	env.height=NUM
handler.setPriceProperty(NUM)	env.price=NUM
handler.addImplementation(CODE)	env.implementations.add(CODE)
handler.stop()	println("The handler completed its work!")

Table 7.1: Effects of the methods in the Handler class on the Environment fields

Each method contains also a command to print on the screen each step of the

execution, and the result of the Parsing is shown on the screen (img. 7.1).



```
<terminated> ParserTester [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (23/Jug/2014 22:53:06)
Parsing with ANTLR lexer
-----

The handler started working!

CompositeName found and memorized: robotEnvironment

Package found and memorized: out_robotEnvironment

Extension found and memorized: resources.robotEnvironment.Robot

RobotName found and memorized: <missing ID>

Robot's NickName found and memorized: Discovery

Robot's Weight found and memorized: 20

Robot's Height found and memorized: 50

Robot's Price found and memorized: 499.99

Implementation found and memorized: resources.robotEnvironment.RobotInterface

Implementation found and memorized: resources.robotEnvironment.GPSInterface

The handler completed its work!
```

Figure 7.1: Screen of a correct Parsing

Other methods such as `printErrors()` and `addErrors(String)` are used to manage potential errors, expecially on writing and saving the new Java file. One more method named `prepareString(Token)` is used to prepare the Token to be written as a String, as it may presents quotes or other symbols as first an last characters.

In the end, there is a method named `writeToFile()`, called by `myScanner.g` at the end of the Parsing, that simply determines the end of the Handler's work and the begin of the Writer's work. To do so, a new `CodeWriter` instance is created and the method `write()` is called. From now on, the working principles is managed by the Writer, and it is shown how in the next chapter.


Chapter 8

Writer

Now that the XML has been totally analyzed, split and memorized in the environment instance's fields, it is necessary to organize the values contained in the fields and to place them in the right place to create a Java class.

A Java class has a standard structure. In this project it is created a simple abstract class implementing and extending predetermined classes (defined in the XML file) and containing four fields initialized with the relative values (found in the XML file). The choice of an abstract class is dictated by the need to create a class as simple and standard as possible. Removing the tag "abstract" directly from the output file, it is possible to see which methods have not been implemented and which imports are needed to complete the class.

One more time, a correct standard output Java file can be seen in listing 3.4. In this case all the fields are correctly initialized because the XML file was well spelled and all the properties were present in the component definition. As said in chapter 5, although there is no error check on the fields variables, greater errors on required variables (such as the ID property) or other compulsory fields (such as the package - targetName) are managed by `IOExceptions` and `StringOutOfBoundExceptions`.

For example, in  8.1 is shown what happens if the Robot property ID is (in-

tentionally) left empty: in this case the Parser completed its work and the exception was managed at the time to create a new file, as there was no name to give to the new Java class.

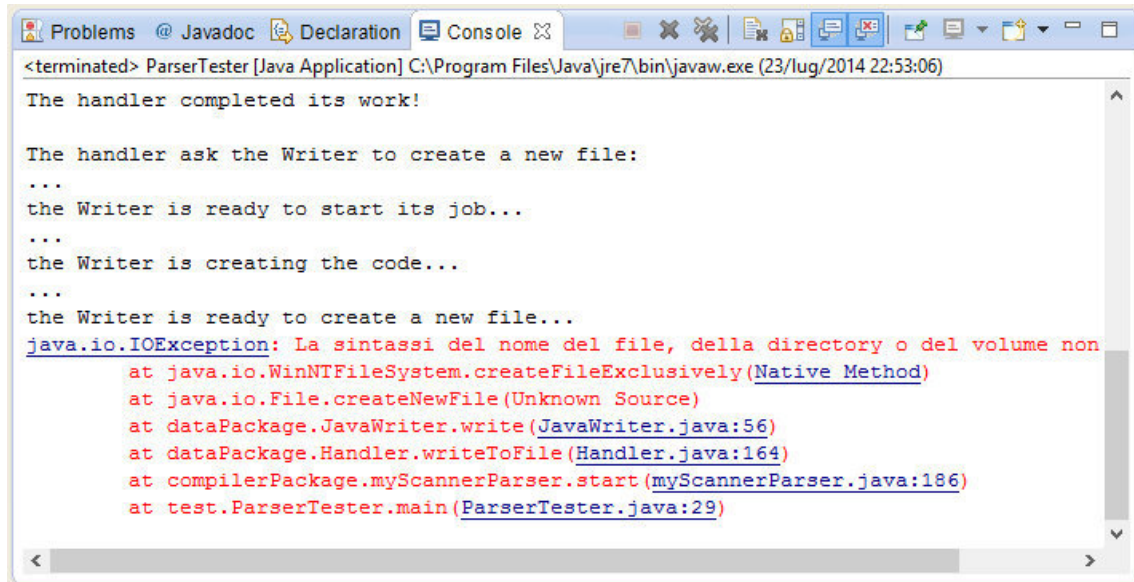


Figure 8.1: Screen of an Exception thrown because of an incorrect ID completion

Moreover, in img. 8.2, is shown what happens if the package field (targetName in the XML file) is left empty or incorrectly spelled: the handler could not even complete its work because of the incorrect spelling and there was no path where to create the new file: the execution was interrupted far before the completion of the execution.

The `JavaWriter` class, that implements `CodeWriter`, has an `Environment` object and a `StringBuilder` as fields. The `StringBuilder` is named "buff" in this project, and at first it is initialized to null. The `JavaWriter` overrides the `write()` method: three methods are called to configure the `StringBuilder` (see listing 8.3). In order, `createIntestation()`, `createPackage()`, `createClass()` are shown in listings 8.4, 8.5 and 8.6. This is the point where the user could modify or add methods to define a different output Java class.

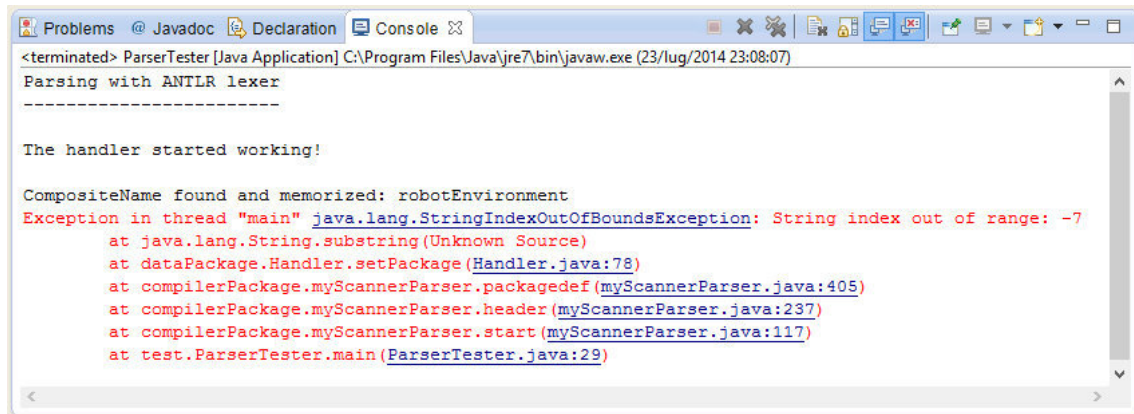


Figure 8.2: Screen of an Exception thrown because of an incorrect targetNamespace completion

```

//--Create a new file method-----
@Override
public void write() throws IOException {
    System.out.println("...\nthe Writer is ready to start its job...");

    //--File structure-----
    System.out.println("...\nthe Writer is creating the code...");
    createIntestation();
    createPackage();
    createClass();
}

```

Figure 8.3: First part of the write method in JavaWriter

```

//--Write methods-----
/**
 * Method to add the intestation comment to the buffer.
 */
private void createIntestation() {
    buff.append("//-----\n");
    buff.append("/// <auto-generated>\n");
    buff.append("/// This code was generated by a tool.\n");
    buff.append("/// @author Yamuna Maccarana.\n");
    buff.append("/// @author Gabriele Maroni.\n");
    buff.append("/// @author Giampietro Bordogna.\n");
    buff.append("/// This file can be modify, changes will not cause incorrect behaviors" +
        "of the tool.\n");
    buff.append("/// ATTENTION: Changes could be lost if the code is regenerated.\n");
    buff.append("/// </auto-generated>\n");
    buff.append("//-----\n\n");
}

```

Figure 8.4: createIntestation() method

```
/**
 * Method to add the package code to the buffer.
 */
private void createPackage(){
    buff.append("package ");
    buff.append(env.packageName);
    buff.append(";\n\n");
}
```

Figure 8.5: createPackage() method

```
/**
 * Method to add the class code to the buffer.
 */
private void createClass(){
    buff.append("public abstract class ");
    buff.append(env.robotName);
    buff.append(" extends ");

    for(int i = 0; i < env.extensions.size(); i++) {
        buff.append(env.extensions.get(i));
        if (i < env.extensions.size() - 1){
            buff.append(" , ");
        }
    }

    buff.append(" implements ");

    for(int i = 0; i < env.implementations.size(); i++) {
        buff.append(env.implementations.get(i));
        if (i < env.implementations.size() - 1){
            buff.append(" , ");
        }
    }

    buff.append(" {\n\n");

    createFields();
    createConstructor();

    buff.append("}\n\n");
}
```

Figure 8.6: createClass() method

In listing 8.6 it is possible to see that `createClass()`, in turn, calls other two methods: `createFields()` and `createConstructor()`. Their implementation can be found in listing 8.7 and 8.8.

```
/**
 * Method to create and initialize the fields in the class.
 */
private void createFields() {
    buff.append("\t@SuppressWarnings(\"unused\")\n");
    buff.append("\tprivate String nickname;\n");
    buff.append("\t@SuppressWarnings(\"unused\")\n");
    buff.append("\tprivate String weight;\n");
    buff.append("\t@SuppressWarnings(\"unused\")\n");
    buff.append("\tprivate String height;\n");
    buff.append("\t@SuppressWarnings(\"unused\")\n");
    buff.append("\tprivate String price;\n\n");
}
```

Figure 8.7: createFields() method

```
/**  
 * Method to create the constructor of the class.  
 */  
private void createConstructor() {  
    buff.append("\tpublic ");  
    buff.append(env.robotName);  
    buff.append "() {\n");  
    buff.append ("\t\tnickname = \"\");  
    buff.append (env.nickname);  
    buff.append ("\";\n");  
    buff.append ("\t\tweight = \"\");  
    buff.append (env.weight);  
    buff.append ("\";\n");  
    buff.append ("\t\theight = \"\");  
    buff.append (env.height);  
    buff.append ("\";\n");  
    buff.append ("\t\tprice = \"\");  
    buff.append (env.price);  
    buff.append ("\";\n");  
    buff.append ("\t }\n\n");  
}
```

Figure 8.8: `createConstructor()` method

The method `buff.append("Text")` is useful to add new text to the *buff* `StringBuilder`. At the end of these section the `buff` `StringBuilder` contains all the text that has to be written in a new `JavaFile`. To do so, in the method `write()` there are several lines, that can be found in listing 8.9. The line: *File file = new File ("newClassPath")*; is the key line to define the new Class path.

```
//--Write to file-----
System.out.println("...\nthe Writer is ready to create a new file...");

try {
    String path = env.packageName;
    File file = new File("./src/out_"+path.substring(4), "/" + env.robotName + ".java");
    file.mkdirs();
    if(file.exists()){
        file.delete();
    }
    file.createNewFile();
    BufferedWriter out = new BufferedWriter(new java.io.FileWriter(file));
    out.write(buff.toString());
    out.close();
    System.out.println("\nNew file created succesfully!");
} catch ( IOException e ) {
    e.printStackTrace();
}
```

Figure 8.9: Creating a new file in the write() method

Now that every step has been explained, the last thing to do is to run the tool with the ParserTester class. In the next chapter it will be explained how to set the input file and how the execution is started.

Chapter 9

Using the tool

The first step to run the tool is to give an input path to a `myScannerLexer` instance (which is the auto-generated file through ANTLR). Then, a `CommonTokenStream` is needed to give a stream to the `myScannerParser` instance (auto-generated file). In the end, the method `start()` is called and the parser starts its execution autonomously.

The input path can be given through a `String` in the main argument or can be manually set in the class itself.

After running the `ParserTester`, a new file will appear (please press F5 on the `src` folder to refresh), as shown in fig. 9.2. The output class can be seen in listing 3.4.

In the next chapter it will be explain how to modify and add new functionalities to the tool.


```
public class ParserTester {
    public static void main(String[] args) {
        //--Fields-----
        String inputPath = null;
        //--Initialize inputPath-----
        if(args.length != 0)
            inputPath = args[0];
        else
            inputPath = "./inputResources/robot.composite";
        //--Start the execution-----
        try {
            System.out.println ("Parsing with ANTLR lexer");
            System.out.println ("-----");
            myScannerLexer lexer = new myScannerLexer(new ANTLRInputStream
                (new FileInputStream(inputPath)));

            CommonTokenStream stream = new CommonTokenStream(lexer);
            myScannerParser parser = new myScannerParser(stream);
            parser.start();
        } catch (FileNotFoundException e) {
            System.out.println ("Parsing failed: FileNotFoundException");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println ("Parsing failed: IOException");
            e.printStackTrace();
        } catch (RecognitionException e) {
            System.out.println ("Parsing failed: RecognitionException");
            e.printStackTrace();
        }
    }
}
```

Figure 9.1: main method of the ParserTester class

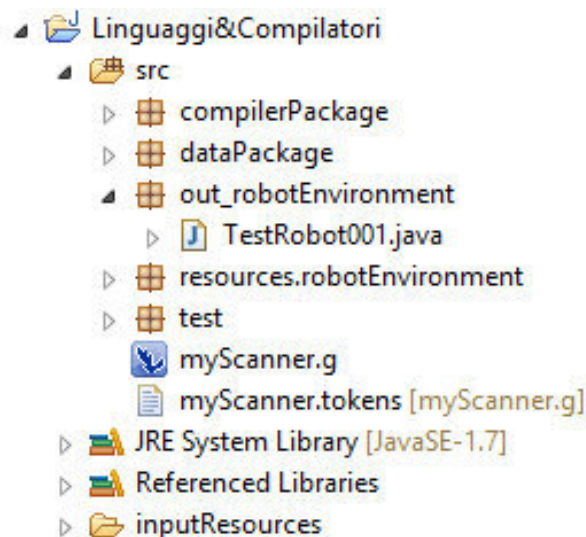


Figure 9.2: Final view of a correct configuration and run of the tool

Chapter 10

Conclusions

This project has been created as a basic tool to be extended for specific purposes. In fact, it is simply adjustable to satisfy specific intention such as, for example, adding new declarations in the XML file or creating new standard lines to the output Java class.

To do so, it is suggested to work on `myScanner.g` to create or modify Tokens and syntactic rules, on `Environment` class to create or modify fields and variables and on `Handler` class to create or modify new methods to manage the Environment's fields and variables.

Instead, to modify the output Java class, it is suggested to work on the `JavaWriter` class, and specifically on the first part of the `write` method: it is simply enough to create new methods such as `createUserMethod()` containing the command `buff.append(" UserText")` to manage the creation of the new code.

Bibliography

- [1] D.Brugali; A.Fernandes da Fonseca; A.Luzzana; Y.Maccarana:

Developing Service Oriented Robot Control System

Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium

- [2] *Extensible Markup Language (XML) 1.0 (Fifth Edition)*

W3C Recommendation 26 November 2008

- [3] S.Laws; M.Combellack; R.Feng; H.Mahbod; S.Nash:

Tuscany SCA in Action

Manning

- [4] D.K.Barry with D.Dick:

Web Services, Service-Oriented Architectures, and Cloud Computing: The Savvy Manager's Guide (Second Edition)

MK

- [5] *The Java Language Environment*

1.2 Design Goals of the *JavaTM* Programming Language. Oracle. 1999-01-01.

Retrieved 2013-01-14.