INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Construção de Linguagens Específicas de Domínio e a sua Integração com IDEs

LUÍS MIGUEL GODINHO PINHO OLIVEIRA MARQUES

Junho de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Domain-Specific Languages Automated Building and their IDE Integration

Luis Marques

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Alexandre Bragança

Porto, June 30, 2022

Abstract

Low-code platforms are presented as model-based software development solutions. In this sense, they could be described as applications of the Model-driven Engineering (MDE) paradigm. Despite the apparent success of these development platforms, they do not appear to adhere to standards and are frequently closed-source solutions. These characteristics may cause problems in the maintenance and evolution of solutions developed on these platforms in the future. One of these issues is the difficulty in migrating solutions to other platforms, implying that the client/user is dependent on the platform.

The goal of the study is to externalize low-code platform modeling or any DSL to more general-purpose integrated development environments (IDE) like Visual Studio Code or Eclipse. As a result, users are able to use DSLs to model their applications in the IDE and integrate them with more general-purpose programming languages.

This dissertation starts by providing an overview of the current state of the use of domain-specific language in general-purpose IDE environments.

Furthermore, several designs were developed to find the best solution that achieves the goal. The designs are then compared, and the best one is selected to be implemented.

The solution developed still has quite some future work to be done. It lacks many of the features found in a full-fledged IDE for a general-purpose language, like Visual Studio Code supports Javascript. Nonetheless, it may be quite useful when deploying a DSL to a general-purpose IDE.

Keywords: Model-driven Engineering, Domain-specific Language, Integrated development environment, Open Source

Resumo

As plataformas low-code são apresentadas como soluções de desenvolvimento de software baseadas em modelos. Nesse sentido, podem ser descritas como aplicações do paradigma Model-driven Engineering (MDE). Apesar do aparente sucesso dessas plataformas de desenvolvimento, não parecem aderir aos padrões e frequentemente são soluções de código fechado. Essas características podem causar problemas na manutenção e evolução das soluções desenvolvidas nessas plataformas no futuro. Um desses problemas é a dificuldade em migrar soluções para outras plataformas, implicando que o cliente seja dependente da plataforma.

O objetivo do estudo é externalizar a modelagem de plataforma low-code ou de uma linguagens específica de domínio (DSL) para ambientes de desenvolvimento integrado (IDE) de propósito geral, como Visual Studio Code ou Eclipse. Como resultado, os usuários poderão usar DSLs para modelar seus aplicativos no IDE e integrá-los com linguagens de programação mais gerais.

Esta dissertação começa fornecendo uma visão geral do estado atual do uso de DSL em ambientes IDE de uso geral.

Além disso, vários designs foram desenvolvidos para encontrar a melhor solução que atinja o objetivo. Os designs são então comparados e o melhor é selecionado para ser implementado.

A solução desenvolvida ainda tem bastante trabalho a ser feito. Faltam muitas das funcionalidades encontrados em um IDE para uma linguagem de uso geral, tal com o Visual Studio Code tem suporte para Javascript. No entanto, pode ser bastante útil ao implantar uma DSL num IDE de uso geral.

Palavras-chave: Engenharia Orientada a Modelos, Linguagem Específica de Domínio, Ambiente de Desenvolvimento Integrado, Código Aberto

Acknowledgement

First and foremost, I would like to thank my supervisor, Professor Alexandre Bragança, for his availability and his interest in guiding me throughout this project.

I'd want to thank my loved ones and friends for their support and encouragement throughout this time.

Contents

List of Figures

List of Tables

List of Source Code

List of Acronyms

AHP	Analytic Hierarchy Process.
CI	Consistency Index.
CLI	Command-line interface.
CR	Consistency Ratio.
DSL	Domain-specific Language.
DSRM	Design Science Research Methodology.
FFE	Fuzzy front end.
GCS	Graphical Concrete Syntaxes.
GPL	General-purpose language.
GUI	Graphical User Interface.
IDE	Integrated development environment.
JVM	Java Virtual Machine.
LCDP	Low-Code Development Platform.
LSP	Language server protocol.
MDE	Model-driven Engineering.
NCD	New Concept Development.
NPD	New product development.
QEF	Quantitative Evaluation Framework.
SWOT	Strengths, Weakness, Opportunities and Threats.
TCS	Textual Concrete Syntaxes.
UML	Unified Modeling Language.
URI	Uniform Resource Identifier.

Chapter 1

Introduction

This Chapter was written to give an overview of the document's structure, the context of this work, and to explain the foundations of this dissertation. Finally, the objectives, approach, and development process will all be specified.

1.1 Context

This section introduces Model-driven Engineering, Low code platforms, and Integrated Development Environments, to ease the understanding of this work.

1.1.1 Model-driven Engineering

Software engineers have long attempted to make software development easier by providing abstractions that allow them to program in terms of design rather than computer environments and technology (Schmidt 2006).

Despite the fact that advances in languages and platforms have raised the level of software abstractions in recent decades, software developers still spend a significant amount of time and effort developing and maintaining applications in third-generation languages, as well as manually porting application code to newer platforms or versions of the same platform.

Third-generation languages are high-level computer programming languages that are more user-friendly and machine-independent than second-generation languages but have a narrower focus than the fourth and fifth generations. Languages like Java and Python are examples of third-generation languages (*What is a Generation Languages?* 2022).

One issue with developing software with third-generation languages is that they require developers to focus on so many tactical imperative programming details that they lose sight of strategic architectural considerations like system-wide accuracy and performance.

Model-Driven Engineering technologies, which integrate Domain-specific Language (DSL), transformation engines, and generators, could be one answer to this problem. However, MDE is not without its limitations; there is a substantial initial cost associated with building or implementing tools and transformations. MDE is a long-term investment and needs customization of environment, tools and processes, and training (Mohagheghi et al. 2008).

Domain-Specific Language

A domain-specific language (DSL) is a computer language specialized to a particular domain or context, in contrast to a General-purpose language (GPL), which as the name implies is general and applicable across domains.

Some DSL examples are HTML, Gherkin, and SQL.

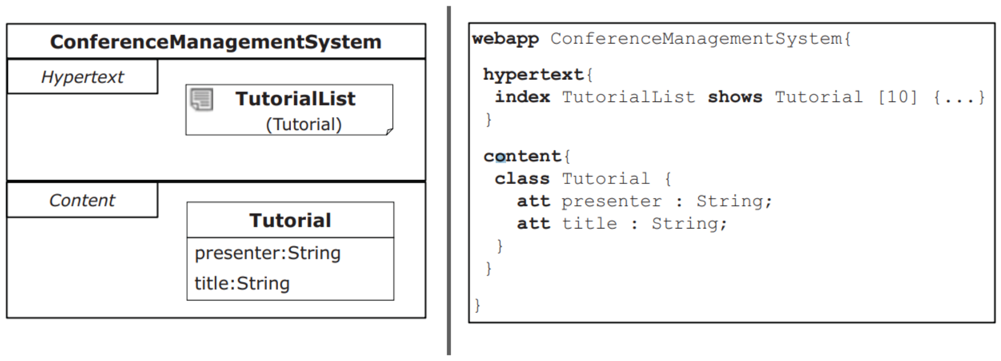
Examples of GPL are XML, Unified Modeling Language (UML), Java.

The DSLs have the two syntactical ingredients of modeling languages, namely the abstract syntax and the concrete syntax of the languages.

- **The abstract syntax** of an implementation is the set of trees used to represent programs in the implementation. This is, the abstract syntax defines the way the programs look to the evaluator/compiler (Meuter 2022).
- **The concrete syntax** of a programming language is the graphical or textual elements used to render the model elements in modeling editors (Schmidt 2006). It consists of a set of rules (productions) that define the way programs look to the programmer (Meuter 2022). We will be focussing more on the concrete syntaxes since it is more important for this dissertation.

Concrete Syntax

Two kinds of concrete syntaxes are currently supported by existing frameworks: Graphical Concrete Syntaxes (GCS) and Textual Concrete Syntaxes (TCS). Having a textual language allows the encoding of information using sequences of characters like in most programming languages, while graphical languages encode information using spatial arrangements of graphical (and textual) elements. Thus, textual representations are onedimensional, while most graphical languages allow for two-dimensional representations. For example, in UML diagrams each model element is located in a two-dimensional modeling canvas (Brambilla, Cabot, and Wimmer 2017).



```
webapp ConferenceManagementSystem{

 hypertext{
  index TutorialList shows Tutorial [10] {...}
 }

 content{
  class Tutorial {
    att presenter : String;
    att title : String;
  }
 }

}
```

Figure 1.1: Example GCS and TCS, From: Brambilla, Cabot, and Wimmer 2017

1.1.2 Low-Code development platform

A Low-Code Development Platform (LCDP) provides a development environment used to create application software through a graphical user interface. A low-coded platform may produce entirely operational applications, or require additional coding for specific situations. Low-code development platforms can reduce the amount of traditional time spent, enabling accelerated delivery of business applications. (Richardson et al. 2014)

One disadvantage of these platforms is that they each have their own visual "editor" and lack interoperability.

1.1.3 Integrated development environment

An Integrated development environment (IDE) software for building applications that combines common developer tools into a single Graphical User Interface (GUI). An IDE typically consists of:

- **Source code editor**: A text editor that can assist in writing software code with features such as syntax highlighting with visual cues, providing language-specific auto-completion, and checking for bugs as code is being written.
- **Local build automation**: Utilities that automate simple, repeatable tasks as part of creating a local build of the software for use by the developer, like compiling computer source code into binary code, packaging binary code, and running automated tests.
- **Debugger**: A program for testing other programs that can graphically display the location of a bug in the original code. (*What is an IDE?* 2022)

1.2 Problem

Low-code platforms are presented as model-based software development solutions. In this sense, they could be described as applications of the Model-driven Engineering (MDE) paradigm (Schmidt 2006). Despite the success that these development platforms seem to be enjoying, they don't seem to follow standards, and most of the time they are closed-source solutions. These characteristics can lead to future problems in the maintenance, and evolution of the solutions developed on these platforms. An example of these problems is the difficulty in migrating solutions to other platforms, that is, the client/user is dependent on the platform.

1.3 Goals

This dissertation proposal arises in the context of the BAMoL research project which aims to develop a language workbench (Fowler 2005) for domain-specific languages (DSL) for low-code platforms. Overall, the aim is to externalize the modeling of low-code platforms to more general-purpose IDE environments, such as Visual Studio Code or Eclipse. In this way, it will be possible for the user of the low-code platform to use DSLs to model their low-code applications in the IDE and integrate them with more general-purpose programming languages. The IDE solution will be integrated with the low-code platform, allowing the user to use the environment that is most convenient for them and, at the same time, can migrate the solution to other low-code platforms. This approach is partially described in (Bragança et al. 2021).

This dissertation, in particular, is intended to explore concrete syntaxes for the domain-specific languages to be made available in the IDE solution mentioned above. As a starting point, we intend to explore the different existing alternatives, as demonstrated in (*EMF.cloud*

Project 2022). If possible, a tool should be produced that fully or partially supports this activity, namely in the automatic generation of support for concrete syntaxes from domain-specific language meta-models.

BAMoL's approach is to produce the low-code platform meta-model in a standard format, Ecore. This project will use the meta-model produced for the low-code platform as a starting point for generating the DSL and IDE support. In the context of BAMoL, it was also decided to use Visual Studio Code extensions and the Language Server Protocol to support the DSL in IDEs.

Since this dissertation is part of the BAMoL research project, it comes with some technological constraints:

- The tool for designing domain-specific languages is Xtext since it supports Ecore;
- The IDE for the DSL has to support the language server protocol.

1.4 Hypothesis

The following hypothesis should be analyzed and validated in order to establish a starting point for this project:

- **H1** - The proposed approach provides an automated and effective way of producing a language server implementation and complete Visual Studio extension based solely on the abstract syntax (i.e., meta-model) of the domain-specific language.

1.5 Research methodology

This dissertation conforms to the Design Science Research Methodology (DSRM) to improve the relevance and quality of the solution to the previously mentioned problem. DSRM is a framework that defines and analyzes the constraints, objectives, processes, and results to be presented in the simplest terms with legitimate value to the audience to evaluate and perform design science research in information systems (Peffers et al. 2007).

This technique comprises six activities, applied in the context of this dissertation as follows:

- **Problem identification and motivation**: consist of describing the problem that will be the focus of the study and demonstrating how the suggested solution will be useful in this context. In this document, the problem is described in section 1.2, and the value of the solution in section 3.2;
- **Objectives of a solution**: consists of identifying the goals depending on the constraints of the problem. This is described in section 1.3;
- **Design and development**: Create and implement the chosen solution. After examining existing state-of-the-art frameworks, patterns, and approaches, the design is completed to create a proof of concept that implements this architecture. This activity is described in Chapters 4 and 5;
- **Demonstration**: includes a description of how well the solution solves the problem. Experimentation and evidence are used to determine how well the implemented solution addresses the problem. This is described in Chapter 6;

- **Evaluation**: consists of utilizing metrics to evaluate how well the implementation holds against the problem's solution. This is also described in the Chapter 6;
- **Communication**: This final activity consists of communicating how successful the proposed solution's outputs are at solving the problem, their utility in the context of the area, and how well they meet the expected value. Communication is discussed in the final section of this dissertation, Chapter 7 - Conclusions, as well as the final presentation to the evaluation audience.

  The source code of the solution is available in a public repository[1].

1.6 Expected Results

When it comes to predicted outcomes, it is expected to provide a tool that allows the user to generate concrete syntaxes using the modeling language's abstract syntax.

Furthermore, the domain-specific language generated should be able to be utilized in an open-source IDE with all of the functionality that a third-generation language IDE has, such as:

- Keywords highlighting
- Auto-completion
- Error highlighting

Finally, the application project should be available in a public repository.

1.7 Dissertation Structure

This document is divided into seven chapters. Those chapters are Introduction, State of the Art, Value Analysis, Analysis and Design, Experiments, and Conclusions.

The introduction contextualizes the project, followed by a brief description of the problem that drove this dissertation, a definition of the objectives, the preferred approach, and work planning.

State of the Art describes the technologies used with some alternatives. Following that, some work related to delivering DSLs to IDEs is presented and compared.

The following chapter depicts the project's Value Analysis. As a result, this dissertation begins with a description of the Innovation Process, followed by a characterization of its theoretical constituents and implementation. Second, the Analytic Hierarchy Process (AHP) is used to select the best IDE to deploy a DSL alternative for this study. Finally, the Value Proposition, Customer Value, and Canvas Business Model in this dissertation are designed to assist the reader in better understanding the value that will be delivered to the clients.

Functional and non-functional requirements are described in the Analysis and Design. There are also three alternatives for satisfying all requirements. Finally, there is a list of the pros and cons of each alternative, as well as the choice of the alternative.

The project structure is described in the implementation. Then, it is documented how each module was developed, including code snippets and problems encountered.

[1]`https://github.com/Lzgpom/xtext-gen-extension`

Chapter 2

State Of The Art

State of the Art will start on addressing the technologies in concern. It will describe them with more focus on the parts that have more interest for this dissertation. Afterward, there is related work. And finally, there is a summary justifying what distinguishes this solution from the others mentioned in the related work.

2.1 Technological

In this section, there will be a demonstration of the chosen technologies as well as worthwhile alternative technologies for this project. The technologies shown will be based on the constraints described in section 1.3.

2.1.1 Xtext

Xtext is an open-source software framework for developing programming languages and domain-specific languages (DSLs). Some of Xtext's features are:

- **Multi Platform** - Supports Eclipse, any editor that supports the Language server protocol (LSP) or browser editors. Table 2.1 shows the editors features by platform.
- **Continuous Integration** - Supports both Gradle and Maven to build the language and use it in downstream projects.
- **JVM languages** - Xtext can build languages for any platform. But if targeted to JVM, Xbase can be used, as a statically typed expression language. Linking against Java types, code generation, and debugging work out of the box.
- **Compatible with Graphical Editors** - The text-based formats created with Xtext can be combined with many graphical editing frameworks, such as (*GEF* 2022), (*Sirius* 2022), and (*Graphiti* 2022).
- **Single Sourcing** - Xtext's grammar definition language is not just for the parser. Many Xtext IDE features automatically adapt to your language, so whenever you change your grammar definition, the behavior of the text editor is adjusted without any further code changes. This includes advanced capabilities like cross-reference handling, code completion, navigation, syntax coloring, validation, and more.

Table 2.1: Editor Features By Platform, From: (*Xtext* 2022)

	LSP	**Eclipse**	**Web Browser**
Syntax Coloring	✓	✓	✓
Semantic Coloring		✓	✓
Error Checking	✓	✓	✓
Auto-Completion	✓	✓	✓
Formatting	✓	✓	✓
Hover Information	✓	✓	✓
Mark Occurrences	✓	✓	✓
Go To Declaration	✓	✓	
Rename Refactoring	✓	✓	
Debugging		✓	
Toggle Comments	✓	✓	
Outline / Structure View	✓	✓	
Quick Fix Proposals	✓	✓	
Find References	✓	✓	
Call Hierarchy		✓	
Type Hierarchy		✓	
Folding		✓	

To specify a language, the developer either writes a grammar in Xtext's grammar language or generates it from an Ecore model. From that definition, a code generator derives an ANTLR parser and the classes for the object model, which can be used independently of Eclipse.

2.1.2 Syntax Highlighters

Syntax highlighting is a feature of text editors that is used for programming, scripting, or markup languages, such as HTML. The feature displays text, especially source code, in different colors and fonts according to the category of terms. (d'Anjou, Fairbrother, and Kehn 2005) This feature makes it easier to write in a structured language, such as a programming language or a markup language because structures and syntax errors are visually distinct.

Syntax Highlighters are technologies that allow IDEs to have this feature. The following sections describe some of these.

ANTLR

ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees. (*ANTRL* 2022)

ANTRL is the technology used for the Xtext Generated Eclipse plugins. It allows syntax highlighting, among other features.

TextMate grammars

TextMate grammars are used to assign names to document elements such as keywords, comments, strings or similar. The purpose of this is to allow styling (syntax highlighting) and to make the text editor "smart" about which context the caret is in. (*Language Grammars* 2022).

Invented for the TextMate editor, they have been adopted by many other editors and IDEs due to large number of language bundles created and maintained by the Open Source community. (*Syntax Highlight Guide* 2022).

2.1.3 Tools supporting the LSP

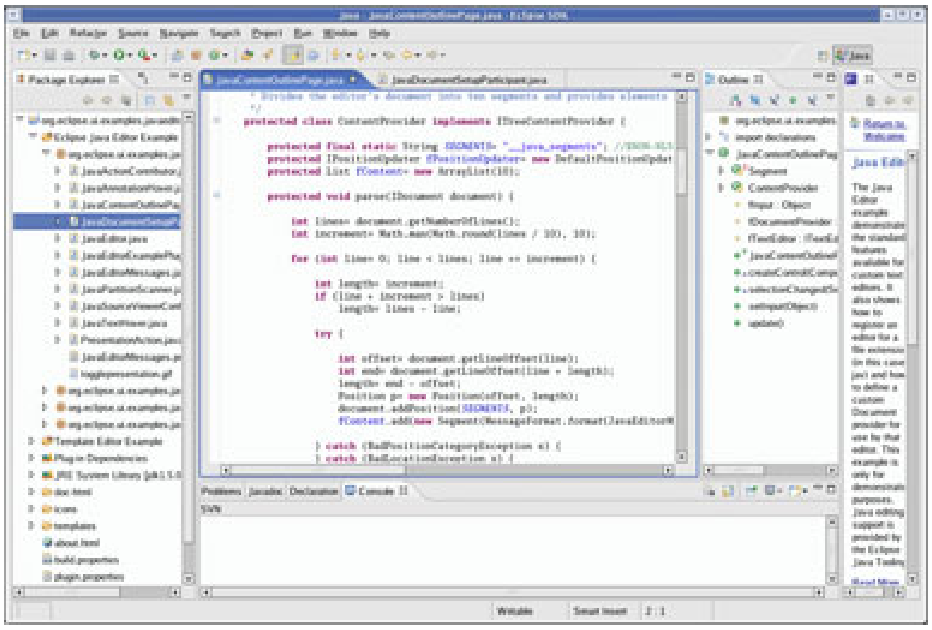
The Language server protocol (LSP) defines the protocol used between an editor or IDE and a language server that provides language features like auto-complete, go to definition, find all references, etc. (*Language Server Protocol* 2022)

A wide range of IDEs support LSP, and Xtext can generate language servers. As a result, LSP will be utilized.

An analysis is required to determine which tool should be used as an LSP client. Of the many tools that support LSP, only a few of the most well-known will be examined. The ones chosen are Visual Studio Code, Eclipse IDE, and Theia.

Eclipse IDE

Eclipse is an integrated development environment (IDE) used in computer programming. It contains a base workspace and an extensible plug-in system for customizing the environment. Eclipse is primarily built in Java and is intended for the development of Java applications. However, it may also be used to develop programs in other programming languages via plug-ins, such as C++, Javascript, and Python.



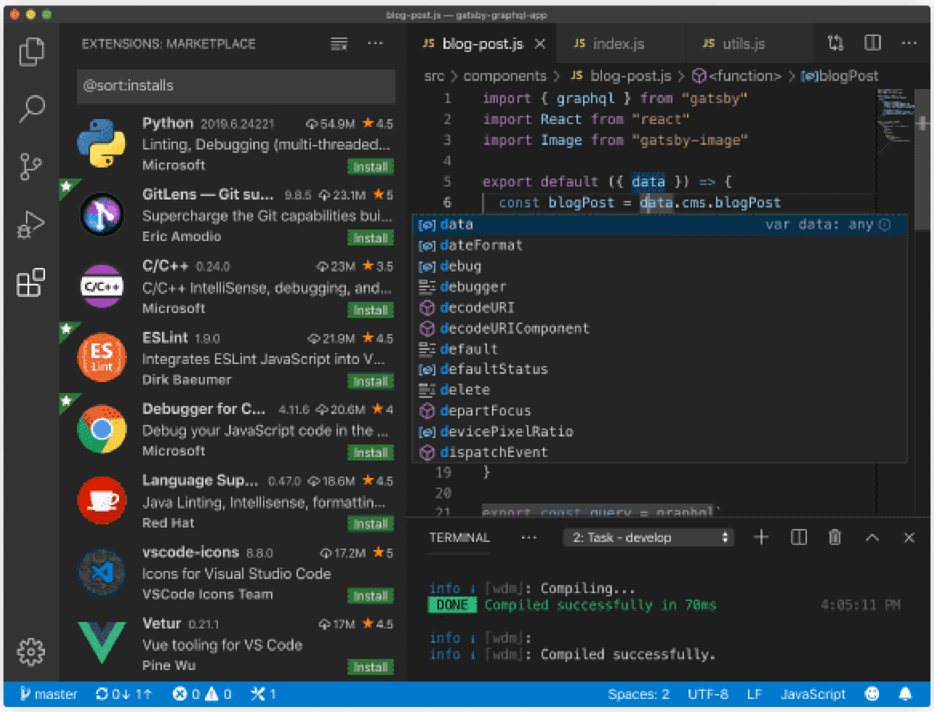
```
     * Divides the editor's document into ten segments and provides elements
     */
    protected class ContentProvider implements ITreeContentProvider {

        protected final static String SEGMENTS= "__java_segments"; //$NON-NLS
        protected IPositionUpdater fPositionUpdater= new DefaultPositionUpdat
        protected List fContent= new ArrayList(10);

        protected void parse(IDocument document) {

            int lines= document.getNumberOfLines();
            int increment= Math.max(Math.round(lines / 10), 10);

            for (int line= 0; line < lines; line += increment) {

                int length= increment;
                if (line + increment > lines)
                    length= lines - line;

                try {

                    int offset= document.getLineOffset(line);
                    int end= document.getLineOffset(line + length);
                    length= end - offset;
                    Position p= new Position(offset, length);
                    document.addPosition(SEGMENTS, p);
                    fContent.add(new Segment(MessageFormat.format(JavaEditorM

                } catch (BadPositionCategoryException x) {
                } catch (BadLocationException x) {
```

Figure 2.1: Desktop Eclipse IDE, From:(Foundation 2022)

Visual Studio Code

Microsoft's Visual Studio Code is an open-source code editor that runs on Windows, Linux, and macOS (Lardinois 2015). It includes features such as support for debugging, syntax highlighting, intelligent code completion, and code refactoring. Also, users can customize the theme, keyboard shortcuts, and preferences, as well as install extensions for extra functionality.

These extensions are available from a centralized repository. One notable feature is the ability to use the Language Server Protocol to construct extensions that add support for new languages, themes, and debuggers, do static code analysis, and add code linters (*Creating Language Servers for Visual Studio Code* 2022).



```
Python 2019.6.24221 54.9M ★4.5
Linting, Debugging (multi-threaded...
Microsoft Install
GitLens — Git su... 9.8.5 23.1M ★5
Supercharge the Git capabilities bui...
Eric Amodio Install
C/C++ 0.24.0 23M ★3.5
C/C++ IntelliSense, debugging, and...
Microsoft Install
ESLint 1.9.0 21.9M ★4.5
Integrates ESLint JavaScript into V...
Dirk Baeumer Install
Debugger for C... 4.11.6 20.6M ★4
Debug your JavaScript code in the ...
Microsoft Install
Language Sup... 0.47.0 18.6M ★4.5
Java Linting, Intellisense, formattin...
Red Hat Install
vscode-icons 8.8.0 17.2M ★5
Icons for Visual Studio Code
VSCode Icons Team Install
Vetur 0.21.1 17M ★4.5
Vue tooling for VS Code
Pine Wu Install
```

```javascript
import { graphql } from "gatsby"
import React from "react"
import Image from "gatsby-image"

export default ({ data }) => {
  const blogPost = data.cms.blogPost
```

```
data                                   var data: any
dateFormat
debug
debugger
decodeURI
decodeURIComponent
default
defaultStatus
delete
departFocus
devicePixelRatio
dispatchEvent
```

```javascript
}

export const query = graphql`
```

```
info ｉ [wdm]: Compiling...
 DONE  Compiled successfully in 70ms                 4:05:11 PM

info ｉ [wdm]:
info ｉ [wdm]: Compiled successfully.
```

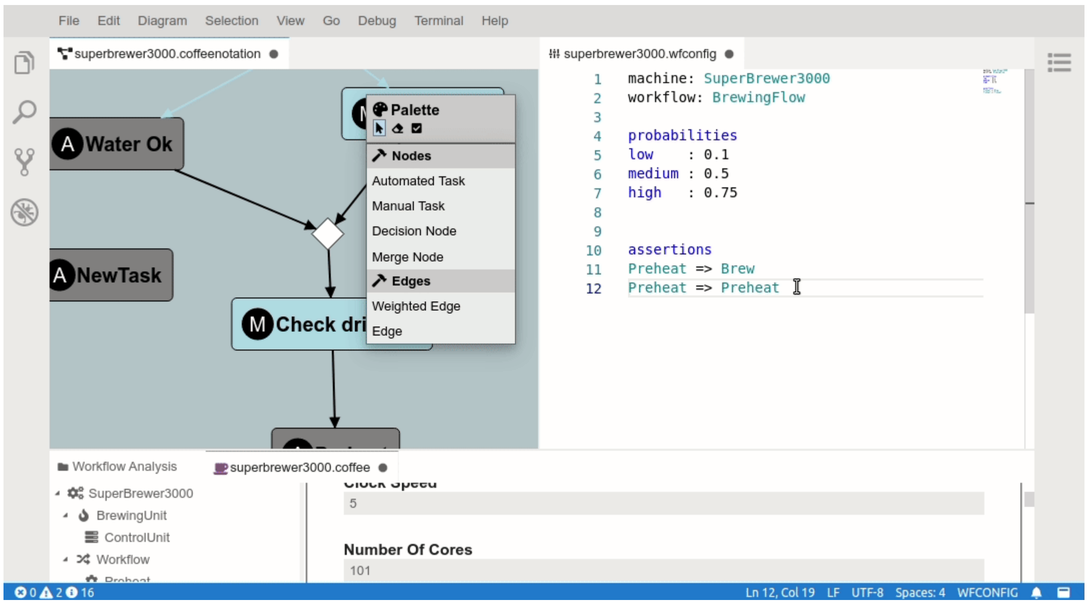
Figure 2.2: Desktop Visual Studio Code IDE, From:(*Visual Studio Code* 2022)

In the recent Stack Overflow 2021 Developer Survey, Visual Studio Code was ranked as the most popular developer environment tool (*Stack Overflow Developer Survey 2021 - Integrated Development Environment* 2022).

Theia

Eclipse Theia is also a free and open-source integrated development environment (IDE) framework based on Visual Studio Code. Theia has multiple features as stated on the official website (*Theia* 2022), in which the following stand out:

- **Cloud and Desktop**: it is possible to develop one IDE and run it in a browser or as a native desktop application from a single source.
- **Extensible**: it is designed in a modular way to allow extenders and adopters to customize and extend every aspect.
- **Modern Tech**: It provides language support via LSP and DAP, which is critical for this dissertation. Further, it can host VS Code extensions and provides full terminal access.



```
machine: SuperBrewer3000
workflow: BrewingFlow

probabilities
low    : 0.1
medium : 0.5
high   : 0.75


assertions
Preheat => Brew
Preheat => Preheat
```

Clock Speed
5

Number Of Cores
101

Figure 2.3: Coffee Editor tool based on Eclipse Theia , From:(*EMF.cloud Project* 2022)

2.2 Related Work

This section contains some related work about generating a plugin/extension for a given DSL to be used in an IDE.

2.2.1 TextX

TextX is a meta-language for domain-specific language (DSL) specification in Python, inspired by Xtext.

From a single grammar description, TextX automatically builds a meta-model (in the form of Python classes) and a parser for your language. The parser will parse expressions of your language and automatically build a graph of Python objects (i.e. the model) corresponding to the meta-model (*textX* 2022).

For this dissertation is of most importance to note that this tool, TextX, there is in an ongoing effort to build tooling support around Visual Studio Code. It is named TextX for VS Code. The input for the generator is the language grammar and some additional information about the DSL. At the time of writing, TextX for VS Code has the following features:

- Install/Uninstall textX Language Project
- Languages/Generators Preview, described in 2.2.1.
- Model/Meta-model Validation, described in 2.2.1.
- Default Syntax Highlighting, described in 2.2.1.
- Live-reload on grammar changes, describe in 2.2.1.

Languages/Generators Preview

In the Visual Studio Code, there is a view where the registered TextX languages and generators are listed.

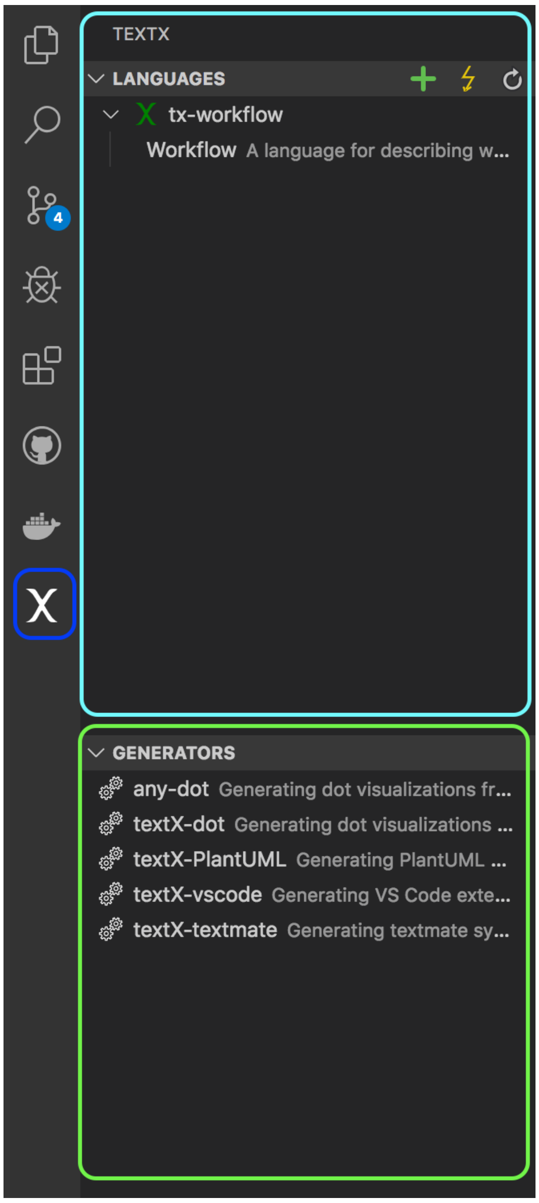
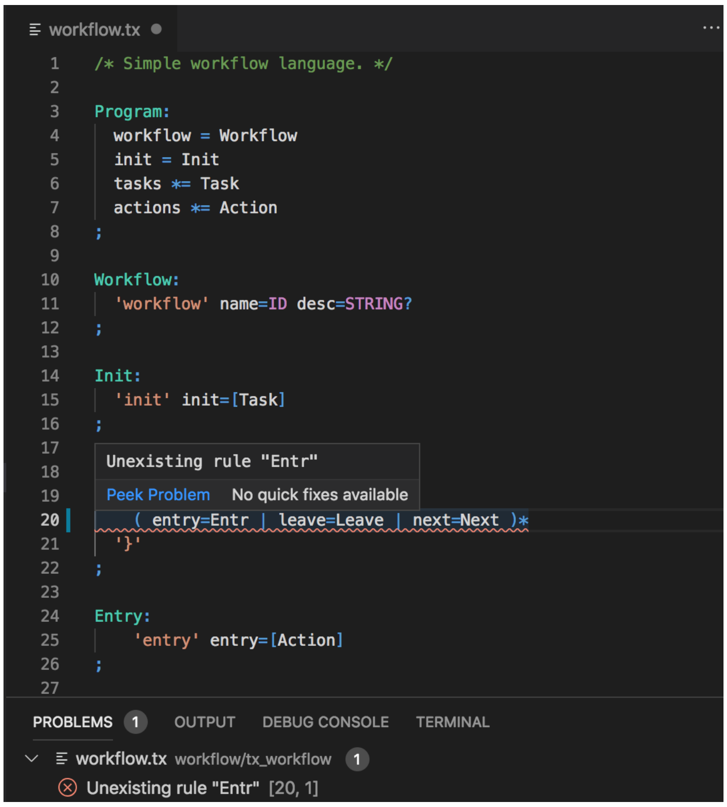

TEXTX
LANGUAGES
tx-workflow
Workflow A language for describing w...
GENERATORS
any-dot Generating dot visualizations fr...
textX-dot Generating dot visualizations ...
textX-PlantUML Generating PlantUML ...
textX-vscode Generating VS Code exte...
textX-textmate Generating textmate sy...

Figure 2.4: TextX View, From:(*textX for VS Code* 2022)

Languages/Generators Preview

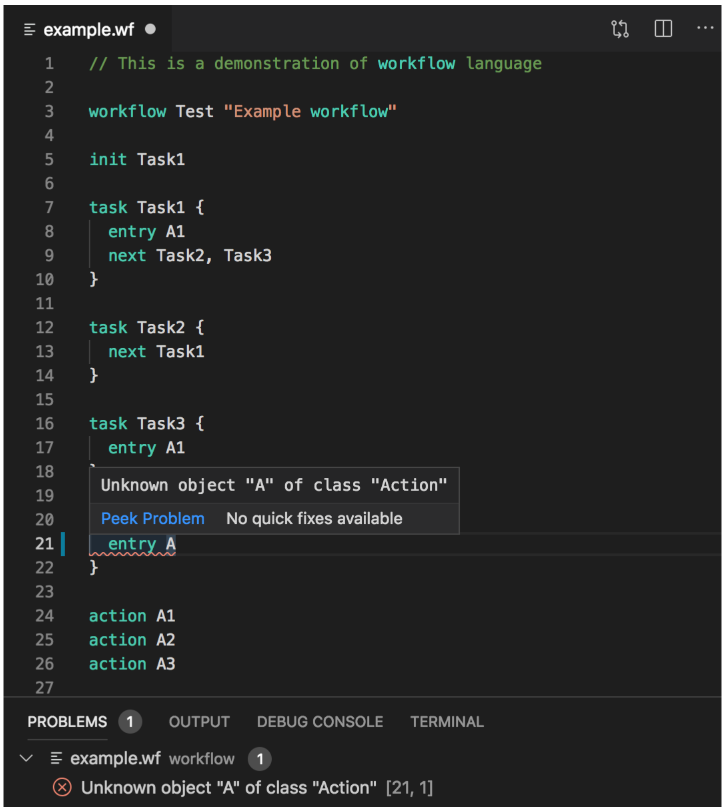
When modifying a DSL's meta-model or model in Visual Studio Code, the editor highlights the errors and displays an error message with the problem. The Figures 2.5 and 2.6 depict some examples of this.



```
/* Simple workflow language. */

Program:
  workflow = Workflow
  init = Init
  tasks *= Task
  actions *= Action
;

Workflow:
  'workflow' name=ID desc=STRING?
;

Init:
  'init' init=[Task]
;
Unexisting rule "Entr"


    ( entry=Entr | leave=Leave | next=Next )*
  '}'
;

Entry:
    'entry' entry=[Action]
;

```

Unexisting rule "Entr" [20, 1]

Figure 2.5: TextX Meta-model validation, From:(*textX for VS Code* 2022)



```
// This is a demonstration of workflow language

workflow Test "Example workflow"

init Task1

task Task1 {
  entry A1
  next Task2, Task3
}

task Task2 {
  next Task1
}

task Task3 {
  entry A1
Unknown object "A" of class "Action"


  entry A
}

action A1
action A2
action A3

```

Unknown object "A" of class "Action" [21, 1]

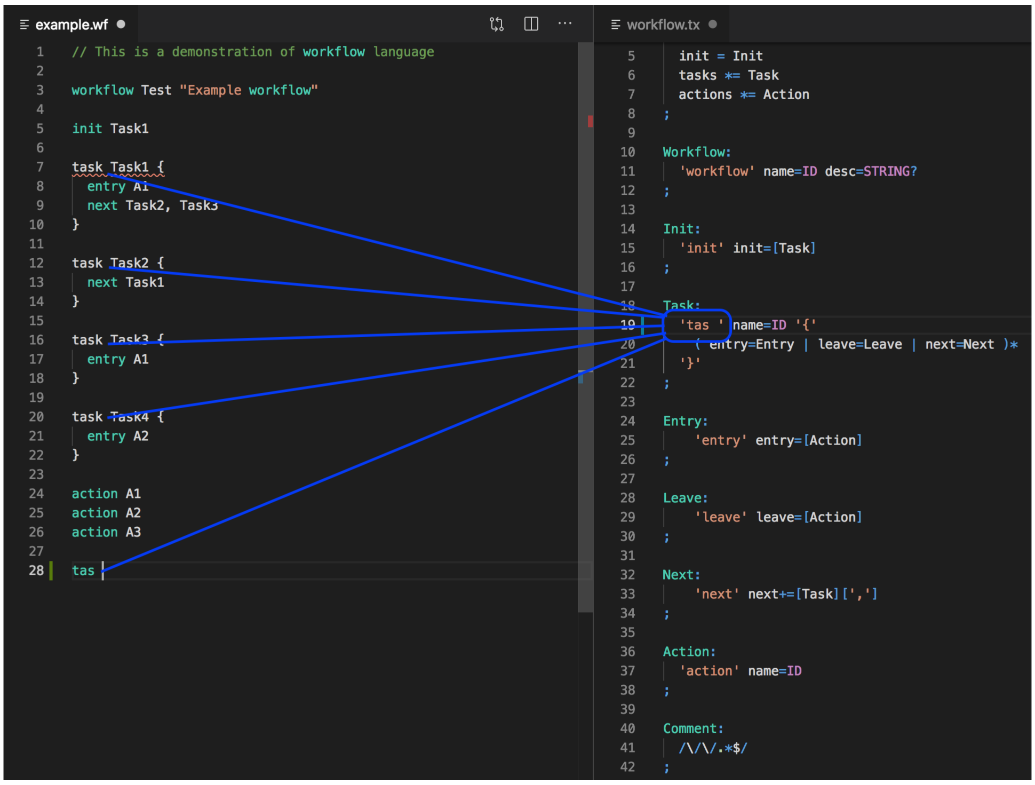
Figure 2.6: TextX Model validation, From:(*textX for VS Code* 2022)

Default Syntax Highlighting

The extension creates syntax highlight for the model files based on the grammar. For example, with the meta-model described in Figure 2.5 the highlight in Figure 2.6 is created.

Live-reload on grammar changes

The extension watches for the grammar file changes and generates a new syntax highlighting information from the changed grammar. The extension then re-paints and re-validates opened files. This can be seen in Figure 2.7.



```
// This is a demonstration of workflow language

workflow Test "Example workflow"

init Task1

task Task1 {
  entry A1
  next Task2, Task3
}

task Task2 {
  next Task1
}

task Task3 {
  entry A1
}

task Task4 {
  entry A2
}

action A1
action A2
action A3

tas 
```

```
  init = Init
  tasks *= Task
  actions *= Action
;

Workflow:
  'workflow' name=ID desc=STRING?
;

Init:
  'init' init=[Task]
;

Task:
  'tas ' name=ID '{'
    ( entry=Entry | leave=Leave | next=Next )*
  '}'
;

Entry:
    'entry' entry=[Action]
;

Leave:
    'leave' leave=[Action]
;

Next:
    'next' next+=[Task][',']
;

Action:
  'action' name=ID
;

Comment:
  /\/\/.*$/
;
```

Figure 2.7: TextX Live-reload, From:(*textX for VS Code* 2022)

2.2.2 Xtext Generator Fragment for Visual Studio Code Extensions

Xtext Generator Fragment for Visual Studio Code Extensions is an open-source project developed by Itemis and published on Github. (*Xtext Generator Fragment for Visual Studio Code Extensions* 2022)

The workflow file in Xtext has the configurations of the language in Xtext. This workflow file allows the customization and or extension of the language through fragments. The fragments can change what modules of the Xtext create, and create custom modules, among other things. Listing 2.1 shows some fragments that Xtext has.

```
language = StandartGenerator {
    name = "org.example.domainmodel.Domainmodel"
    fileExtensions = "dmodel"
    referencedResource = "platform:/resource/org.eclipse.xtext.xbase/
   model/Xbase.genmodel"

    fragment = grammarAccess.GrammarAccessFragment2 {}
    fragment = ecore.EMFGeneratorFragment2 {}
    fragment = serializer.SerializerFragment2 {}
    fragment = resourceFactory.ResourceFactoryFragment2 {}
    fragment = parser.antlr.XtextAntlrGeneratorFragment2 {}
    fragment = validation.ValidatorFragment2 {}
    fragment = scoping.ImportNamespacesScopingFragment2 {}
    fragment = exporting.QualifiedNamesFragment2 {}
    fragment = builder.BuilderIntegrationFragment2 {}
    fragment = generator.GeneratorFragment2 {}
    fragment = formatting.Formatter2Fragment2 {}
    fragment = ui.labeling.LabelProviderFragment2 {}
    fragment = ui.outline.QuickOutlineFragment2 {}
    fragment = ui.outline.OutlineTreeProviderFragment2 {}
    fragment = ui.quickfix.QuickfixProviderFragment2 {}
    fragment = ui.contentAssist.ContentAssistFragment2 {}
    fragment = junit.JunitFragment {}
    fragment = ui.refactoring.RefactorElementNameFragment2 {}
    fragment = types.TypesGeneratorFragment2 {}
    fragment = xbase.XtypeGeneratorFragment2 {}
    fragment = xbase.XbaseGeneratorFragment2 {}
    fragment = ui.templates.CodetemplatesGeneratorFragment2 {}
    fragment = ui.compare.CompareFragment2 {}
    fragment = web.WebIntegrationFragment {
        framework = "Ace"
    }
    fragment = ui.projectWizard.TemplateProjectWizardFragment {}
    fragment = ui.fileWizard.TemplateFileWizardFragment {}
}
```

Listing 2.1: Xtext Language workflow fragments

This project makes use of a custom fragment. This fragment generates a new module containing the generated visual studio code extension. To achieve the creation of the extension, the fragment does the following steps:

- Obtains all the keywords from the Xtext grammar to create the TextMate file containing the syntax highlighting.
- Uses the language server created by Xtext to provide code suggestions and error highlighting.

Listing 2.2 is an example of the usage of this fragment.

```
language = StandardLanguage {
    ...
    fragment = com.itemis.xtext.generator.vscode.VSCodeExtensionFragment
    {
      provider = "My Company" // this property is mandatory
      // fragment configuration goes here...
    }
}
```

Listing 2.2: Xtext Generator Fragment

2.3 Summary

The following sections show the technologies that are going to be used and some comparisons with the related work.

2.3.1 Technologies Comparison and Decision

In regards to the syntax highlighter, **TextMate grammar** is the only supported by Visual Studio Code, so this is the one chosen for this purpose.

2.3.2 Related Work Comparison

Now that all the related work is known, certain distinctions can be established from this solution. Firstly, there is a comparison with TextX for VS Code, then with the Xtext Fragment from Itemis.

TextX for VS Code

The TextX solution accomplishes all the objectives this dissertation is trying to achieve. Unfortunately, the TextX solution does not work with ecore meta-models, which is a technological constraint of this dissertation.

Xtext Fragment

The Xtext Fragment solution generates an extension for Visual Studio Code from an Xtext project, which is also one objective of this dissertation.

One problem is that to generate the Xtext project from a meta-model, Eclipse is needed. This is one part where this dissertation is improving.

Another problem with the Xtext fragment is that, at the time of writing, it does not work for the new Xtext versions as it is outdated and has no support.

Chapter 3

Value Analysis

This section focuses on the envisioned value that this solution provides, utilizing the New Concept Development Model (NCD) to identify and analyze the opportunity, as well as define and analyze the perceived value, or value proposition, using the Quality Function Deployment and Analytic Hierarchy Process methods.

3.1 New Concept Development Model

As shown in Figure 3.1, the innovation process is separated into three main sections: Fuzzy front end (FFE), New product development (NPD), and commercialization (Koen et al. 2002).

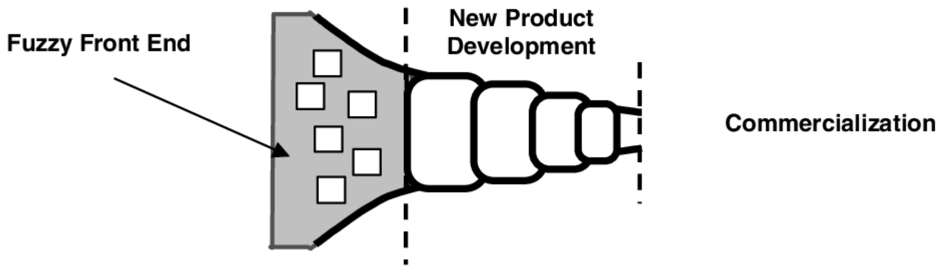


Figure 3.1: The Innovation Process, From: (Koen et al. 2002)

The Fuzzy Front End represents an unstructured, chaotic, and experimental state when commercialization and finance variables are unpredictable and company conceptions are improving and strengthening.

The New Concept Development (NCD) Model, as defined by Peter Koen (Koen et al. 2002), is a model created using standard language and terminology that aims to help improve operations in the FFE, resulting in a higher number of lucrative concepts entering the New product development (NPD).

The NCD model is a nonlinear process that is divided into three sections:

- **Bull's-eye section** - Also known as the engine, represents the factors that drive the five key elements that are controllable by the corporation, such as the organization's leadership, culture, and business strategy.

- **Inner spoke** - Identifies the five key elements under the organization's control, namely opportunity identification, opportunity analysis, idea generation, and enrichment, idea selection, and concept definition
- **Influencing Factors** - The organization's capabilities, the outside world, and the enabling sciences and technologies are all variables that affect the entire innovation process and influence the notion and its viability.

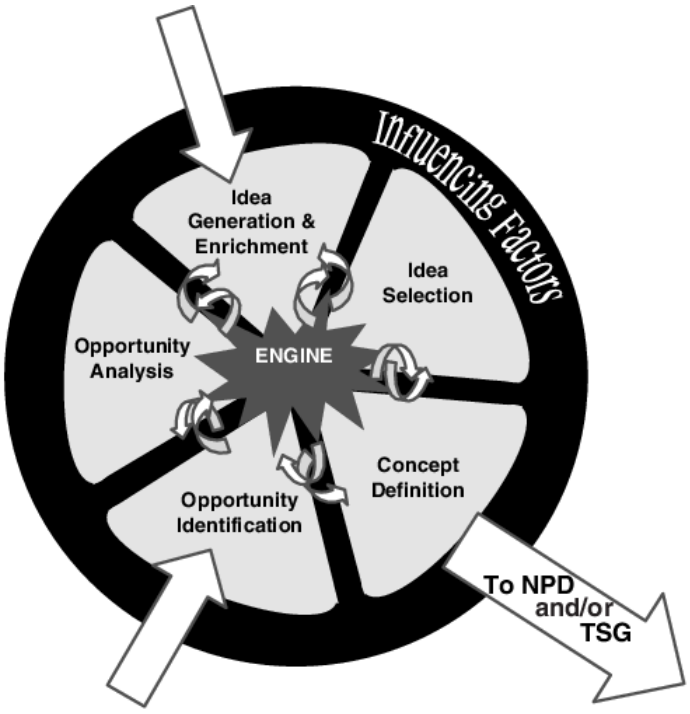


Figure 3.2: NCD model, From: (Koen et al. 2002)

Furthermore, the two arriving arrows show that a project can start at the "opportunity identification" or "idea generation and enrichment" stage.

3.1.1 Opportunity Identification

Opportunities can appear in a variety of forms, such as the launch of a new product or the addition of new features to existing products/services.

Given the growing popularity of low-code tools, it would make sense to try to make their development faster and easier. One area where developing low-code tools is slow is creating concrete syntax. So there are some options:

- A fully automatic concrete syntax generator;
- A generator that assists in the generation of the concrete syntax.

Another opportunity is in the deployment of the DSL into an IDE. Making that process automatic would also be of great help.

3.1.2 Opportunity Analysis

An opportunity is assessed to see whether it is worth pursuing. More information is needed to convert opportunity identification into particular business and technology prospects.

As a result, the SWOT analysis was used in order to better comprehend the highlighted opportunity. SWOT stands for Strengths, Weaknesses, Opportunities, and Threats, and so a SWOT analysis is a technique for assessing these four aspects of your business. The presence of internal and external components with positive and negative aspects results in a reasonably comprehensive SWOT instrument. Because of these multiple advantages, the SWOT analysis is still relevant when examining an opportunity (Sevgili Koçak and GÜVEN 2020). The SWOT analysis is divided into four parts:

- **Strength**: This statistic represents the favorable internal factors that bring value.
- **Weakness**: Internal defects that push back and result in fallacies and disadvantages for the opportunity when compared to others.
- **Opportunities**: External factors that enhance the potential.
- **Threats**: Identifies external negative forces that may cause the project to fail.

Strengths	Weaknesses
• Speed up the development process of DSLs. • Fast deployment of the language to an IDE.	• Hard to make complex language editors with many features.
Opportunities	**Threats**
• Most low-code solutions are not open source. • Increase use of domain-specific languages.	• Users are most likely to use other low-code platforms instead of creating their own DSL.

3.2 Value Proposition

The value proposition is a statement that identifies clear, measurable, and demonstrable benefits consumers get when buying a particular product or service. It should convince consumers that this product or service is better than others on the market. This proposition can lead to a competitive advantage when consumers pick that particular product or service over other competitors because they perceive a higher value (Hassan 2012).

The following statement proposition is the value proposition pitch for this project:

"The automatic language deployment to an IDE increases development efficiency, allowing developers and companies to be more productive."

3.2.1 Value Proposition Canvas

Dr. Alexander Osterwalder created the Value Proposition Canvas as a framework to guarantee that the product and market are a good match. It provides a detailed examination of the relationship between two components of Osterwalder's larger Business Model Canvas: client segmentation and value propositions. The Value Proposition Canvas can be used to refine an existing product or service offering or to create a new offering from the scratch (Osterwalder, Pigneur, et al. 2015).

The Figure 3.3 is the value proposition canvas for this project.

Customer Profile

Customer profiling consists of three primary components: gains, pains, and customer jobs:

- **Gains**: the benefits that the customer expects and requires, what would please customers, and what could boost the possibility of adopting a value proposition;
- **Pains**: the negative experiences, emotions, and risks that the customer encounters while performing a job;
- **Customer jobs**: the functional, social, and emotional tasks that customers are attempting to complete, the issues that they are trying to solve, and the requirements that they wish to satisfy.

Value Map

- **Gain creators**: how the product or service generates customer gains and adds value to the customer;
- **Pain relievers**: a detailed description of how the product or service alleviates the customer's pains;
- **Products and services**: the items and services that increase gains and alleviate pain, and which underpin the creation of value for the customer.

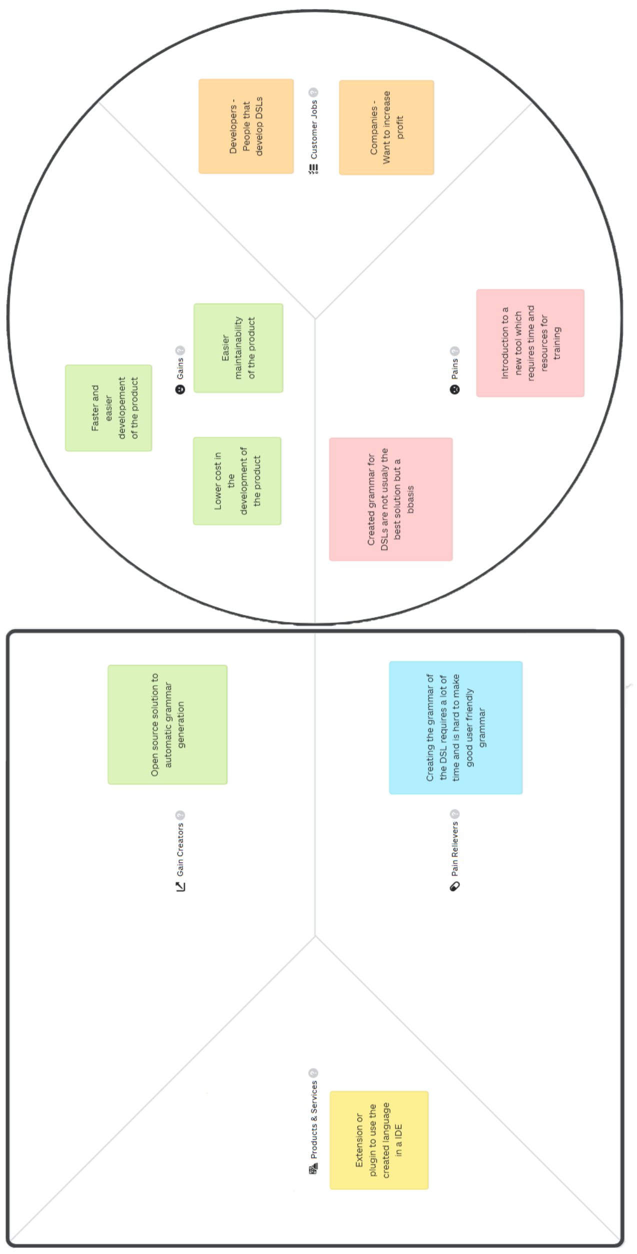

Developers - People that develop DSLs
Customer Jobs
Companies - Want to increase profit
Easier maintainability of the product
Gains
Faster and easier developement of the product
Lower cost in the development of the product
Pains
Introduction to a new tool which requires time and resources for training
Created grammar for DSLs are not usualy the best solution but a bbasis
Open source solution to automatic grammar generation
Gain Creators
Creating the grammar of the DSL requires a lot of time and is hard to make good user friendly grammar
Pain Relievers
Products & Services
Extension or plugin to use the created language in a IDE

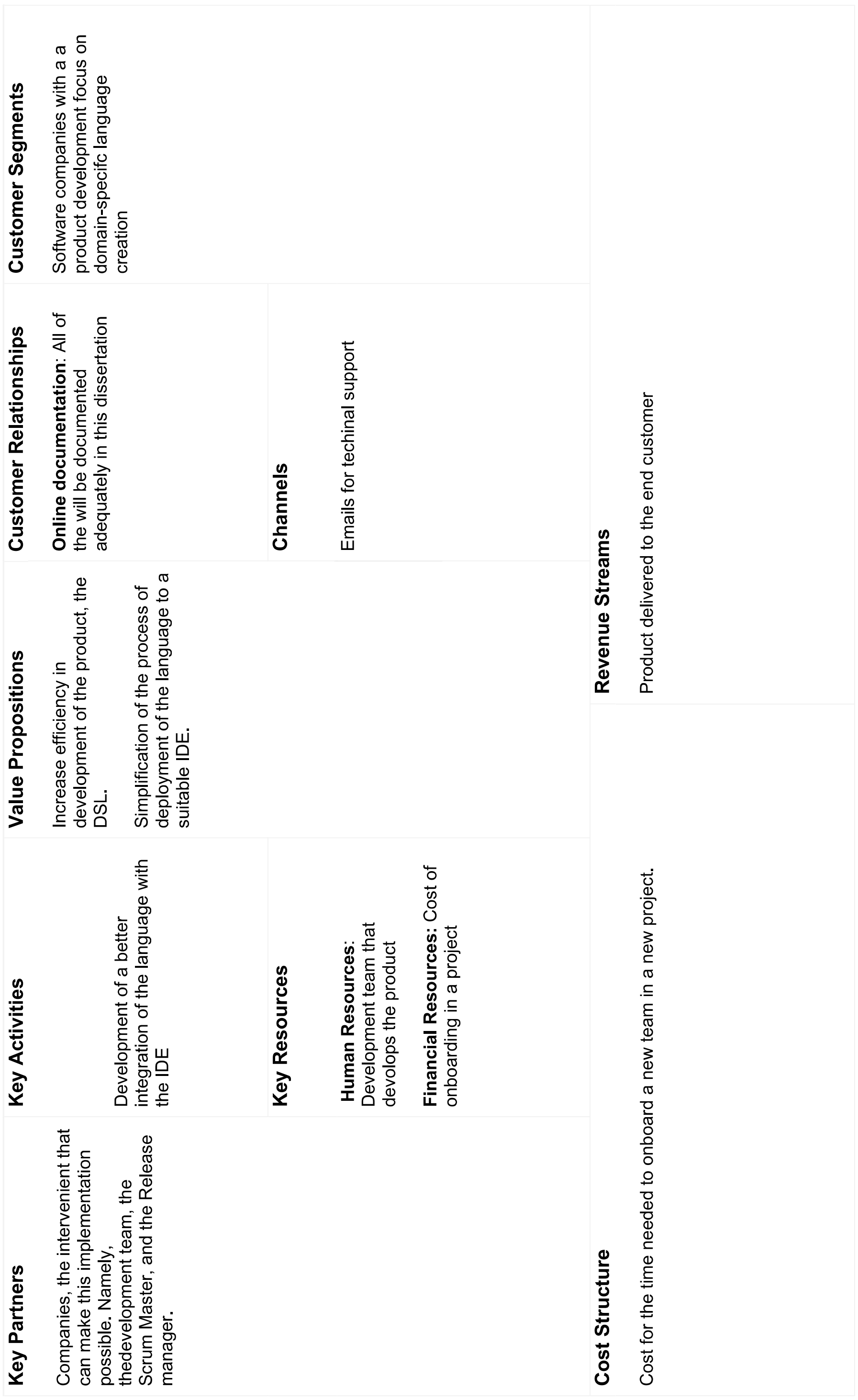
Figure 3.3: Value Canvas

The Business Model is "a conceptual tool that contains a set of elements and their relationships and allows expressing the business logic of a specific firm" (Osterwalder and Pigneur 2003). The Canvas Business Model will be used to quickly develop the business model. Canvas Business Model is separated into nine sections that are critical for the development of the business model. The following sections will be presented:

- **Key Partners**: the network of suppliers and partners that ensures the business model's viability;
- **Key Activities**: the most critical things a company must do to ensure the viability of its business model;

- **Key Resources**: the most critical assets required to make your business model function;
- **Value Propositions**: a collection of products and services that add value to a specific Customer Segment;
- **Customer relationships**: the kinds of partnerships your organization establishes with specific Customer Segments;
- **Channels**: how your organization connects with and reaches out to your Customer Segments in order to convey your value proposition;
- **Customer Segments**: are the various groups of people or organizations that your business hopes to reach and service;
- **Revenue Streams**: reflect the various methods in which your organization produces revenue from each Customer Segment;
- **Cost Structure**: all cost incurred in order to run your business model.

Figure 3.4 shows the model for this project.



Key Partners

Companies, the intervenient that can make this implementation possible. Namely, thedevelopment team, the Scrum Master, and the Release manager.

Key Activities

Development of a better integration of the language with the IDE

Key Resources

Human Resources: Development team that devolops the product

Financial Resources: Cost of onboarding in a project

Value Propositions

Increase efficiency in development of the product, the DSL.

Simplification of the process of deployment of the language to a suitable IDE.

Customer Relationships

Online documentation: All of the will be documented adequately in this dissertation

Channels

Emails for techinal support

Customer Segments

Software companies with a a product development focus on domain-specifc language creation

Cost Structure

Cost for the time needed to onboard a new team in a new project.

Revenue Streams

Product delivered to the end customer

Figure 3.4: Value Canvas

3.3 Analytic Hierarchy Process (AHP)

Thoma L. Saaty invented the Analytic Hierarchy Process (AHP) in 1980 as a multi-criteria decision method. AHP is an accurate method for measuring the weights of decision criteria that may be utilized with both qualitative and quantitative criteria. The goal of AHP is to break the problem into hierarchical decision levels, making it easier to understand.

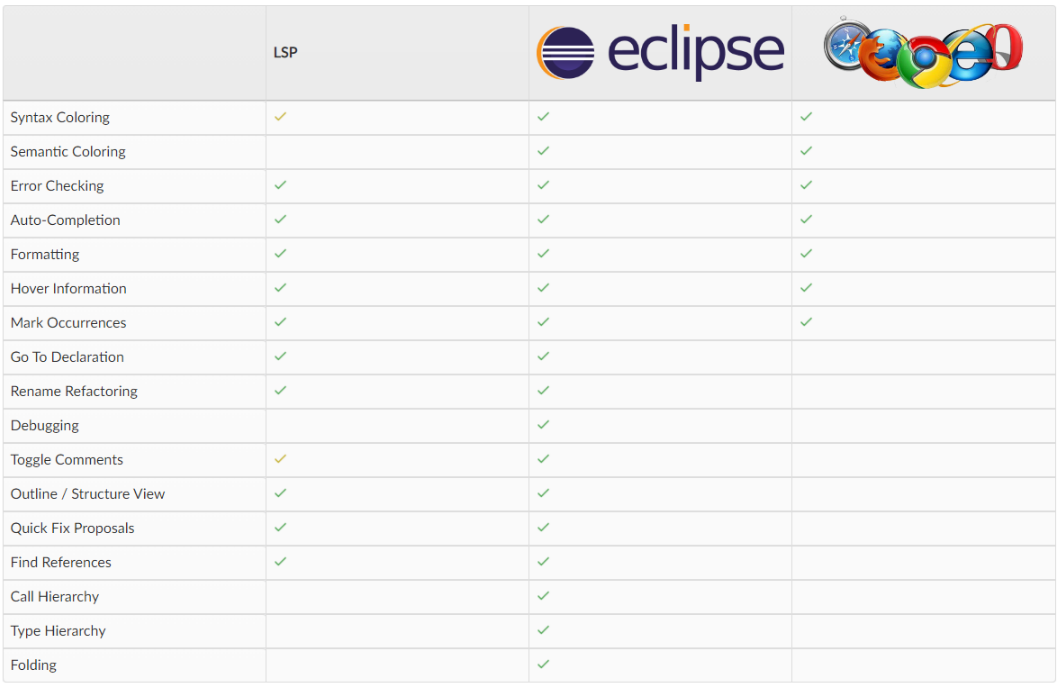
The AHP technique begins with the construction of a hierarchical decision tree, with three levels representing the problem, the criteria, and the choices, respectively (Vaidya and Kumar 2006).

The only problem to solve is the tool to choose as IDE for language usage since the tool for designing domain-specific languages is Xtext. These are the alternatives for this project:

- Eclipse IDE, using the built-in Xtext integration with eclipse;
- Visual Studio Code, via an extension and language server protocol;
- Theia, via an extension and language server protocol.

The following criteria were used to determine which is the best tool to use in this project:

- **Integration** - How much the tool can be integrated with Xtext, according to the Figure 3.5.
- **Potential** - The features that can be created within the tool, to take best advantage of DSL.
- **Active Users** – In comparison, how many active users the tool has, according to the recent Stack Overflow 2021 Developer Survey (*Stack Overflow Developer Survey 2021 - Integrated Development Environment* 2022)



	LSP	eclipse	
Syntax Coloring	✓	✓	✓
Semantic Coloring		✓	✓
Error Checking	✓	✓	✓
Auto-Completion	✓	✓	✓
Formatting	✓	✓	✓
Hover Information	✓	✓	✓
Mark Occurrences	✓	✓	✓
Go To Declaration	✓	✓	
Rename Refactoring	✓	✓	
Debugging		✓	
Toggle Comments	✓	✓	
Outline / Structure View	✓	✓	
Quick Fix Proposals	✓	✓	
Find References	✓	✓	
Call Hierarchy		✓	
Type Hierarchy		✓	
Folding		✓	

Figure 3.5: Xtext Editor Features by Platform , From:(*Xtext* 2022)

The Figure 3.6 represents the Hierarchical Decision Tree.

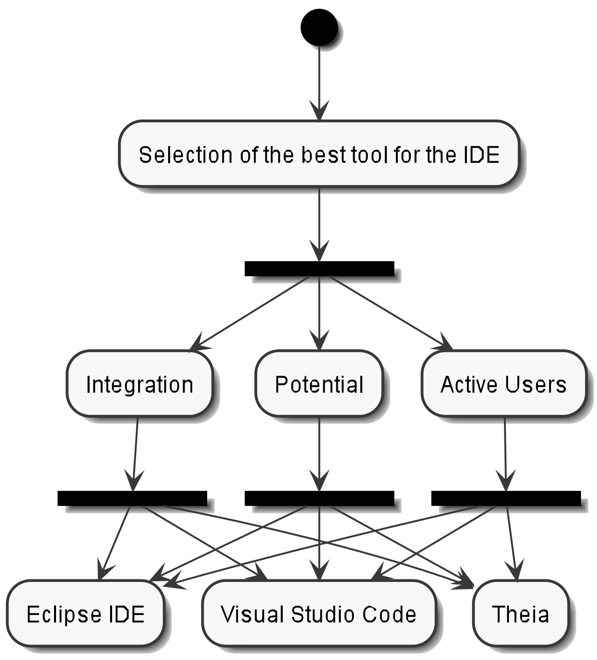


Figure 3.6: Hierarchical Decision Tree

Alternatives and Criteria In the next phase, a comparison matrix is utilized to assign priority to all alternatives and criteria. The Saaty fundamental scale is used to assign priority levels to the criterion, as shown in Table 3.1.

Table 3.1: Saaty fundamental scale, From: (Saaty 1990)

Importance Level	**Definition**	**Explanation**
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance of one over another	Experience and judgment strongly favor one activity over another
5	Essential of strong importance	Experience and judgment strongly favor one activity over another
7	Very strong importance	An activity is strongly favored and its dominance demonstrated in practice
9	Extreme importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgment	When compromise is needed

The pairwise comparison of the defined criteria may be observed in Table 3.2.

After the comparison matrix, the next step is to calculate the relative priority of each criterion by normalizing the pairwise comparison matrix (Table 3.2) and then computing the

Table 3.2: Comparison Matrix between Criteria

	Integration	**Potential**	**Active Users**
Integration	1.000	2.000	3.000
Potential	0.500	1.000	2.000
Active Users	0.333	0.500	1.000

arithmetic average of the values in each line of the normalized matrix. Table 3.3 displays the acquired results.

Table 3.3: Normalized Comparison Matrix and Relative Priority Vector

	Integration	**Potential**	**Active Users**	**Relative Priority**
Integration	0.545	0.571	0.500	0.539
Potential	0.273	0.286	0.333	0.297
Active Users	0.182	0.143	0.167	0.164

According to the results, the most critical criterion is the **Integration**. This result is as expected since it is desired that the IDE works as well as possible with Xtext. The criterions are sorted in the following order: **Integration** > **Potencial** > **Active Users**.

The consistency ratio (RC) is then calculated to evaluate the consistency of the judgments. If the value of RC is less than one, the judgments are regarded as dependable (Vaidya and Kumar 2006). First, λ_{max} is calculated using the formula:

$$Ax = \lambda_{\text{max}} x \tag{3.1}$$

Where A is the criteria comparison matrix and x is the priority vector. The result of λ_{max} is approximately 3.009.

With this value it is possible to calculate the Consistency Index (CI) using the following formula:

$$CI = \frac{\lambda_{\text{max}} - n}{n - 1} \tag{3.2}$$

When the formula is replaced with the previously obtained values and n with the number of criteria, the result obtained is:

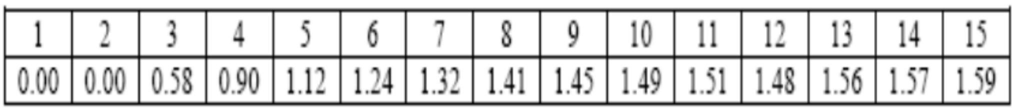
$$CI = \frac{3.09 - 3}{3 - 1} \approx 0,0046 \tag{3.3}$$

Finally, the Consistency Ratio (CR) can be calculated using the formula below:

$$CR = \frac{CI}{RCI} \tag{3.4}$$

As a result, the RC still requires the Random Consistency Index for our matrix, which corresponds to the RCI.

Table 3.4: Random Consistency Index, From:(Vaidya and Kumar 2006)



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49	1.51	1.48	1.56	1.57	1.59

As this is a 3x3 matrix, the Random Consistency value (RCI) for order 3 matrices is 0.58 according to Table 3.4. So the CR is:

$$CR = \frac{CI}{RCI} = \frac{0.0046}{0.58} \approx 0.008 \tag{3.5}$$

Because the resulting CR value was roughly 0.008, which is less than 0.1, it is possible to assume that the priority values are considered reliable.

After calculating the global priority vector, the next step is to generate a comparison matrix for each criterion. Each matrix uses all alternatives to determine the best approach based on the criteria. The procedure is done by repeating the creation of the comparison matrices (this time, between each criterion and alternative), the normalizing matrix, and the priority vector calculation. Tables 3.5, 3.6 and 3.7 show the comparison using the fundamental scale (Table 3.2).

Table 3.5: Comparison Matrix between Integration in Alternatives

Integration			
	Eclipse IDE	**VS Code**	**Theia**
Eclipse IDE	1.000	2.000	2.000
VS Code	0.500	1.000	1.000
Theia	0.500	1.000	1.000

Table 3.6: Comparison Matrix between Potential in Alternatives

Potential			
	Eclipse IDE	**VS Code**	**Theia**
Eclipse IDE	1.000	0.333	0.200
VS Code	3.000	1.000	0.500
Theia	5.000	2.000	1.000

Table 3.7: Comparison Matrix between Active Users in Alternatives

Active Users			
	Eclipse IDE	**VS Code**	**Theia**
Eclipse IDE	1.000	0.200	2.000
VS Code	5.000	1.000	7.000
Theia	0.500	0.143	1.000

The matrices are then normalized, and the local priority vector for each is calculated. Tables 3.8, 3.9 and 3.10 display the results.

Table 3.8: Normalized Comparison Matrix and Local Priority for the comparison between Alternatives regarding Integration Criterion

Integration				
	Eclipse IDE	**VS Code**	**Theia**	**Local Priority**
Eclipse IDE	0.500	0.500	0.500	0.500
VS Code	0.250	0.250	0.250	0.250
Theia	0.250	0.250	0.250	0.250

Table 3.9: Normalized Comparison Matrix and Local Priority for the comparison between Alternatives regarding Potential Criterion

Potential				
	Eclipse IDE	**VS Code**	**Theia**	**Local Priority**
Eclipse IDE	0.111	0.100	0.118	0.110
VS Code	0.333	0.300	0.294	0.309
Theia	0.555	0.600	0.589	0.581

Table 3.10: Normalized Comparison Matrix and Local Priority for the comparison between Alternatives regarding Active Users Criterion

Active Users				
	Eclipse IDE	**VS Code**	**Theia**	**Local Priority**
Eclipse IDE	0,154	0.149	0.200	0.168
VS Code	0.769	0.745	0.700	0.738
Theia	0.077	0.106	0.100	0.094

The result from Tables 3.8, 3.9 and 3.10 are then combined with Table 3.3's relative priority to create a new matrix. The Composite Priority Vector can be calculated using these values by multiplying the outcome of each alternative with the priority vector. Based on the criteria, this vector will indicate the relevance of each alternative.

Table 3.11: Criteria/Alternatives Classification Matrix and Global Priority

	Integratioon	**Potential**	**Active Users**	**Global Priority**
Eclipse IDE	0,500	0.110	0.168	0.330
VS Code	0.250	0.309	0.738	0.348
Theia	0.250	0.581	0.094	0.323

To sum up, the AHP method and the related findings of the global priority vector from Table 3.11 indicate that **VS Code** is the preferred tool as the IDE, followed **Eclipse** and **Theia**.

Chapter 4

Analysis and Design

This chapter focuses on the functional and non-functional needs of the project. Also, the description of the requirements and the design that results from the project requirements is shown.

4.1 Functional and Non-Functional Requirements

A functional requirement in software engineering describes a function of its component, where the function is defined as a definition of behavior between inputs and outputs (Fulton and Vandermolen 2017). Non-functional requirements (sometimes known as "quality requirements") impose constraints on the design or implementation of functional requirements (such as performance requirements, security, or reliability). In general, functional requirements are expressed as "system must do <requirement>," whereas non-functional requirements are expressed as "system shall be <requirement>" (Loucopoulos 2005).

Requirements for this project are as follows.

4.1.1 Functional Requirements

- **FR1**: Users can create the grammar given the meta-model, and the root object;
- **FR2**: Users can deploy the language created to Visual Studio Code.

It is important to note that to generate a grammar of a meta-model, the meta-model has to be valid.

4.1.2 Non-Functional Requirements

The **FURPS+** model will be used to analyze non-functional requirements. “FURPS stand for functionality, usability, reliability, performance and supportability” (Shaha and Pawar 2018). The "+" in the FURPS+ acronym provides for the specification of constraints such as design, implementation, interface, and physical constraints.

Functionality Specifies the features that are not related to the use cases, namely: auditing, reporting, interoperability, and security. No non-functional requirement was identified to fit this attribute.

Usability Evaluates the user interface. It has several subcategories, including error prevention, aesthetics, design, aids, documentation, consistency, and standards. In this project, the non-functional requirements that fit this attribute are:

- **NFR1**: The extension has to have keyword and error highlighting.
- **NFR2**: The extension has to have some features such as formatting, quick fix, auto-completion, and go-to the declaration.

Reliability Refers to software integrity, compliance, and interoperability. The requirements to be considered are frequency and severity of failures, the possibility of recovery, extent, duration of failure (recovery/survival), and predictability (stability). No non-functional requirement was identified to fit this attribute.

Performance Evaluates the performance requirements of the software, namely: response time, resource consumption, capacity, and scalability. No non-functional requirement was identified to fit this attribute.

Supportability Testability, adaptability, maintainability, compatibility, configurability, installability, and scalability are all examples of supportability requirements. In this project, the non-functional requirements that fit this attribute are:

- **NFR3**: The visual studio extension should be tested.
- **NFR4**: The grammar generation should support ecore meta-models as input.

Design Constraints A design constraint, as the name implies, restricts the design. Design constraints include Design Patterns and Software Development Processes. No non-functional requirement was identified to fit this attribute.

Implementation Constraints Specifies or constrains a system's code or construction using constraints such as Resource Limits and Operating Systems. No non-functional requirement was identified to fit this attribute.

Interface Constraints Specifies or restricts the functionality inherent in different component interfaces. External modules are commonly used, and the constraints that come with them must be considered in this section. No non-functional requirement was identified to fit this attribute.

Physical Constraints Specifies a physical constraint imposed by the hardware used to deploy the system. No non-functional requirement was identified to fit this attribute.

4.2 Design

The C4 Model and the 4+1 architectural view model are going to be used in conjunction to better describe the design of this project.

C4 Model The C4 model is an "abstraction-first" approach to software architecture diagramming that is based on abstractions that reflect how software architects and developers think about and build software. The C4 model considers the static structures of a software system in terms of containers, components, and code. It is important to note there is no need to use all 4 diagram levels, only those that add value (*The C4 model for visualising software architecture* 2022). The levels are used in this project are:

- **Context**: which is the highest level of abstraction and describes something that delivers value to its users.
- **Container**: which represents an application or a data store.
- **Component**: which is a grouping of related functionality encapsulated behind a well-defined interface.
- **Code**: which represents how each component is implemented as code.

4+1 architectural view model 4+1 is a view model that is used to "describe the architecture of software-intensive systems using multiple, concurrent views." (Kruchten 1995) The model has four views: logical, development, process, and physical. In addition, selected use cases or scenarios are used as the 'plus one' view to illustrate the architecture. As a result, there are 4+1 views in the model: (Kruchten 1995)

- **Logical view**: The logical view is concerned with the system's functionality as it pertains to end-users.
- **Process view**: The process view focuses on the system's run-time behavior and deals with the system's dynamic aspects. It explains the system processes and how they communicate.
- **Development view**: The development view depicts a system from a programmer's standpoint and is concerned with software management.
- **Physical view**: The physical view portrays the system from the perspective of a system engineer. The physical layer is concerned with the topology of software components and the physical connections between these components.
- **Scenarios**: A small number of use cases, or scenarios, that become the fifth view, are used to illustrate the description of architecture. Sequences of interactions between objects and processes are described in the scenarios.

Figure 4.1 shows a diagram with the relation between the views.

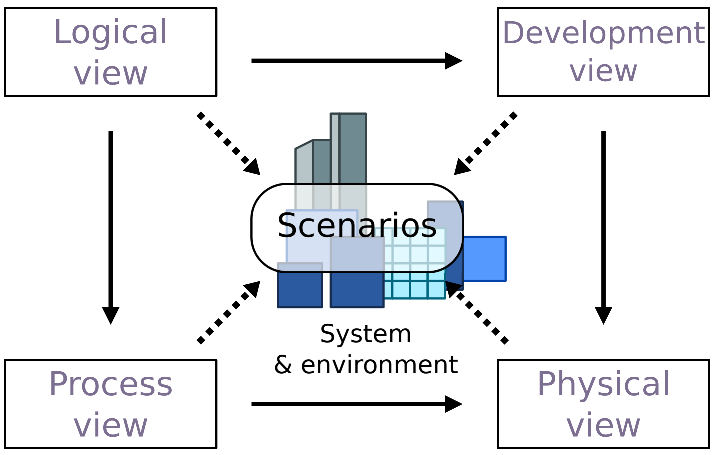


Figure 4.1: Illustration of the 4+1 Architectural View Model, From (Kruchten 1995)

4.2.1 Context

To better understand the context of this project, the diagram in Figure 4.2 was created.

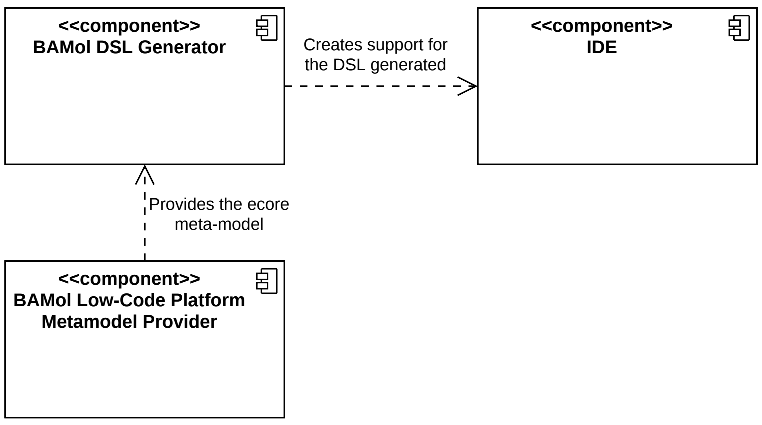

<<component>> BAMol DSL Generator
Creates support for the DSL generated
<<component>> IDE
Provides the ecore meta-model
<<component>> BAMol Low-Code Platform Metamodel Provider

Figure 4.2: System Context Logical View

The component that is going to be developed is the **BAMol DSL Generator**. To generate the DSL, BAMol DSL Generator takes the ecore meta-model from the component **BAMol Low-Code Platform Metamodel Provider**. Then, BAMol DSL Generator creates an extension for Visual Studio Code, the **IDE**, that adds support for the language.

4.2.2 Alternatives

For the design, three possible alternatives were created. To describe the alternatives, the container logical view and component logical view are used.

Alternative 1

In this first alternative, the application developed in this dissertation only allows the generation of the Visual Studio Code extension of the DSL. That process requires the Xtext project with the correct modules created.

Since Xtext does not have a standalone application to generate Xtext projects, Eclipse is required. Figure 4.3 shows the container logical view.

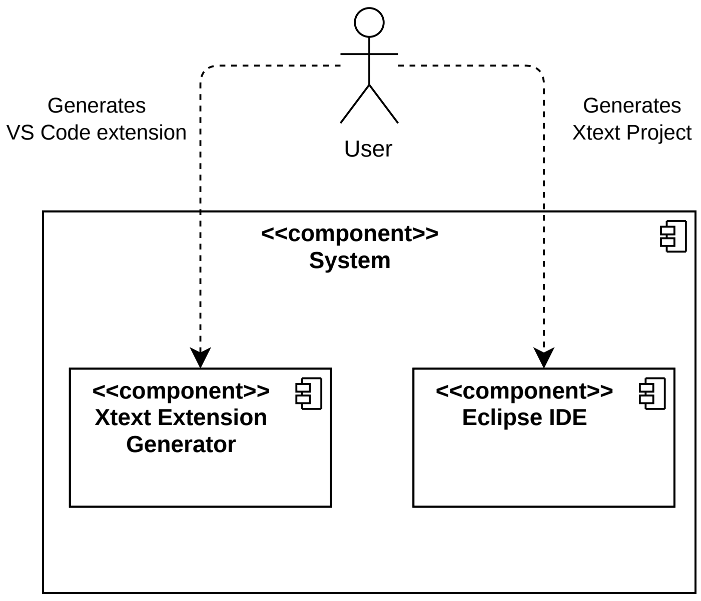


Figure 4.3: Container Logical View of Alternative 1

Let's call the application created in this alternative Xtext Extension Generator.

Now getting into a finer level of detail, the component level, it is possible to see how the component Xtext Extension Generator is decomposed, in Figure 4.4. The components are:

- **Controller**: the component that handles and performs requests with the use of other components. It only supports requests for the generation of Visual Studio Code extensions in this scenario.
- **Xtext**: allows the Xtext grammar to be loaded. This grammar is generated by the Eclipse component. It should be noted that this is not Xtext itself, but rather a module with Xtext-related functionality.
- **VS Code**: allows the generation of the Visual Studio Code extension given the grammar. This component uses the TextMate component for the syntax highlighting in the extension.
- **TextMate**: given the grammar of the DSL, it creates a TextMate file syntax highlight file.

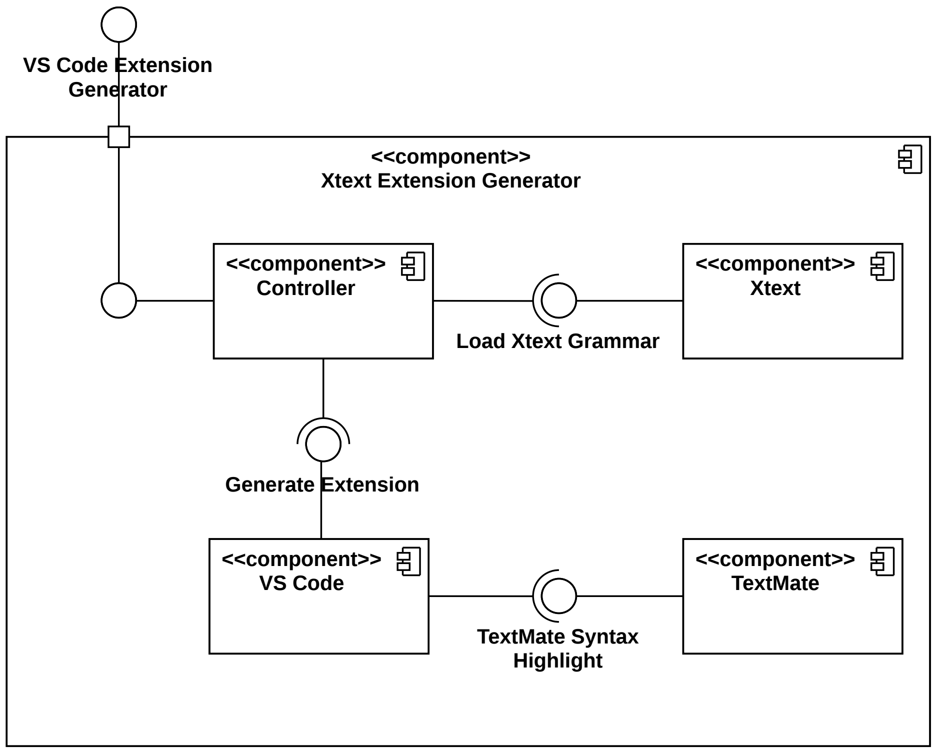


Figure 4.4: Component Logical View of Alternative 1

Figure 4.5 shows a better view of how the components of the application work to fulfill FR1, to create the Xtext project, and FR2, to deploy the language to Visual Studio Code.

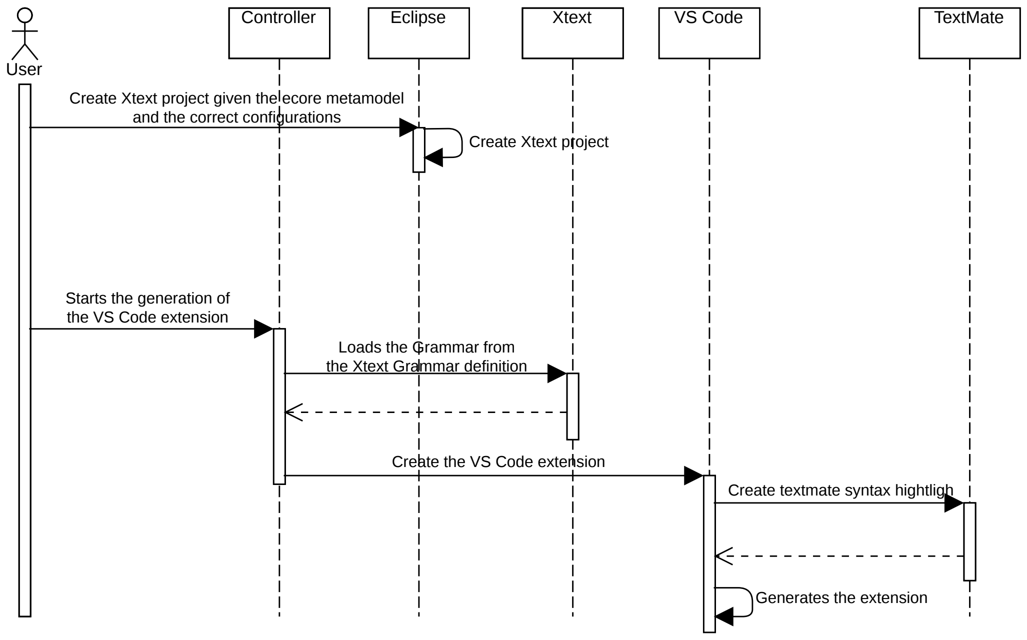

User
Controller
Eclipse
Xtext
VS Code
TextMate
Create Xtext project given the ecore metamodel and the correct configurations
Create Xtext project
Starts the generation of the VS Code extension
Loads the Grammar from the Xtext Grammar definition
Create the VS Code extension
Create textmate syntax hightligh
Generates the extension

Figure 4.5: Component Process View for FR1 and FR2 of Alternative 1

Alternative 2

For this alternative, the application developed allows the creation of both the Xtext project and the Visual Studio Code extension for the DSL. This alternative joins both functionally from alternative 1 in one application. One bonus of this is that the configuration of the Xtext project is predefined to work with the extension generator. Figure 4.6 shows the container logical view.

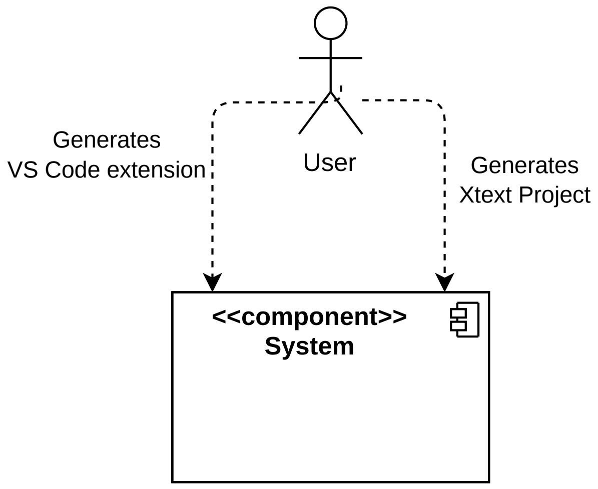


Figure 4.6: Container Logical View of Alternative 2

Given that the application to develop fulfills all the functional requirements, the components change in comparison to the first alternative. Therefore the components in this alternative are:

- **Controller**: the component that handles and performs requests with the use of other components. It supports requests for the generation of Visual Studio Code extensions and the generation of Xtext projects.
- **Xtext**: allows the Xtext grammar to be loaded and the generation of Xtext projects. It should be noted that this is not Xtext itself, but rather a module with Xtext-related functionality.
- **Ecore**: allows Ecore models to be loaded.
- **VS Code**: allows the generation of the Visual Studio Code extension given the grammar. This component uses the TextMate component for the syntax highlighting in the extension. It should be noted that this is not Visual Studio Code itself, but rather a module with VS Code-related functionality.
- **TextMate**: given the grammar of the DSL, it creates a TextMate file syntax highlight file. It should be noted that this is not TextMate itself, but rather a module with TextMate-related functionality.

The diagram in Figure 4.7 shows how the components communicate with each other.

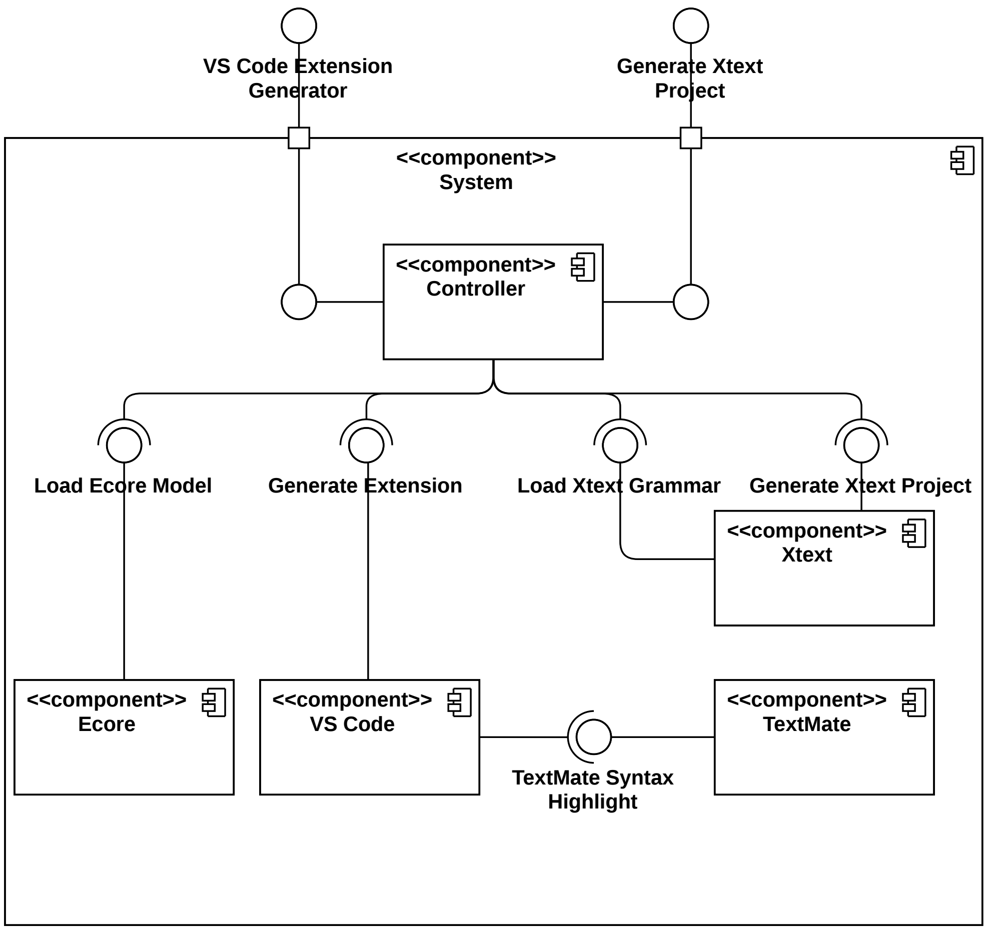


Figure 4.7: Component Logical View of Alternative 2

To fulfill FR1, to create the grammar, meaning the Xtext project, Figure 4.8 shows a better view of the process between the components.

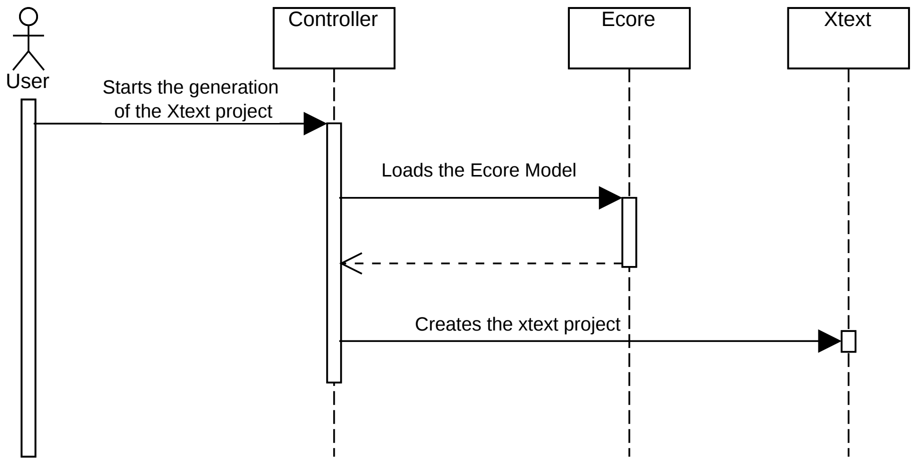

User
Controller
Ecore
Xtext
Starts the generation of the Xtext project
Loads the Ecore Model
Creates the xtext project

Figure 4.8: Component Process View for FR1 of Alternative 2

Figure 4.9 shows a better view of how the components of the application work to fulfill FR2, to deploy the language to Visual Studio Code.

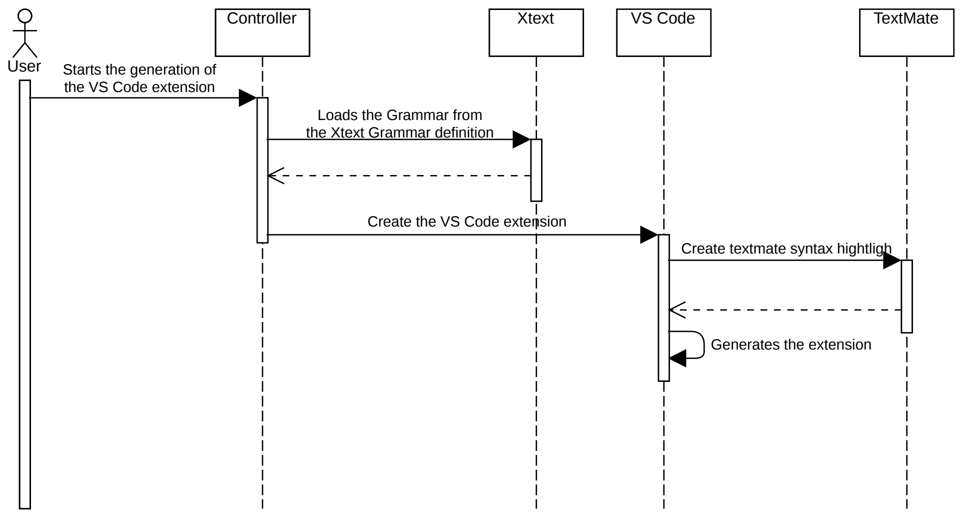

User
Controller
Xtext
VS Code
TextMate
Starts the generation of the VS Code extension
Loads the Grammar from the Xtext Grammar definition
Create the VS Code extension
Create textmate syntax hightligh
Generates the extension

Figure 4.9: Component Process View for FR2 of Alternative 2

Alternative 3

In this last alternative, the application would be a lot like the related work in 2.2.2. The fragment would only create the Visual Studio Code extension. Like alternative 1, the generation of the Xtext project would be through Eclipse IDE.

The fragment would have to be in the Xtext project workflow. Executing the default generation of the language process would also create the Visual Studio Code extension.

Figure 4.10 shows the container logical view.

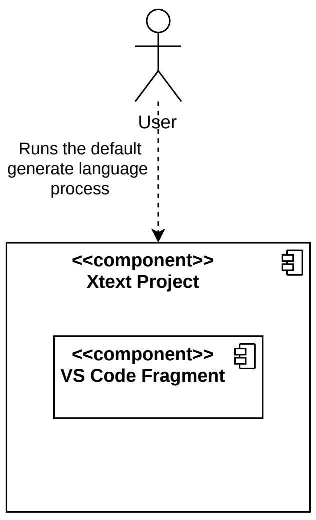

User
Runs the default generate language process
<<component>> Xtext Project
<<component>> VS Code Fragment

Figure 4.10: Container Logical View of Alternative 3

Now that the fragment is within the Xtext project workflow, it can access the grammar and other settings, reducing the components needed for it, as can be seen in Figure 4.11. So the components in this alternative are:

- **VS Code**: allows the generation of the Visual Studio Code extension given the grammar. This component uses the TextMate component for the syntax highlighting in the extension.
- **TextMate**: given the grammar of the DSL, it creates a TextMate file syntax highlight file.

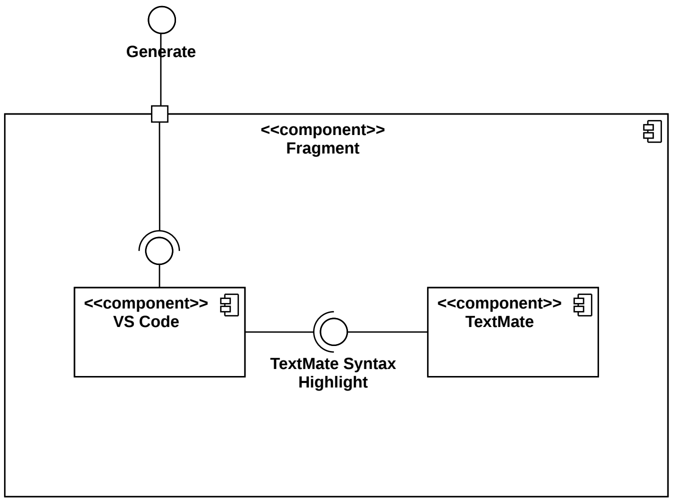


Figure 4.11: Component Logical View of Alternative 3

Summary

To determine which alternative to choose, they are compared against one another. This comparison is based on the advantages and disadvantages of each solution. Tables 4.1, 4.2 and 4.3 show the comparisons. Finally, the ideal option for the requirements is chosen.

Table 4.1: Alternative 1 Advantages and Disadvantages

Alternative 1	
Advantages	**Disadvantages**
• Generation of the Xtext projects is maintained by Eclipse. • Visual Studio Code extension generation is in a separate application.	• Uses multiple application to fulfills all requirements.

Table 4.2: Alternative 2 Advantages and Disadvantages

Alternative 1	
Advantages	**Disadvantages**
• Visual Studio Code extension generation and Xtext project generation are in the same application. • No need for other applications	• Slightly more complex.

Table 4.3: Alternative 3 Advantages and Disadvantages

Alternative 1	
Advantages	**Disadvantages**
• Generation of the Xtext projects is maintained by Eclipse. • Easier access to language grammar and other settings.	• Needs setup for every project.

Given all the advantages and disadvantages of all alternatives, the ideal solution is **Alternative 2**. It offers more pros, and the only con is that more complex that might require more work.

Chapter 5

Implementation

After finishing the Design phase, which comprises the layout of all components and the description of the Architecture, the developer can safely begin the implementation phase. As a result, this chapter will focus on implementation details and reveal the necessary components so that the reader can gain a better grasp of the application's inner workings. First, the project structure will be described. Following that, the implementation of each component of the chosen design is documented.

5.1 Project Structure

Gradle is being used as the build automation tool framework in this project. Each component for the selected design 4.2.2 is a module in Gradle project. The project structure is shown in Figure 5.1.

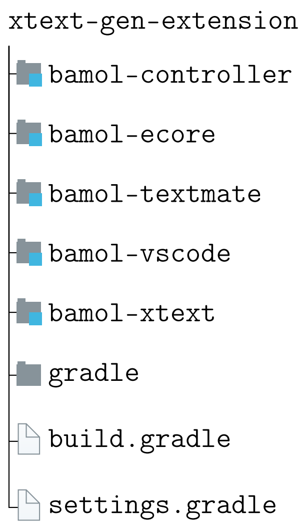


Figure 5.1: Project Structure Tree

Each module contains the following folders and file:

- **src/main/kotlin** - contains the source code of the application.
- **src/mainresource** - contains resources of the project. This is usually templates for file creation.
- **src/test** - contains tests of the source code.

- **build.gradle** - contains the dependencies for the module.

The module structure is as follows:

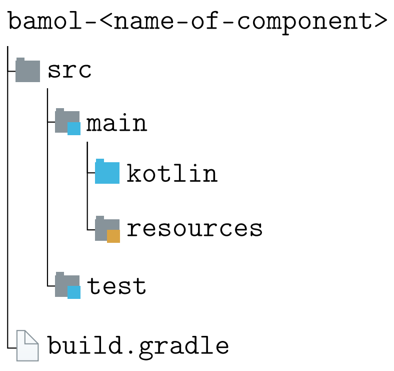


Figure 5.2: Module Structure Tree

5.2 Ecore Module

This module allows the loading of ecore models. An eclipse library for ecore models is utilized to accomplish this. Given that there is a library, this module should be simple, but unfortunatly some resources have to be instantiated.

To give a bit of context, eclipse libraries read resources from files using Resource Sets. This Resource Sets use factories to parse the content of a file. The factory to use is determined by the extension of the file.

Ecore models is decomposed into two files with file extensions **.ecore** and **.genmodel**. So the resource factory has to be set for both extensions.

```kotlin
private fun createResourceSet(): ResourceSet {
    val resourceSet = ResourceSetImpl()
    val ecoreFactory = EcoreResourceFactoryImpl()

    val registry = resourceSet.resourceFactoryRegistry
    val map = registry.extensionToFactoryMap
    map["ecore"] = ecoreFactory
    map["genmodel"] = ecoreFactory

    //Inits the GenModelPackage
    GenModelPackage.eINSTANCE
    return resourceSet
}
```

Listing 5.1: Creating Resource Set to load Ecore Models

Now loading a ecore model is as simple as:

```
val resourceSet: ResourceSet = createResourceSet()

// Model Uniform Resource Identifier
val genModelURI = URI.createFileURI(genModel.absolutePath)

// The Ecore model loaded.
val resource = resourceSet.getResource(genModelURI, true)
```

Listing 5.2: Loading Ecore Models

With the model resource loaded, the model resource is manipulated to ease the creation of Xtext projects. This module also contains some utilitarian methods.

5.3 Xtext Module

This module allows the creation of Xtext projects and the loading of Xtext grammar.

5.3.1 Creation of Xtext Project

For this feature, the Xtext library implemented in Eclipse is used. So to create a project is as simple as calling the library with some configuration.

The configuration that is needed to provide contains the following parameters:

- **targetDirectory** - The directory where the projects are going to be created.
- **languageName** - The language name.
- **baseName** - The base name of the project.
- **fileExtensions** - The file extensions of the dsl.
- **packageInfos** - The set of EPackageInfo for the ecore configuration. The EPackageInfo contains the Package that has all the elements of the language.
- **defaultPackage** - The default EPackageInfo for the ecore configuration.
- **rootElementClass** - The root element class for the ecore configuration.

Some configurations are necessary for the visual studio extension generation. These configurations are preset and are the following:

```
val PROJECT_LAYOUT = ProjectLayout.HIERARCHICAL
val SOURCE_LAYOUT = SourceLayout.MAVEN
val BUILD_SYSTEM = BuildSystem.GRADLE
val JAVA_VERSION = JavaVersion.JAVA11
val JUNIT_VERSION = JUnitVersion.JUNIT_5
val LANGUAGE_SERVER = LanguageServer.FATJAR
val ENCODING = Charsets.UTF_8
val LINE_DELIMITER: String = LineDelimiter.UNIX.value
```

Listing 5.3: Xtext preset configurations

The following Listing 5.4 shows the implementation of the creation of a project.

```
fun create(xtextConfiguration: XtextConfiguration) {
    val config = createXtextConfiguration(xtextConfiguration)
    val creator = CliProjectsCreator()

    creator.lineDelimiter = config.lineDelimiter
    creator.createProjects(config)
}
```

Listing 5.4: Create Xtext Project

5.3.2 Load Xtext Grammar

Like loading ecore models, loading an xtext grammar requires a resource set. So there was a need to also register a factory for **.xtext** files. However there was another problem.

A Resource Set to locate files uses Uniform Resource Identifier (URI). URIs uses a protocol and a path to locate the resource. For example, to locate a file in the system, the URI can be: **"file:/root/file.txt"**.

The Xtext Resource Set uses the **classpath** protocol, which locates resources inside the JVM. This protocol is not implemented by default, so a custom implementation was needed for this to work.

```
XtextStandaloneSetup.doSetup()
Resource.Factory.Registry.INSTANCE.protocolToFactoryMap["classpath"] =
    object : ResourceFactoryImpl() {
        override fun createResource(uri: URI): Resource? {
            val resource: URL? = this.javaClass.getResource(uri.path())
            return XtextResourceSet().createResource(URI.createURI(
    resource!!.toExternalForm()))
        }
    }
```

Listing 5.5: Xtext Resource Set

5.4 TextMate Module

This module creates TextMate files for Xtext grammar. TextMate files are necessary for the syntax highlighting of a language in the VS Code extension.

The Xtext grammar contains all the information necessary to create the TextMate file. The information read is the keywords, language name, and comments tokens. Some treatment might be needed in the keywords, like escaping special characters and sorting by tokens and class keywords.

```
fun fromGrammar(grammar: Grammar, fileExtensions: Iterable<String>):
    GrammarInfo {
    val name = nameFromGrammar(grammar)
    val keywords = keywordsFromGrammar(grammar)
    val operations = operationsFromGrammar(grammar)
    val comment = Comment.fromGrammar(grammar)

    return GrammarInfo(
        name,
        fileExtensions,
        keywords,
        operations,
        Lists.newArrayList(),
        comment
    )
}
```

Listing 5.6: Loading Information from Grammar

After the information is loaded, it is passes into a template that generates a TextMate json file. The template library used is **Apache Velocity** [1]. Apache Velocity allows the use of templates and, given a context, it produces a String. The Listings 5.7 and 5.8 shown an example of this library is used.

```
...
"language_keyword": {
    "patterns": [
        #foreach( $keyword in $grammar.keywords )
            {
                "match": "$keyword.match",
                "name": "$keyword.type.value"
            }
            #if ($foreach.count != $grammar.keywords.size())
                ,
            #end
        #end
    ]
},
"language_operation": {
    "patterns": [
        #foreach( $operation in $grammar.operations )
            {
                "match": "$operation.match",
                "name": "$operation.type"
            }
            #if ($foreach.count != $grammar.operations.size())
                ,
            #end
        #end
    ]
}
...
```

Listing 5.7: Velocity Template Snippet

[1] `https://velocity.apache.org/`

```
fun create(grammar: Grammar, fileExtensions: Iterable<String>): String {
    val grammarInfo = GrammarInfo.fromGrammar(grammar, fileExtensions)

    // Preparing the context for the template.
    val context = VelocityContext()
    context.put(GRAMMAR_TEMPLATE_VARIABLE, grammarInfo)

    // Read template.
    val template = ve.getTemplate(TEMPLATE_PATH)

    // Executing the template with the context.
    val writer = StringWriter()
    template.merge(context, writer)

    return Utils.prettifyTextMateResult(writer.toString())
}
```

Listing 5.8: Velocity Usage

5.5 Vscode Module

This module creates a Visual Studio Code extension development workspace. This workspace is a node project. It also contains all the configurations for the language extension.

The workspace contains the following files:

- **.vscode/launch.json** - The launch configuration of the extension.
- **language-configuration-json** - Contains the information for autocompletion and validation.
- **syntaxes/language.tmLanguage.json** - The TextMate file for the DSL highlighting.
- **package.json** - Contains the language name, file extensions, node module dependencies, etc.
- **bin/language-server.jar** - The Xtext language server of the DSL.
- **src/extension.js** - The script to start and communicate with the language server.

```
extension
├─ .vscode
│  └─ launch.json
├─ bin
│  └─ language-server.jar
├─ src
│  └─ extension.js
├─ syntaxes
│  └─ language.tmLanguage.json
├─ .vscodeignore
├─ language-configuration.json
└─ package.json
```

Figure 5.3: Extension Structure Tree

5.6 Controller Module

The Controller Module is an application that has a Command-line interface (CLI). The **picocli** [2] library is used to simplify application development. It contains two commands, one for each functional requirement.

5.6.1 Functional Requirements 1

The command to fulfill this requirement allows the creation of an Xtext project. The following Listing 5.9 shows documentation of how the command works.

```
Usage: xtext [-o=<outDir>] <genmodel> <packageName> <rootElement>
                     <languageName> <baseName> [<extensions>[,<extensions>...]]
Creates an xtext project given its configuration
      <genmodel>          The genmodel file with the metamodel.
      <packageName>       The default package.
      <rootElement>       The root element.
      <languageName>      The language name.
                          The last name of the language as to be capitalized.
                          Example: org.bamol.Dsl
      <baseName>          The base name of the language.
                          Example: statemachine
      [<extensions>[,<extensions>...]]
                          The file extensions of the dsl.
  -o, --outDir=<outDir>   The path to output the xtext project created.
                          By default is the current path.
```

Listing 5.9: Xtext Command usage

[2] https://picocli.info/

The diagram in Figure 5.4 shows a part of the implementation of this functional requirement.

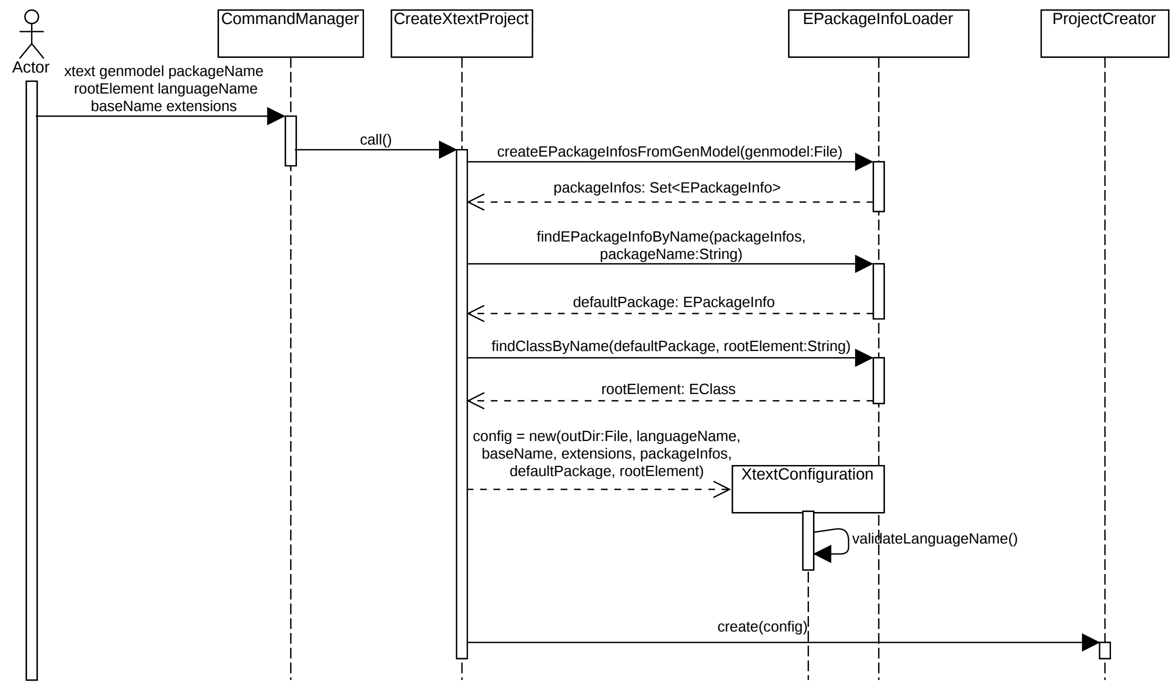

Actor
CommandManager
CreateXtextProject
EPackageInfoLoader
ProjectCreator
xtext genmodel packageName rootElement languageName baseName extensions
call()
createEPackageInfosFromGenModel(genmodel:File)
packageInfos: Set<EPackageInfo>
findEPackageInfoByName(packageInfos, packageName:String)
defaultPackage: EPackageInfo
findClassByName(defaultPackage, rootElement:String)
rootElement: EClass
config = new(outDir:File, languageName, baseName, extensions, packageInfos, defaultPackage, rootElement)
XtextConfiguration
validateLanguageName()
create(config)

Figure 5.4: Code Process View for FR1

5.6.2 Functional Requirements 2

The command to fulfill this requirement allows the creation of a Visual Studio Code extension with the DSL information. The following Listing 5.10 shows documentation of how the command works.

```
Usage: vscode [-o=<outDir>] <grammar> <languageServer> [<extensions>[,
                         <extensions>...]]
Creates an vscode extension for a xtext language.
      <grammar>            The grammar file of the language.
                           This is a .xtext file.
      <languageServer>     The language server jar file created from
    xtext.
      [<extensions>[,<extensions>...]]
                           The file extensions of the dsl.
  -o, --outDir=<outDir>    The path to output the xtext project created.
                           By default it is a extension folder in the
    current
                             path.
```

Listing 5.10: Vscode Command usage

The diagram in Figure 5.5 shows a part of the implementation of this functional requirement.

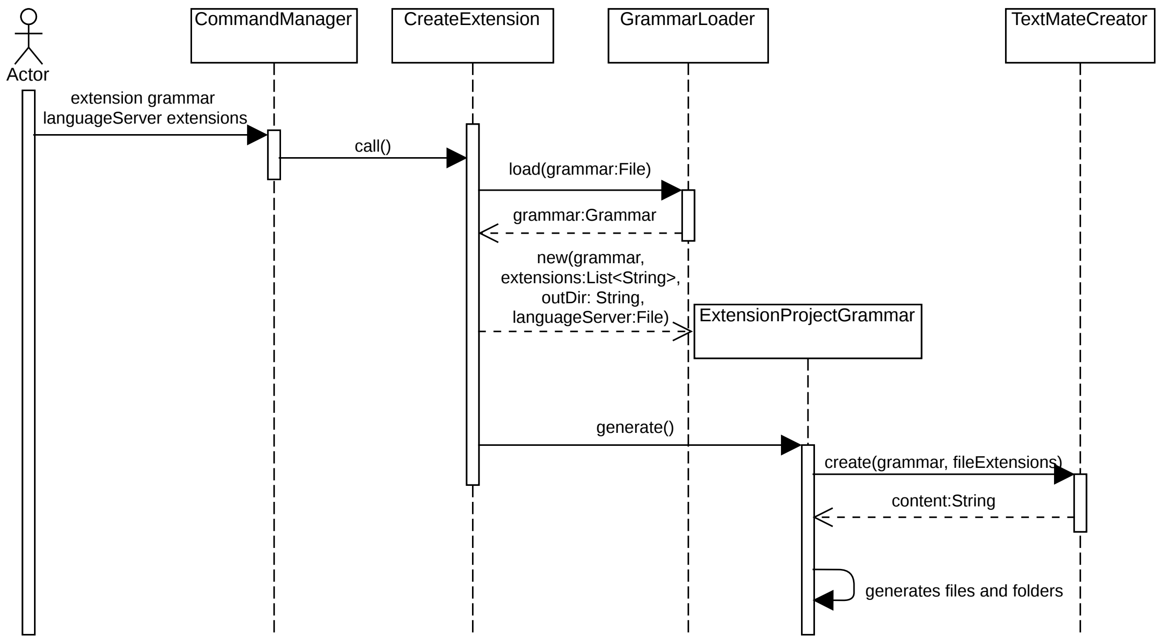

Actor
CommandManager
CreateExtension
GrammarLoader
TextMateCreator
extension grammar languageServer extensions
call()
load(grammar:File)
grammar:Grammar
new(grammar, extensions:List<String>, outDir: String, languageServer:File)
ExtensionProjectGrammar
generate()
create(grammar, fileExtensions)
content:String
generates files and folders

Figure 5.5: Code Process View for FR2

5.6.3 Summary

In this section, implementation details of the designed solution were presented. First, an overview of the developed system modules was provided, along with their intrinsic aspects. The following section will focus on the evaluation of the implementation.

Chapter 6

Experiments

This chapter describes the experiences and evaluation carried out on the implemented solution. Initially, it is shown an explanation of the evaluation methodology used to test the solution, followed by the results.

6.1 Hypothesis

A hypothesis is an assumption that serves as a starting point for further investigation. The following hypothesis was identified, as indicated before in the first chapter (section 1.4):

- **H1** - The proposed approach provides an automated and effective way of producing a language server implementation and complete Visual Studio extension based solely on the abstract syntax (i.e., meta-model) of the domain-specific language.

6.2 Assessment Methodology

For the evaluation, two methodologies are used:

- **Quantitative Evaluation Framework (QEF)** for Quality Assessment.
- **Comparison Testing** for Functionality Assessment.

6.2.1 QEF

Quantitative Evaluation Framework (QEF) evaluates how close the intended solution is to the initially set objectives, allowing the development team to focus on fixing those flaws and directing the product to meet the desired criteria. (Escudeiro and Bidarra 2008)

In this dissertation, QEF uses all functional and non-functional requirements of the project. As a result, the solution can be evaluated using predefined criteria that measure the overall completion of each requirement.

6.2.2 Comparison Testing

Comparison testing refers to a type of testing where the strength and weaknesses of the currently developed software produced are compared with already existing software products in the market. It helps to assess how the current software product performs against the market competition along with this, the comparison testing helps for the development of a high-quality software product with improved performance and functionality. (*Comparison Testing in Software Engineering* 2021)

For this project, the approach for language deployment to Visual Studio Code is compared against the Xtext-generated Eclipse plugins. This feature of creating plugins for Eclipse is made by Xtext developers.

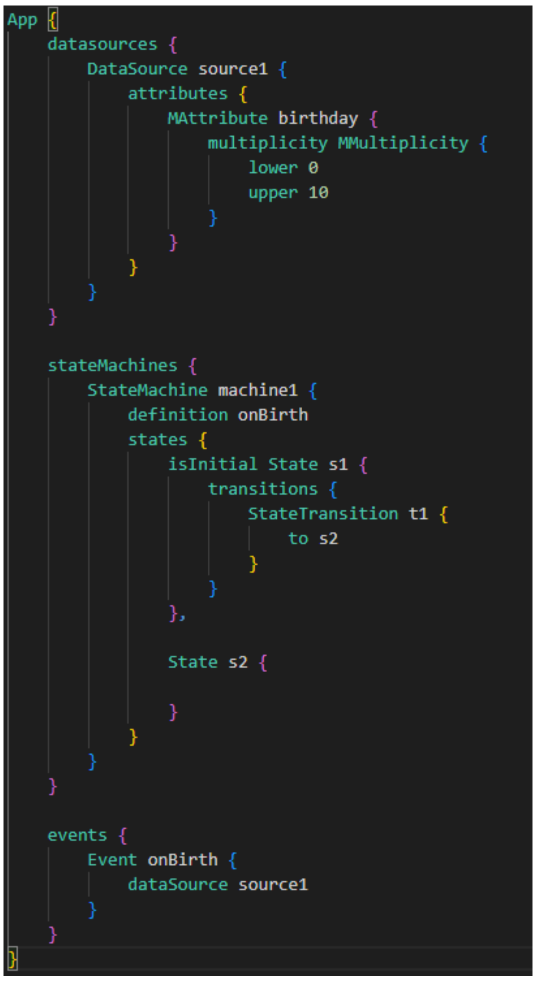
6.3 Results

The project source code and all the extensions created are in an open-source GitHub project[1].

This section is responsible for presenting the system's quality assessment. Firstly it is shown an example of an extension for the Omnia meta-model. Then this project solution is compared to the Xtext Eclipse plugins using comparison testing. The results of the tests are compiled and used to validate the Functionalities Dimension in QEF.

6.3.1 Omnia extension

Using the Omnia ecore metamodel, the grammar and the extension are generated. Figure 6.1 shows an example of using the DSL. Note that the contents are only for demonstration purposes.



```
App {
    datasources {
        DataSource source1 {
            attributes {
                MAttribute birthday {
                    multiplicity MMultiplicity {
                        lower 0
                        upper 10
                    }
                }
            }
        }
    }

    stateMachines {
        StateMachine machine1 {
            definition onBirth
            states {
                isInitial State s1 {
                    transitions {
                        StateTransition t1 {
                            to s2
                        }
                    }
                },

                State s2 {

                }
            }
        }
    }

    events {
        Event onBirth {
            dataSource source1
        }
    }
}
```

Figure 6.1: Omnia Extension usage example

[1]https://github.com/Lzgpom/xtext-gen-extension

6.3.2 Xtext Eclipse Plugin Comparison Test Results

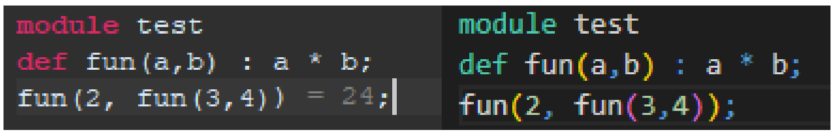
As previously stated, comparison testing is going to be used to compare the Visual Studio Code extensions generated by this project with the Xtext Eclipse plugins. This comparison test was conducted to evaluate the hypothesis **H1**. To create the extensions the application was used by giving the grammar of each language.

Multiple DSLs were used to compare both solutions. The DSLs are the following:

- **Arithmetics**[2] - Allows the creation of formulas and calculates the results.
- **Domain Model**[3] - Allows the creation of a domain model with properties and functions with Java classes.
- **State Machine**[4] - Represents a statemachine.
- **Home Automation**[5] - Represents some devices and some operations that can have a trigger.

Arithmetics

Figure 6.2 shows a snippet of the Arithmetics DSL in the Eclipse plugin on the left and the same snippet in the Visual Studio Code extension on the right.



```
module test
def fun(a,b) : a * b;
fun(2, fun(3,4)) = 24;
```

```
module test
def fun(a,b) : a * b;
fun(2, fun(3,4));
```

Figure 6.2: Arithmetics DSL Comparison

One significant disadvantage of this project solution, the Visual Studio Code extension, is that it cannot display extra information. In this example, the plugin can execute the calculation and display the result, whereas the extension cannot.

Other than that, both behave similarly.

Domain Model

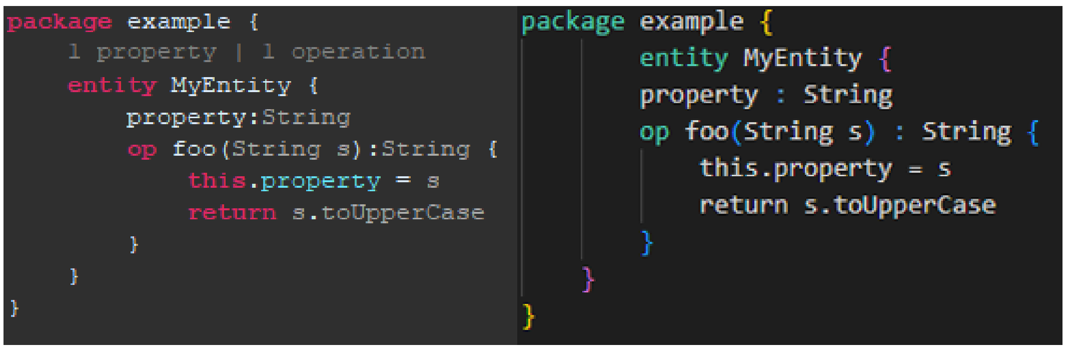
Figure 6.3 shows a snippet of the Domain Model DSL in the Eclipse plugin on the left and the same snippet in the Visual Studio Code extension on the right.

[2] https://github.com/eclipse/xtext-eclipse/tree/master/org.eclipse.xtext.xtext.ui.examples/projects/arithmetics

[3] https://github.com/eclipse/xtext-eclipse/tree/master/org.eclipse.xtext.xtext.ui.examples/projects/domainmodel

[4] https://github.com/eclipse/xtext-eclipse/tree/master/org.eclipse.xtext.xtext.ui.examples/projects/fowlerdsl

[5] https://github.com/eclipse/xtext-eclipse/tree/master/org.eclipse.xtext.xtext.ui.examples/projects/homeautomation



```
package example {
    1 property | 1 operation
    entity MyEntity {
        property:String
        op foo(String s):String {
            this.property = s
            return s.toUpperCase
        }
    }
}
```

```
package example {
    entity MyEntity {
        property : String
        op foo(String s) : String {
            this.property = s
            return s.toUpperCase
        }
    }
}
```

Figure 6.3: Domain Model DSL Comparison

Like in the Arithmetics DSL, the extension cannot display extra information. In this case, the plugin shows the number of properties and operations an entity has.

An advantage of the plugin is that the String class, which is a java class, is highlighted differently. Also, when attempting to go to the class definition it shows the Java class definition, whereas in the extension this is not possible.

Other than that, both behave similarly.

State Machine

Figure 6.4 shows a snippet of the State Machine DSL in the Eclipse plugin on the left and the same snippet in the Visual Studio Code extension on the right.

In this DSL, both behave similarly.

Home Automation

Figure 6.5 shows a snippet of the Home Automation DSL in the Eclipse plugin on the left and the same snippet in the Visual Studio Code extension on the right.

Like in the State Machine DSL, the extension and plugin for this DSL behave similarly.

Summary

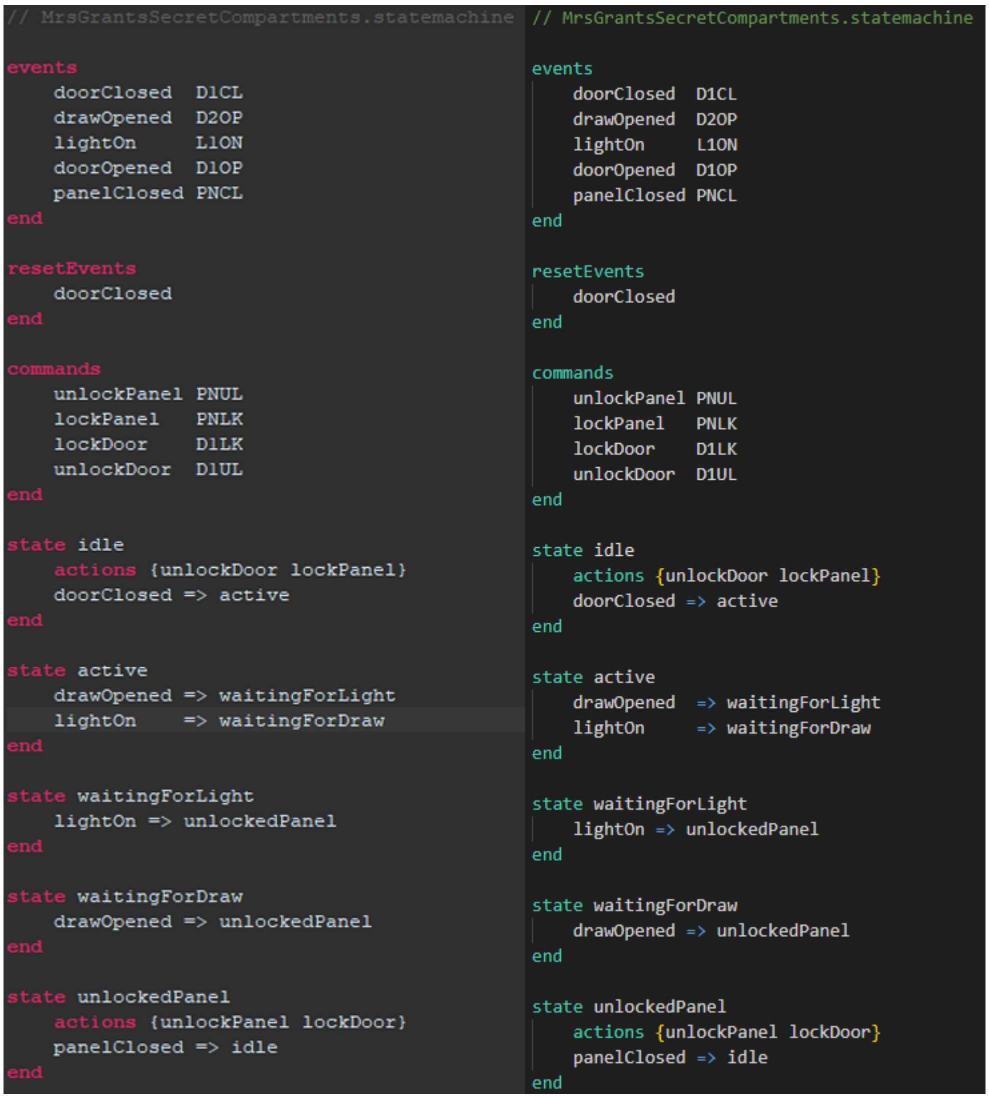
Given all the previous comparisons, it can be concluded that the solution developed, the extension, does not offer as many features as the Xtext Eclipse plugin. This is due to the limitations of the language server protocol, shown in Table 2.1.

Nevertheless, the extension is pretty complete, and the grade from 0 to 100% is **75%**.

6.3.3 Assessment Completion

The solution is assessed using QEF in Table 6.4, which depicts the primary dimensions:

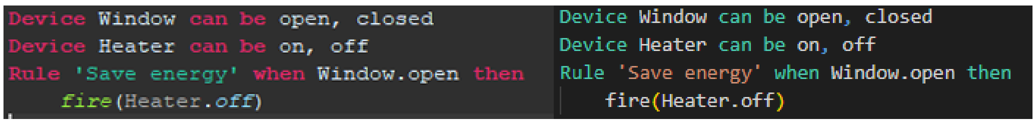
- **Functionality**: includes all the functional requirements. Table 6.1 shows the scales used for the functional requirements.
- **Usability**: includes all the non-functional requirements for usability, more precisely for the User Interface. Table 6.2 shows the scales used for the usability.



```
// MrsGrantsSecretCompartments.statemachine

events
    doorClosed  D1CL
    drawOpened  D2OP
    lightOn     L1ON
    doorOpened  D1OP
    panelClosed PNCL
end

resetEvents
    doorClosed
end

commands
    unlockPanel PNUL
    lockPanel   PNLK
    lockDoor    D1LK
    unlockDoor  D1UL
end

state idle
    actions {unlockDoor lockPanel}
    doorClosed => active
end

state active
    drawOpened => waitingForLight
    lightOn    => waitingForDraw
end

state waitingForLight
    lightOn => unlockedPanel
end

state waitingForDraw
    drawOpened => unlockedPanel
end

state unlockedPanel
    actions {unlockPanel lockDoor}
    panelClosed => idle
end
```

```
// MrsGrantsSecretCompartments.statemachine

events
    doorClosed  D1CL
    drawOpened  D2OP
    lightOn     L1ON
    doorOpened  D1OP
    panelClosed PNCL
end

resetEvents
    doorClosed
end

commands
    unlockPanel PNUL
    lockPanel   PNLK
    lockDoor    D1LK
    unlockDoor  D1UL
end

state idle
    actions {unlockDoor lockPanel}
    doorClosed => active
end

state active
    drawOpened   => waitingForLight
    lightOn      => waitingForDraw
end

state waitingForLight
    lightOn => unlockedPanel
end

state waitingForDraw
    drawOpened => unlockedPanel
end

state unlockedPanel
    actions {unlockPanel lockDoor}
    panelClosed => idle
end
```

Figure 6.4: State Machine DSL Comparison



```
Device Window can be open, closed
Device Heater can be on, off
Rule 'Save energy' when Window.open then
    fire(Heater.off)
```

```
Device Window can be open, closed
Device Heater can be on, off
Rule 'Save energy' when Window.open then
    fire(Heater.off)
```

Figure 6.5: Home Automation DSL Comparison

- **Suportability**: includes all the non-functional requirements for supportability, more precisely for Maintenance and Compatability. Table 6.3 shows the scales used for the supportability.

Table 6.1: QEF Functionality Scale

		Wfk - Fullfilment (%)				
Requirement	**Metric Evaluation**	**0**	**25**	**50**	**75**	**100**
FR1 - Create the grammar given the meta-model and the root object	User can create an xtext project.	No access to functionality	-	-	-	Full access to functionality
FR2 - Deploy the language created to Visual Studio Code	User can create an visual studio code extension for the language.	Based on the comparison test	Based on the comparison test	Based on the comparison test	Based on the comparison test	Based on the comparison test

Table 6.2: QEF Usability Scale

		Wfk - Fullfilment (%)		
Requirement	**Metric Evaluation**	**0**	**50**	**100**
NFR1 - The extension has to have keyword and error highlighting.	The extension has the features.	No	-	Yes
NFR2 - The has to have some features such as: formatting, quick fix, auto-completion and, go to declaration.	The extension has the features.	No	-	Yes

Table 6.3: QEF Supportability Scale

		Wfk - Fullfilment (%)		
Requirement	**Metric Evaluation**	**0**	**50**	**100**
NFR3 - The visual studio code extension should be tested.	How much the extension is tested.	No tests	Some tests	Fully tested
NFR4 - The grammar generation should support ecore meta-models as input.	The level of support of ecore meta-models.	No support	Support to single file ecore models	Full support

The ratings and results of QEF are shown in Table 6.4.

Table 6.4: QEF Assessment

q	**D**	Qi	**Dimension**	Q_j	W_{ij}	**Factor**	rw_{jk}	**Requirement**	w_{fk} %
90%	0.28	87.5	Functionality	87.5	1.00	Functional	10	**FR1** - Create the grammar given the meta-model and the root object	100
							10	**FR2** - Deploy the language created to Visual Studio Code	75
		100	Usability	100	1.00	User Interface	10	**NFR1** - The extension has to have keyword and error highlighting.	100
							8	**NFR2** - The extension has to have some features such as: formatting, quick fix, auto-completion and, go to declaration	100
		75	Supportability	100	0.50	Maintenance	8	**NFR3** - The visual studio code extension should be tested.	100
				50	0.50	Compatibility	10	**NFR4** - The grammar generation should support ecore meta-models as input.	50

Throughout the QEF, six requirements were individually reviewed, using each own rating system.

Only one Functional requirement was completed receiving a **75%** score. This is justified in the section 6.3.2. **NFR4** was another criterion that was not fully met in Supportability. This is due to the solution's inability to read multi-file ecore models, only single-file ecore models. For this reason, it received a **50%** score.

When all of the assessed dimensions are considered, the final system score is **90%** of the ideal solution. This score answers the hypothesis raised in (section 6.1).

Chapter 7

Conclusion

This chapter will go over all of the conclusions from the investigation. Its goal is to connect previously established objectives with accomplishments. This chapter will demonstrate the goals achieved. Second, it will take into account the system's limitations as well as future work. Finally, there will be a section for some personal remarks ending the conclusion.

7.1 Goals Achieved

The main goal of this work was to build an application to automatically generate and deploy domain-specific languages to an Integrated Development Environment. This goal was achieved and the solution is available for the community in an open-source repository[1].

The results were great as the application can create a language and deploy it to Visual Studio Code. Another good point of the application is that it is not for a specific project as it can work with any ecore meta-model.

The application might be handy if the user wants to quickly deploy a language to Visual Studio Code. Or the automatically generated Visual Studio Code extension can be the basis for a more complex extension.

7.2 Limitations and Future Work

The generated language extensions were simple, and it would be interesting to add some features, including a live reload of grammar, finding all references, and code folding, among others. Adding those features would take a significant amount of time, which was out of the time scope available for this work.

7.3 Personal Remarks

This project allowed the author to investigate some of the main concepts of the master's degree in software engineering, namely MDE.

This project also provided a challenge, learning how to reverse engineer, which happened when understanding how the Eclipse IDE uses Xtext.

Furthermore, because MDE is a popular software development methodology, it was an excellent opportunity to learn more about it and acquire relevant skills from a professional standpoint.

[1]`https://github.com/Lzgpom/xtext-gen-extension`

References

ANTRL (2022). https://www.antlr.org/. Accessed: 2021-12-16.

Bragança, Alexandre et al. (2021). "Towards supporting SPL engineering in low-code platforms using a DSL approach". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, pp. 16–28.

Brambilla, Marco, Jordi Cabot, and Manuel Wimmer (2017). "Model-driven software engineering in practice". In: *Synthesis lectures on software engineering* 3.1, pp. 1–207.

Comparison Testing in Software Engineering (2021). https://www.geeksforgeeks.org/comparison-testing-in-software-engineering. Accessed: 2022-06-01.

Creating Language Servers for Visual Studio Code (2022). https://code.visualstudio.com/docs/extensions/example-language-server. Accessed: 2021-12-18.

d'Anjou, Jim, Scott Fairbrother, and Dan Kehn (2005). *The Java developer's guide to Eclipse*. Addison-Wesley Professional.

EMF.cloud Project (2022). https://www.eclipse.org/emfcloud/. Accessed: 2021-11-30.

Escudeiro, Paula and José Bidarra (2008). "Quantitative evaluation framework (QEF)". In: *Conselho Editorial/Consejo Editorial* 16.

Foundation, Eclipse (2022). *Eclipse IDE*. https://www.eclipse.org/ide/. Accessed: 2021-12-16.

Fowler, Martin (2005). *Language workbenches: The killer-app for domain specific languages*. Accessed: 2022-02-22.

Fulton, Randall and Roy Vandermolen (2017). *Airborne Electronic Hardware Design Assurance: A Practitioner's Guide to RTCA/DO-254*. CRC Press.

GEF (2022). https://www.eclipse.org/gef/. Accessed: 2022-06-10.

Graphiti (2022). https://www.eclipse.org/graphiti/. Accessed: 2022-06-10.

Hassan, Almoatazbillah (2012). "The value proposition concept in marketing: How customers perceive the value delivered by firms-A study of customer perspectives on supermarkets in Southampton in the United Kingdom". In: *International journal of marketing studies* 4.3, p. 68.

Koen, Peter A et al. (2002). "Fuzzy front end: effective methods, tools, and techniques". In: *The PDMA toolbook* 1, pp. 5–35.

Kruchten, Philippe B (1995). "The 4+ 1 view model of architecture". In: *IEEE software* 12.6, pp. 42–50.

Language Grammars (2022). https://macromates.com/manual/en/language_grammars. Accessed: 2021-12-16.

Language Server Protocol (2022). https://microsoft.github.io/language-server-protocol/. Accessed: 2021-12-18.

Lardinois, Frederic (2015). "Microsoft launches visual studio code, a free cross-platform code editor for os x, linux and windows". In: *United State: TechCrunch*.

Loucopoulos, Pericles (2005). "Requirements engineering". In: *Design process improvement*. Springer, pp. 116–139.

Meuter, Wolfgang De (2022). *Concrete vs. Abstract Syntax*. `http://pico.vub.ac.be/mc/absconc.html`. Accessed: 2021-11-22.

Mohagheghi, Parastoo et al. (2008). “MDE adoption in industry: challenges and success criteria”. In: *International Conference on Model Driven Engineering Languages and Systems*. Springer, pp. 54–59.

Osterwalder, Alexander and Yves Pigneur (2003). “Modeling value propositions in e-Business”. In: *Proceedings of the 5th international conference on Electronic commerce*, pp. 429–436.

Osterwalder, Alexander, Yves Pigneur, et al. (2015). *Value proposition design: How to create products and services customers want*. Vol. 2. John Wiley & Sons.

Peffers, Ken et al. (2007). “A design science research methodology for information systems research”. In: *Journal of management information systems* 24.3, pp. 45–77.

Richardson, Clay et al. (2014). “New development platforms emerge for customer-facing applications”. In: *Forrester: Cambridge, MA, USA* 15.

Saaty, Thomas L (1990). “How to make a decision: the analytic hierarchy process”. In: *European journal of operational research* 48.1, pp. 9–26.

Schmidt, Douglas C (2006). “Model-driven engineering”. In: *Computer-IEEE Computer Society-* 39.2, p. 25.

Sevgili Koçak, Seda and MEHMET GÜVEN (2020). “Aile içi iletişimi geliştirmeye yönelik grupla psikolojik danışmanın bireylerin iletişim beceri düzeylerine etkisi”. In.

Shaha, Manali and Meenakshi Pawar (2018). “Transfer learning for image classification”. In: *2018 second international conference on electronics, communication and aerospace technology (ICECA)*. IEEE, pp. 656–660.

Sirius (2022). `https://www.eclipse.org/sirius/`. Accessed: 2022-06-10.

Stack Overflow Developer Survey 2021 - Integrated Development Environment (2022). `https://insights.stackoverflow.com/survey/2021`. Accessed: 2021-12-16.

Syntax Highlight Guide (2022). `https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide`. Accessed: 2021-12-16.

textX (2022). `http://textx.github.io/textX/3.0/`. Accessed: 2021-11-30.

textX for VS Code (2022). `https://github.com/textX/textX-LS/tree/master/client`. Accessed: 2021-12-13.

The C4 model for visualising software architecture (2022). `https://c4model.com/`.

Theia (2022). `https://theia-ide.org/`. Accessed: 2021-12-18.

Vaidya, Omkarprasad S and Sushil Kumar (2006). “Analytic hierarchy process: An overview of applications”. In: *European Journal of operational research* 169.1, pp. 1–29.

Visual Studio Code (2022). `https://code.visualstudio.com/`. Accessed: 2021-12-16.

What is a Generation Languages? (2022). `https://www.computerhope.com/jargon/num/1gl.htm`. Accessed: 2021-12-16.

What is an IDE? (2022). `https://www.redhat.com/en/topics/middleware/what-is-id`. Accessed: 2021-12-16.

Xtext (2022). `https://www.eclipse.org/Xtext/`. Accessed: 2021-12-25.

Xtext Generator Fragment for Visual Studio Code Extensions (2022). `https://github.com/itemis/xtext-generator-vscode`. Accessed: 2021-12-14.