



Faculty of Economic Sciences, Communication and IT  
Department of Computer Science

Roman Yusupov  
Stefan Artan

# Domain Specific Test Language

Degree Project of 15 ECTS credit points  
Bachelor of Computer Science Engineer

Date/Term: 2012-06-05  
Supervisor: Donald F Ross  
Examiner: Thijs J Holleboom  
Serial Number: C2012:05



# **Domain Specific Test Language**

**Roman Yusupov**

**Stefan Artan**



This thesis is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previous been conferred.

---

Stefan Artan Roman Yusupov

Approved, 2012-06-05

---

Advisor: Donald F Ross

---

Examiner: Thijs J Holleboom



## **Abstract**

Testing is an important part of software development. It ensures that the developed product is of high standard and quality. Tieto frequently develops large complex systems which require comprehensive testing. Testing employs manually designed test cases. According to recent development within software testing it has been shown that design of test cases can be simplified with domain specific test languages (DSTL).

The thesis project is a research and development study in the field of software testing and has been performed at Tieto office in Karlstad. The project concerns the development of a DSTL and a suitable development environment based on the Eclipse Platform. The project evaluates the development of a domain specific test language using the tools Eclipse JDT, developed by Eclipse Foundation, and Xtext, developed by itemis AG. The experiment was evaluated by its functionality and integrability.

The project has shown promise in using a domain specific test language with a suitable development environment. The result and evaluation have shown that the subject shows promise, but needs further development if it is to be adapted within Tieto's organization.





## **Acknowledgments**

We would like to thank our supervisor Donald F Ross at Karlstad University for his guidance, support and encouragement in writing the thesis report. We would also like to thank the supervisors at Tieto, Lars Ohlén and Robert Magnusson for their support during the thesis project and for instructive mentoring in the area of testing and software development.



## Contents

Chapter 1: Introduction .....	1
1.1 Purpose .....	1
1.2 Disposition .....	2
Chapter 2: Background.....	3
2.1 Introduction .....	3
2.2 Software Development .....	3
2.3 Software Testing .....	6
2.4 Domain Specific Language .....	10
2.5 Project Discussion .....	11
2.6 Summary .....	13
Chapter 3 Implementation/Experiment .....	15
3.1 The Tester's Perspective.....	15
3.2 The Developers Perspective .....	16
3.3 Experior DSTL Project Details .....	19
3.4 Xtext Hello World Example .....	27
3.5 Xtext Hello World Language Demonstration.....	29
3.6 Experior DSTL Example.....	30
3.7 Experior DSTL Demonstration .....	35
3.8 Summary .....	41
Chapter 4: Results and Evaluation .....	44
4.1 Experior DSTL Project.....	44
4.2 The Language .....	45
4.3 Execution Mechanism .....	46
4.4 Extended Eclipse Environment .....	47
4.5 Integration .....	48
4.6 Comparison with Prior System .....	48
4.7 Project Evaluation .....	49
Chapter 5: Conclusion.....	53
5.1 The Project .....	53
5.2 Project Evaluation .....	53
5.3 Future Development .....	54
References: .....	55
Appendix A .....	57
A.1 GenerateExperior.mwe2 .....	57
A.2 Experior.xtext (EBNF) .....	60



## List of Figures

Figure 1.1: Simplified view of system .....	1
Figure 2.1: Steps of the Waterfall model .....	4
Figure 2.2 Box testing .....	8
Figure 2.3 Planning stage of Keyword-driven testing .....	10
Figure 3.1: Test View Steps .....	15
Figure 3.2: Overview of the development process .....	17
Figure 3.3: MWE2 Hello World .....	18
Figure 3.4: MWE2 Generated Code .....	18
Figure 3.5: Detailed representation of the project .....	19
Figure 3.6 Eclipse Platform .....	20
Figure 3.7 TestCase: hej.....	22
Figure 3.8: Log4j properties file .....	24
Figure 3.9: EBNF Hello World .....	27
Figure 3.10: Terminal Rules .....	27
Figure 3.11: TestCase .....	29
Figure 3.12 .....	31
Figure 3.13: ExperiorUiModule .....	33
Figure 3.14: ExperiorEObjectHoverProvider .....	33
Figure 3.15: Configuration Tab .....	38
Figure 3.16: CLI .....	39
Figure 3.17: Result in HTML .....	40
Figure 3.18(a): HelloExperior.exp.....	35
Figure 3.18(b): HelloExperior.exp.....	36
Figure 3.19 HelloExperior.exp Error.....	36
Figure 3.20 HelloExperior.exp Navigation.....	37
Figure 3.21 ExperiorDoc Example.....	37



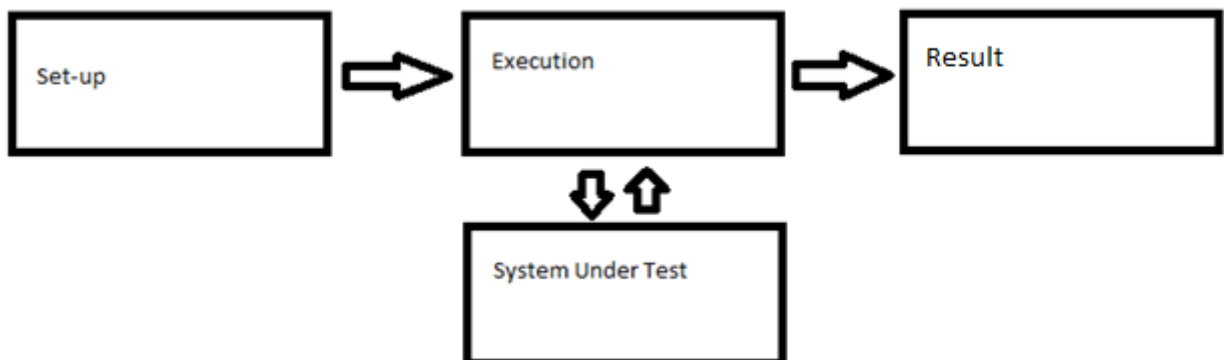
# Chapter 1: Introduction

## 1.1 Purpose

Tieto [23] is an IT service company in Northern Europe providing IT and product engineering services. The purpose of this thesis is to create a Domain Specific Test Language (DSTL) and test environment using Eclipse to test the Radio I&V system, where radio uplinks and downlinks are tested under various conditions.

The purpose of a DSTL is to simplify test scripting. The syntax of the DSTL should contain the methodology of keyword-driven testing. This project requires the development of a working environment based on the Eclipse Platform [21] for the DSTL. The tools Eclipse Java Development Tool (JDT) [20] and Xtext [28] were required as working tools for this project.

Eclipse JDT is utilized by the Eclipse Platform which is widely used as a development environment for applications, plug-ins and platforms.



*Figure 1.1: Simplified view of system*

The DSTL system of this thesis project involves implementing the following three steps: Set-up, Execution and Result as shown in Figure 1.1.

The final result of the DSTL project consists of: A domain specific test language, a working test environment and information output.

## **1.2 Disposition**

The thesis report is structured as follows.

### **Chapter 2 – Background**

This chapter presents background information about the project described in this report. A brief introduction to the underlying components and methodology are presented to give a better understanding of the project development. The chapter also presents software testing, discussing different testing methods and techniques.

### **Chapter 3 – Implementation/Experiment**

The chapter contains implementation details of the project development along with the examples of the resulting product. The implementation details will describe different components used in production. The examples will demonstrate the functionality of the produced result.

### **Chapter 4 – Results and Evaluation**

This chapter contains the evaluation on the choices made during the development described in chapter 3, along with evaluation of the project as a whole. The evaluation will contain a discussion of choices made regarding the use of tools and implementation choices. This chapter will also contain some reflections on the project.

### **Chapter 5 – Conclusion**

This chapter concludes and summarizes the thesis rapport. It contains a summary of the result of the project, a project evaluation and thoughts on future development.



## **Chapter 2: Background**

### **2.1 Introduction**

This chapter presents background information for the project. Section 2.2 covers the software development processes, different models of product development and activities. Section 2.3 is discussing the software testing techniques, methods and aspects of testing. Section 2.4 briefly introduces the term Domain-specific language. Finally section 2.5 covers some preliminary thoughts about the project.

### **2.2 Software Development**

The software development process can vary in its approach depending on what software development model is being employed.

#### **2.2.1 Models**

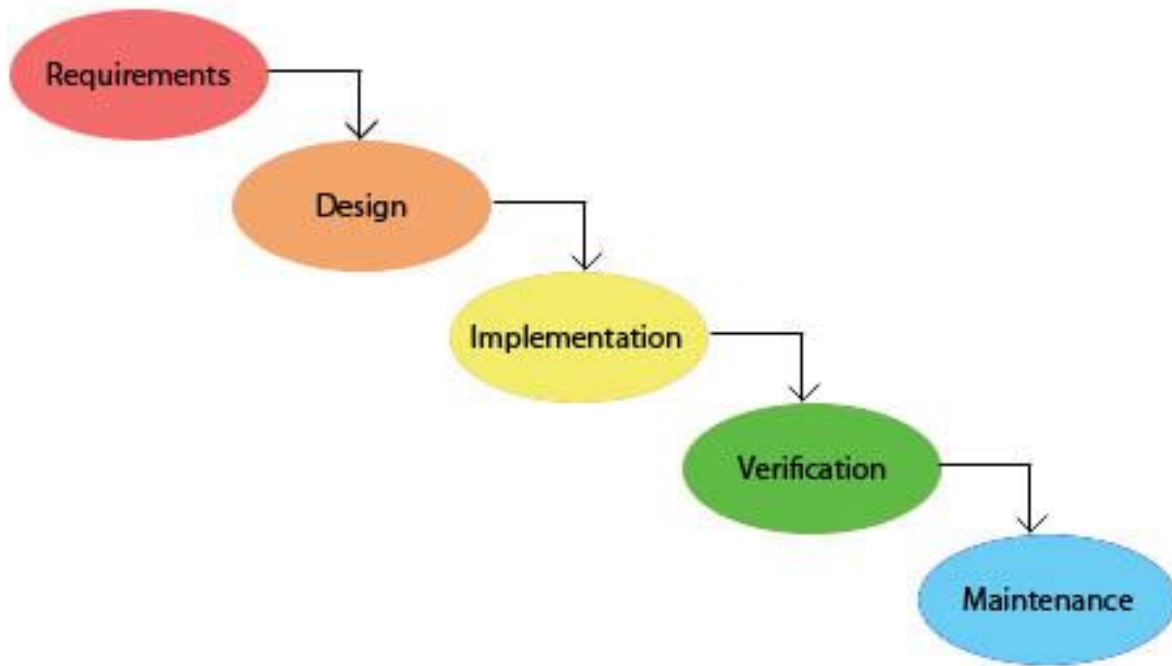
Software development process models are techniques that help manage projects. Each one has its advantages and disadvantages. Based on those advantages and disadvantages the developers should consider which model is the most appropriate to employ on the project development.

#### **Waterfall model**

The Waterfall Model [1] is a step by step approach that was invented by Dr Winston Royce in 1970s where a project is developed in different phases (Figure 2.1).

The thought is to approach a problem by first analyzing it, making a design based on requirements and the best way to fulfill them. The implementation phase should only start after the project is well thought out. Verification is the process of testing and improvement of the program after which the product is ready to be deployed and maintained.

This development model can help to create stable well thought out projects. However it could only work on smaller projects due to bigger projects having higher risks since the planning process takes longer time, which might cause the project not to be finished on time. By the time the implementation is finished to fit the design and the requirements those might already be out of date and the requirements may have changed. [2]



*Figure 2.1: Steps of the Waterfall model*

### **Iterative and Incremental Development Model**

As the name suggest the Iterative and Incremental Development Model (IIDM) [3] consists of two development strategies: incremental and iterative. In incremental development the developers develop different parts of the systems at different times and rates, integrating them as each part is finished. The iterative development dictates to set aside time to revise and improve parts of the system.

### **Spiral Model**

The Spiral Model [5] combines the IIDM and Waterfall Model. The model is based on incremental releases of the product refining the key elements with each release. The spiral model can be seen as an iterated version of the Waterfall Model where it releases several prototypes improving each prototype by adding or refining elements.

## **Agile Development Model**

The Agile Development Model (ADM) [6] uses the iterative development model (IIDM) as basis, but focuses its development on general feedback instead of the planning. The feedback is the control mechanism based on continuous testing and releases. The agile development model divides tasks into much smaller parts of a functioning program. The model time boxes<sup>1</sup>, and the time spent on development of different parts. The emphasis is on adding/removing functions depending on general feedback.

### **2.2.2 Activities**

Essentially, regardless of the model the activities are the same: Planning, implementation, testing, documenting, deployment and maintenance. Looking at Figure 2.1 as reference, the following sections will show examples and explain different activities. [7]

#### **Planning**

The first two phases in the waterfall model: the requirement and the design are the phases of the planning activity. In the planning activity the developers look at the requirements and make design plans on how to structure the product. Developers also look for any unnecessary requirements and missing components.

#### **Implementation, Testing and Documentation**

These three may be considered as one coherent activity. In the waterfall model these three take place in the implementation and verification phases. The implementation activity consists of the implementation (i.e writing the code and structuring the components). The implemented code should always be tested in several ways (different levels of testing are covered in part 2.3 Software Testing). Nearly all code written needs to be well documented for further use and maintenance since documentation of the code makes it easier for other developers to work with the code.

---

<sup>1</sup> <sup>1</sup>

Time boxing is a term in time management meaning that a fixed period of time is allowed for a given activity.

## **Deployment and Maintenance**

Finally the last phase in the model is deployment and maintenance. The product is deployed only when it has been properly tested and has been approved for the deployment. After the product has been distributed it is maintained by updating to newer versions or releasing an update that removes issues found after the product has been deployed.

## **2.3 Software Testing**

“Software testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.” [8]. The main purpose of software testing is to find errors in an executing program or system. Testing is an important part of software development. The main purpose of testing a system is to improve the overall quality of the system. [9]

### **2.3.1 Software Testing Methods**

There are several different software testing methods but the most popular is the so called “box testing”. Box testing consists of two main methods White- and Black-box testing, but there is also a third method called Grey-box testing which is a combination of the two. [8, 9, 10]

#### **White-box testing**

White-box testing (Figure 2.2 (a)) completely involves the tester in all aspects of the test. The tester is exposed to the data structures and the algorithms including the code that implement these. While making a white-box test the tester has to take in to consideration all the details about the software, such as the programming language used. The tester does this in order to more easily single out the point of failure. White-box testing is often called glass-box testing.

## **Black-box testing**

Black-box testing (Figure 2.2 (b)) only requires the testers to be aware of the input and output for each function. The testers do not have to have any knowledge of how the application itself is structured or implemented. For instance, the testers are creating a database system. The testers create a number of queries and only check if the results match the expectations. It is called black-box testing because during the test phase the testers handle the software under test as a black-box (hence the name), only the input and the output is visible. To check if the functionality of the software is correct the testers have to match the input with the corresponding output.

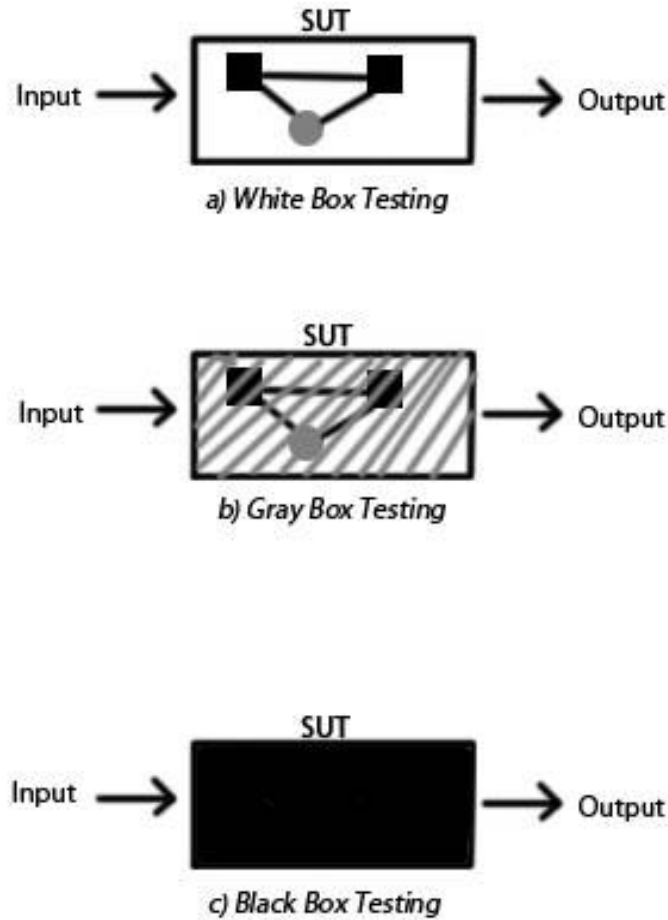
## **Grey-box testing**

Grey-box testing (Figure 2.2 (c)) is often seen as a combination of the white-box testing and the black-box testing, the tester does not have to have any knowledge of the internal data structures and algorithms in order to design tests. The tester is only to some degree exposed to the underlying concept of the tests, but not in the same way as in white-box testing.

### **2.3.2 Software Testing Levels**

Software testing can be divided into levels depending on what stage of the software development process they are implemented at. The different levels are:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Regression Testing



(Fig 2.2 Box testing)

## Unit Testing

Unit testing [30] or component testing is isolation of components in a test case. This means that a larger system is divided into smaller “Units” where their functionality is tested. A unit can contain smaller sections of code to test a specific function. Unit tests are often written by the programmers at the same time as the code is being developed. This means that they have full contact with the code (White-box testing). These tests are called “test cases” and are created to test a specific case, test cases can also be combined with other test cases to create a “test suite” which is a collection of several test cases that are executed during the same test session.

## **Integration Testing**

Integration tests [31] are made to test the interaction between system units (e.g computers and servers) such as interfaces and plug-ins. These system units are typically integrated step by step to be able to perform tests at each step of the implementation. A different approach would be to implement the system units at the same time. In most cases the step by step implementation is preferred due to the fact that it is easier to isolate and locate problems. This means that after each step if any problem should occur it can be handled at once. The main purpose of integration testing is to identify the faults in the interaction between the system units.

## **System Testing**

System testing [32] is testing of a completed integrated system to check that the system fulfills its requirements. This kind of testing usually involves testing of load, performance and security testing. At this stage most resources are used for testing, this is performed by the entire test team and is executed on a separate machine for testing the system on an external machine (or typically on a simulator or virtual machine).

## **Acceptance Testing**

Acceptance testing [33] is the only tests that are performed by the customer, often on their own hardware. This is typically referred as user acceptance testing. These kind of tests are often performed simultaneously with the system being deployed at the customer side. Acceptance testing are in most cases merged with the system testing.

## **Regression Testing**

During upgrades of functionality to a system, the previous functionality may regress (to revert to an earlier or less advanced state) or fail. Regression testing [11] is aimed to catch these kinds of errors as soon as possible. The purpose of regression testing is to encounter errors not obvious to the developers and to make sure that the additions to the system do not break it.

### 2.3.3 Keyword-driven Testing

Keyword driven testing (KDT) [12] is scripting of test cases for various systems which until now have mostly been written in non-domain specific languages. This is a method of writing tests that separates the programming aspect of testing from the actual tests. In its essence keyword driven testing is a generalization of the technique of testing software. Keyword driven testing can be divided into two stages:

- Planning Stage
- Implementation Stage

The planning stage is where all the keywords are defined. More complex operations are simplified into keywords such as “start”, “get” and “check” (consider Figure 2.3 as example). These keywords are determined by analyzing the application and finding which operations are most used. The next step is to implement a framework where the testers can use the keywords defined in the planning stage. By doing this testers can write test cases based on the defined keywords and thus eliminating the need for any previous knowledge in programming. A test case consists of a series of operations with at least one keyword that tests a given application’s behavior and functionality. [12]

Object	Action	Data
Textfield (domain)	Enter text	<domain>
Textfield (username)	Enter text	<username>
Textfield (password)	Enter text	<password>
Button (login)	Click	One left click

Figure 2.3 Planning stage of Keyword-driven testing [12]

### 2.4 Domain Specific Language

DSL or Domain Specific Language [13] is a specific programming language which handles a specific problem area. This means that the language is only created to handle a specific task. The focus of the domain area can differ from a language developed for testing to a language for flight trajectories. In contrast to “general purpose programming languages” which are made for programming for any general purpose, the Domain Specific Test Languages (DSTL) are custom made for a specific task. A domain specific language is categorized between a scripting language<sup>3</sup> and Tiny programming language<sup>4</sup>.

---

3

A programming language that supports the writing of scripts. [29]



When developing a DSL there are usually two parts, the definition of the language i.e the grammar and syntax defined usually in a mathematical notation to escape ambiguity. The second part is an execution mechanism in form of (and in some cases a combination of) compiler, interpreter or code generator. Briefly a compiler transforms the code into executable machine code that is used by the processor to execute it. The interpreter goes through source code, executing each command encountered. The code generator translates the DSL code into one or several general-purpose languages.

## **2.5 Project Discussion**

The concept behind this project is considered new in the world of software testing. Even though domain specific languages or keyword driven testing has existed for a longer period of time the idea to create a domain specific language for testing is new. Therefore it will to some extent be hard to find some means or examples for how it could or should be done. The concept of DSTL is to build a language on a higher level than of the high level general-purpose language, so that writing test cases would not require thinking about code implementation of the tests. In other words testers would compose test cases regarding what should be done and not how it should be done.

### **2.5.1 Current System**

The current system requires the testers to have programming skills. The tests are written in the E language [24]. When the tester has written the test a java tool called Ateng [25] generates XML [18] and PERL [26] files from the written E code together with the executable code which is to be executed to test the underlying system.

### **2.5.2 Goal**

The main goal of the project is to make the product useable for testers who are lacking programming skills. For instance tester can come up with hundreds of examples where faults may occur, but by not being a programmer this might inhibit their ability to test a system. So the goal is to reduce or remove the programming aspect of test scripting. All that testers would have to do is to write the test cases that consist of commands with self explanatory keywords like: `DELETE-CONTENT <field> or ASSERT <field> IS EMPTY.`

---

4

Tiny is an actual name of a programming language and not the size measurement. [14]

### 2.5.3 Different parts of the product

The development begins by developing the product in three parts: the Language and definition part, the Launcher and executor part and the interpreter part.

#### **Language and Definition**

In the language and definition part it is required to structure and provide a good documentation for the language. This means that the documentation presented will explain how the grammar of the language is structured. It is also required to provide a list of all the keywords, their meaning and how to use them. The language part also defines the language grammar definition in Extended Backus Naur Form (EBNF).

EBNF is an extended version of Backus Naur Form (BNF) [16]. BNF is a formal mathematical way to describe the grammar of a language. The Xtext tool uses the EBNF definition to generate a parser for the DSL.

#### **Launcher and executor**

The launcher and executor are divided into two parts. It is required to have a graphical development environment for test developers to be able to work with the language and execute the tests without a need to switch to another environment. The part of the launcher and executor will be developed as an Eclipse Plug-in.

#### **Interpreter**

The interpreter is implemented in several basic steps. The interpreter should be able to be started as a runnable application. The interpreter should be able to report results. That means that it should be able to write output to the tester which tests have passed and which ones have failed and also be able to report any errors in the language or some exceptions that it has come across while in execution stage. The interpreter should contain some external drivers to simplify the work of defining different protocol standards. The interpreter also needs to be extracted as an external package so that it is not bound to the development environment and if needed can be launched from the command line interface (CLI). And finally the interpreter should be able to interpret the different commands and execute them.

## 2.5.4 SCRUM and Agile Development

The project is to be performed using agile development and SCRUM [15] techniques. It means that the project is to be divided into several sprints (a time-boxed period of development time), each sprint being two to three weeks long. It also means that a horizontal approach to the product development will be taken. It means that all parts of the project will be implemented during the first sprint and then the functionality of those parts will be extended during each sprint. This is an agile way of developing a system, it provides simplicity for making changes in the system when needed and being able to demonstrate results to the customers.

## 2.5.5 Advantages / Disadvantages

The major advantage that comes with the DSTL is that tester's main focus becomes the actual testing, instructing what to test instead of how to test it. Another advantage is that the code becomes more self-documenting. As mentioned earlier the keywords of the DSTL are quite self-explanatory. Therefore test cases made with the DSTL should be easy to understand, since each command tells almost as in common spoken language what it does. There is also the advantage of re-usability of the language on different systems. The language can be wrapped around other languages or even use a code generator to generate tests in code of specific general-purpose language.

For testers who already are programmers, there is a slight consumption of time, learning new language that perhaps is unnecessary for them. Since the language is no longer a general-purpose language it is limited to its purpose. Also by being one level above a general-purpose language it reduces the efficiency from the machine perspective i.e it takes longer to process.

## **2.6 Summary**

In this chapter the background to the thesis subject of DSTL and introduction to some of the components used for research of the subject has been presented. An overview of software development processes has been introduced, which covers models and activities of software development such as planning, implementation, testing, documentation, deploying and maintenance.

White, Gray and Black-box testing was all part of the software test methods discussed in the part about Software testing which also included software test techniques such as keyword-driven testing. Finally software testing levels were discussed and explained.

Section 2.4 gave a brief introduction to DSL and its components divided in grammar and compiler/interpreter/code generator.

The last section 2.5 included the discussion about the DSTL project of the thesis including the goals, main components, advantages/disadvantages and the work methodology.

## Chapter 3 Implementation/Experiment

This chapter presents the structure of the Experior DSTL and describes the different parts and components as well as the tester's and developer's perspectives. After the brief introduction on the structure of the Experior DSTL, further implementation and structure details will be presented as examples.

The Experior DSTL project can be divided into two different perspectives: the tester's perspective and the developer's perspective. The outcome of the development process is a test environment in which the testers can design and execute tests.

### 3.1 The Tester's Perspective

Essentially the tester's view can be divided into three steps: Set-up, Execution and Result. (Figure 3.1) As result the test environment and DSTL will look as presented in Figure 3.18.

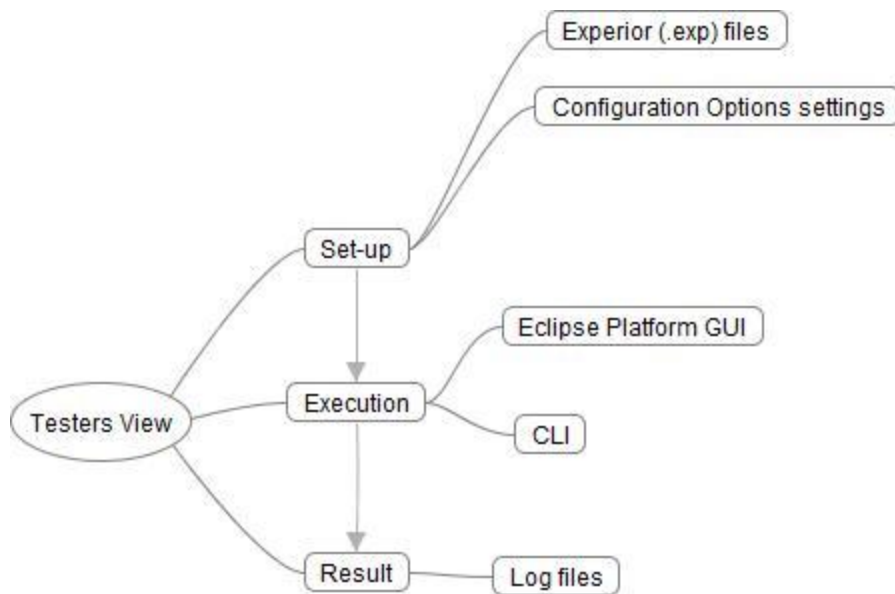


Figure 3.1: Test View Steps

The first step is the “Set-up”, which is writing the test cases in text files with the .exp extension and configuring the root folder which is the destination of the result files. All test cases end with one or more ASSERT commands where the tester confirms whether the results of a test match the expectations. The test scripts may be written in any text editor, but it is preferable to write them in an Eclipse based environment which is produced for the Experior DSTL. The Experior DSTL environment will assist in implementing test cases with various features.

In the second step, the “Execution”, the test cases may be executed in two different ways. One of the options is to execute them in the Experior DSTL development environment (GUI<sup>1</sup>) where there is a “Run” button that executes the test. The second option is to invoke the interpreter from CLI<sup>2</sup> with the help of the Java Runtime Environment, input files and configuration options as arguments.

The last step the “Result” has to be analyzed manually by the tester, meaning that the tester looks at the information that was produced as output from the execution step. The output information consists of log files containing information about the test cases’ executions. The result stage yields the test cases’ verdicts (assertion results: either passed or failed), collected information and possible errors that might have occurred along with the execution.

### ***3.2 The Developers Perspective***

To produce the development environment as described in section 3.1 (The Tester’s Perspective) the project has to undergo several development stages as described in Figure 3.2. This stage of the development process in developing a DSTL and a fitting environment is the purpose of the thesis and thus the largest part of the thesis report.

The development process can be described in three steps namely “Set-up, Execution and Result” (Figure 3.2).

---

<sup>1</sup>  
Graphical User Interface

<sup>2</sup>  
CLI or Command Line Interface is a non-graphical interfaces that exists in nearly all operating systems (e.g Command Prompt in Windows or Terminal in Linux)

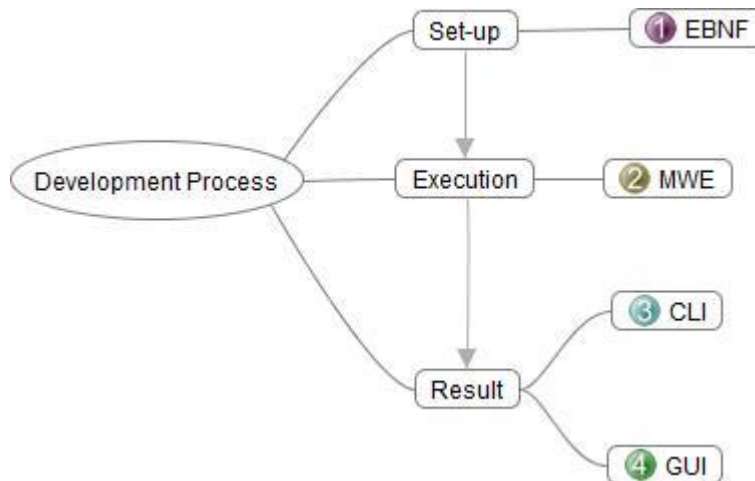


Figure 3.2: Overview of the development process

### 3.2.1 EBNF

EBNF [16] or Extended Backus-Naur Form, which belongs to the “Set-up” step, is a meta-language notation to define a programming language. EBNF defines the grammar of a programming language using a quadruple  $G = (S, P, NT, T)$  where  $S$  is the start symbol,  $P$  a set of production rules,  $NT$  the set of non-terminal symbols and  $T$  the set of terminal symbols of the grammar. The terminal symbols are very basic components usually defined as symbols or sequences of symbols most commonly known as integers, strings or IDs (rule for variable naming). The production rules are for combining the non-terminal symbols and terminal symbols.

### 3.2.2 MWE

MWE [17] or in case of the thesis MWE<sup>3</sup> (Modeling Workflow Engine 2 ) belongs to the “Execution” step. The MWE<sup>2</sup> is a declarative, configurable generator engine. Using EBNF as an input it can declare object instances, attribute values and references. There

<sup>3</sup>

The “2” signifies that it is a second version. MWE<sup>2</sup> is a rewritten, backwards compatible implementation of MWE.

are components in the work-flow that read EMF<sup>4</sup> resources, generates and writes back a number of artifacts.

The simplest MWE2 module, the “Hello World” is presented in Figure 3.3:

```
module HelloWorld

SayHello {
    message ="Hello World!"
}
```

(Figure 3.3: MWE2 Hello World)

From which MWE2 will generate:

```
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowComponent;
import org.eclipse.emf.mwe2.runtime.workflow.IWorkflowContext;
public class SayHello implements IWorkflowComponent {
    private String message = "Hello World!";
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
    public void invoke(IWorkflowContext ctx) {
        System.out.println(getMessage());
    }
    public void postInvoke() {}
    public void preInvoke() {}
}
```

(Figure 3.4: MWE2 Generated Code)

To summarize, with EBNF as input MWE2 will produce necessary artifacts that are essential to the language.

### 3.2.3 CLI

The first part of the “Result” step is the Command Line Interface (CLI) which is a non-graphical user interface for the system. The CLI of the Experior DSTL is used by executing a jar file containing the interpreter and all of its dependencies. The execution of a jar file is completed through the Java Runtime Environment (JRE) with following the structure: `java -jar Interpreter.jar <Interpreter arguments>`

---

4



### 3.2.4 GUI

The second part of the “Result” step is the GUI of the Experior DSTL development environment which is a stand-alone Plugin to the Eclipse Platform. This means that the Eclipse environment acquires extended functionality to be able to work with the Experior DSTL projects and recognize its files (.exp) as a known file format. The work on the GUI was briefly described in section 3.1 (Testers View) as it is the visible result of the project. The GUI offers the standard features of the Eclipse Environment such as an editor (including text highlighting, real-time graphical support and content assist), a console, a project browser, all of the general Eclipse Views and also views developed specifically for the Experior DSTL test development. Because the Eclipse Platform is self contained the user can run all the projects within the platform by pressing the run button, at which point a new Java Virtual Machine (JVM) instance is started and controlled within Eclipse.

### ***3.3 Experior DSTL Project Details***

So far this chapter has only presented an overview of the main components of the project. This section will present the details of the development and structure of the Experior DSTL project including descriptions of all the main components.

Figure 3.5 is an extended version of Figure 3.2 presenting an elaborated version of the Experior DSTL system.

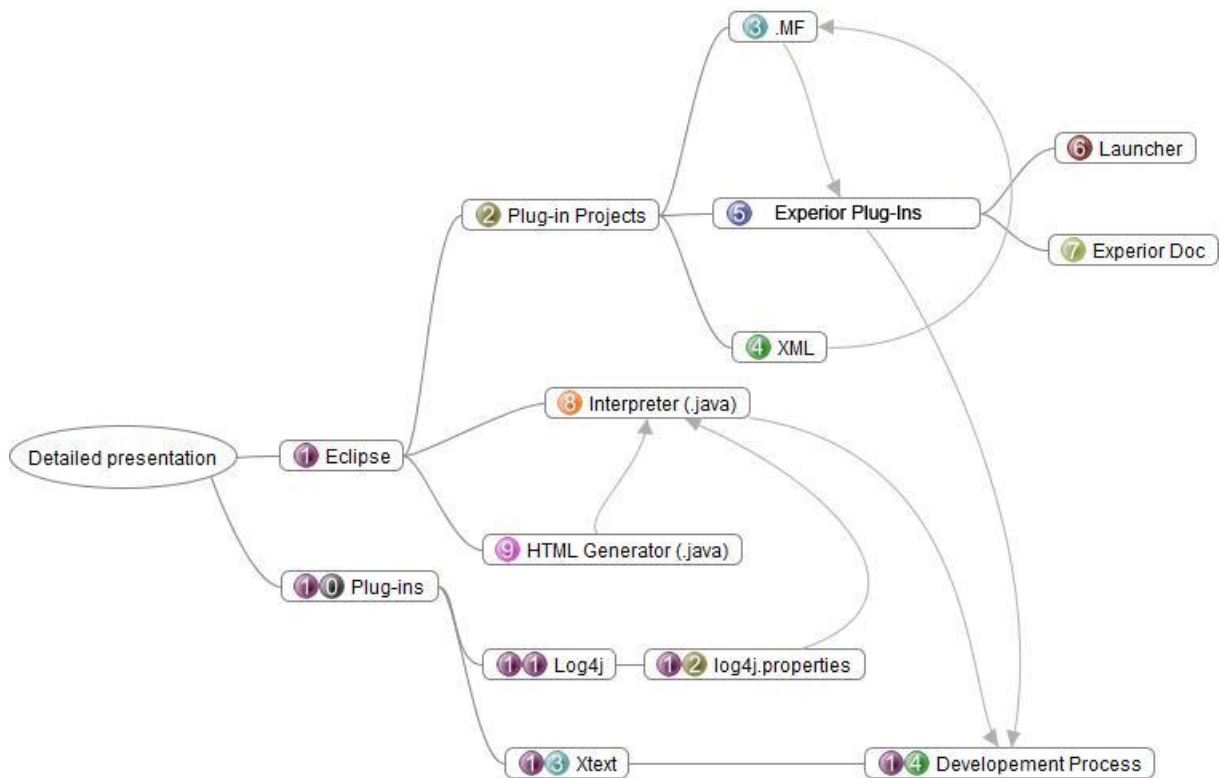


Figure 3.5: Detailed representation of the project

### 3.3.1 Eclipse

When talking about “Eclipse” it is generally meant the Eclipse Software Development Kit (SDK) which is a leading Java Integrated Development Environment (IDE). Eclipse SDK is a combination of the following components: Eclipse Projects, Eclipse Java Development Tools (JDT) [20], The Eclipse Platform [21] and the Eclipse Plug-in Development Environment (PDE) [22]. The Eclipse Platform provides the functionality required to develop an IDE. As presented in fig 3.4 the Eclipse Platform itself is a set and a subset of other components that make Eclipse Platform capable of serving as an environment not only for IDE development but also for the development of arbitrary tools and applications. The Eclipse RCP is a platform for building and providing rich client applications.

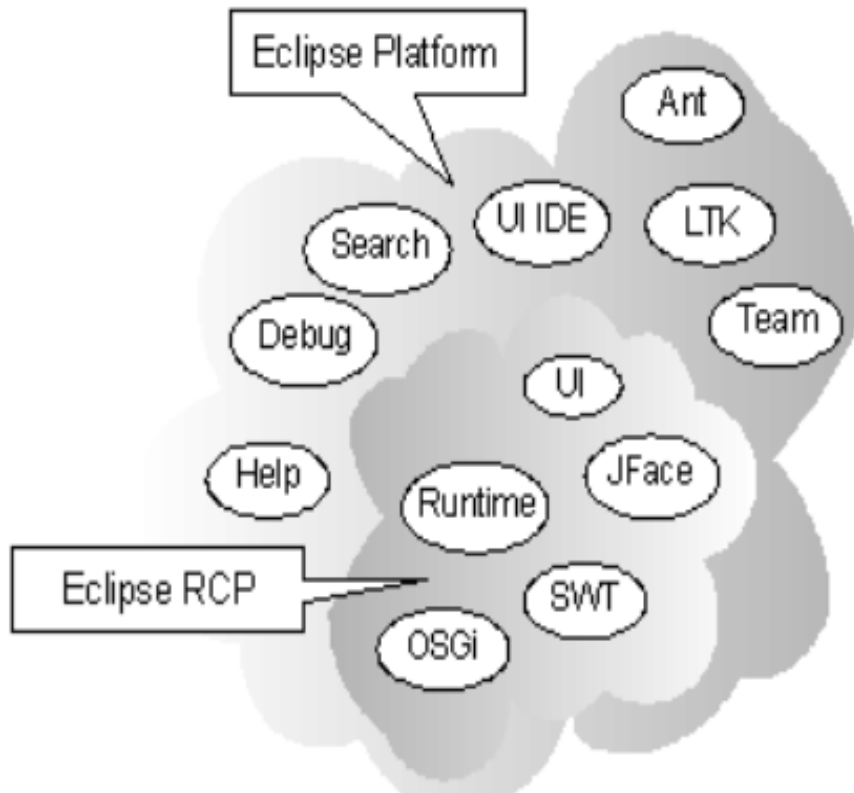


Figure 3.6 Eclipse Platform

As presented in Figure 3.6 the Eclipse RCP does not provide any specific development tools, The Eclipse RCP is included in the Eclipse Platform which adds the programming tools and services to the environment.

The key benefit of using the Eclipse Platform is its use as an integration point. The tools and applications developed in Eclipse become integrable with each other within the Eclipse Platform. This benefit allows the development of the Experior DSTL by using the Xtext tool (which is presented in 3.3.13) and Eclipse JDT.

### 3.3.2 Plug-in Projects

The plug-in projects are separate “Eclipse plug-in development projects” where the plug-ins and features for the Eclipse Platform are developed. Eclipse Plug-ins are described in section 3.3.10. Besides the functionality of the plug-in written in Java, the plug-in project contains a manifest (.MF section 3.3.3) file that summarizes the project.

In Experior DSTL a plug-in development project is created to extend the Eclipse functionality to make the Experior DSTL a recognizable language to Eclipse (Experior Plug-Ins section 3.3.5).

### 3.3.3 .MF (Manifest File)

The manifest file is a Java feature that defines the components, dependencies and contributions. The manifest file refers to the imported packages of the project, packages from the project that are exported and a plugin.xml (XML section 3.3.4) file.

### 3.3.4 XML<sup>6</sup> (plugin.xml)

The plugin.xml file is an Eclipse feature which is used to define what a component provides and uses. The plugin.xml file is written in XML syntax. The plugin.xml file describes which points of the Eclipse platform the project extends the functionality of (i.e. which extensions are made and which extension points are used). When a description of an extension is made it also gives the extension an ID and refers to the package(s) that contain(s) the extension classes.

### 3.3.5 Experior Plug-Ins

One of the requirements for the thesis project is to develop a suitable development environment for the Experior DSTL. It is necessary to develop a plug-in that can recognize the Experior DSTL as a language within the Eclipse Platform, to be able to run Experior (.exp) files within the Eclipse Platform, to provide graphical support such as content assist and real-time syntax checker and to provide some additional functionality. A plug-in managing the content assist and graphical support is provided by the Xtext tool (section 3.3.13). The Experior DSTL plug-in has two extensions. One extends the run configuration options providing launcher (section 3.3.6) for Experior DSTL. The second is ExperiorDoc (section 3.3.7) which extends the Eclipse Platform with an additional view that presents multiline comment documentation (much like JavaDoc).

### 3.3.6 Launcher

The Experior DSTL Launcher extension provides an additional run option in the “Run Configurations” window in Eclipse. The launcher has two functions: to set the launch configurations and to bridge (start) the interpreter. The Experior DSTL Launcher configuration has two tabs: “Main” and “Configuration”. In the Main Tab the user is able to choose which Experior Project and which test script (.exp) file they are going to execute. The

---

<sup>6</sup>

XML or Extensible Markup Language is not presented in this thesis, for more information see [18]

second tab “Configuration” is for specifying a root folder path where all the output will be stored. In the backend the launcher is responsible for setting up a runtime environment for the Experior DSTL. It injects all the dependencies, retrieves all the user input and sets up an environment based on this information and passes the rest of the information to the appropriate components such as the Interpreter.

### 3.3.7 ExperiorDoc

The ExperiorDoc extension provides an additional viewpart to the Eclipse Platform. This view collects information from the editor, much like JavaDoc, the ExperiorDoc extracts a multiline comment from the editor when a marker is placed on a declaration of a test case printing it in a viewpart window. Taking figure 3.6 as example the declaration of a test would be `TestCase hej` so placing a marker anywhere on that line would trigger the Experior Doc view to display a multiline comment above which in this case would be “Hej hej hej”. Alternatively hovering over the same line with cursor will bring up a tooltip window with the same documentation, parsing HTML tags if there are any.

A screenshot of an IDE editor window titled 'hej.exp'. The editor shows a multiline comment `/** * Hej hej hej */` followed by a test case declaration `TestCase hej` and its implementation `GET "http://www.google.se" ASSERT http.response 200`. A tooltip window is displayed over the `TestCase hej` line, showing the comment text: `/** * Hej hej hej */`.

Figure 3.7: *TestCase hej*

### 3.3.8 Interpreter

The interpreter of the Experior DSTL is the execution mechanism used to interpret the Experior Test Case script. It reads the Experior (.exp) file and executes the commands line by line.

The interpreter is a stand-alone java application called by the Experior DSTL environment. When a user presses the “Run” button the launcher configuration runs the `Interpreter.class` and hands it the arguments as if through a command line interface. The first argument is a root folder where the output containing the logs of execution results will be stored. To produce output the interpreter uses an `HTMLGenerator` class (section 3.3.9) and a logger provided by Apache’s `log4j` (section 3.3.11). The remaining arguments to the `Interpreter` are one or more .exp files that are to be executed. The interpreter makes a call to the parser to first check if the syntax of the file is correct before it can begin to interpret the

file. It does that to make sure that the system will not crash because of a syntactical error along the process of execution. The interpreter executes the commands of the Experior file and stops when it reaches the end of the file, at which point it moves on to the next file provided in arguments and if there are none the execution stops.

### 3.3.9 HTML Generator

The function of the HTMLGenerator is to generate an HTML file to display the logger produced output in a more reader friendly markup rather than raw data. Each time HTMLGenerator is called it creates a new `htmllog.html` file or overwrites an existing one. After generating an empty HTML file it reads a `userlog.log` file produced by log4j logger (section 3.3.11) parsing the information into expandable tables highlighting the test case names with green (passed) or red (failed). Once expanded, the tables show detailed information about the test along with the verdicts.

### 3.3.10 Eclipse Plug-ins

Eclipse Plug-ins are tools added to the Eclipse Platform as “pluggable” components. An Eclipse plug-in is a component that provides an extended service to the Eclipse development environment. The Eclipse plug-in architecture supports several plug-ins being activated and operated on at the same time in order to provide the service needed to accomplish a given task. Plug-ins exist in a running instance of Eclipse as plug-in classes, these classes provide configuration options and support for the plug-in. Since Eclipse plug-ins are configurable and integrable with the rest of the platform, all the Eclipse Platform’s functionality (except for a small kernel known as Platform Runtime) is located in plug-ins.

To develop Experior DSTL there is a need to use Eclipse Plug-ins. In this thesis report, two of the major plug-ins used for this project will be presented. One of them is the plug-in known as Xtext (section 3.3.13). Xtext is an open source tool developed by itemis AG and comes as a separate plug-in (not installed by default) to the Eclipse Platform. The second plug-in is log4j (section 3.3.11) developed by Apache Logging Services™. log4j is a tool to help the programmer output log statements to a variety of output targets.

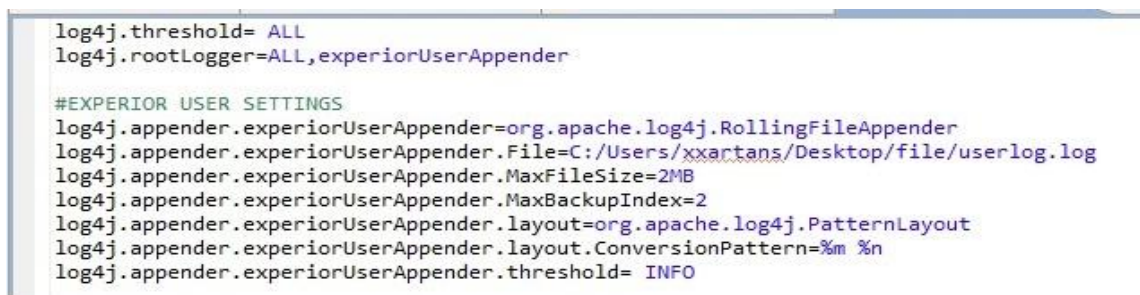
### 3.3.11 log4j

log4j is an information logging utility. It is widely used in a variety of systems to replace the use of a common system output. The benefit of using log4j is that to change the

format of information output (e.g. to switch from run mode to debugging mode), a programmer does not need to change the code of the system, but instead only change the configurations of the log4j. Configurations are made by modifying a log4j.properties file (in some cases log4j.xml file) (Figure 3.8). This file contains information regarding the logging of information such as: the structure of the output, the level of logging the file destinations, output file extension and how to divide the information among files.

log4j is used in both the development and testers phases of this project. In the development phase the plug-in Xtext uses the log4j logger to log information about MWE Workflow while building a language (generating the necessary files for the language parser). However this logger is of no relevance to this thesis project and is not modified. This logger is there to collect information about the language building. The log4j logger that is of concern is the one that will log information gathered from executing test cases from the Experior DSTL.

The final output from the logger is either an .log file or a .HTML file. The .log file is generated from log4j while the .HTML file is generated by Experior.



```
log4j.threshold= ALL
log4j.rootLogger=ALL,experiorUserAppender

#EXPERIOR USER SETTINGS
log4j.appender.experiorUserAppender=org.apache.log4j.RollingFileAppender
log4j.appender.experiorUserAppender.File=C:/Users/xxartans/Desktop/file/userlog.log
log4j.appender.experiorUserAppender.MaxFileSize=2MB
log4j.appender.experiorUserAppender.MaxBackupIndex=2
log4j.appender.experiorUserAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.experiorUserAppender.layout.ConversionPattern=%m %n
log4j.appender.experiorUserAppender.threshold= INFO
```

Figure 3.8: Log4j properties file

### 3.3.13 Xtext

Xtext is a set of features<sup>7</sup> for the Eclipse Platform that supports the creation of a Domain Specific Language (DSL). Programmers start by defining a grammar that describes the syntax of the language in EBNF notation. When this is done Xtext will derive an infrastructure including a parser and an editor for the DSL. When employed the Eclipse Workbench will include the editor that recognizes the DSL defined in the EBNF file providing

<sup>7</sup>

A feature is a set of plug-ins.

it with syntax highlighting. The content assist will also include an outline view showing the code of the DSL in a structured tree with DSL rules as nodes.

In this project it is not enough to only use files provided by Xtext plug-in. Firstly, all of the Xtext projects must be referred to by the Launcher plug-in (described in section 3.3.6). Secondly, since the Xtext Plug-in does not provide anything to make a functional language, an execution mechanism has to be implemented manually. The execution mechanism can be implemented in form of compiler, interpreter or code generator. For this project it was decided that the most appropriate choice for the Experior DSTL would be an interpreter (which is described in section 3.3.8). Other files that have manually been created within Xtext, such as a file to display tooltip, had no major function. Those files have been derived with inheritance from existing files either within the Eclipse Platform or the Xtext and had been slightly modified to adapt better to the Experior DSTL and later bound to be called instead of the default files. In conclusion to develop a functional DSTL the developer has to both add additional packages and files to Xtext project to make the language functional and modify the Xtext-generated files to adapt Xtext behavior to language's functionality.

### 3.3.14 Development Process

This sub-section describes the section 3.2 in detail regarding development of the Experior DSTL. As presented in section 3.2 along with the figure 3.2, the process of development is passing an EBNF (.xtext) file to MWE2 and retrieves the results in form of a user interface (language infrastructure + editor).

In reality since the Experior DSTL employs an extra (Plug-in development) project the development process is slightly more complicated. Since the Launcher plug-in relies on the files generated by the MWE2 it has to make references to the packages in the Xtext projects which have to be integrable with the Launcher. To satisfy those requirements, additional (not only default ones) packages have to be exported from the Xtext project for the Launcher to be able to use the generated files. The same applies to the Interpreter which refers to the generated files. All of these references are made indirectly. A developer essentially has no control over the files generated by the MWE2, therefore the UI module has to be bound to the files that the Interpreter and the Launcher are referring to.

The user interface described in the section 3.2 depends on all three parts: the Interpreter, the Launcher and the Xtext project. However since no work environment is



defined for the CLI it is not dependent on the Launcher part, only on the Xtext and the Interpreter.

To make the CLI available, the Interpreter is exported as a stand-alone runnable .jar file. The jar archive has to include all of the imported packages (dependencies) that are not included in the standard JRE library. The components of the Xtext project are exported by placing binaries of the Xtext project into the jar archive along with the interpreter.

In case of the GUI all three parts are necessary. When exporting the projects as a stand-alone plug-in, all of the projects need to be exported as separate .jar files with references to each other along with contents and artifacts. When and only when all three are installed in the Eclipse Platform will the Experior DSTL plug-in have the intended functionality.

### **3.4 Xtext Hello World Example**

This section will present a basic example of an Xtext project, the Hello World language. The Hello World language is generated automatically by the Xtext at the creation of a new Xtext project. The content of this Hello World is a brief introduction of the syntax and structure of DSL programming in Xtext.

#### **3.4.1 Xtext Generated Components**

Xtext projects are divided into three separate sub-projects meant for programming language infrastructure, user interface and tests implementation.

The project containing the files that make the programming language infrastructure contains two significant files, `Generate<Language Name>.mwe2` and `<Language Name>.xtext`.

#### **3.4.2 EBNF Example**

`<Language Name>.xtext` is an EBNF file meaning it contains the definition of the DSL rules.

```

MyDsl.xtext
grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals

generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

Model:
  greetings+=Greeting*;

Greeting:
  'Hello' name=ID '!';

```

Figure 3.9: EBNF Hello World

Initially the EBNF implementation is presented in Figure 3.9. The first line: `grammar org.xtext.example.mydsl.MyDsl with org.eclipse.xtext.common.Terminals` Defines initial grammar rules for terminal rules. The keyword “with” works as a kind of import saying that these terminal rules exist in the package `org.eclipse.xtext.common.Terminals`. If those rules are extracted (which in case of Experior DSTL they were, to apply modifications) they would look as in Figure 3.9.

```

// Terminals rules
terminal ML_COMMENT: '/*' -> '*/*';
terminal SL_COMMENT: '//' !('\n'|\r)* ('\r'? '\n')?;
terminal ID : '^?('a'..'z'|'A'..'Z'|'_'|'0'..'9')*';
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING :
  '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'"'|'\\') | !('\n'|\r) )* '"' |
  "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'"'|'\\') | !('\n'|\r) )* "'";
terminal WS : (' |\t|\r|\n')+;
terminal ANY_OTHER: .;

```

Figure 3.10: Terminal Rules

In this case the keyword “with” along with the package is replaced with the keyword “hidden” followed by terminal rules that will remain hidden from the parser, which in this case would be ML\_COMMENT (Multi-Line Comment), SL\_COMMENT(Single-Line Comment) and WS(White Space). To define `ecore::EInt` a package <http://www.eclipse.org/emf/2002/Ecore> needs to be imported as `ecore`.

Except for the hidden terminal rules described above, ID, INT and STRING are presented. ID is a rule for naming a variable and with the regular expressions it is declared to start with a letter or an underscore followed by any number of letters, numbers and underscores. An INT is declared to be a positive integer and a STRING includes all of the characters placed between citation marks as a common string rule within any other language.

The definition of the Hello World language that follows is declared to be a Model that contains 0 or more Greetings. A Greeting is a non-terminal symbol. It is required

to start with the keyword “Hello” followed by any variable name and concluded with the exclamation mark.

### 3.4.3 MWE2 Example

The file `Generate<Language Name>.mwe2` is a MWE2 (See 3.2.2) that generates DSL infrastructure from the EBNF file presented in Figure 3.8, most of which are java files containing functionality and additional files with various formats (.xtend, .ecore etc.) that marshal the .java files. The modification of the .mwe2 file is for advanced DSL development. The modification of the .mwe2 file is to change the behavior of the generated content. The .mwe2 file can be found in appendix A: A.1.

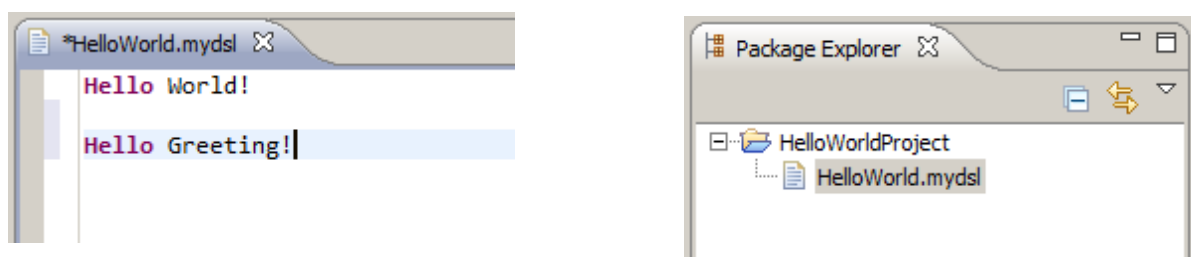
To understand the Hello World example the programmer does not have to take tests and ui projects into account.

## 3.5 Xtext Hello World Language Demonstration

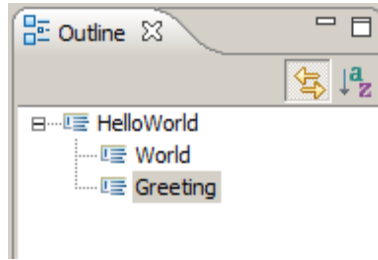
After the MWE2 has generated the language the quickest way to test the language is to open a new instance of Eclipse that will inherit the Xtext nature of the project.

After the Eclipse environment has started, a general project and a DSL file can be created and recognized by the Eclipse Platform by its file extension in this case (.mydsl).

By using the recognized file extension the project and file will inherit the Xtext nature. After this the empty file can be filled with code according to the grammar rules defined in the EBNF file.



As declared in the EBNF file each “Hello <variable> !” is a separate Greeting. Each of these Greetings is shown in an Outline View as a navigation help.



Since the structure of the Hello World language is missing any execution mechanism there is no way of using it in a productive way i.e the file containing Greetings cannot be executed in any way. The Hello World example is used for demonstrating the EBNF syntax, how to generate an infrastructure and how to run an environment that recognizes the DSL.

### 3.6 Experior DSTL Example

This section will present how the Experior DSTL is built, its infrastructure, execution mechanisms and additional extensions. It will also explain manually implemented files, modifications made to generated files.

#### 3.6.1 Experior EBNF

The project starts by modifying the default EBNF to suit the purpose of Experior. First the terminal symbols are being extracted as shown in the picture in the section 3.4.2 with slight modification such as negative integers being accepted.

<pre> Model:   (elements += Expression)* ; </pre>	<pre> Expression:   'TestCase' name = ID code = TestCase ; </pre>
---------------------------------------------------	-------------------------------------------------------------------

In contrast to the Hello World model the Experior model consists of zero or more expressions. The rule for Expression is as shown in figure: Expression it has to start with the keyword “TestCase” followed by a variable that will serve as a name for the test case and followed by a non-terminal symbols “TestCase”.

```

TestCase:
  XMLCommand += (XMLCommand)* GetCommand = GetCommand AssertCommand += (AssertCommand)*
;

```

Figure 3.11: TestCase

As presented in the Figure 3.11: TestCase a test case consists of zero or more XMLCommands, one GetCommand, and zero or more AssertCommands.

An XMLCommand is a command used to import variables containing URL paths needed for the GetCommand. An XMLCommand starts with a keyword “xml.load” followed by a string with the path to an xml file. The structure of the xml file that can be loaded needs to follow the following structure:

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
    <variable value="http://www.tieto.com" name="Tieto" />
    <variable value="http://www.google.se" name="Google" />
    ...
</data>
```

Since the EBNF is for demonstrative purpose, there is not much use for the XMLCommand since there is only one GetCommand allowed in the test case. But it was intended that in cases where the URLs are used repeatedly a tester only needs to write variable names such as Tieto or Google instead of their URLs.

The GetCommand has the purpose of sending an HTTPGET request to a URL and store the result. It starts with the keyword “GET” followed by an argument which can be either a variable or a string.

The AssertCommand is used to produce a verdict where it start with the keyword “ASSERT” followed by a keyword “http.response” or “http.size” depending on which part is in question of the test(since multiple AssertCommands are allowed one may follow the other), followed by a value that tester expects to retrieve as the result of a GetCommand.

The assignment shown in the figures above (accumulative in Model and simple in Expression) has the purpose of storing the information in internal variables of language infrastructure with the purpose of being able to retrieve them during the execution state.

A complete EBNF can be found in Appendix A: A.2

### 3.6.2 Experiore Interpreter

The interpreter is responsible for reading semantics of the Experiore DSTL script. Its responsibility is go through the script line by line interpreting the semantics and taking actions in order to execute the commands.

The Interpreter is written in Java and is implemented manually. It is located within an additional package inside the project containing language infrastructure. The interpreter class is required to have at least two arguments. One is the path to the rule folder where the results are stored, the following arguments are one or more script files(.exp). At the start of execution the interpreter configures all the paths and retrieves global variables. After that the interpreter loops through all of the argument files executing them by first checking with the parser if the file content follows the syntax of Experior DSTL and is executable. If the parser validates the file content interpreter loops through each semantic element of the file identifying the rule and executing commands accordingly.

Each semantic element that the interpreter retrieves is an Ecore Object (EObject) implemented in files generated by the MWE2. Referring back to part 3.6.1: “Experior EBNF” it is describing assignment of elements to variables. Since the interpreter requires these elements it retrieves them by calling internal get commands such as `getCommand().getName()`; in an EObject that is an instance of an element contained by the Model (See figure 3.12).

All of the output produced by the interpreter is stored within a logfile (.log), an html file (.html) and console output. The log file and html file are produced with help of log4j logger configured in `log4j.properties` and an HTMLGenerator.

```
content app = (content) bElement.getGetCommand().getName();
if (app instanceof link)
    executeGET (((link) app).getName());
else if (app instanceof variable)
{
```

*Figure 3.12*

### 3.6.3 Output Generators

The output generators of Experior DSTL are the log4j logger installed as a external component provided by Apache Logging Services™ and the HTMLGenerator implemented manually.

The log4j logger is configured by a `log4j.properties` file (See figure 3.8). The content of this file will decide the level of logging and layout. In the `log4j.properties` shown in the figure 3.7 the first two lines decide the threshold meaning what kind of information that should filtered and set what loggers handles what information. The following paragraph handles the details such as appender package, destination and layout.

The information gathered from executing Experior DSTL files is stored in “userlog.log” file which stores information cumulatively and can be viewed in a text editor.

The HTMLGenerator creates an HTML file in the same folder as userlog.log file. The information presented in the HTML file is gathered from the userlog.log file and filtered to produce a more reader friendly markup including HTML and JavaScript characteristics.

### 3.6.4 Launcher

The Launcher project is an Eclipse Platform’s Plug-in Development project manually implemented to extend functionality of Eclipse Workbench to suit the needs of Experior DSTL. It has two extensions a Run Configuration extension (i.e Experior Launcher) and documentation view called ExperiorDoc.

To be able to start and run an Experior project a specific “Experior Launcher” was created. The Experior Launcher is similar to the Eclipse launcher by containing tabs for different launch options. To access the Experior launcher one must first access the Run Configurations option in eclipse, from here Experior should be visible as an option. This is all implemented in the LauncherConfiguration class.

The LauncherConfiguration class has a container with fields intended for browsing project, script and root folder. It has listeners that activate “Run” button only when all the fields are filled with fitting information. When the run button is clicked the launcher starts and configures a new JVM running the interpreter, sending the folder and the script as argument.

The second extension ExperiorDoc is a documentation view extension to the Eclipse Workbench. It works by adding a post-selection listener to the selection provider within the shell. It is triggered every time the marker has changed locations at which point the documentation view is being updated by retrieving the semantic object at the markers position and sending it to the default documentation provider receiving a string with comment in return. The MultiLineCommentDocumentationProvider.java provided by Xtext Plugin has been selected to be the default documentation provider. It is done by binding it to the IEObjectDocumentationProvider interface within the ExperiorUiModule inside UI project of Xtext as shown in figure 3.13. The MultilineCommentDocumentationProvider extracts the comment belonging to the EObject written between `/**` and `*/` tags if such a comment exists.

Inside the plugin.xml file in launcher these two extensions are give IDs and referred to extension points they will be “plugged” in to, including types, type images, tab groups and views.

```

public class ExperiorUiModule extends com.tieto.experior.ui.AbstractExperiorUiModule {
    public ExperiorUiModule(AbstractUIPlugin plugin) {
        super(plugin);
    }

    public Class<? extends IEObjectHoverProvider> bindIEObjectHoverProvider() {
        return ExperiorEObjectHoverProvider.class;
    }

    public Class<? extends IEObjectDocumentationProvider> bindIEObjectDocumentationProvider() {
        return MultiLineCommentDocumentationProvider.class;
    }
}

```

Figure 3.13: ExperiorUiModule

### 3.6.5 Smaller file-modifications

To adjust smaller details in this case the Outline View and hover tooltip to make them work as intended additional smaller modifications had to be made.

The hover tooltip is triggered when the cursor hovers over the declaration of a test case. To modify its functionality a new java class had to be implemented, inheriting a default hover provider and overriding the header output layout as shown in the figure 3.14 and binding it to IEObjectHoverProvider interfaces in ExperiorUiModule as shown in figure 3.13

```

package com.tieto.experior.ui;

import org.eclipse.emf.ecore.EObject;

public class ExperiorEObjectHoverProvider extends DefaultEObjectHoverProvider {
    @Override
    protected String getFirstLine(EObject o) {
        if (o instanceof Expression) {
            return "TestCase" + " <b>" + getLabel(o) + "</b>";
        }
        return super.getFirstLine(o);
    }
}

```

Figure 3.14: ExperiorEObjectHoverProvider

The code presented above returns a header containing “TestCase” and a test case name written in declaration of a test case in bold font.

There was an issue with the default implementation of the Outline View’s OutlineTreeProvider, where it was not compatible with the language defined in the EBNF (Experior DSTL). Some of the semantic objects within the implementation were unnamed



according to Xtext standards. To handle this issue, names for the “unnamed” objects were dynamically preset.

Other than the handled issues described in this sub-section the UI project of the Xtext remained unchanged for time being.

### 3.6.6 MANIFEST.MF

During the development of the Experior DSTL the MANIFEST.MF files of all the projects have been both directly and indirectly modified. The indirect modifications of the MANIFEST.MF file occur when a file which the MANIFEST.MF refers to, has been refracted (e.g renamed) the Eclipse Platform modifies the reference automatically.

The only direct changes made to the MANIFEST.MF file were selecting the IDs, names and bundles of the projects so that they can identify themselves when being integrated.

The indirect modification that regards the imported and the exported packages to and from the projects that were not contained by standard JRE library. When a not yet imported package was imported to a class within the java code the Eclipse Platform modified the MANIFEST.MF to import that package into the project or if a project were not even exported it suggested doing it over those project that were implemented manually that users had control over.

### 3.6.7 Making Experior DSTL Deployable

The final requirement of the thesis is to make Experior DSTL a deployable plug-in. Since it is dependent on Xtext it either has to include the Xtext plug-in or be installed on Eclipse that already has the Xtext plug-in installed. For the sake of simplicity the latter one was chosen.

To make Experior DSTL a deployable it first needs to be exported as a JAR file. After that it needs to be installed on a running Eclipse Platform. This is done automatically (if default configurations are chosen) by the Eclipse export option where the JAR files are being exported into a folder containing all the plug-ins used by the Eclipse Platform.

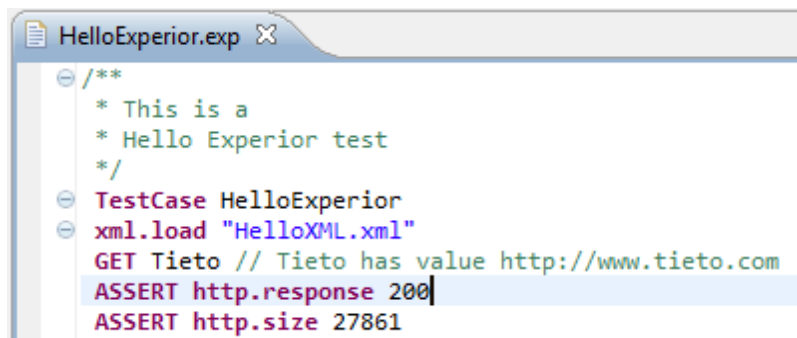
## **3.7 Experior DSTL Demonstration**

This section is presenting the result of Experior DSTL Project which is a Development Environment for Experior DSTL built in the Eclipse Platform. The section contains both creating the project and the Experior files, the demonstration of written

Experior code with examples of content assists outline view, documentation as set-up stage. After in the execution stage there will be presented how to set up run configurations and execute the code. The set-up and execution stages will be shown in both GUI and CLI. Finally this section will present the result files, taking a look back at logger configurations and comparing them with the output format of the result.

### 3.7.1 Set-up

The demonstration set-up starts almost exactly as it did in the demonstration of Hello World example. However the difference is that Experior DSTL has been made deployable and has no need to be created within another instance of Eclipse to be able to install customized Xtext plug-ins and Launcher plugin. Nonetheless the set-up requires the creation of a new project and a new file with an .exp extension. And as it did in the hello world example the program asks if Xtext nature should be added to the project.



```

HelloExperior.exp
/**
 * This is a
 * Hello Experior test
 */
TestCase HelloExperior
xml.load "HelloXML.xml"
GET Tieto // Tieto has value http://www.tieto.com
ASSERT http.response 200
ASSERT http.size 27861

```

Figure 3.18 (a): HelloExperior.exp

Next a test script is written. The chosen test demonstrates most features of the Experior DSTL language.

As seen in Figure 3.18 the Experior DSTL Environment provides a standard to Eclipse syntax highlighting for keywords, strings, and comments. This is a functional test that will send an HTTPGET request to <http://www.tieto.com> and store the response in language variables for response and size that are retrieved at combination of command ASSERT and either http.response or http.size keywords. In this case a user assumes that the HTTP response will be of code 200 (OK) and that content-length will be 27861 bytes. This case also uses a variable name Tieto which is declared in the HelloXML.xml along with other variables, but since the GET may also take a string as an argument this will also be a valid script code:

```
/**
 * This is a
 * Hello Experior test
 */
TestCase HelloExperior
GET "http://www.tieto.com"
ASSERT http.response 200
ASSERT http.size 27861
```

Figure 3.18 (b): HelloExperior.exp

To take a look at the error message that will be produced in case of an invalid code written in the script the GET command will be removed. Recalling the EBNF definition, that will be invalid code because every test case needs to have at least one GET command.

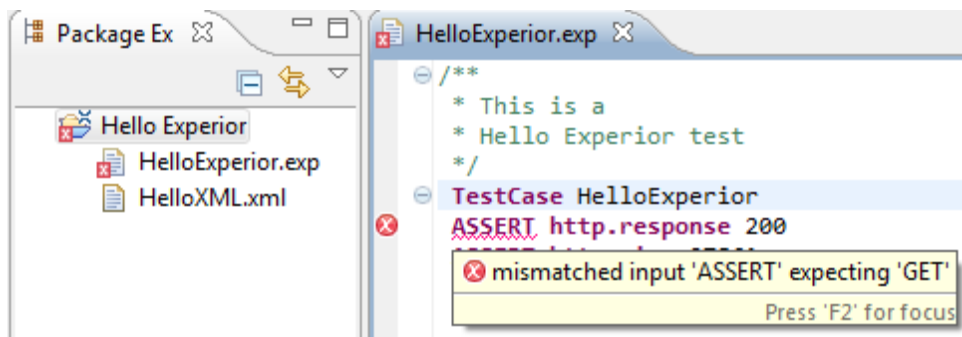


Figure 3.19: HelloExperior.exp Error

In the Figure 3.19 it is shown that if a GET command is missing a white cross with red background appears at the corner of the project and file icons and at the line of code where the error has occurred. As a help a user can hover over the underlined word or click on the red circle with a cross to get a tool-tip text that explains why parser recognizes it as an erroneous code and suggest an alternative. In this case it is dissatisfied with the mismatched input where GET command is supposed to be. The same kind of error will be printed in the output files in case a tester runs the script despite seeing the error message or in case the user is using CLI and has written the script in a simple text editor without having instant feedback on the written code.

It has also been discussed that during the course of the project the outline view has been modified in order to fit the Experior DSTL. Outline View works as a navigational tool where a file is seen in a tree form displaying rules as nodes of the tree. A tester can click on a tree node to jump to and select the piece of code that represents that node of the tree.

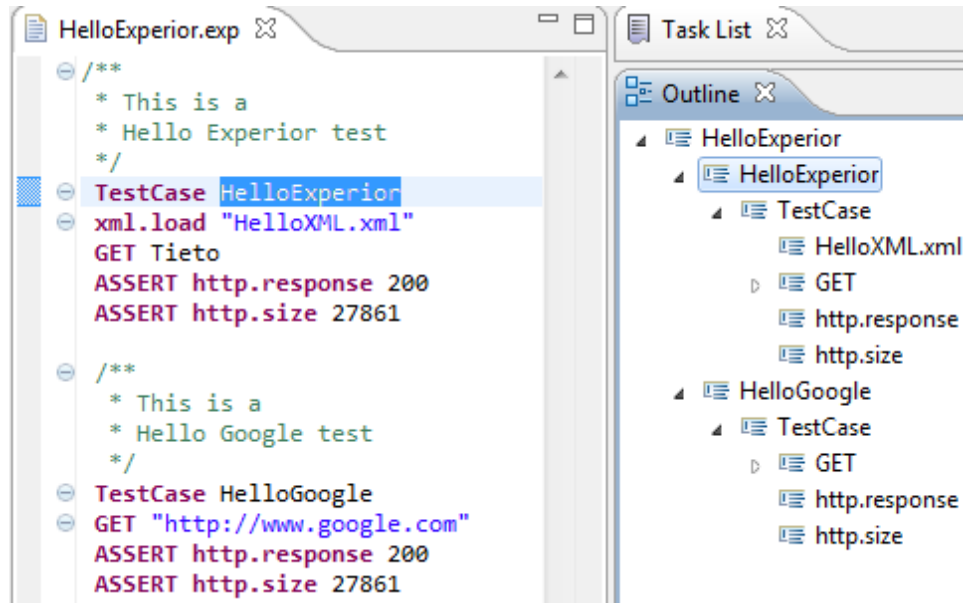


Figure 3.20 HelloExperior.exp Navigation

The last component of the set-up demonstration will be ExperiorDoc which is not very useful at this stage of the Experior DSTL development when the language is this small, but will become more useful when the language is big enough to write test suites and will get additional markup options in future releases. The comment documentation provider has been described earlier (in sections 3.6.4 and 3.6.5) to provide documentation output in two different places: in the tool-tip box while hovering the cursor over the declaration of the test case and in an additional view in Eclipse Workbench:

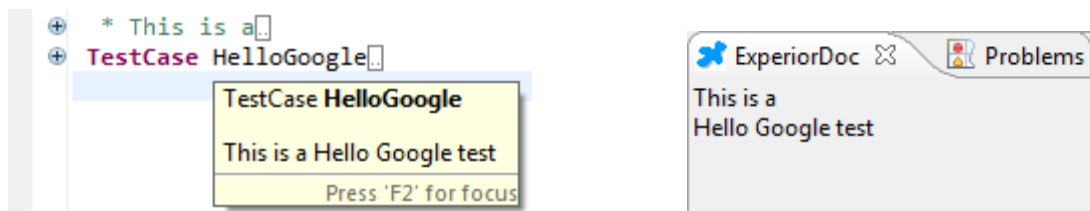


Figure 3.21 ExperiorDoc Example

## 3.7.2 Execution

### GUI:

In the execution process of the Experior DSTL in order to be able to run the script a run configuration options have to be set. In the section of the Experior Development it has been mentioned that the configuration options consisted of a Main tab and a Configuration tab (see Figure. 3.15) where in a main tab a tester has to select the project in

question and a script (.exp) file to be executed. In the Configuration tab a user has to select a root folder for output files.

As seen in Figure 3.15 the “Run” button is disabled. It depends on one or more of those fields not being filled with correct input. Each of the browse buttons starts a dialog, browse project dialog for projects, browse script dialog for scripts and browse directory dialog for directories. When all of those have been filled correctly the Run button is enabled. Since the information is stored in program variables this configuration needs to be done only once while working on the same script file. After it has been configured once, the user can always run the script with “Hello\_Experior” configuration (user can also have different configuration for same file or several configurations for several files).

When the user selects to run the configuration (i.e to execute the script) the automatic execution mechanism takes over passing output information into desired folder.

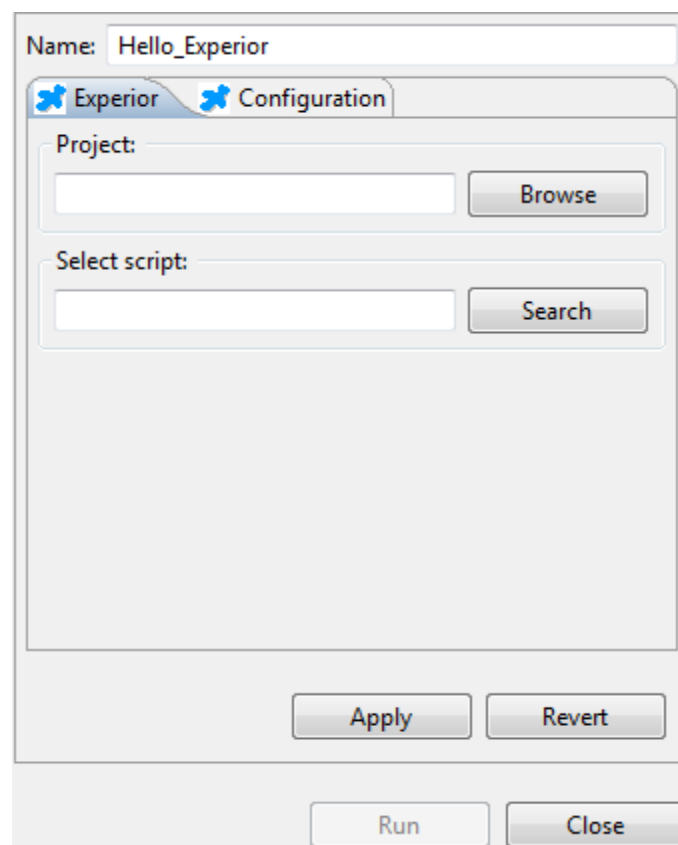


Figure 3.15: Configuration Tab

## CLI

If a user would rather work with Command Line Interface and execute files from there, all they need to do is to call the exported Interpreter through the java program

providing the path of the root directory as the first argument and one or several script files as following arguments (see Figure 3.16)

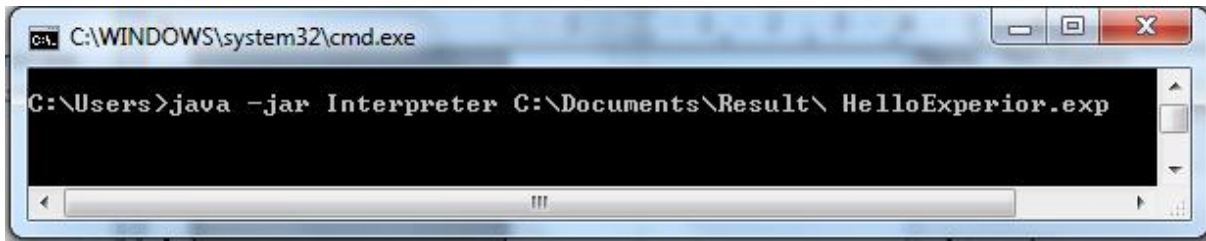


Figure 3.16: CLI

### 3.7.3 The Results

For the demonstration of the Exterior DSTL the executed script has been the one that was shown as the first example in the set-up part of this section the TestCase HelloExterior.

As a result the user gets the two files in the directory configured to be directory for output files. The userlog.log file contains the the raw information regarding the HelloExterior test execution (assuming that there was no prior execution that stored information within the same folder):

```
TestCase: HelloExterior
Interpreter 0.1
Current user is: xxyusupr
GET command name: http://www.tieto.com
Response code = 200 Size = 28117
ASSERT command http.response Expected: 200
Assertion Passed!
http.response = 200
ASSERT command http.size Expected: 27861
Assertion Failed!
http.size=28117
Starttime: 2012/05/08 11:04:00 Endtime: 2012/05/08 11:04:01
```

The same information has been output to the console to skip the confusion of tricking the user into thinking that nothing has happened.

The other file inside the output folder is the htmllog.html containing an expandable table with marked up information (see figure 3.17)

The reason that the table is expandable and has green header is because when the tests have accumulated into dozens of tests it is simpler to scroll through the results picking out only those that are highlighted red (failed) instead of green and expanding them to see additional information to why the test has been failed.

<b>TestCase: HelloExperior -Passed</b>
Interpreter 0.1
Current user is: xxyusupr
GET command name: http://www.tieto.com
Response code = 200 Size = 28117
ASSERT command http.response Expected: 200
Assertion Passed!
http.response = 200
ASSERT command http.size Expected: 28117
Assertion Passed!
http.size = 28117
Starttime: 2012/05/08 11:16:41 Endtime: 2012/05/08 11:16:43

Figure 3.17: Result in HTML

### 3.8 Summary

This chapter has presented detailed information about the experiment Experior. It has described the tester's perspective on the complete project where the structure only contains of Set-up, Execution and Result. It has also described how the testers proceed to execute a test using one of two interfaces, the GUI and the CLI.

The developer's perspective has shed some light on the underlying work of developing a DSL. The structure of Set-up, Execution and Result stages was employed in developer's perspective as well with writing the EBNF as a Set-up, passing it over to the MWE2 at Execution stage and receiving the environment described in the Tester View as result.

In the section 3.3 Experior DSTL Project Details all of the implementation and structure was described in detail showing a complete picture of the project development. All of the tools used within the project such as Eclipse, Xtext, log4j, Interpreter etc were abstractly described to show the context of the different sub-projects.

Later examples were presented containing the generating of Hello World language along with its usage. How the Hello World generated contents were modified to create Experior DSTL including modification of EBNF, UI project, MANIFEST files and implementation of additional interpreter package to the project containing language infrastructure and implementation of a plug-in project extending run configurations and views. After that the result of the development of Experior DSTL was demonstrated in utmost detail.





## Chapter 4: Results and Evaluation

This chapter contains the evaluation of the thesis project. The results described and evaluated are the Experior DSTL project as a whole along with major contributions implemented by the developers.

The major contributions to the project which have been presented in chapter 3 will be evaluated in this chapter. These are: Experior DSTL (the language), Experior execution mechanism, an extended Eclipse environment compatible with Experior DSTL and the integration of all the components with each other.

This chapter will include a comparison of Experior DSTL with the existing similar system.

Finally this chapter will contain the developers' thoughts on the project summarizing the experience and acquired knowledge.

### ***4.1 Experior DSTL Project***

Initially the goal of the Experior DSTL project was to create a functional language and environment to be used for the Radio I&V system. Since the test scripting language is somewhat complicated, the Experior DSTL is planned to replace the current test scripting language.

During the course of development of the Experior DSTL, the production shifted focus to the development environment instead of the language itself.

At the beginning of the project some time was spent researching and understanding the current system. The current testing system is intended to test radio up-links and down-links under various circumstances.

At the beginning of the Experior DSTL Project the so called Sprint 0 was used to allow the developers to get to know the working environment and learn to use the tools required for the project. As one of the goals of the Sprint 0 was to design a language with some basic functions such as sending an HTTP GET request and handling the response. Even though Sprint 0 included meetings with Tieto's staff to learn about the systems and to try to reach a compromise about how the Experior DSTL should look like and work, the development turned more towards developing the development environment by adding some additional small functions to the language during each sprint.

It was decided that the whole project will be implemented within Eclipse Platform, using Java to program its functionality. The developers did not have a choice in the matter since the thesis project description had a requirement for developing DSTL using the plug-in Xtext which is compatible with Eclipse. The reason behind these tools is because of the requirements made by Tieto, the Tieto developers spend much time working within the Eclipse Platform and as a company Tieto is one of the contributors in the Eclipse Foundation [27]. Given the simplicity in generating the language infrastructure, the use of the Eclipse Platform and Xtext has proven to be a smart choice.

## **4.2 The Language**

The syntax and semantic choices of the Experior DSTL were almost solely based on the sprint requirements made by specifiers of the projects with suggestions and feedback from the developers.

Experior DSTL is a domain-specific test language that is similar to keyword driven test development scripts. This is the explanation for the fact that each command contains a keyword and one or two arguments. The keywords should be self-explanatory or consists common tester expressions.

The Experior DSTL's keyword and language development is the part that suffered from the shifting of the project goals from the development of the language to that of the test environment system. The focus on the Development Environment prevented further research and development of a DSTL that could be applied to the current testing system. Instead Experior DSTL has been implemented in small parts adding functionality during each sprint.

The difficulty of designing a DSTL was the need to take to account both the language design, i.e how the language would look like to the testers, and from that be able to structure the EBNF of the language that would be possible to interpret with an execution mechanism. The learning curve for the design of the language was significant, since even though Experior DSTL is a usable language there are better ways to achieve same language functionality, but in a more stable way.

The current Experior DSTL version cannot be employed by the Radio I&V without further expansion of functionality of both the language and the execution mechanism to interpret it. However the contribution of this project has been to extend the development environment.

### **4.3 Execution Mechanism**

The execution mechanism was one of the larger tasks. It had a moderate learning curve but it took some time to understand the nuances of integrating additional files to the Xtext tools.

In the Experior DSTL the choice was made that the execution mechanism for the Experior DSTL would be in form of an interpreter. The interpreter was not the only option for the execution mechanisms and perhaps in the future if the work on the project continues there will be several alternatives to the execution mechanisms. The alternative solutions to the execution mechanisms that were discussed were: compiler and code generator. The interpreter alternative has been chosen to make it as simple as possible to faster produce a usable version of the project since the construction of a compiler or code generator would require more time and effort. That was the advantage in implementing the interpreter. The second implementation choice for the execution mechanism is the code generator that could be able to based on the Experior DSTL script generate a general-purpose programming language code. The advantages of a code generator would be that Experior DSTL could be translated to different languages and become machine independent, but this independence is also offered by the interpreter since the binaries are the same for all the JVM and nearly all machines have the JVM installed.

Once it became clear how the interpreter could be integrated with the Xtext tools, the task became to make it as stable as possible. Testers have been consulted on how they would like to have their output information what should happen to the execution once an error is encountered and if an error could lead to the follow-up errors (if the test execution should stop completely or just skip some part of the test), and how those errors should be presented in the output files. The rest was java programming, identifying the command written in Experior DSTL, executing it and moving on to the next command.

Another important part of the execution mechanism is the result output. It was developers' choice to use the log4j plug-in to handle the logging of output information. The first alternative to log the information was to write the information to the console with the use of system prints and to use the Java IO packages to create and write to the files. For the sake of convenience it was a logical choice to employ a logging service to handle the information output. The first logging service that was encountered was the Apaches log4j, which is also the one used in this project since it provided both convenience and structured output layout. In

retrospective the logging service that could and perhaps even should have been used was Logback [19] which is intended to be a successor to the log4j.

#### ***4.4 Extended Eclipse Environment***

Extending the Eclipse Development Environment by the development of the Experior DSTL Plug-in has been most time consuming part. With no prior experience in Eclipse Plug-in development this part presented a challenge in form of programming certain components instead of learning curve. The idea of extension points and extensions was not difficult. The extensions that were made reused existing methods by inheriting classes and adding or overriding methods to achieve the desired result.

The first developed extension of the Plug-in was a launcher. The launcher was a necessity because no application can run in Eclipse without a specific launcher initiating a JVM Environment and setting up the application specific configurations. When the Plug-in is deployed and installed in Eclipse Platform, the Experior DSTL launcher is found in “Run Configurations” option where the user can choose to run the code as Experior DSTL application. Compared to the “Run Configurations” of the Java Application the Experior DSTL does not collect nearly as much information. This is due to the fact that the information collecting functions of the configuration window were implemented progressively as the information was needed for the execution mechanism.

The extension ExperiorDoc was aimed to replicate the functionality and behavior of JavaDoc in Eclipse Platform. Since the JavaDoc is not extended on to the Xtext projects and is not compatible with the Xtext-provided editor to be able to collect the information the ExperiorDoc extension was implemented. The ExperiorDoc extension uses the comment documentation provided by the Xtext to collect the written comment information from the editor, much like the JavaDoc does. However since the project of replicating all of the JavaDoc functions would be too big of task, for now the function of the ExperiorDoc is to extract the commentary text into a separate view and into a tooltip window. The alternative to this solution could be to replace the Xtext-provided editor with a standard Eclipse editor which is compatible with JavaDoc and add the JavaDoc package as an extension, but it would cause the stop of functionality of content assist provided by Xtext.

## 4.5 Integration

There is not much difference on how the integration between the execution mechanism and how the plug-in could be integrated. The integration between the execution mechanism and the Xtext project was simplified by adding the interpreter as one of the packages within the Xtext project that defines the Experior DSTL. That way the interpreter has access to all of the components provided by the Xtext and it is only the matter of locating the right component and extracting the right information.

The integration of the Plug-in project with the Xtext project was of a higher complexity. The issue was resolved through the imported packages and dependency injections. The dependency injection allows the plug-ins to load the components during the run-time instead of the compile-time.

## 4.6 Comparison with Prior System

During the course of the thesis project in research for DSTL production we were able to interview Hans Ström, Christoffer Johannesson, Christer Jansson, the developers at Ericsson Karlskrona who already have developed a functional domain specific test language to fit their system.

### 4.6.1 Ericsson DSTL

Ericsson Flexible Test Framework (EFTF) is an execution environment containing DSTL, created and deployed at Ericsson Karlskrona. The purpose of using DSTL is to simplify testing of the system, meaning that a tester does not have to be a programmer. Instead “test-commands” are implemented to specify what is to be done. For example:

```
start probe | on 10.10.2.2 | of_type CPU | interval 5 min  
wait | for_time 2 min
```

The syntax of the DSTL was structured with greatest concern towards testers and simplicity of implementing test cases. In the syntax of the DSTL only base terms were used to ease creating tests, no default values were set, which gave the tester full control. At the execution step the code is sent to a server where it is first checked for errors that may disrupt the execution (e.g syntax errors) and other forms of validation. After that the server sends the code to its interpreter where code is executed. If a runtime failure occurs at the execution step the test case is aborted and the interpreter jumps to the next test case. When the

interpreter is finished executing the code, the result is sent to the tester in HTML format with verdict of tests, detailed descriptions and charts.

The EFTF system was finished in approximately half a year. It does not need any maintenance; any new features are updated with backwards compatibility. The system has received positive feedback from the testers. The system is superior to the previous way of testing where the wrappers for test cases were written in PERL. Both testers and developers are happy with the new system and intend to keep working with it.

#### 4.6.2 The Comparison

From the descriptions of Experior DSTL and Ericsson DSTL one conclusion that can be drawn that both languages aim for the same goals, to produce a domain specific test language with the greatest concerns towards satisfying the testers and their needs. In contrast to Experior DSTL the EFTF is a complete and usable system that has been in use while Experior DSTL is still in development phase and is also released as an alpha (0.1) version.

The other difference between the two is that there is no specific graphical interface to the EFTF where the greatest focus of the Experior DSTL project became the development of the graphical development environment.

### **4.7 Project Evaluation**

This section will contain our evaluation of the project, our conclusions about what we have learned during the thesis project. The section will reflect the thought process during different phases of this project.

Overall the thesis project was a positive experience. We have been provided with a number of resources during the research phase of the project such as books, interviews with the staff and people with experience in the area of DSL development. We have also have gotten weekly feedback on our work from the Tieto mentors and help when needed.

The project of developing a domain specific test language was very new to us since neither of us had any prior experience with either development of a domain specific language or test development. There was a three weeks learning period (Sprint 0) to prepare us for the project. It was just enough to get to know the working environment and develop basic functions to use as a base for the project.

Some confusion arose in the middle of the project when the priorities shifted and the development environment for the Experior DSTL had more or less been made the main focus of the project. We stopped looking at the Tietos current testing system trying to find a way to improve the testing process with the Experior DSTL, instead we continued with the development of the development environment with smaller modifications to the Experior DSTL.

At the end of the project we came to the conclusion that even though we saw the project as a pleasant experience, our time management could vastly improve. Often we found ourselves stressed with the work on the project or with the report writing because we did not set aside enough time for either of them.

The Experior DSTL project was an enriching and positive experience. We have acquired additional programming skills and gained knowledge in software testing and environment development. We have also gained experience in customer contact as well as established contact with co-workers. We have also learned how to prepare meetings, handle them and document the outcome.







## **Chapter 5: Conclusion**

The purpose of this chapter is to summarize the project, the course of development and the result and to give an overview of the report. This chapter will also contain thoughts on further development of the Experior DSTL project.

### ***5.1 The Project***

The purpose of the project was to develop a domain specific test language to replace the current test system used by the testers of the Radio I&V system. The final product and original product idea differ due to shift of focus and priorities during the course of the development. During the course of the development of the Experior DSTL additional knowledge and experience has been acquired.

### ***5.2 Project Evaluation***

Since the idea of domain specific test languages is relatively new, it has been very interesting and informative to work with the development of DSTL. The development environment used was the Eclipse platform with Eclipse JDT which is familiar since the first year of studies. In addition to the Eclipse JDT the tool Xtext was used, which was originally unfamiliar to the developers. The project development combined the commonly used tools such as the Eclipse JDT with unusual tools such as Xtext. This has kept the project at reasonable level of difficulty.

The time required according to the curriculum to spend on the thesis is considered not enough for the project of this magnitude. Considering Eclipse IDE being the basis for the development environment of the Experior DSTL it took extra time to make sure that each component developed was integrated properly with the rest of the system to avoid issues or system crashes. Because of this continuous integration, some sprints have not gone according to the planning nor to the SCRUM ideology.

During the course of the development a good contact was established between the developers and product owners, peers and the target audience. This was seen as a positive experience because it gave us the opportunity to establish work contacts and train for work within the software development field. Aside from special meetings with the target audience and other developers in similar area of development, there were weekly meetings with product owners to discuss the progress of the development, get feedback and discuss ideas on

solving occurring issues. This was one of the most valuable aspects of the development progress because it meant we could always get instant feedback on newly added components.

### ***5.3 Future Development***

There are various plans for the future development of the Experior DSTL. Most of those are the original ideas that there was no time to develop. These come in two aspects, the language and the GUI.

The goal was for the Experior DSTL to be the language used to write tests for the Radio I&V system, which meant expanding the Experior DSTL and its functionality.

The further development of the GUI requires adding additional tools to the development environment. There are no specifications at the moment on which additional features are required, but it is assumed that it is desired for both launcher and the documentation tool to be expanded. The expansions will include additional options in the configuration options window of the launcher and text markup functionality of the documentation tool. In association with the expansion of the configuration options it is desired to modify the output produced by the system. The requirement for the future development to add an option to filter level of information output to the user.

## References:

- [1] <http://www.waterfall-model.com/>
- [2] Royce, Dr Winston; Managing the Development of Large Systems
- [3] <http://alistair.cockburn.us/Using+both+incremental+and+iterative+development>
- [3] <http://dl.acm.org/citation.cfm?doid=12944.12948>
- [5] B Boehm; A spiral model of software development and enhancement
- [6] [http://en.wikipedia.org/wiki/Agile\\_software\\_development](http://en.wikipedia.org/wiki/Agile_software_development)
- [7] [http://en.wikipedia.org/wiki/Software\\_development\\_process#Software\\_development\\_activities](http://en.wikipedia.org/wiki/Software_development_process#Software_development_activities)
- [8] [Hetzel88] Hetzel, William C., *The Complete Guide to Software Testing, 2nd ed.* Publication info: Wellesley, Mass. : QED Information Sciences, 1988. ISBN: 0894352423. Physical description: ix, 280 p. : ill ; 24 cm.
- [9] [http://www.ece.cmu.edu/~koopman/des\\_s99/sw\\_testing/](http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/)
- [10] Patrick Oladimeji: Levels of Testing; Advance Topics in Computer Science
- [11] <http://www.wrox.com/WileyCDA/Section/id-291252.html>
- [12] [http://en.wikipedia.org/wiki/Keyword-driven\\_testing](http://en.wikipedia.org/wiki/Keyword-driven_testing)
- [13] <http://martinfowler.com/tags/domain%20specific%20language.html>
- [14] [http://en.wikipedia.org/wiki/Tiny\\_programming\\_language](http://en.wikipedia.org/wiki/Tiny_programming_language)
- [15] Henrik Kniberg: Scrum and XP from the trenches; how we do scrum
- [16] <http://www.garshol.priv.no/download/text/bnf.html>
- [17] [http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.tmf/org.eclipse.xtext/plugins/org.eclipse.xtext.doc/help/MWE2.html?root=Modeling\\_Project&view=co](http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.tmf/org.eclipse.xtext/plugins/org.eclipse.xtext.doc/help/MWE2.html?root=Modeling_Project&view=co)
- [18] <http://en.wikipedia.org/wiki/XML>
- [19] <http://logback.qos.ch/>
- [20] <http://www.eclipse.org/jdt/>
- [21] <http://www.eclipse.org/platform/>
- [22] <http://www.eclipse.org/pde/>
- [23] <http://www.tieto.com/about-us>
- [24] <http://www.erights.org/>
- [25] reference missing
- [26] <http://www.perl.org/>
- [27] <http://www.eclipse.org/org/>
- [28] <http://www.eclipse.org/Xtext/>
- [29] <http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html>
- [30] <http://msdn.microsoft.com/en-us/library/aa292197%28v=vs.71%29.aspx>

[31] <http://msdn.microsoft.com/en-us/library/aa292128%28v=vs.71%29.aspx>

[32] [http://en.wikipedia.org/wiki/System\\_testing](http://en.wikipedia.org/wiki/System_testing)

[33] <http://www.extremeprogramming.org/rules/functionaltests.html>

## Appendix A

### A.1 GenerateExperior.mwe2

```
module com.tieto.experior.GenerateExperior

import org.eclipse.emf.mwe.utils.*
import org.eclipse.xtext.generator.*
import org.eclipse.xtext.ui.generator.*

var grammarURI = "classpath:/com/tieto/experior/Experior.xtext"
var file.extensions = "exp"
var projectName = "com.tieto.experior"
var runtimeProject = "../${projectName}"

Workflow {
    bean = StandaloneSetup {
        scanClassPath = true
        platformUri = "${runtimeProject}/.."
        // The following two lines can be removed, if Xbase is not used.
        registerGeneratedEPackage = "org.eclipse.xtext.xbase.XbasePackage"
        registerGenModelFile =
"platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel"
    }

    component = DirectoryCleaner {
        directory = "${runtimeProject}/src-gen"
    }

    component = DirectoryCleaner {
        directory = "${runtimeProject}.ui/src-gen"
    }

    component = Generator {
        pathRtProject = runtimeProject
        pathUiProject = "${runtimeProject}.ui"
        pathTestProject = "${runtimeProject}.tests"
        projectNameRt = projectName
        projectNameUi = "${projectName}.ui"
        language = {
            uri = grammarURI
            fileExtensions = file.extensions

            // Java API to access grammar elements (required by several other
            fragments)
            fragment = grammarAccess.GrammarAccessFragment {}

            // generates Java API for the generated EPackages
            fragment = ecore.EcoreGeneratorFragment {
                // referencedGenModels = "
                // platform:/resource/org.eclipse.xtext.xbase/model/Xbase.genmodel,
                //
                platform:/resource/org.eclipse.xtext.common.types/model/JavaVMTypes.genmodel
                // "
            }

            // Serializer 2.0
            fragment = serializer.SerializerFragment {
```

```

        generateStub = false
    }

    // the serialization component (1.0)
    // fragment = parseTreeConstructor.ParseTreeConstructorFragment {}

    // a custom ResourceFactory for use with EMF
    fragment = resourceFactory.ResourceFactoryFragment {
        fileExtensions = file.extensions
    }

    // The antlr parser generator fragment.
    fragment = parser.antlr.XtextAntlrGeneratorFragment {
    // options = {
    //     backtrack = true
    // }
    }

    // java-based API for validation
    fragment = validation.JavaValidatorFragment {
    //     composedCheck =
"org.eclipse.xtext.validation.ImportUriValidator"
    //     composedCheck =
"org.eclipse.xtext.validation.NamesAreUniqueValidator"
    }

    // scoping and exporting API
    // fragment = scoping.ImportURIScopingFragment {}
    // fragment = exporting.SimpleNamesFragment {}

    // scoping and exporting API
    fragment = scoping.ImportNamespacesScopingFragment {}
    fragment = exporting.QualifiedNamesFragment {}
    fragment = builder.BuilderIntegrationFragment {}

    // generator API
    fragment = generator.GeneratorFragment {
        generateMwe = false
        generateJavaMain = false
    }

    // formatter API
    fragment = formatting.FormatterFragment {}

    // labeling API
    fragment = labeling.LabelProviderFragment {}

    // outline API
    fragment = outline.OutlineTreeProviderFragment {}
    fragment = outline.QuickOutlineFragment {}

    // quickfix API
    fragment = quickfix.QuickfixProviderFragment {}

    // content assist API
    fragment = contentAssist.JavaBasedContentAssistFragment {}

    // generates a more lightweight Antlr parser and lexer tailored for
content assist
    fragment = parser.antlr.XtextAntlrUiGeneratorFragment {}

```



```

// generates junit test support classes into Generator#pathTestProject
fragment = junit.Junit4Fragment {}

// project wizard (optional)
// fragment = projectWizard.SimpleProjectWizardFragment {
//     generatorProjectName = "${projectName}"
//     modelFileExtension = file.extensions
// }

// rename refactoring
fragment = refactoring.RefactorElementNameFragment {}

// provides the necessary bindings for java types integration
fragment = types.TypesGeneratorFragment {}

// generates the required bindings only if the grammar inherits from
Xbase
fragment = xbase.XbaseGeneratorFragment {}

// provides a preference page for template proposals
fragment = templates.CodetemplatesGeneratorFragment {}

// provides a compare view
fragment = compare.CompareFragment {
    fileExtensions = file.extensions
}
}
}
}

```



```
httpChoice:
    'http.response' | 'http.size'
;
```