INSTITUTE FOR
SOFTWARE

OST
Eastern Switzerland
University of Applied Sciences

Bringing Context Mapper to the Developer's Workflow

Enhanced IDE Integration and Tooling Support

Department of Computer Science
OST - Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Bachelor Thesis, Spring Term 2025

Author: Lukas Streckeisen
Advisor: Stefan Kapferer
Project Partner: IFS Institute for Software
External Co-Examiner: Roman Blum
Internal Co-Examiner: Prof. Dr. Farhad D. Mehta

Abstract

Context Mapper provides a Domain-Specific Language (DSL) for modelling software systems using Domain-Driven Design patterns. The Context Mapper DSL (CML) language supports patterns from strategic and tactic DDD, as well as Value-Driven Analysis and Design. Context Mapper currently offers an Eclipse and VSCode plugin. IntelliJ, a popular IDE among Java developers, is not yet supported, potentially preventing Context Mapper's widespread adoption. This thesis aims to enhance the developer's workflow by developing a proof of concept for a Context Mapper IntelliJ plugin and outline a path for the plugin to be extended to full functionality. To achieve this goal, this thesis provides an overview of current language workbenches (frameworks for creating DSLs) and options for integrating DSLs in IntelliJ. From these technologies, the thesis evaluates the most suited technology to develop the proof of concept (PoC). The implemented plugin uses LSP4IJ, an open-source IntelliJ plugin based on the Language Server Protocol, and Langium, a TypeScript DSL framework. The PoC successfully implemented important editor features, such as syntax highlighting, hyperlinking, autocomplete and a PlantUML component diagram generator. Future work includes providing a Java library for reading and writing CML models, so Context Mapper's existing Java tools can be migrated as well.

Keywords: Context Mapper, Domain-Specific Language, Language Server, IntelliJ, Editor Support

Management Summary

Introduction

Context Mapper provides a Domain-Specific Language called Context Mapping Language (CML) for modelling software systems on the basis of Domain-Driven Design patterns. The language supports patterns from strategic and tactic Domain-Driven Design, as well as Value-Driven Analysis and Design. The current implementation of Context Mapper is based on Xtext, which is a Java-based framework for creating Domain-Specific Languages. With Xtext, Context Mapper offers an Eclipse plugin out of the box. Additionally, based on the language server that comes with Xtext, it also provides an extension for VSCode.

IntelliJ has become a popular development environment among Java developers, for which Context Mapper does not yet offer editor support. This situation requires developers to use two development environments, which potentially prevents the widespread adoption of Context Mapper.

Objective

The goal of this thesis is to enhance the workflow for developers using IntelliJ, by developing a proof of concept for a Context Mapper IntelliJ plugin. To achieve this goal, this thesis provides an overview of current language workbenches (frameworks for creating Domain-Specific Languages) and options for integrating languages in IntelliJ. From these technologies, the thesis selected the most suited technology to develop the proof of concept. For the not-implemented features, the thesis outlines how these features can be implemented in a future project.

Results

The technology analysis covered three options each for language workbenches and IntelliJ integration options. The covered language workbenches are JetBrains MPS, Langium and Rascal. Based on a utility analysis, Langium, a TypeScript framework, proved to be the most suitable, capable and stable workbench for Context Mapper. The analysed integration options are a language server integration via IntelliJ's LSP (Language Server Protocol) support, a language server integration via the LSP4IJ IntelliJ plugin, and native integration. Since a language server is the most suitable approach from an architectural perspective, the language server integration options received high scores in the utility analysis. In the end, LSP4IJ received the highest score due to its availability in both IntelliJ Community and Ultimate versions, as well as its superior set of supported LSP capabilities.

The proof of concept was implemented using both Langium and LSP4IJ. Figure 1 shows the resulting architecture.

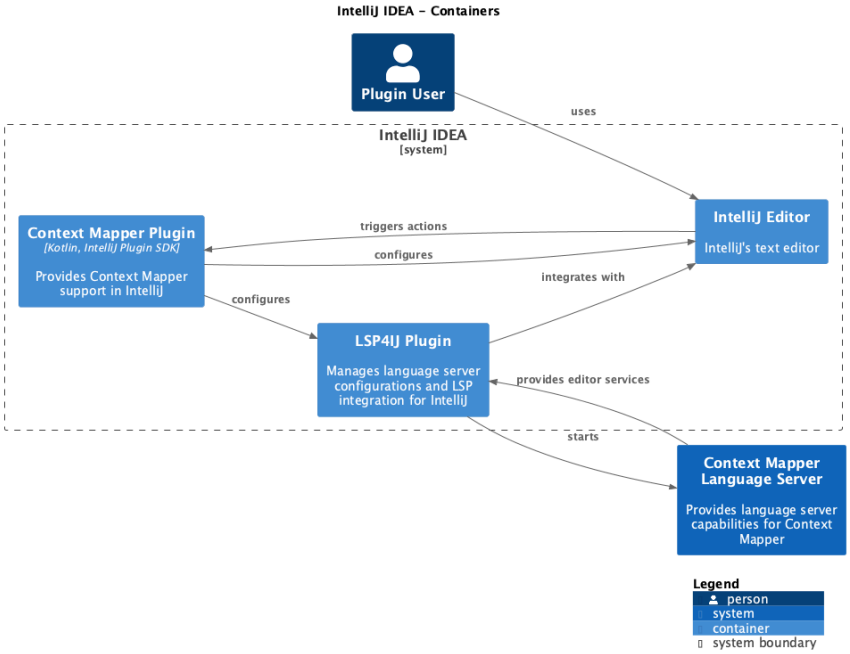

Figure 1: C4 container diagram of the developed proof of concept

With LSP4IJ in charge of interactions between the language server and IntelliJ, the Context Mapper plugin limits itself to configuring LSP4IJ. All feature logic is placed in the language server, which gives Context Mapper the flexibility to target other development environments at a later point.

The proof of concept successfully implemented most of the selected subset of Context Mapper features. Figure 2 shows the Context Mapper IntelliJ editor and a generated PlantUML component diagram.

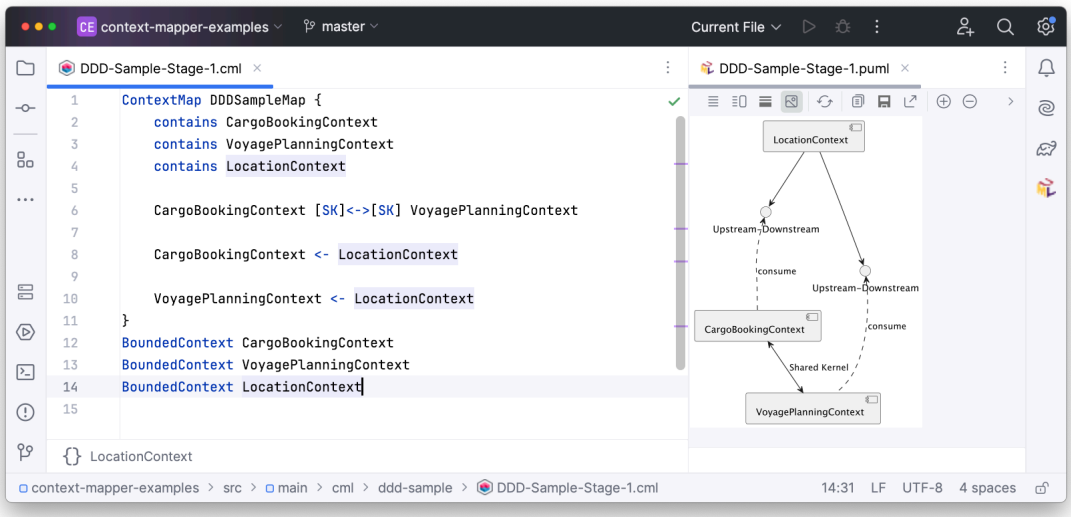

Figure 2: Screenshot of the CML editor in IntelliJ

Future work includes providing a Java library for reading and writing CML models, so Context Mapper's existing Java tools can be migrated as well.

Acknowledgements

I would like to thank Stefan Kapferer for his guidance during my thesis and for providing valuable feedback on my work. I am also grateful to Jann Flepp for sharing his experiences from his own thesis. Furthermore, I would like to thank the maintainers of LSP4IJ and Langium for their prompt and helpful responses to my questions. Finally, I sincerely thank my family and friends for their unwavering support.

Table of Contents

Part I - Technical Report

1. Introduction

This section describes the context (Section 1.1), motivation (Section 1.2) and goals (Section 1.3) of this thesis. The thesis context describes Context Mapper itself and frameworks relevant to Context Mapper. The motivation and goals are based on the thesis task description, which can be found in full in Appendix A.

1.1. Project Context

Context Mapper [1] provides a DSL (Domain Specific Language) and tools to model software systems on the basis of DDD (Domain-Driven Design) patterns. It was mainly created to support strategic DDD patterns like Context Mapping, but also has elements for tactic DDD like Domain modelling. The CML (Context Mapping Language) also offers support for process flows, user requirements, stakeholders, and values from Value-Driven Analysis & Design[1]. For editor support, Context Mapper offers plugin extensions for Eclipse[2] and VSCode[3]. These extensions include the CML editor (syntax highlighting, autocomplete, etc.), as well as the capabilities for story splitting[4] and architectural refactorings[5]. In addition, the extensions include generators that convert the defined contexts to, e.g. a PlantUML diagram.

Context Mapper also offers additional tools, such as a discovery library to generate a CML model from an existing project. However, these tools are out of scope for this thesis.

Xtext (see Section 1.1.1) lays the foundation of the CML. Context Mapper heavily relies on Xtext for the CML parser, Eclipse plugin and LSP (Language Server Protocol) language server (see Section 1.1.2). The existing VSCode plugin leverages the language server generated by Xtext.

1.1.1. Xtext

Xtext [3], part of the Eclipse project, is a framework for creating DSLs. By defining a language grammar for the DSL, Xtext automatically generates a parser and text editor, among other resources. The text editor includes code completion, syntax highlighting, syntactic validation, hyperlinking and more. Xtext can also build an Eclipse plugin, which makes editor support for DSLs available to others. For integration with other IDE (Integrated Development Environment)s, Xtext supports the generation of LSP language servers.

In 2020, the core maintainers made a call to action [4] regarding the future maintenance of Xtext. The blog post outlines the decreasing numbers of active

[1] Process aiming to combine value-driven approaches with software engineering practices [2]
[2] https://eclipseide.org/
[3] https://code.visualstudio.com/
[4] https://socadk.github.io/design-practice-repository/activities/DPR-StorySplitting.html
[5] https://contextmapper.org/docs/architectural-refactorings/

contributors and the difficulties of keeping up with Eclipse releases. As of February 2024, Xtext is still being maintained; however, there is no indication that the situation has changed much.

1.1.2. Language Server Protocol

The purpose of LSP [5] is to enable language servers to integrate with different development tools without having to consider their implementation specifics. LSP defines a protocol for development tools to communicate with language servers via JSON-RPC (JavaScript Object Notation - Remote Procedure Call)[1]. When interacting with a language file in a development tool, the tool sends a request to the language server, which then replies with the information required for the tool to offer editor support. Figure 3 shows an example of how such an interaction could look like.

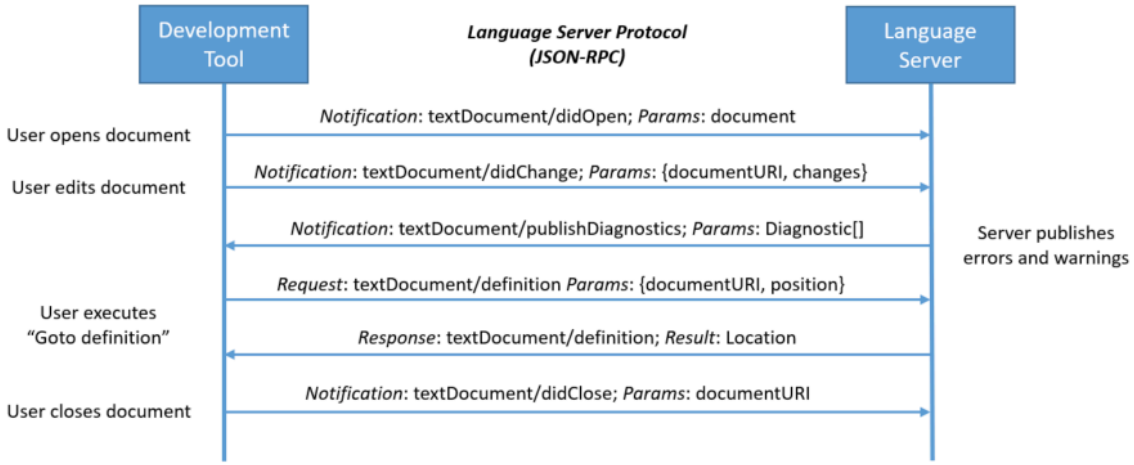

Figure 3: Example sequence between a language server and a development tool [6]

Supported language features are grouped into capabilities. Not every language server and development tool supports the same capabilities. During initialisation, the supported capabilities are negotiated. This capability negotiation allows language servers to be reused, but it also means that the editing experience for a language can differ from tool to tool.

1.2. Motivation

According to the 2024 StackOverflow developer survey [7], VSCode is used by 73.6%, IntelliJ by 26.8%, and Eclipse by 9.4% of developers. Currently, Context Mapper offers plugins for Eclipse and VSCode, so developers using IntelliJ either do not use Context Mapper or have to use a second IDE. This situation is inconvenient for IntelliJ developers and potentially hinders the widespread use of Context Mapper. Therefore, Context Mapper should offer an editor for IntelliJ, improving the developer's workflow. In light of Eclipse's low popularity, Context Mapper's editor support for Eclipse might be abandoned in the future.

[1] https://www.jsonrpc.org/specification

1.3. Goals

This thesis aims to create a PoC (Proof of Concept) IntelliJ plugin for Context Mapper. To do that, first, all available options for integrating Context Mapper's DSL into IntelliJ were analysed. After identifying the most suitable technology for Context Mapper, a PoC plugin was implemented. The timeframe available for this thesis was not sufficient to implement all features included in Context Mapper's Eclipse plugin. Instead, the thesis analyses possible challenges and limitations to implementing the remaining features in a future project.

2. System Analysis & Requirements

To define the requirements of this thesis, an inventory of Context Mapper's current features was made, which can be found in Section 2.1. Section 2.2 provides a more detailed description of the Context Mapper features selected for implementation in the PoC.

2.1. Existing System Analysis

The following subsections list the features available in Context Mapper [8], [9] as of February 2025. The list differentiates between supported concepts in the CML grammar and available editor features.

2.1.1. Context Mapper DSL

The CML supports concepts from DDD, VDAD (Value-Driven Analysis & Design), requirement analysis as well as application and process layers:

- **Context Map**: Definition of relationships between Bounded Contexts
- **Bounded Context**: Definition of Bounded Contexts
- **Bounded Context Relationships**: Definition of Bounded Context relationships, including Partnership, Shared Kernel, Customer/Supplier, Conformist, Open Host Service, Anticorruption Layer and Published Language
- **Domain & Subdomain**: Definition of Domains and Subdomains
- **Tactic DDD Modelling**: Definition of Domain Models using tactic DDD patterns, including Aggregate, Entity, Value Objects, Domain Elements, Commands, Services and Repositories
- **Application & Process Layer**: Definition of Application Layers, including process flows with Event and Command events
- **User Requirements**: Definition of Use Cases and User Stories
- **Stakeholder** (VDAD): Definition of project stakeholders
- **Value Register** (VDAD): Definition of stakeholder values

The CML is contained in `.cml` files, in which elements for the concepts above can be declared. CML also supports cross-references between files via `import` statements.

```
ContextMap DDDSampleMap {
  contains CargoBookingContext
  contains VoyagePlanningContext
  contains LocationContext

  CargoBookingContext [SK]<->[SK] VoyagePlanningContext

  CargoBookingContext [D]<-[U,OHS,PL] LocationContext

  VoyagePlanningContext [D]<-[U,OHS,PL] LocationContext

}
BoundedContext CargoBookingContext
BoundedContext VoyagePlanningContext
BoundedContext LocationContext
```

Listing 1: Example of a Context Map modelled in the CML

Listing 1 gives an example of a Context Map defined in the CML. The Context Map shown defines relationships with three Bounded Contexts using Shared-Kernel and

Upstream-Downstream relationships with the Open Host Service and Published Language pattern.

2.1.2. Use Cases

To give an overview of all features included in the current version of Context Mapper, all Use Cases were collected and displayed in Figure 4 and Figure 5. The Use Cases relevant to the PoC are displayed in Figure 4.

In previous work [1], three actors were identified for Context Mapper:

- Business Analyst / Domain Expert
- Software Engineer
- Software Architect

Since the DSL already exists and the thesis does not aim to change the available concepts, the actors are merged into one actor: Context Mapper User.

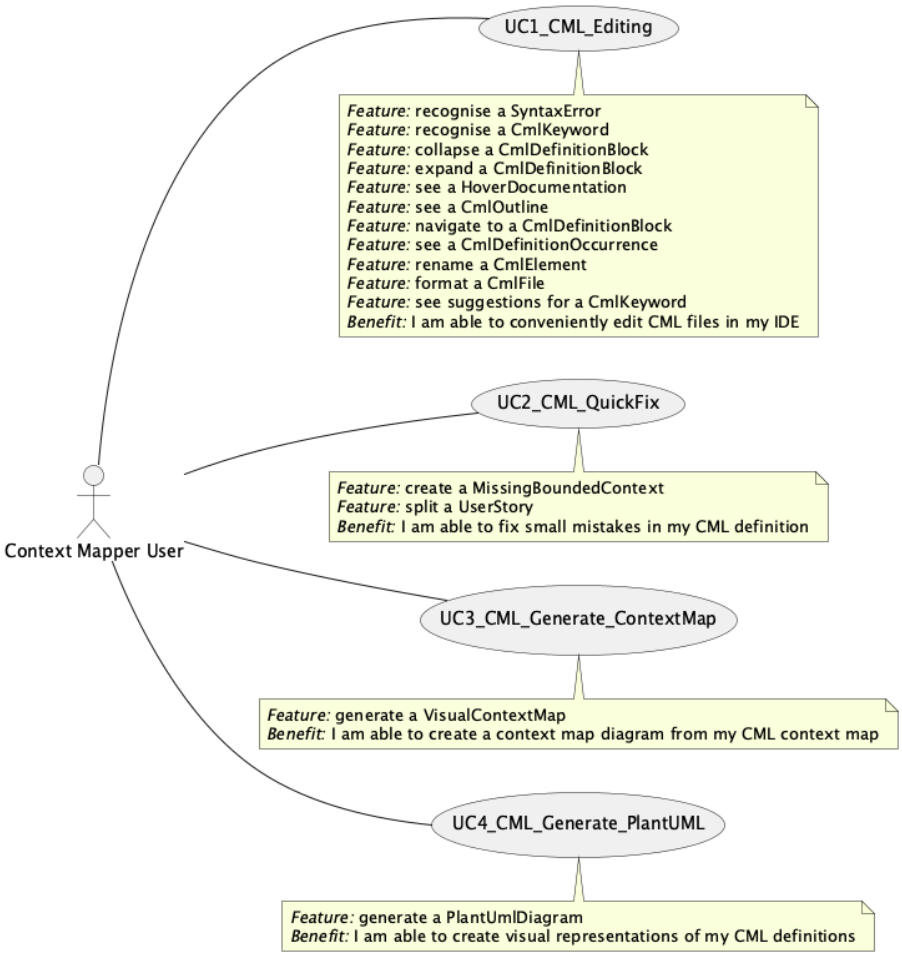

Figure 4: Context Mapper Use Cases (Part 1)

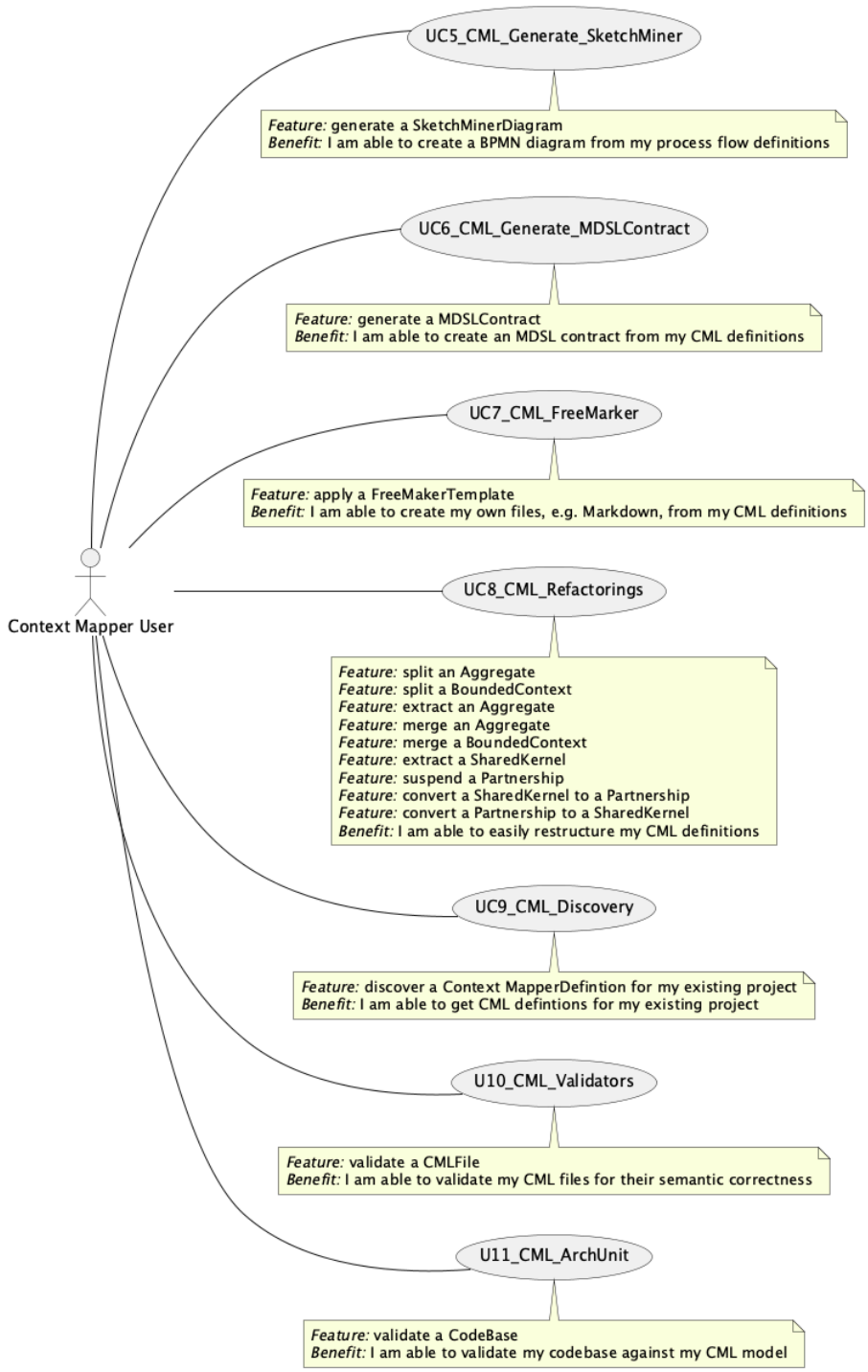

Figure 5: Context Mapper Use Cases (Part 2)

2.2. Requirements

The Use Cases from Figure 4 are documented as functional requirements for the PoC in Section 2.2.1. Additional non-functional requirements were defined and documented in Section 2.2.2.

2.2.1. Functional Requirements

Table 1 contains an overview of all the defined requirements, including their priority. More detailed requirement descriptions following the casual description format defined by Larman [10] can be found below.

Key	Summary	Priority
FR 1.1	Syntax highlighting - Visual distinction of CML keywords and syntax errors	High
FR 1.2	Hyperlinking - Navigation from a CML element usage to its definitions	High
FR 1.3	Occurrence highlighting - Occurrences of a selected CML element are highlighted in the same file	High
FR 1.4	Autocomplete - IDE suggests CML elements to complete existing fragments	High
FR 1.5	Code folding - Expanding and collapsing CML code blocks	Medium
FR 1.6	Keyword tooltips - Hover tooltip for CML keyword usages	Medium
FR 1.7	Structure outline - Concise overview of CML element definitions in a file	Medium
FR 1.8	Find usages - List of all usages of a CML element in the project	Medium
FR 1.9	Document formatting - “Reformat Code” action formats CML file	Medium
FR 2.1	Missing Bounded Context quick fix - Inline suggestion to create a Bounded Context if it is not defined yet	Medium
FR 3.1	Generate a visual Context Map	Medium
FR 1.10	Definition tooltips - Tooltip for CML element usages	Low
FR 4.1	Generate PlantUML diagrams	Low

Table 1: Functional Requirements for IntelliJ plugin PoC

FR 1.1 - Syntax Highlighting

Primary Actor	Context Mapper User
Goal	Ensure users can visually distinguish CML keywords and comments from element names.

Main Scenario

The IntelliJ CML editor distinguishes in colour between CML keywords, such as `BoundedContext`, and the name of a Bounded Context. The used colours follow the IntelliJ colour schemes, i.e. Dark, Light, etc.

FR 1.2 - Hyperlinking

Primary Actor	Context Mapper User
Goal	Users can easily navigate from a CML element usage to its definition.

Main Scenario

A Ctrl-/Cmd-Click with the mouse on the usage of a CML element, such as the name of a Bounded Context, should navigate the plugin user to the definition of that element in its CML file.

FR 1.3 - Occurrence Highlighting

Primary Actor	Context Mapper User
Goal	Users can visually detect other usages of the selected CML element.

Main Scenario

If the user places the caret within the name of a CML element, such as the name of a Bounded Context, the CML editor should highlight all other usages of that CML element within the same file.

FR 1.4 - Autocomplete

Primary Actor	Context Mapper User
Goal	The IDE makes suggestions while the user is typing.

Main Scenario

The CML editor automatically makes suggestions based on the already given structure. For example, if the user starts typing “Bou” on the top level of a CML file, the editor automatically suggests the keyword “BoundedContext.”

Extensions

The editor also suggests, e.g. names of defined Bounded Contexts when defining Context Map relationships.

FR 1.5 - Code Folding

Primary Actor	Context Mapper User
Goal	Users can collapse and expand CML blocks.

Main Scenario

The IntelliJ CML editor recognizes definition blocks, for example the Bounded Context in Listing 2, and allows the user to collapse and expand these definition blocks.

```
BoundedContext LanguageCore {
  domainVisionStatement "Provides the Context Mapper DSL (CML) modelling
language to express architectures based on Strategic Domain-driven
Design (DDD) patterns."

  Aggregate StrategicDesign
  Aggregate TacticDesign
}
```

Listing 2: CML definition block example [11]

FR 1.6 - Keyword Tooltip

Primary Actor	Context Mapper User
Goal	Users can learn about DDD concepts while editing CML files.

Main Scenario

When moving the mouse cursor over a CML keyword, a brief description of that keyword should be displayed as tooltip documentation.

FR 1.7 - Structure Outline

Primary Actor	Context Mapper User
Goal	Users can see the structure of their CML files at one glance.

Main Scenario

The IntelliJ structure tool window should outline the structure of a CML file. The breadcrumbs of the IntelliJ structure toolbar should also display the path from the document root to the current caret position in the file.

FR 1.8 - Find Usages

Primary Actor	Context Mapper User
Goal	Users can easily find usages of their CML definitions.

Main Scenario

Both a Ctrl-/Cmd-Click with the mouse on a CML definition name and the “Find Usages” action, report all usages of that CML definition.

FR 1.9 - Document Formatting

Primary Actor	Context Mapper User
Goal	Users can easily reformat their CML files.

Main Scenario

The “Reformat Code” action ensures proper indentation and removes syntactically irrelevant whitespaces from a CML file.

FR 1.10 - Definition Tooltip

Primary Actor	Context Mapper User
Goal	Users can see descriptions of the used CML element.

Main Scenario

CML definitions can be documented using comments, as shown in Listing 3. The comment text preceding a CML element should be displayed in a hover tooltip when hovering the mouse cursor over the usage of that element.

```
...

/* The original booking application context */
BoundedContext CargoBookingContext

...
```

Listing 3: CML definitions with documentation [11]

FR 2.1 - Missing Bounded Context Quick Fix

Primary Actor	Context Mapper User
Goal	Users can easily create a missing Bounded Context.

Main Scenario

If a Bounded Context is used in a Context Map, but its definition is missing, the CML editor provides a quick fix for creating the definition.

FR 3.1 - Generate a Visual Context Map

Primary Actor	Context Mapper User
Goal	Users can create a visual representation of their CML Context Map.

Main Scenario

The context menu of a CML file displays the option “Generate Visual Context Map”. If the user clicks on the menu item, the editor gives the user a choice of output format (png, svg, or dot). After the selection, the plugin generates the visual Context Map using `graphviz` and stores the output file in a `src-gen` folder in the project root.

FR 4.1 - Generate PlantUML Diagrams

Primary Actor	Context Mapper User
Goal	Users can generate PlantUML diagrams from their CML definitions.

Main Scenario

The option “Generate PlantUML Diagrams” is displayed in the context menu of a CML file. If the user clicks on the menu item, the plugin collects all CML definitions of the current file and generates the following diagram types:

1. Use Case Diagram
2. Component Diagram
3. Class Diagram
4. State Diagram
5. Stakeholder Map
6. Value Impact Map

The generated files are stored in the output folder `src-gen` in the project root.

2.2.2. Non-Functional Requirements

The NFR (Non-Functional Requirement)s below concern the plugin itself or are cross-cutting and therefore concern all Use Cases. The ISO-25010 [12] standard was used for classification.

Table 2 overviews all defined NFRs.

ID	Summary
NFR 1	Minimise duplicated code between IntelliJ and VSCode implementations
NFR 2	Plugin stability
NFR 3	CML editor efficiency
NFR 4	PlantUML diagram generation efficiency
NFR 5	Extensibility for additional generators
NFR 6	Code quality
NFR 7	Separation of concerns
NFR 8	IntelliJ compatibility
NFR 9	Licence compatibility
NFR 10	IntelliJ best practices

Table 2: Overview of project NFRs

NFR 1: Reusability - Minimise duplicated code between IntelliJ and VSCode implementations

Both the VSCode and IntelliJ plugins will offer the same features. While the implementations cater to the respective IDE platform, the core logic stays the same and should be reused if possible.

Verification

Review of extension point classes

Acceptance Criteria

Classes implementing IntelliJ extension points do not contain implementation logic and use either existing Context Mapper logic or delegate to a reusable component.

Realisation

Creating adapters for existing Context Mapper logic in case it should not be directly reusable. Discussion of potentially necessary changes to Context Mapper to improve reusability. Separation of IntelliJ specific code and feature logic where using the existing implementation was not possible, so the logic may easily be extracted/ replaced later.

NFR 2: Reliability - Plugin stability

The IntelliJ plugin should not terminate unexpectedly, e.g. if `graphviz` is not installed and the user wants to generate a visual Context Map

Verification

Unit and component tests

Acceptance Criteria

Unit and component tests for error cases pass.

Realisation

Proper error handling, e.g. checking if `graphviz` is available before generating visual context maps.

NFR 3: Performance Efficiency - CML editor
The editor must remain responsive during normal editing activities, such as typing.

Verification
Informal user test with the thesis advisor
Acceptance Criteria
No reports of editor freezes or lags.
Realisation
Avoiding synchronised, performance-intensive actions in editor functions.

NFR 4: Performance Efficiency - PlantUML diagram generation
The generation of the PlantUML component diagram for the stage 3 DDD-Sample from the context-mapper-examples repository[1], is completed within 0.5 seconds.

Verification
Performance Test
Acceptance Criteria
Performance measurement of the PlantUML component diagram generation completes within 500ms.
Realisation
Avoiding costly CML model conversions in the generator.

NFR 5: Modifiability - Extensibility for additional generators
The PoC plugin can easily be extended with more generators.

Verification
Creation of a dummy generator
Acceptance Criteria
A new generator can be added by using one extension point.
Realisation
Generators implement the same interface and are dynamically loaded by the plugin.

[1] https://github.com/ContextMapper/context-mapper-examples/blob/master/src/main/cml/ddd-sample/DDD-Sample-Stage-3.cml

NFR 6: Maintainability - Code quality
The plugin code passes linting & static code analysis.

Verification
Code linting with KtLint[1] & code analysis in CI (Continuous Integration) pipeline with JetBrains Qodana[2]
Acceptance Criteria
KtLint and Qodana report no problems or warnings.
Realisation
Following coding best practices & implementation of Qodana suggestions.

NFR 7: Modularity - Separation of concerns
Plugin code is structured into feature-based packages to improve code discoverability. Cross-package dependencies are only allowed for shared helper/utility classes in a `utils` package.

Verification
Automated structure checks with ArchUnit
Acceptance Criteria
ArchUnit tests pass.
Realisation
Strictly separate features from each other. Move shared code to helper classes.

NFR 8: Installability - IntelliJ compatibility
The plugin can be installed in the Community and Ultimate versions of IntelliJ, starting from version 2024.3.

Verification
Manual installation test with the final PoC plugin
Acceptance Criteria
Successful installation of the PoC plugin in the mentioned IntelliJ versions.
Realisation
Avoiding IntelliJ features only available in the Ultimate version of IntelliJ or features only available in most recent IntelliJ versions.

[1] https://github.com/pinterest/ktlint
[2] https://www.jetbrains.com/qodana

NFR 9: Legal Compliance - Licence compatibility
All used frameworks & libraries have to be compatible with the Apache 2.0 licence.

Verification
Manual dependency check
Acceptance Criteria
No dependency with a licence incompatible with Apache 2.0 (according to Apache guidelines [13]) is found.
Realisation
Licence as a decision factor for technology decision, checking licences before using libraries.

NFR 10: Appropriateness/Recognizability - IntelliJ best practices
Features implemented in the IntelliJ plugin should follow IntelliJ best practices. Features included in the Context Mapper Eclipse plugin should only be rebuilt in the same way if they fit into the IntelliJ best practices.

Verification
Informal user test with thesis advisor
Acceptance Criteria
No reports of confusion about the way a feature is visible in IntelliJ.
Realisation
Studying IntelliJ documentation and tips.

3. Technology Exploration

This section explores the different technological options to create a DSL and to integrate a DSL in IntelliJ. Each option was evaluated regarding its suitability for Context Mapper. The best-suited option for the IntelliJ integration was then used to implement the PoC.

Descriptions of the tests performed with the evaluated technologies can be found in Appendix B.

3.1. Language Workbenches

DSL workbenches are tools that help developers create DSLs. More specifically, they support modelling the language itself, provide at least one editing environment for the language, and define its behavioural semantics [14].

This section gives an overview of such language workbenches that could be used to replace Xtext.

3.1.1. JetBrains MPS

JetBrains MPS (Meta Programming System) [15] is a language workbench to create DSLs and comes with its own IDE. MPS works differently than most language workbenches. While a DSL creator usually defines a grammar in "Backus-Naur form"[1] or a similar notation, this is not the case with MPS. Instead, JetBrains created its own DSL to define language structure, editor views, constraints, quick fixes, etc.

MPS uses projectional editing [16], which is why the editing experience and the experience of creating the DSL are different from those of other language workbenches. Most programming languages store their code in text files. These files are then parsed, which results in an AST (Abstract Syntax Tree). With projectional editing, the user does not write a text file but directly modifies the AST. This allows the editor to use graphical formats (e.g. to display mathematical equations in their correct visual representations) or to hide unnecessary information from the user.

Languages constructed with MPS can be packaged into a plugin, which can then be installed in the MPS IDE. The plugin can also be installed in, e.g., IntelliJ, but the "MPS Core" plugin[2] is required as a dependency since IntelliJ itself is not capable of handling projectional editing.

3.1.2. Langium

Langium [17] is a language workbench written in TypeScript and is part of the Eclipse project. While Xtext is closely integrated with Eclipse, Langium focuses more on VSCode, which is why TypeScript has been chosen as its implementation language. The integration of a DSL into IDEs is achieved via LSP, which means that other IDEs than VSCode can be targeted. A language plugin for VSCode can be built in the same project as the language itself. Plugins for other IDEs require a separate project.

[1] https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form
[2] https://plugins.jetbrains.com/plugin/7075-mps-core

Langium itself does not provide a direct integration into IntelliJ. It would require combining it with one of the LSP integration options.

3.1.3. Rascal

Rascal [18] is a meta-programming language that allows the creation of DSLs. Its features include the definition of language syntax, parser generators, and hooks for IDE integration. The documentation mentions IDE integration into Eclipse via the Eclipse Meta-Tooling Platform and VSCode via LSP.

Since Rascal does not integrate into IntelliJ by itself, one of the described LSP options would also have to be used.

3.1.4. Spoofax

Spoofax [19] is a language workbench that allows developers to create their languages using meta-languages. Using the "Syntax Definition Formalism"[1], developers can define their language's syntax, and Spoofax generates parsers, type checkers, compilers, etc., for the language. Spoofax comes with an Eclipse extension and can generate Eclipse editors for defined languages.

Attempts to set up a sample project with Spoofax proved cumbersome, and error messages suggested that Spoofax's technology stack was outdated (Java 8). It seems that a new version of Spoofax is under development [20], which should also support IntelliJ plugins in the future. However, since the current stable version of Spoofax only supports the creation of Eclipse editor extensions and the documentation does not mention the generation of a language server, Spoofax was not considered for the PoC in this thesis.

3.1.5. Others

There are more language workbenches out there than the ones introduced above. Two worth mentioning are MetaEdit+[2] and MontiCore[3]. Both do not offer the possibility of IDE integrations for the created languages, which is why they are not included in the evaluation.

While researching language workbench, other names of language workbenches popped up, but these either have not reached a mature level or have been abandoned.

3.2. IntelliJ Integration Options

There are two possible ways to create a plugin for custom language support in IntelliJ: LSP integration or native development. Both integration styles are further explained below.

3.2.1. LSP Integration

As described in Section 1.1.2, LSP is a protocol for language servers and development environments to communicate with each other. To integrate a language server in IntelliJ, there are two possible options:

[1] https://spoofax.dev/references/sdf3/
[2] https://www.metacase.com/
[3] https://monticore.github.io/monticore/

- IntelliJ LSP Support
- External Plugins like LSP4IJ

The capabilities of these two options are further explained below.

Language servers cannot fully replace a native IDE integration. Some native customisation is still required, e.g., for the Context Mapper generators.

IntelliJ LSP Support
JetBrains added LSP support for IntelliJ Ultimate as of version 2023.2 [21]. The Community version of IntelliJ does not include LSP support, so plugins using this integration style cannot be installed in IntelliJ Community IDEs.

The 2024.3 Ultimate version supports the most important LSP features. They include:
- Go-To declaration
- Code completion
- Quick fixes
- Error & warning highlighting
- Quick (hover) documentation
- Code formatting
- Semantic highlighting
- Find usages

Plugins can bundle a binary for the language server or let the user define the language server location.

LSP4IJ
LSP4IJ [22] is an IntelliJ plugin developed by the Red Hat developer community. It allows users or developers to configure language servers for specific languages/file types. Users can configure language servers via the UI, and developers can build their own plugins based on the LSP4IJ extension points and API (Application Programming Interface). LSP4IJ's advantage over IntelliJ's native LSP support is that the plugin is also available for IntelliJ Community versions.

LSP4IJ 0.11.0 does not yet support all LSP features, but it does support the most important ones.

3.2.2. Native IntelliJ Integration
Integrating a DSL natively into IntelliJ is an approach many DSL plugins like Structurizr[1] use. A native integration requires a lexer, a parser and an implementation of PSI (Program Structure Interface) classes [21]. The PSI is a layer in the IntelliJ platform, responsible for parsing files and creating code models. Many IntelliJ editor features are built upon the PSI.

JetBrains offers various extension points through which syntax highlighting, autocomplete, custom actions, etc., can be implemented. While an LSP integration

[1] https://plugins.jetbrains.com/plugin/20606-structurizr-dsl-language-support

adds many editor features, a native integration allows the implementation of a broader range of IDE features.

The required lexer and parser can either be written manually or generated by a parser generator. The easiest way to generate a lexer and parser is to use JFlex[1] and GrammarKit[2] since the generated parser integrates into IntelliJ's PSI. Other parser generators, like ANTLR[3], can also be used, but these parsers must be manually adapted to the PSI to be useful.

3.3. Technology Decision

The technology decision is based on a utility analysis [23]. In a utility analysis, the criteria that influence the decision must be defined first. Then, these criteria are weighted, and scoring scales are defined for each criterion. Finally, each option is evaluated according to the criteria and scoring scale. The weighted scores are summed up per option, allowing the quantification of each option's utility and making it possible to compare options. The option with the highest total score provides the best utility.

The defined criteria can be found in Section 3.3.1. The evaluation of language workbenches and integration options is summarised in Section 3.3.2 and Section 3.3.3. For the detailed evaluation with reasoning for the given scores, see Appendix C. Finally, the resulting scores, including the technology decision, can be found in Section 3.3.4.

[1] https://www.jflex.de/
[2] https://plugins.jetbrains.com/plugin/6606-grammar-kit
[3] https://www.antlr.org/

3.3.1. Criteria

The criteria in Table 3 have been set in collaboration with the thesis advisor. Each criterion includes a brief statement clarifying its goal, weight, and scoring system.

Criterion	Goal	Weight	Scoring
Future Proofing	The technology is well maintained and will not be abandoned anytime soon	3	3 = Large group of maintainers (30+ active maintainers) or maintained by a (mid-size to large) company, future of technology is secured 2 = Small group of maintainers (4-30 active maintainers) or future support of the technology is unknown 1 = Less than four active maintainers or technology is/will be abandoned
Ease of use	The technology has a low learning curve, is not overly complicated and easy to use	2	3 = Technology concepts are straightforward and easy to understand, and its usage is straightforward 2 = Some of the involved concepts require a more profound understanding, and its usage is straightforward in most cases 1 = The technology requires a deep understanding of all involved concepts, and its usage is complicated in most cases

Criterion	Goal	Weight	Scoring
Documentation	The technology provides a well-maintained documentation	2	3 = The documentation is easy to read, complete and up-to-date 2 = The documentation has some gaps, is not clear in some details, or some parts have not been updated in a while 1 = The documentation is very minimalistic, complicated or outdated
Feature Support	The technology supports features that already exist in the Eclipse plugin of the Context Mapper (unless there is no equivalent of that feature in IntelliJ)	3	3 = The technology supports all major and minor features of Context Mapper 2 = The technology supports all major features of Context Mapper; some minor features are not supported 1 = The technology does not support a major feature of Context Mapper
Licence	The technology licence is compatible with the Apache 2.0 licence of Context Mapper.	3	3 = Licence is compatible without additional restrictions 2 = Licence is compatible with some additional restrictions 1 = Licence is not compatible
IDE Compatibility	The technology supports both IntelliJ and VSCode	3	3 = Support for IntelliJ (Ultimate and Community) and VSCode 2 = Support for IntelliJ (Ultimate only) and VSCode 1 = Support only for IntelliJ (Ultimate or Community)

Criterion	Goal	Weight	Scoring
Reusability	Existing Context Mapper logic can be reused	2	3 = Most logic can be reused 2 = Partial rewrites are necessary 1 = Full rewrite necessary

Table 3: Technology evaluation criteria

3.3.2. Workbench Evaluation

The following summarises the evaluation of the language workbenches JetBrains MPS, Langium and Rascal. For the conclusion, see Section 3.3.4.

JetBrains MPS

JetBrains MPS is a future-proof option by JetBrains. Its projectional editor provides interesting possibilities but is incompatible with the current version of Context Mapper. MPS would require a complete rewrite and could only be used in the MPS IDE and IntelliJ (though 2024 versions of IntelliJ do not support MPS yet).

With a guided tutorial (~2 hours), a sample language (chemmastery) could be extended within a few minutes. The key concepts became clear after working through the tutorial, but understanding the MPS DSL beyond the tutorial scope requires much more time.

Langium

Written in TypeScript, using Langium would require a complete rewrite of Context Mapper. Langium itself is well-documented and easy to understand. By leveraging LSP, the resulting language server can be used in various IDEs. Langium is actively maintained and promotes a clear feature-based structure.

Following the tutorial in the documentation [17], allowed the creation of a VSCode plugin and language server for a sample language within 30 minutes.

Rascal

Since there is no tutorial on IDE integration in the documentation [18], and IDE integration requires knowledge of Rascal's LSP package, a Rascal sample language could not be tested in an IDE. Rascal as a language is well documented and similar to Haskell's syntax. Using Rascal would require a complete rewrite of Context Mapper.

3.3.3. Integration Option Evaluation

The evaluation of LSP integration (IntelliJ LSP support and LSP4IJ) and the native integration is outlined below. For the conclusion, see Section 3.3.4.

IntelliJ LSP

The documentation for IntelliJ's LSP support is up-to-date but limited. Further information has to be obtained from existing plugins using LSP support, e.g. the Vue.js

plugin[1]. While only available for Ultimate versions of IntelliJ, the language server can be reused.

In a brief test with the existing Xtext CML language server, some features like the keyword tooltip documentation worked instantly, while other LSP features like semantic tokens did not work yet.

LSP4IJ
LSP4IJ supports the majority of LSP capabilities and works in both IntelliJ Community and Ultimate versions. The plugin is actively maintained and receives frequent updates. It would allow the reuse of the language server, as with IntelliJ's LSP support.

A brief test with the LSP4IJ configuration UI delivered the same results as the test with IntelliJ's built-in LSP support.

Native Integration
Integrating CML natively into IntelliJ would offer the broadest range of editor features but also means that other IDE plugins have to be developed separately. The documentation [21] is extensive but not complete. Some editor features require a deeper understanding of PSI and IntelliJ's extension points.

Following the tutorial on custom language plugins in the documentation made it possible to create a plugin for a sample language within a day.

The native integration approach was evaluated using JFlex and Grammar-Kit as lexer and parser generators. ANTLR was briefly considered as an option to retain a common grammar base between a VSCode and IntelliJ plugin, but has not been evaluated further as integration of the parser into the PSI is crucial and not provided for ANTLR.

3.3.4. Result

Before discussing the evaluation's results, it is important to highlight that the combination of the best-suited language workbench and the most suitable integration option is not always compatible. For example, combining JetBrains MPS with an integration option does not work as the MPS has its own IDE and its own way of creating plugins. Conversely, a native integration does not require a language workbench at all. Workbenches like Langium and Rascal can be combined with one of the LSP-based integration options but are not compatible with the native integration option.

Table 4 provides an overview of the evaluation results, which contain the weighted score totals for the language workbenches and IntelliJ integration options.

[1] https://github.com/JetBrains/intellij-plugins/tree/master/vuejs

Criterion	Language Workbenches			Integration Options		
	MPS	Langium	Rascal	IntelliJ LSP	LSP4IJ	Native
Future Proofing	9	6	6	9	6	9
Ease of use	2	4	4	4	4	4
Documentation	4	6	2	4	6	4
Feature Support	3	9	6	6	9	9
Licence	9	9	9	9	6	9
IDE Compatibility	3	9	9	6	9	3
Reusability	2	2	2	6	6	4
Total	**32**	**45**	**38**	**44**	**46**	**42**

Table 4: Utility analysis results

For the integration option, the highest utility is achieved through an LSP-based integration with LSP4IJ. Considering the broader context of Context Mapper, this result is logical. A language server ensures that editor features can be reused in different IDEs, giving Context Mapper the flexibility to target different IDEs in the future. At this time, LSP4IJ supports more LSP capabilities than IntelliJ's LSP support and is available in the IntelliJ Community version. While a native integration offers the broadest range of supported features, it requires maintaining separate codebases for every IDE plugin. It would theoretically be possible to have a shared grammar between the IDE plugins, for example, using ANTLR. However, integrating such a generated parser into the IntelliJ PSI requires considerable effort. Therefore, the PoC will use LSP4IJ as an integration option.

For the workbenches, JetBrains MPS is an intriguing option, but its incompatibility with existing CML files speaks against its use for Context Mapper. Langium made a more mature impression than Rascal. It provides better documentation on how to create a language server and has more active maintainers. That makes Langium the most suitable language workbench for Context Mapper.

The complete reasoning behind the evaluation results can be found in Appendix C.

LSP4IJ requires a language server to provide editor services for the CML. The existing Xtext language server or a newly created one could be used. The adoption of Langium was initially postponed to a later project, and the reuse of the existing Xtext language server was preferred. However, it was discovered that customisation to the Xtext language server would have been necessary. Since Xtext is not considered future-proof, a new Langium language server was created as well.

The decisions made in this section qualify as architectural decisions. Detailed ADR (Architectural Decision Record)s can be found in Appendix D.

4. Proof of Concept Implementation

This section describes essential aspects, like architecture and necessary steps to implement features, as well as challenges encountered during the PoC implementation, such as issues with the CML grammar.

4.1. Architecture

The PoC architecture is based on analysing architecturally significant requirements and architectural decisions. The resulting architecture is documented using a Context Map and the C4[1] model.

4.1.1. Architecturally Significant Requirements

To find architectural requirements, the requirements from Section 2.2 were evaluated regarding their architectural significance, using the ASR (Architecturally Significant Requirement)-Test [24] method. The ASR Test entails seven criteria that indicate that a requirement is significant to a system's architecture. A requirement is architecturally significant if the requirement:

1. …is associated with high business value
2. …is a concern of an important stakeholder
3. …includes QoS (Qualify-of-Service) characteristics
4. …causes new or deals with existing unpredictable/unreliable external dependencies
5. …has a cross-cutting nature
6. …has a first-of-a-kind character
7. …has been troublesome in a previous project

Table 5 shows which of the requirements qualify as architecturally significant.

Requirement	ASR-Test Criteria	Reasoning
NFR1: Minimise duplicated code between IntelliJ and VSCode implementations	• Concern of a stakeholder • Cross-Cutting nature	Code reusability is an essential concern of the Context Mapper maintainer. Reusability affects the VSCode and IntelliJ plugins.
NFR2: Plugin stability	• High business value • Cross-Cutting nature	Prevention of crashes, etc., affects all plugin components. Plugin stability increases user satisfaction.

[1] https://c4model.com/

Requirement	ASR-Test Criteria	Reasoning
NFR3: Editor performance	• High business value • Quality-of-Service characteristic • Cross-Cutting nature	Editor performance directly affects the editing experience. Low response times for syntax highlighting, autocomplete, etc., are important to make the editor convenient and, therefore, increase user satisfaction. It affects all editor features.
NFR5: Extensibility for additional generators	• Concern of stakeholder	Making it easy to add new generators increases the maintainability of the plugin. Design considerations are necessary to achieve that.
NFR7: Separation of concerns	• Cross-Cutting nature • Concern of stakeholder	Separation of concerns impacts the scope of plugin components. By clearly separating feature logic, it becomes easier to replace or reuse code at a later point.

Table 5: Architectural Significance of non-functional requirements

4.1.2. Strategic Design

In DDD, bounded contexts describe parts of a system with their own Domain Model and maybe even a different technology stack [25]. There are two bounded contexts in this project: The language server and the IntelliJ plugin.

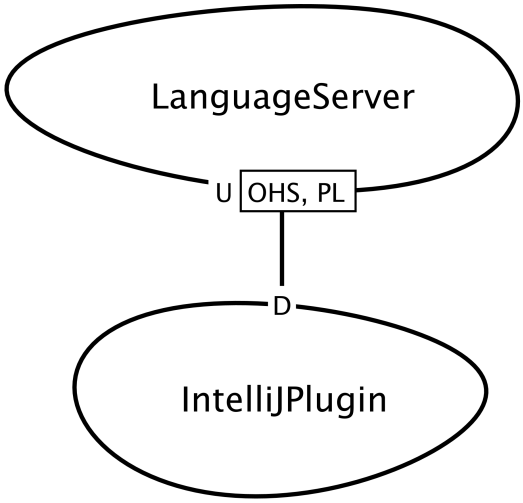

Figure 6: Context map of the project

Figure 6 shows the contexts in an upstream-downstream relationship, where the language server is the upstream, and the IntelliJ plugin is the downstream context. The language server implements the LSP as the Published Language. Development environments (clients) can freely access the services from the language server, making it an Open Host Service.

The following technical architecture description models the language server as the software system "Context Mapper Language Server" and the IntelliJ plugin context as the container "Context Mapper Plugin".

4.1.3. C4 Model

The C4 model [26] defines four detail levels to describe software architecture: (System) Context, Containers, Components, and Code.

Software systems The definition of a software system is often different from organisation to organisation. It can be defined as a team boundary or as a collection of software contributing to the same goal.

Containers A container represents a runtime boundary around code. Containers are individually deployable units of code.

Components Components are part of a container and group related functionality.

Code On the code level, the C4 model refers to UML (Unified Modelling Language) diagrams.

Software System

This thesis defines the term software system as a deployable (group of) container(s) that serve a common goal. Since the IntelliJ plugin cannot run independently, it does not qualify as a software system. The software system of the plugin is IntelliJ itself. However, since JetBrains maintain IntelliJ, it is not described in detail.

The language server runs independently in its own process and can be used by more IDEs than just IntelliJ, which is why it is considered a software system. Figure 7 shows the interactions between the software systems.

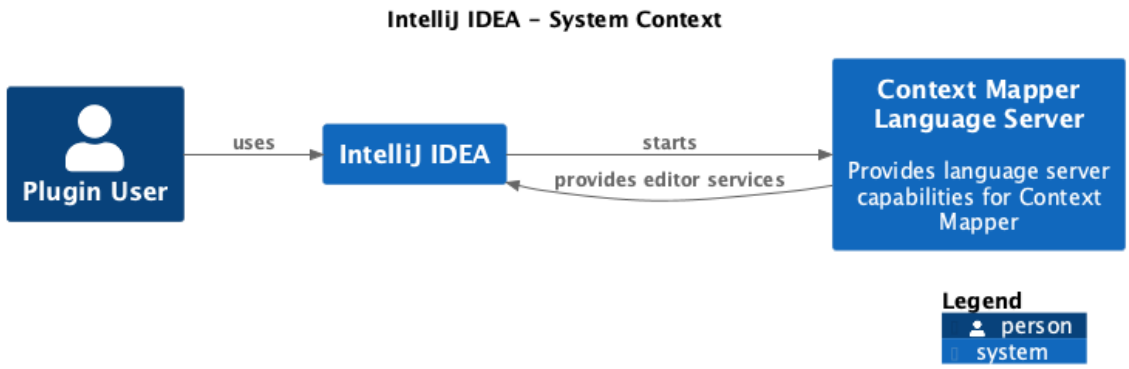

Figure 7: C4 system context diagram of the Context Mapper IntelliJ plugin

Containers

Figure 8 shows the containers involved in this PoC. The diagram shows the perspective of IntelliJ as the software system, which is why the language server (another software system) is drawn outside the system boundary.

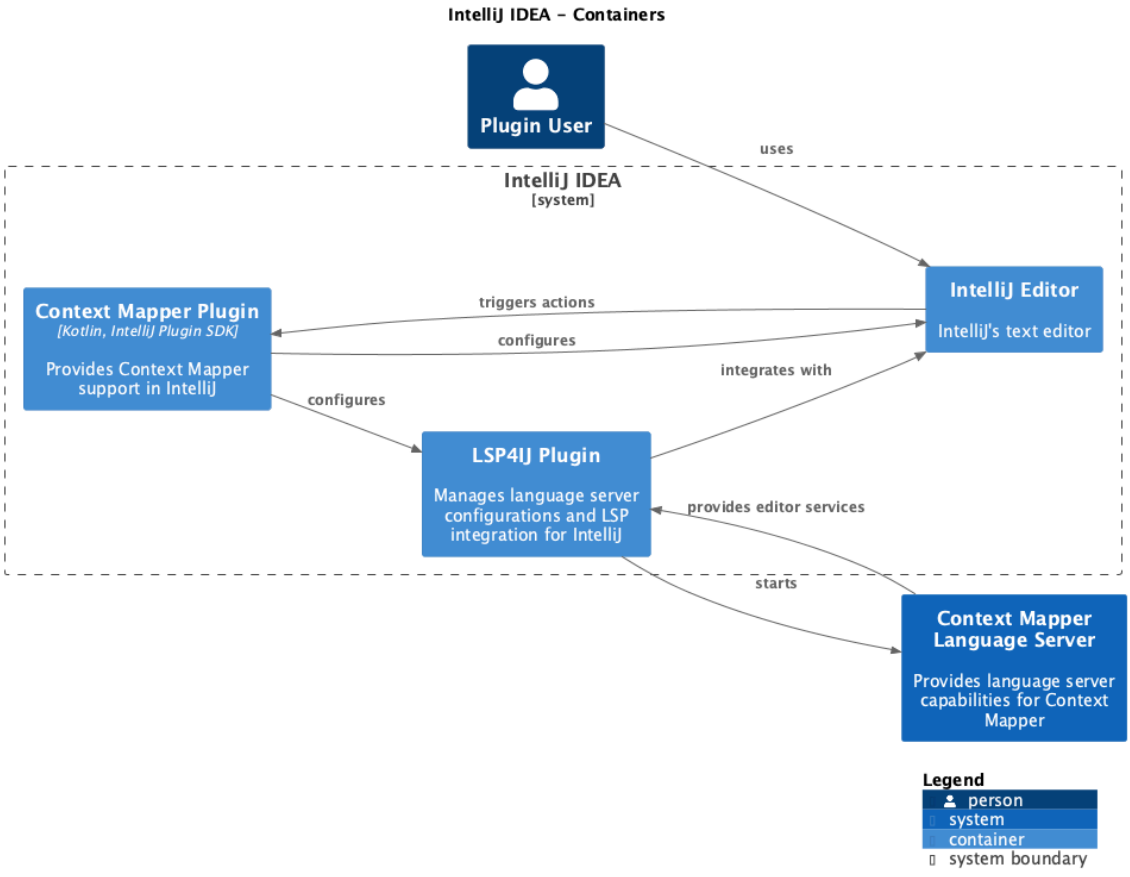

Figure 8: C4 container diagram of the Context Mapper IntelliJ plugin

The three main containers are:

Context Mapper plugin The IntelliJ plugin configures LSP4IJ and adds additional features to the IntelliJ editor.

LSP4IJ plugin The LSP4IJ plugin is in charge of managing language servers and communicating with them. To do that, LSP4IJ is deeply integrated into the IntelliJ editor features.

Context Mapper language server A Langium language server that provides information on CML files to the LSP4IJ plugin. The language server is a one-container system, which is why there are no containers displayed in the diagram above.

This setup keeps the IntelliJ plugin lightweight, reducing the required maintenance work on the plugin.

Language Server Components

The components in the language server, displayed in Figure 9, correspond to the LSP capabilities that are either not supported out-of-the-box or required customisation.

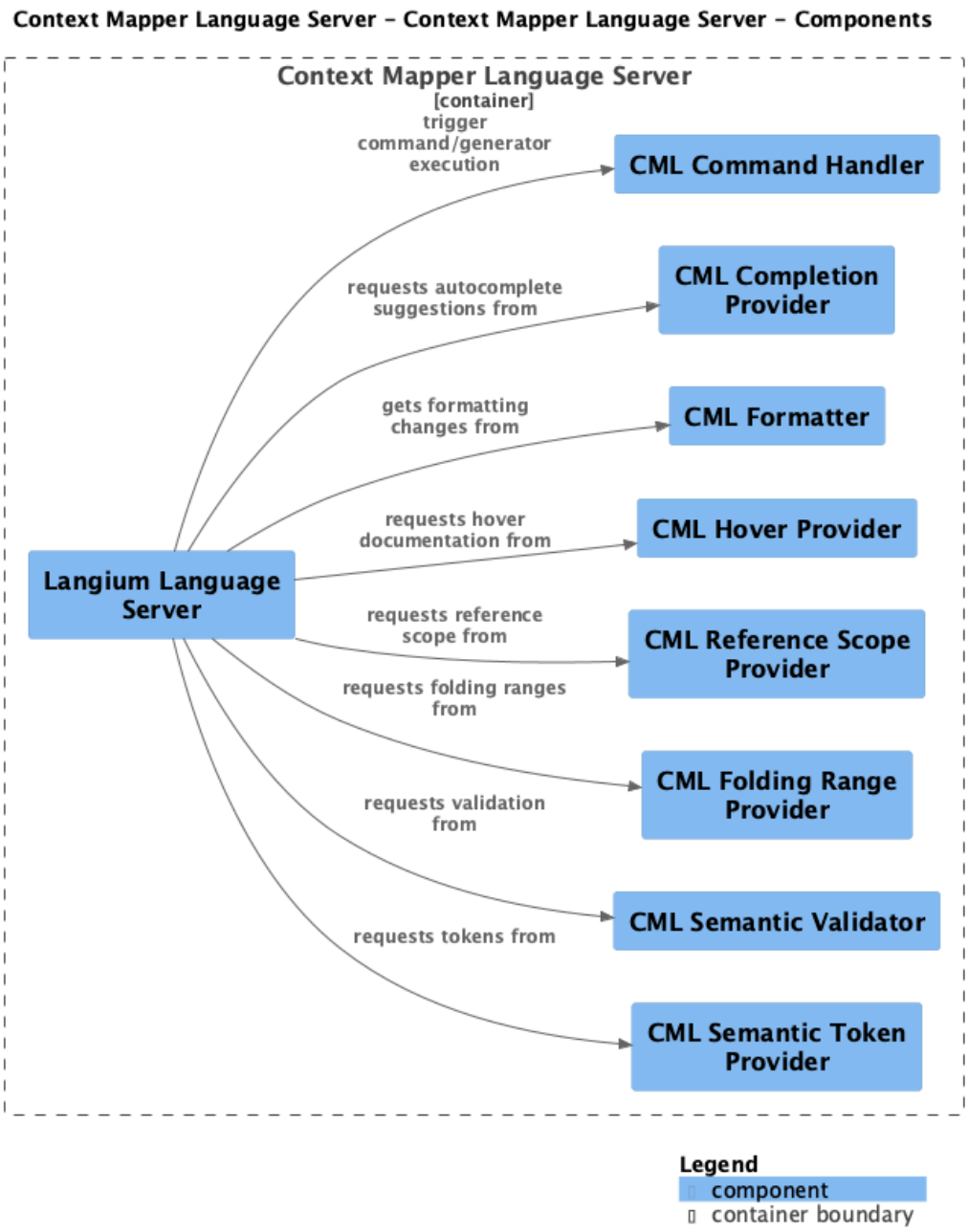

Figure 9: C4 component diagram of the Context Mapper Language Server

The language server is structured according to Langium's component-based architecture. At its core is the server component, which is responsible for interacting with the development environment. It receives requests via LSP and forwards the request to the responsible feature component, such as the semantic token provider. The feature components return their response to the server component, which then sends the LSP response to the development environment. Each feature component's logic is encapsulated behind one entry-point class registered with the Langium server module. This design promotes a clear separation of feature implementations, improving their maintainability.

Plugin Components

The IntelliJ plugin does not contain much implementation logic, as shown in Figure 10.

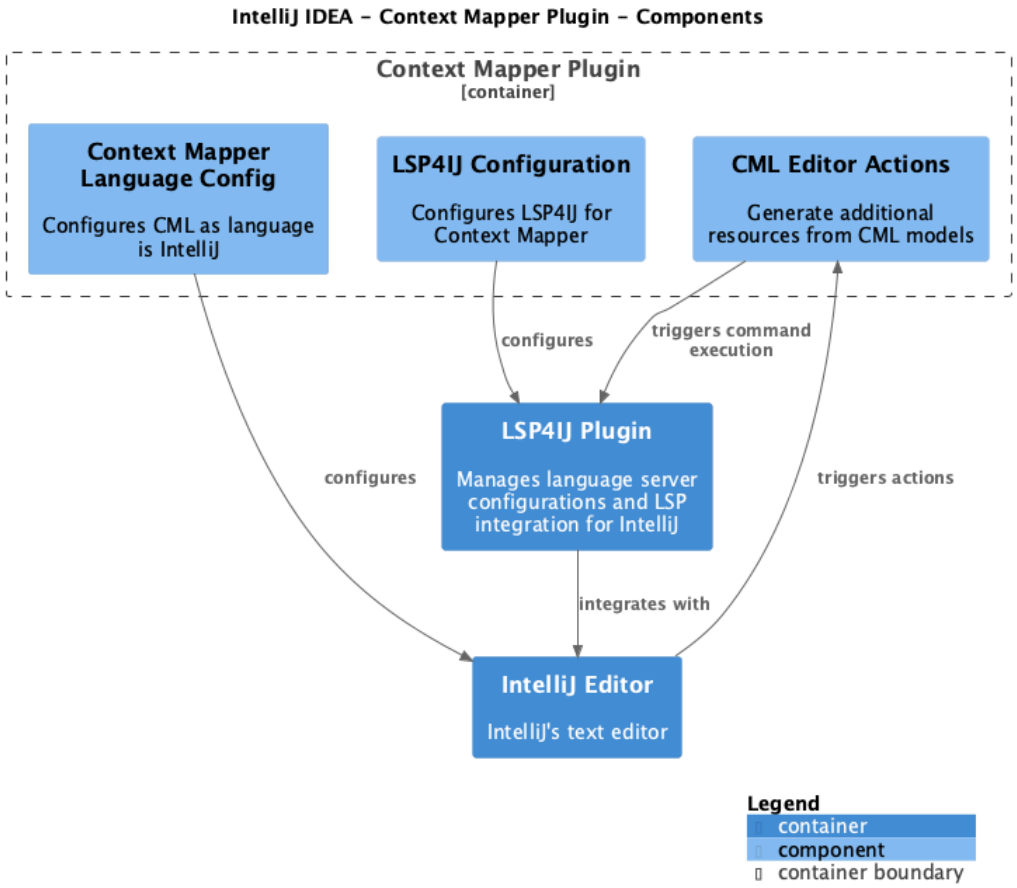

Figure 10: C4 component diagram of the Context Mapper Plugin

The Context Mapper language config defines CML as an editor language, associating `.cml` files with it. The LSP4IJ configuration component defines the language server in LSP4IJ and configures individual editor features. Generators are implemented as editor actions, which trigger an LSP command in LSP4IJ.

Code

Details on the implementation of the plugin and the language server can be found in Appendix E.

4.1.4. Deployment

The IntelliJ plugin and the language server are bundled and deployed as a single package, as illustrated in Figure 11.

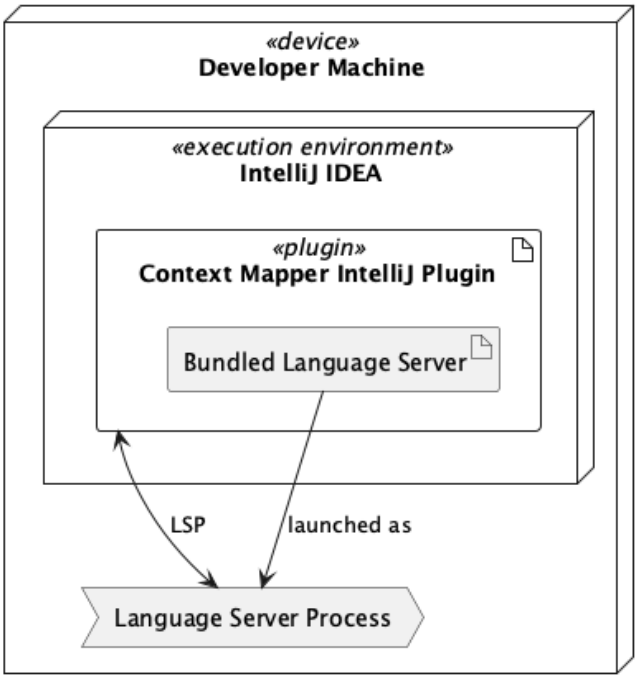

Figure 11: Deployment diagram of the Context Mapper IntelliJ plugin

This setup ensures that the plugin can reliably access and launch the language server, eliminating the need to check for an external dependency. Bundling the language server also improves the plugin usability, as users do not need to install an additional package for the language server.

4.2. Context Mapper Grammar Changes

Since creating a new language server was not part of the original scope of this thesis and the project has a fixed timeframe, the CML grammar was reduced by omitting tactic DDD and the `import` grammar elements.

While Langium's grammar syntax is similar to that of Xtext, there are a few key differences. TypeFox provides a tool[1] to convert Xtext grammars to Langium. However, the tool produced overly complex enum definitions and generated a grammar that was not valid in Langium.

Instead, the grammar was converted manually. The encountered grammar issues are explained in detail below.

4.2.1. Comments

The Xtext CML grammar allows comments to appear anywhere in a `.cml` file. However, comments are also an explicit part of the CML grammar, an example of which can be seen in Listing 4.

[1] https://github.com/TypeFox/xtext2langium

```
ContextMappingModel:
 (
  (topComment=SL_COMMENT | topComment=ML_COMMENT)? &
    ...
    (boundedContexts += BoundedContext)* &
    ...
  )
;

BoundedContext:
 (comment=ML_COMMENT | comment=SL_COMMENT)?
  ...
;
```

Listing 4: Ambiguous comment grammar rules in CML Xtext grammar [8]

These rules are ambiguous for Chevrotain[1], the parser used by Langium. A top-level comment can be a ContextMappingModel topComment, a BoundedContext comment, or a regular comment. The ANTLR parser used by Xtext could handle this ambiguity by backtracking if a rule did not match, but Chevrotain is stricter in that regard and requires clear grammar rules.

As a consequence of this issue, comments were excluded entirely from the grammar rules and converted to hidden terminal rules, as shown in Listing 5.

```
hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Listing 5: Langium grammar comment terminal rules

Making the comment terminal rules hidden still allows users to add comments to their CML files, but the parser will ignore them so that they will not appear in the AST nodes.

4.2.2. Optional Elements in Unordered Groups

The CML Xtext grammar heavily relies on optional elements in unordered groups. Listing 6 shows an example of this. The `*` or `?` cardinalities make grammar elements optional. The `&` operator chains the elements together, indicating to the parser that the defined elements can appear in any order.

```
ContextMappingModel:
  (
    (topComment=SL_COMMENT | topComment=ML_COMMENT)? &
    (imports+=Import)* &
    (map = ContextMap)? &
    (boundedContexts += BoundedContext)* &
    (domains += Domain)* &
    (userRequirements += UserRequirement)* &
    (stakeholders += Stakeholders)* &
    (valueRegisters += ValueRegister)*
  )
;
```

Listing 6: Use of optional elements in an unordered group in the CML Xtext grammar [8]

[1] https://chevrotain.io

Chevrotain does not support combining optional elements and unordered groups in this way. According to Mark Sujew, a maintainer of Langium, there are two possible workarounds to this issue [27]:

1. Grammar elements in unordered groups are optional by default, so removing the `?` cardinality still has the desired effect, and the parser accepts the rule.
2. The second recommended workaround is to use a rule in the style of `(A | B | C)*`. This rule allows zero or more repetitions of the grammar elements A, B and C in any order. A semantic validator that enforces non-repetition needs to be registered in the language server to enforce a maximum of one repetition per grammar element.

The second workaround also allows one to define clear error messages, while the first workaround may produce cryptic error messages for invalid CML content.

An issue with unordered groups and autocomplete was also discovered. After one attribute was set, e.g. the `domainVisionStatement` in a Bounded Context, autocomplete stopped making suggestions.

For that reason, the `(A | B | C)*` workaround was applied. Most unordered groups contained elements with the `?` cardinality, which with this change turned from simple properties like `string` to array properties, making access to these properties in the AST inconvenient.

4.2.3. User Requirement Linking in Aggregate

The CML Aggregate structure allows four ways to specify related user requirements. Listing 7 contains the related Xtext grammar rule. As can be seen, an Aggregate can be related to only Use Cases or User Stories by using the `useCases` or `userStories` keyword. It can also be related to both Use Cases and User Stories by using either the `features` or `userRequirements` keywords.

```
Aggregate:
  ...
  (
    (('useCases' ('=')? userRequirements += [UseCase]) ("," 
userRequirements += [UseCase])*) |
    (('userStories' ('=')? userRequirements += [UserStory]) ("," 
userRequirements += [UserStory])*) |
    ((('features' | 'userRequirements') ('=')? userRequirements += 
[UserRequirement]) ("," userRequirements += [UserRequirement])*)
  )? &
  ...
;
```

Listing 7: Aggregate User Requirements rule in the CML Xtext grammar [8]

In the AST, though, this separation is not relevant as all the requirements end up in the `userRequirements` property. Chevrotain can handle these rules for parsing but not for linking. User Stories specified using the `userStories` keyword were attempted to be linked to a Use Case.

This issue could be resolved by separating the properties in which the `UserStory`, `UseCase` and `UserRequirement` values are stored in the AST (see Listing 8). Separating

the AST properties makes it clear during linking which values should be related to which type. This change does not have any impact on users.

```
Aggregate:
  ...
  (('useCases' ('=')? useCases+=[UseCase]) ("," useCases+=[UseCase])*) |
  (('userStories' ('=')? userStories+=[UserStory]) ("," 
userStories+=[UserStory])*) |
  ((('features' | 'userRequirements') ('=')? 
userRequirements+=[UserRequirement]) ("," 
userRequirements+=[UserRequirement])*) |
  ...
;
```

Listing 8: Resolved Aggregate user requirements linking issue

4.3. Implemented Features

This section outlines the features implemented in the language server and IntelliJ plugin as part of the PoC. The goal is to provide a concise but comprehensive picture of the PoC's current capabilities.

4.3.1. Semantic Validation

The grammar changes described in Section 4.2 made it necessary to add semantic validation so that the supported CML elements are equivalent to their Xtext implementation.

Semantic validation performs checks beyond what the parser can check based on the language grammar. A validation registry and validator need to be created for validations to be executed [17].

Multiple specialised validators have been created to keep the validator implementation clean. For more details, see Appendix E.1.6.

The PoC only includes semantic validations necessary to compensate for the required grammar changes described in Section 4.2.2. The Xtext implementation of Context Mapper includes more validation rules that have not been migrated.

Figure 12 shows an example of a syntax error in the CML editor that was detected through semantic validation.

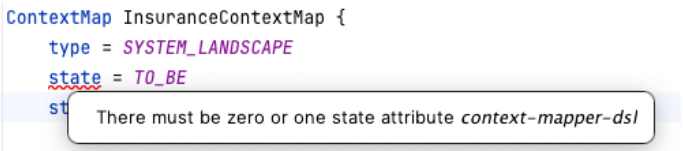

Figure 12: Screenshot of an error message in the CML editor from semantic validation

4.3.2. Syntax Highlighting (FR 1.1)

Syntax Highlighting via LSP is based on so-called "semantic tokens". A semantic token describes an element's position in a source file, as well as its type and modifier (declaration, static, etc.) [5].

When an editor requests the semantic tokens for a file, Langium traverses a source file's AST and requests tokens for each node from the semantic token provider [17].

Since Langium does not provide a default implementation, a semantic token provider had to be implemented. The created token provider delegates token requests to specialised providers. For more details, see Appendix E.1.1.

Generating tokens for all AST nodes is not enough, though. As comments are not part of the AST, they will not get semantic tokens created for them. So, to enable syntax highlighting for comments, a RegEx (Regular Expression) search on the root node had to be implemented to locate all comments and generate tokens for them.

In the IntelliJ plugin, syntax highlighting is handled by the LSP4IJ plugin. The only customisation necessary was translating the semantic token types and modifiers to editor text attributes. For more details, see Appendix E.2.1.

4.3.3. Hyperlinking (FR 1.2)

The navigation between a language element usage and its definition worked out of the box. However, as shown in Figure 13, the displayed hyperlink covered the whole file instead of just one element.

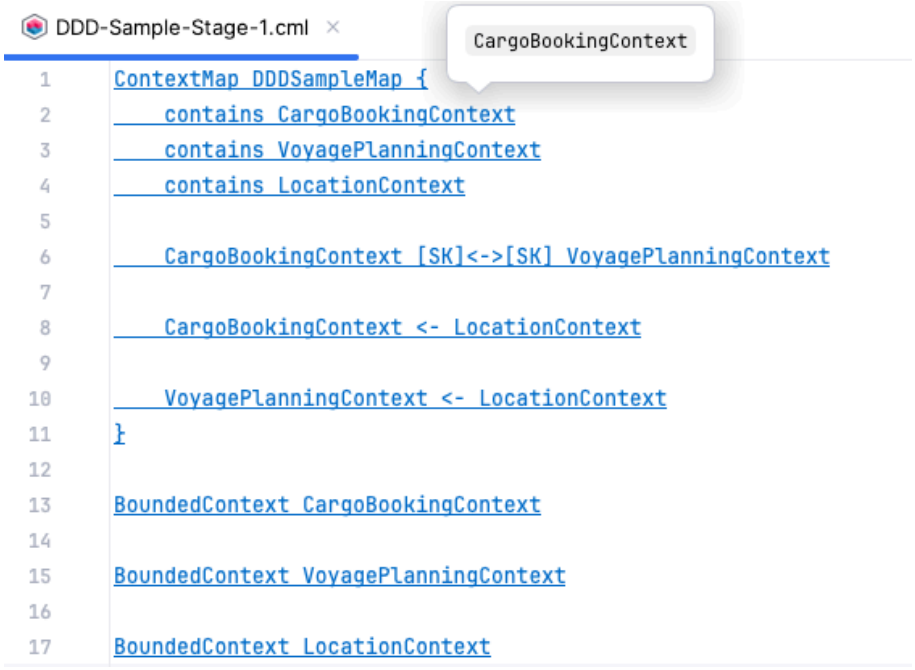

Figure 13: Hyperlinking after the initial LSP4IJ setup

The LSP4IJ plugin is responsible for displaying these hyperlinks. To resolve the issue, a configuration change had to be applied (see Appendix E.2.1).

4.3.4. Occurrence Highlighting (FR 1.3)

Occurrence highlighting worked out of the box. Placing the cursor in a Bounded Context name, for example, highlights all occurrences of the same Bounded Context in the file.

No custom configuration/implementation was necessary.

4.3.5. Autocomplete (FR 1.4)

The language server provides autocomplete suggestions without any special configuration or implementation [17]. However, the default implementation hides non-alphabetic keywords from autocomplete suggestions [28]. For CML, this means that Context Map relationship arrows like <-> are not automatically suggested by the language server.

An implementation change in the language server (see Appendix E.1.3) was required to change this behaviour.

An example for an autocomplete suggestion in the editor can be seen in Figure 14.

Figure 14: Screenshot of an autocomplete suggestion in the CML editor

4.3.6. Code Folding (FR 1.5)

Code folding required applying a configuration change in the IntelliJ plugin, see Appendix E.2.1. However, when collapsing a code block, it was impossible to see what CML element was hidden in the collapsed block (see Figure 15).

Figure 15: A collapsed Context Map after the initial LSP4IJ setup

By default, the language server provides folding ranges for code blocks [17]. A custom folding range provider was implemented to improve the folding ranges so that the first line of a block is still visible when collapsed (see Appendix E.1.4).

With invalid syntax embedded in a valid element, code folding cannot correctly determine the folding ranges since the language server cannot parse the whole document structure.

4.3.7. Keyword Tooltips (FR 1.6)

By default, the language server returns a JSDoc[1] comment directly preceding a language element (if available) as tooltip documentation [17]. In Langium, these tooltips are handled by so-called “hover providers”. Hover providers return documentation text for any given position in a document.

A custom hover provider (see Appendix E.1.5) was implemented to extend Langium’s default behaviour so that the keyword description is returned when a user hovers over a CML keyword.

[1] https://jsdoc.app/about-getting-started

Figure 16 shows the keyword tooltip for the `ContextMap` keyword.

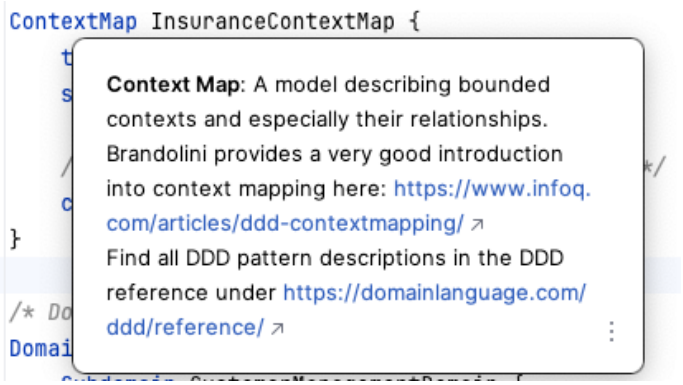

Figure 16: Screenshot of a keyword tooltip

4.3.8. Structure Outline (FR 1.7)

IntelliJ's structure view outlines the language elements in a file and relies on document symbols provided by the language server [21]. The language server automatically provides these symbols [17].

An adjustment in the LSP4IJ configuration was necessary for IntelliJ to display the received document symbols (see Appendix E.2.1).

An example of how a populated structure outline can look like is shown in Figure 17.

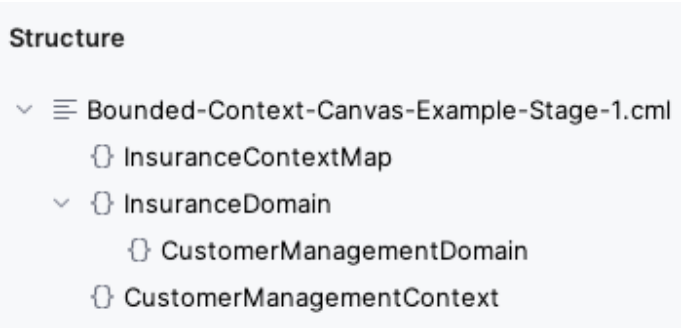

Figure 17: Screenshot of the IntelliJ structure tool window

4.3.9. Display Usages (FR 1.8)

LSP4IJ supports displaying the usages of a language element by default (see Figure 18). No customisation in the language server or the IntelliJ plugin was required.

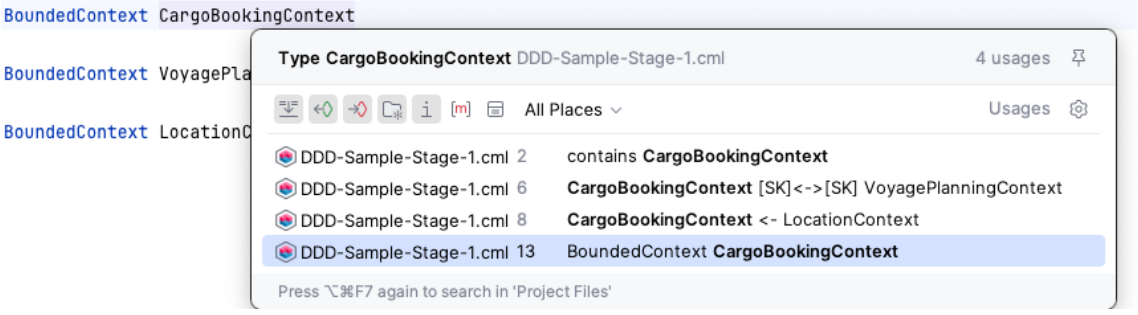

Figure 18: Screenshot of the "Find Usages" action

4.3.10. Document Formatting (FR 1.9)

LSP can transmit instructions for a development environment to reformat a file according to the rules defined in the language server [5]. Langium does not provide formatters by default [17]. Therefore, one had to be implemented.

As with semantic tokens and validation, specialised formatter classes were created to avoid a cluttered formatter implementation. For more details, see Appendix E.1.7.

4.3.11. Definition Tooltips (FR 1.10)

By default, the language server returns JSDoc comments preceding a language element as its documentation text [17].

A customisation of this type of tooltips was not made, but it would be necessary to extend this feature to regular multiline comments.

4.3.12. PlantUML Generator (FR 4.1)

The PlantUML generator in Context Mapper's current Xtext documentation generates different diagram types. The component diagram generation was picked for the PoC to showcase how a generator can be implemented.

The generator's implementation leverages the `workspace/executeCommand` [5] LSP capability. For a detailed explanation of the generator implementation in the language server, see Appendix E.1.8.

LSP4IJ supports the execution of LSP commands, but a generator "Action" had to be created, to display the generator in IntelliJ's editor popup menu and instruct LSP4IJ to execute the generator command. For more details, see Appendix E.2.2.

Figure 19 shows a generated component diagram.

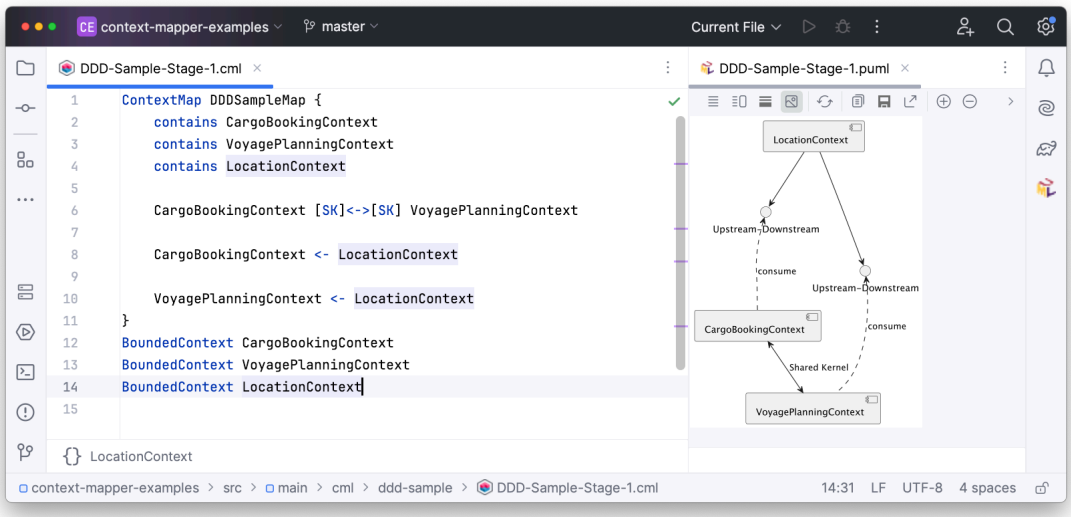

Figure 19: Screenshot of a generated PlantUML component diagram in IntelliJ

5. Results

This section evaluates the results of the project, including a review of which of the initially stated requirements have been implemented and what features remain unresolved. An experience report reflects on the technology evaluation and points out general issues encountered during the development of the PoC.

5.1. Fulfilment of Requirements

The thesis task description set the following goals that needed to be achieved:

1. Researching and analysing technical options to integrate a DSL into IntelliJ
2. A technology and architectural decision based on criteria defined in accordance with the thesis advisor
3. A prototype plugin implementation
4. A migration path for Context Mapper features that were not implemented in the plugin

Section 3 covers the first two goals. The prototype plugin was implemented as a PoC and documented in Section 4. Finally, a migration path for remaining features is outlined in Section 6.2.1. Therefore, the goals of this thesis could be achieved.

In addition to the general thesis goals, Section 2.2.1 and Section 2.2.2 defined requirements towards the PoC. Automated tests have been implemented to verify the correctness of the language server and plugin features. The language server includes 418, the IntelliJ plugin nine, automated tests. These tests include unit tests, component tests and ArchUnit tests to enforce separation of concerns. Manual tests were executed for the IntelliJ plugin (see Appendix F).

Table 6 overviews which requirements were fulfilled.

Requirement	Fulfilment status	Comment
FR 1.1 - Syntax highlighting	Fulfilled	
FR 1.2 - Hyperlinking	Fulfilled	
FR 1.3 - Occurrence highlighting	Fulfilled	
FR 1.4 - Autocomplete	Fulfilled	There are a few remaining open issues, see Section 5.2, but overall the requirement was fulfilled
FR 1.5 - Code folding	Fulfilled	
FR 1.6 - Keyword tooltips	Fulfilled	
FR 1.7 - Structure outline	Fulfilled	
FR 1.8 - Find usages	Fulfilled	
FR 1.9 - Document formatting	Fulfilled	
FR 1.10 - Definition tooltips	Partially fulfilled	see Section 5.1.2
FR 2.1 - Missing Bounded Context quick fix	Not fulfilled	see Section 5.1.1

Requirement	Fulfilment status	Comment
FR 3.1 - Generate visual Context Map	Not fulfilled	see Section 5.1.1
FR 4.1 - Generate PlantUML diagrams	Partially fulfilled	see Section 5.1.3
NFR 1 - Minimise duplicated code between IntelliJ and VSCode	Fulfilled	see Section 5.1.4
NFR 2 - Plugin stability	Fulfilled	see Section 5.1.5
NFR 3 - CML editor efficiency	Fulfilled	see Section 5.1.6
NFR 4 - PlantUML diagram generation efficiency	Fulfilled	see Section 5.1.7
NFR 5 - Generator extensibility	Fulfilled	see Section 5.1.8
NFR 6 - Code quality	Fulfilled	see Section 5.1.9
NFR 7 - Separation of concerns	Fulfilled	see Section 5.1.10
NFR 8 - IntelliJ compatibility	Fulfilled	see Section 5.1.11
NFR 9 - Licence compatibility	Fulfilled	see Section 5.1.12
NFR 10 - IntelliJ best practices	Fulfilled	see Section 5.1.13

Table 6: Fulfilment status of project requirements

5.1.1. Changed project scope

When the project requirements were defined, a new language server was not planned. The mid-project decision to create a new language server with Langium not only reduced the CML grammar supported by the PoC but also reprioritised the planned features. The following features were given a lower priority in favour of the PlantUML generator feature (FR 4.1):

- FR 2.1 - Missing Bounded Context quick fix
- FR 3.1 - Generate visual Context Map

Due to time constraints, these features could not be implemented during this thesis.

5.1.2. FR 1.10 - Definition tooltips

The current Eclipse extension of Context Mapper is capable of using regular multiline comments (`/*...*/`) as tooltip documentation for a CML element. The PoC can only use JSDoc comments (`/**...*/`) for tooltip documentation.

5.1.3. FR 4.1 - Generate PlantUML diagrams

The PlantUML generator in Context Mapper's current Eclipse & VSCode extension creates multiple diagram types. Due to time constraints, the component diagram generation was selected to showcase how a generator can be implemented in Langium. All remaining PlantUML diagram types have not been implemented.

5.1.4. NFR 1 - Minimise duplicated code between IntelliJ and VSCode

While the PoC could not directly reuse existing Context Mapper code, the feature logic was placed in the newly created language server. That way, a future VSCode extension can be built upon the language server without adding additional feature logic. The verification criteria were, therefore, fulfilled.

5.1.5. NFR 2 - Plugin stability

The language server has been extensively tested with automated unit & component tests executed as part of the CI pipeline. Where possible, unit tests have been created for the IntelliJ plugin. The plugin stability was also observed during manual tests.

5.1.6. NFR 3 - CML editor efficiency

No performance issues were discovered during manual tests and an informal user test with the thesis advisor.

5.1.7. NFR 4 - PlantUML diagram generation efficiency

An automated performance test has been created in the language server project that verifies the creation of a PlantUML component diagram within the set time constraint. The test is executed as a part of the CI pipeline.

5.1.8. NFR 5 - Extensibility for additional generators

A new generator can be added to the plugin using one extension point each. First, the generator needs to be registered with the command handler in the language server. A corresponding generator action then has to be registered within the Context Mapper action group in the plugin.xml of the IntelliJ plugin.

A dummy generator was added in a branch of the language server[1] and IntelliJ plugin[2] repositories, to showcase their extensibility.

5.1.9. NFR 6 - Code quality

JetBrains Qodana has been set up to analyse code in the language server and IntelliJ plugin CI pipeline. At the time of the thesis submission, there were no open problems in the IntelliJ plugin code. As Figure 20 shows, Qodana reports one open problem for the language server regarding the test coverage of the `main.ts` file. This problem was accepted since the code in `main.ts` is equivalent to the standard script generated when creating a Langium project.

[1] https://github.com/lstreckeisen/context-mapper-language-server/compare/dummy-generator
[2] https://github.com/lstreckeisen/context-mapper-intellij-plugin/compare/dummy-generator

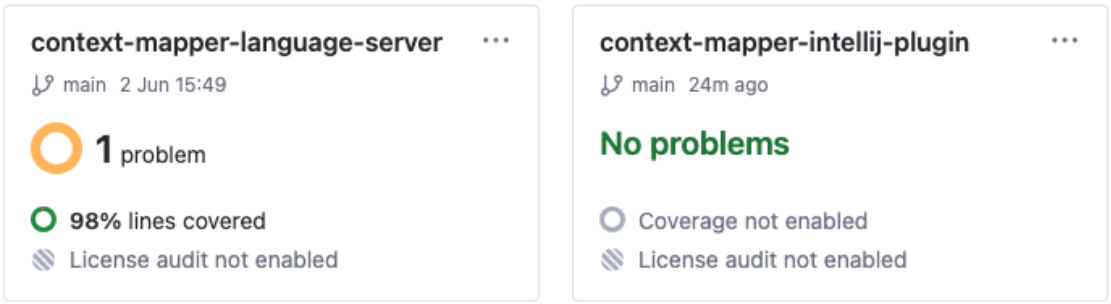

Figure 20: Screenshot of Qodana scan results after the PoC was finalised

5.1.10. NFR 7 - Separation of concerns

To enforce separation of concerns, ArchUnit tests have been created. These tests check that features in a top-level package/folder do not depend on code in a sister-package/folder. The ArchUnit tests are executed as part of the CI pipeline.

5.1.11. NFR 8 - IntelliJ compatibility

A manual compatibility check with the IntelliJ versions 2024.3.5 and 2025.1.1.1 was performed. The plugin could be installed in both versions.

5.1.12. NFR 9 - Licence compatibility

JetBrains Qodana is capable of performing licence compatibility checks for project dependencies. However, due to an unknown issue, this feature did not work. Therefore, the licences of all used libraries were checked for compatibility according to Apache's third-party licence policy [13]. All dependency licences were found compatible.

5.1.13. NFR 10 - IntelliJ best practices

Most features in the PoC IntelliJ plugin are provided through the LSP4IJ plugin. LSP4IJ and the added generator conform to the IntelliJ best practices.

5.2. Open Issues

The informal user test with the thesis advisor led to the discovery of a few issues with the PoC implementation that could not be resolved as of the end of this thesis. These issues are outlined below.

5.2.1. Brace Matching

IDEs usually automatically complete a brace pair when typing "{" in the editor. IntelliJ also has this capability, but custom languages such as CML require a custom brace pair matcher [21]. A brace pair matcher defines brace, bracket and parenthesis pairs using lexer tokens, which requires the IntelliJ plugin to have a lexer.

To resolve this issue in a future project, a lexer could be created. For the purpose of this feature, it would be enough for the lexer to be capable of recognizing brace pairs and not the full CML grammar.

5.2.2. Autocomplete

There are two known cases where autocomplete is not yet on the same level as in the VSCode extension.

1. When defining a Context Map relationship, autocomplete can only make suggestions for keywords within brackets after the user starts to type
2. The VSCode extension suggests placeholder values, e.g. for a stakeholder description, which benefits users unfamiliar with the CML grammar.

Modifications to the completion provider in the language server should resolve these issues.

5.2.3. IntelliJ Plugin Testing

The developed IntelliJ plugin is currently only tested with unit tests. However, integration tests would be necessary for the CI pipeline to detect an issue in the plugin.

Attempts were made to set up integration tests according to the JetBrains documentation [21], but they broke the existing unit tests. As of the end of this thesis, the source of this issue has not yet been discovered and requires further investigation.

5.3. Experience Report

This experience report reflects on the technology decision and highlights both positive and negative experiences during the PoC development.

As described in the technology evaluation, Langium is well-designed and generally easy to understand for developers. However, the documentation [17] had more gaps regarding implementation details than anticipated. The documentation appears to be heavily VSCode-oriented and omits supported LSP capabilities that are not required in VSCode. Syntax highlighting is a good example of that. In VSCode, syntax highlighting is provided through TextMate[1] and does not require semantic tokens. IntelliJ, on the other hand, needs semantic tokens to highlight source files. Since the Langium documentation does not provide guidance on implementing a semantic token provider, public GitHub repositories that also use Langium had to be consulted instead.

On the IntelliJ side, the young age of the LSP4IJ plugin became apparent. Syntax highlighting was very unstable up to version 0.13.0, which was released towards the end of the PoC implementation. Without a fix in version 0.13.0, the Context Mapper IntelliJ plugin would have been unusable for end users.

During the PoC implementation, both the Langium and LSP4IJ maintainers have been contacted for help with minor issues. The interaction with both maintainers proved to be very pleasant, with quick and constructive responses.

Based on the experiences collected during this thesis, the technology decision can be confirmed. Langium is currently the most capable alternative to Xtext and works very well once it becomes clear how to implement certain LSP features. LSP4IJ, apart from

[1] https://macromates.com/manual/en/language_grammars

the syntax highlighting issues, worked very well. It is currently also the most effective option for integrating a language server in IntelliJ.

6. Outlook

This section identifies possible long-term risks with the newly selected technologies and outlines a migration path for Context Mapper features that have not been implemented in the PoC.

6.1. Long-Term Risk Analysis

To analyse the risks involved with using Langium and LSP4IJ in the long term, the identified risks are evaluated using the factors “probability” and “severity”. The risk matrix in Table 7 describes the scale for both factors and assigns colour codes.

Probability / Severity	1 Almost Impossible	2 Unlikely	3 Moderate	4 Likely	5 Almost Certain
4 Catastrophic					
3 Critical					
2 Major					
1 Minor					

Table 7: Risk matrix

Section 6.1.1 and Section 6.1.2 highlight the risks associated with Langium and LSP4IJ, respectively, and discuss their mitigation. A conclusion on the associated risks can be found in Section 6.1.3.

6.1.1. Langium

Table 8 contains the identified long-term risks for Langium.

ID	Risk	Probability	Severity	Reasoning
RSK-1	Maintainers stop contributing and the project is abandoned	2	3	Langium is backed by TypeFox[1]. TypeFox maintained Xtext for more than 10 years and decided to start fresh with Langium. It cannot be ruled out that this will happen again, but given Langium’s age of 3 years, this seems unlikely at this time.

[1] https://www.typefox.io/language-engineering/

ID	Risk	Probability	Severity	Reasoning
RSK-2	Documentation is missing vital information for a future feature	4	2	Based on the experiences made during the PoC development, the information needed for a future feature is likely missing from the documentation. Gaps in the documentation impact the time required to implement the feature but not the feature's technical feasibility.
RSK-3	Breaking changes are introduced	4	2	The introduction of breaking changes in future versions of Langium is likely based on past releases[1]. Breaking changes would increase the effort required to upgrade to the next version but should not have a bigger impact.

Table 8: Identified long-term risks for Langium

The possibilities to mitigate the mentioned risks are limited. To mitigate the impact of breaking changes, it could be attempted to abstract feature logic in a way that limits dependencies on Langium to a few classes. However, this might not be easy to achieve. To mitigate the risk of abandonment, DSL developers could be encouraged to make code or financial contributions to the Langium project.

[1]https://github.com/eclipse-langium/langium/blob/main/packages/langium/CHANGELOG.md

6.1.2. LSP4IJ

Table 9 shows the identified long-term risks for the LSP4IJ plugin.

ID	Risk	Probability	Severity	Reasoning
RSK-4	Maintainers stop contributing and the project is abandoned	3	3	LSP4IJ is maintained by the Red Hat developer community and is approximately one year old as of the end of this thesis. Among other things, the future of LSP4IJ depends on its user base. The plugin is currently used by 27 other plugins[1] and has 155′000 downloads in the JetBrains marketplace[2]. At this time, it is unlikely that the plugin will be abandoned, but it cannot be ruled out.
RSK-5	LSP4IJ prevents plugin support for latest IntelliJ version	3	3	If LSP4IJ becomes incompatible with the latest version of IntelliJ, the Context Mapper plugin cannot be installed with that version of IntelliJ either. A situation like that could be caused by a breaking change in the IntelliJ Platform and Plugin API, which does tend to happen[3]. As of the end of this thesis, LSP4IJ had a fast release cycle and supports IntelliJ versions as of 2023.2.

[1] https://github.com/redhat-developer/lsp4ij/blob/main/README.md
[2] https://plugins.jetbrains.com/plugin/23257-lsp4ij
[3] https://plugins.jetbrains.com/docs/intellij/api-changes-list.html

ID	Risk	Probability	Severity	Reasoning
RSK-6	Breaking changes are introduced	3	1	As of the end of this thesis, there are no known breaking changes up to version 0.13.0. Since the PoC primarily configures LSP4IJ, a breaking change would likely necessitate the use of a new or different configuration option.

Table 9: Identified long-term risks for LSP4IJ

As with Langium, the best way to mitigate the abandonment of LSP4IJ is to encourage developers to make contributions and to promote the plugin. The mitigation of the Context Mapper plugin being blocked from running in the latest IntelliJ versions is, due to the nature of its dependency on LSP4IJ, not possible. In the event of breaking changes to the LSP4IJ configuration, the required adjustments should require minimal effort and, therefore, do not require mitigation.

6.1.3. Conclusion

Software dependencies always carry risks when used. They lead to a loss of control over code that is sometimes essential to a project. However, it is not always an option to replace a dependency with your own code, as the required effort would be too high. The use of dependencies, therefore, is always a tradeoff between the associated risks and the benefits of using them [29].

Langium, as a language workbench, is an essential part of a DSL project. The effort required to switch to a different language workbench or implement a language server from scratch would be very high, and any issues related to the language server also impact IDE plugins that are based on it. With Xtext seemingly reaching the end of its life, Langium is currently the most capable option for building a language server. There are risks associated with using Langium, but they are at an acceptable level.

LSP4IJ bears fewer risks than Langium, as its role as an integration option is not as essential as with a language workbench. The risks involved in using LSP4IJ are also on an acceptable level. In case LSP4IJ becomes an unreliable option, a fallback to IntelliJ's LSP support is always possible, with the consequence that the plugin becomes unavailable in IntelliJ Community versions. It is also possible that JetBrains will add more LSP capabilities to IntelliJ in the future and make LSP support available in IntelliJ's Community version. In that case, a switch to JetBrains' LSP support would make sense from a risk perspective.

6.2. Future Work

This section describes how not-implemented Context Mapper features can be migrated to the technology stack used in the PoC. Given the complexity of migration, a future project is proposed.

6.2.1. Migration of Remaining Context Mapper Features

The following ContextMapper features have not been implemented in the PoC.

- Quick fixes
- Architectural refactorings
- Validators (other than the ones implemented due to grammar differences)
- Generators
  - PlantUML (other than component diagram)
  - Visual Context Map
  - Sketch Miner
  - MDSL contract
  - FreeMarker
- Discovery
- ArchUnit extension

Grammar Completion

The developed PoC does not include the grammar elements for tactic DDD and imports. To offer the complete set of Context Mapper features, the grammar in the language server must be fully implemented. The Xtext grammar for the remaining grammar elements needs to be adjusted according to the Langium grammar differences described in Section 4.2.

Quick Fixes and Architectural Refactorings

Since quick fixes and architectural refactorings are offered in the same way in the editor, their implementation follows the same path. The LSP handles quick fixes via “code actions” [5]. The language server computes available code action commands for a document and returns them to the editor. The execution of code actions is handled via the `workspace/executeCommand` request, which is already used in the PoC to execute generator actions.

In the language server, code actions are provided by a `CodeActionProvider`, which needs to be implemented to offer quick fixes and architectural refactoring actions in the editor [17]. Additionally, a command must be registered in the `ContextMapperCommandHandler` for every quick fix and refactoring.

In the IntelliJ plugin, it should not be necessary to make customisations for these features [22].

The quick fixes and refactorings implemented in the Xtext version of Context Mapper manipulate the AST and do not have any dependencies on other libraries [8]. Their reimplementation in TypeScript is, therefore, unproblematic.

Validators
The PoC showcased the implementation of semantic validation. The validators implemented in Xtext do not depend on any non-Xtext libraries [8] and can, therefore, be rewritten in TypeScript without any issues.

Generators
The partially implemented PlantUML generator shows how generators can be implemented in a Langium language server. The PlantUML generator can be completed in the same manner as the component diagram implementation in the PoC.

The Context Map generator in the Xtext language server depends on the `graphviz-java`[1] library. A potential TypeScript alternative is `ts-graphviz`[2]. If this library proves insufficient to replace `graphviz-java`, the Context Map diagram can be generated by creating a `.dot` file - similar to the approach for the PlantUML diagrams - and then using the `graphviz` CLI (Command Line Interface) tool to generate the desired output formats.

The migration of the FreeMarker, MDSL, and Sketch Miner generators, which all utilise FreeMarker[3] templates, requires more effort. There are currently no maintained Node.js libraries for FreeMarker, which prevents a straightforward migration to Langium. To work around that, a Java-based CLI tool could be created that reads a CML file and renders a FreeMarker template. The language server or the IDE plugin would then call the CLI tool to generate resources based on the CML models defined by users. These generators therefore still require a Java library to parse CML files.

Discovery
The Context Mapper discovery feature, which is separate from the current Xtext language server, utilises the Context Mapper standalone library to translate discovered Bounded Contexts and their relationships into CML. Since the discovery library specifically targets Java projects, it still requires a Java library to create CML files.

ArchUnit Extension
The Context Mapper ArchUnit extension can be used to check if the model defined in CML is reflected in a Java codebase. To do that, the ArchUnit extension relies on the Context Mapper standalone library.

There is an ArchUnit implementation for TypeScript: `ts-arch`[4]. `ts-arch` would allow writing ArchUnit tests based on a CML model, but the library cannot target Java code. The Context Mapper ArchUnit extension, therefore, still requires a Java library for parsing CML.

[1] https://github.com/nidi3/graphviz-java
[2] https://github.com/ts-graphviz/ts-graphviz
[3] https://freemarker.apache.org/
[4] https://github.com/ts-arch/ts-arch

Context Mapper Java Library

Context Mapper should continue to support Java developers, as it does today, by offering an ArchUnit extension and a standalone library for CML. To do that, a Java parser for CML will still be required in the future.

There are currently plans for Langium to refactor its parsing engine[1], allowing parsers other than Chevrotain to be used. These plans also include offering an ANTLR parser out of the box.

With an ANTLR grammar as the base for the Langium language server, a Java parser could be generated. This parser could then be used for a standalone Java library, the discovery library and the ArchUnit extension. A CLI tool based on that ANTLR parser could implement the Context Mapper generators. This approach would allow partial reuse of the existing implementations for the discovery library, ArchUnit extension and generators.

However, it is to be expected that an ANTLR parser will not be available in Langium for some time. In the meantime, building the Java libraries on an ANTLR parser is still possible by using separate grammars for the Java tools and Langium. This separation would require maintaining both grammars until Langium supports the ANTLR parser, at which point the Langium grammar could be discarded.

6.2.2. Suggestion for a Future Project

Given the need for a Java library that allows automated parsing and writing of CML, an additional project is required to provide this library. One possible implementation path is to use ANTLR alongside Langium, as mentioned above. Another option is to build a language server from scratch on the JVM (Java Virtual Machine) stack. A JVM-based language server would have the advantages of having a single technology base and being able to reuse existing feature implementations more easily. The parser from the language server could then be reused for a standalone library as well.

The project would need to assess the trade-offs between building a custom language server and going ahead with Langium, as well as evaluate the risks associated with using ANTLR alongside Langium.

[1] https://github.com/eclipse-langium/langium/issues/1742

7. Conclusion

This thesis analysed the Context Mapper features included in the existing Context Mapper implementation. An overview of current language workbenches and integration options to add custom language support in IntelliJ was provided. A technology selection based on a utility analysis was made, resulting in the use of Langium and LSP4IJ. A PoC was implemented, showcasing how Context Mapper can be integrated into IntelliJ and how Xtext can be replaced as the base for the language server. A risk analysis identified long-term risks associated with the PoC technologies and concluded that there are no uncommonly significant risks associated with them. Finally, a migration path for expanding the PoC to the complete Context Mapper feature set was outlined. Since Context Mapper should continue to offer Java libraries, not all features can be based on the Langium sources from the PoC. An additional project is required to provide a new Java library for reading and writing CML models, so Context Mapper's existing Java tools, such as the ArchUnit extension, can be migrated as well.

Part II - Appendix

A: Task Description

A.1: Initial Situation

Context Mapper (contextmapper.org) is an open source modelling tool and framework that uses a Domain-Specific Language (DSL) to enable the modelling of software systems based on Domain-Driven Design (DDD) patterns. Various artefacts such as context maps, PlantUML diagrams and interface contracts can be generated from the models.

The Context Mapper is currently available as an Eclipse plugin and as a Visual Studio Code (VSCode) extension. There is also a CLI (Command Line Interface), standalone Java library for generating Context Mapper DSL (CML) models using code, a discovery library for generating models from existing source code, and an ArchUnit extension for comparing code and model. Technologically, the entire framework is implemented in Java and uses the Xtext framework (eclipse.dev/Xtext) for language engineering. Integration into VSCode is realised via the so-called Language Server Protocol (LSP). The Xtext framework automatically generates an LSP server based on the grammar of the CML language. The VSCode extension of Context Mapper uses this LSP server as a “backend”.

Since many Java developers today use IntelliJ IDEA from JetBrains, they would like to see Context Mapper integrated into this Integrated Development Environment (IDE). The lack of such integration means that software developers have to use VSCode in addition to IntelliJ IDEA to edit CML models, or that this hurdle is perceived as too high and Context Mapper is therefore not used.

A.2: Goals and Deliverables

The lack of a Context Mapper integration for IntelliJ IDEA makes it difficult to spread the modelling tool, especially in the Java community, where IntelliJ has established itself as the standard IDE. In addition, the current technology stack based on the Xtext framework is getting older and maintainability is not ideal. The spread of Eclipse has declined in recent years and the Xtext framework is no longer really developed further.

The main objective of this thesis is to demonstrate through a prototype how the Context Mapper DSL (CML) language and its tools can be integrated into IntelliJ IDEA - as a plugin.

The first step is to analyse and research the options available for this. Integration on the basis of the existing Xtext/LSP stack is only one possible variant. Within the scope of this work, an open overview of the possibilities, independent of the existing technology, is to be created.

After a technology and architecture decision has been made (according to criteria to be defined), a prototype (IntelliJ plugin) is to be implemented. The plugin should offer the basic functionalities of the CML-language in a corresponding editor and integrate one or two of the existing generators. For the other existing Context Mapper features,

it should be shown how these can be implemented or migrated in the new plugin (no complete implementation required as part of the prototype).

The critical success factors for this work are defined as follows:
1. An analysis and research shows which technical options are available for integrating the CML-language and the other Context Mapper tools into IntelliJ.
2. A well-founded technology and architecture decision is made on the basis of jointly defined criteria (with the supervisor).
3. A prototype of a plugin shows how Context Mapper and its CML language can be integrated into the IntelliJ IDEA IDE.
4. For existing Context Mapper features that are not implemented (in the prototype), it is at least analysed how they can be integrated or implemented later.
5. The report of the bachelor thesis documents the analysis (technology research, etc.), the decisions for the prototype (including the underlying criteria), the implementation of the prototype, and gives an outlook on how the integration can be completed in subsequent projects.

A.3: Support

The expected and effectively received support is recorded by the student.

A.4: Project Execution

The Bachelor's thesis is about applying the knowledge learned in the various OST modules to a project. In particular, software engineering skills will be required. Students are expected to apply this knowledge and use methods such as unit testing, clean code, SCM and continuous integration wherever possible. The usability of the results should also be checked using suitable means and representatives of the target group.

The preliminary study, requirements documentation and architecture documentation should be approved in a stable state during the course of the project by means of milestones with the client and supervisor. Preliminary feedback is given on the submitted work results. A definitive assessment is made on the basis of the documentation delivered by the deadline.

The rights to the results of the Bachelor's thesis are defined in a separate agreement (report public, no non-disclosure agreement required).

The requirements are specified by the student in consultation with the supervisor. In the event of disputes, the supervisor decides on the definitive requirements relevant to the Bachelor's thesis in consultation with the student.

As a rule, weekly meetings are held with the supervisor (meeting at OST or video conference). Additional meetings are to be arranged as required. All meetings that require preparation by the supervisor must be prepared by the student with an agenda. This must be sent to the supervisor at least half a working day in advance. Decisions must be documented in minutes and then sent to the supervisor or filed in a defined location (e.g. wiki). A project plan must be drawn up for the realisation of the

work. Attention must be paid to continuous and visible work progress. Working hours must be documented.

A.5: Tools

Unless specified in the assignment, students are responsible for selecting their own tools, libraries, frameworks, etc.

A.6: Documentation

This work must be documented in accordance with the guidelines of the Department of Computer Science (regulations and instructions in MS Teams of the Computer Science programme). The documents to be created must be recorded in the project plan. All documents must be kept up to date, i.e. they should document the status of the work in a consistent form at the time of submission. Time records must be kept and analysed in the report.

B: Technology Exploration Tests

This appendix contains a brief documentation of tests performed with the technologies from the technology elaboration phase.

B.1: JetBrains MPS

The introduction course for MPS[1] uses the ChemMastery sample project included in MPS. Each MPS project comes with a sandbox to test the language and its definition. The structure of the language definition is clear and easy to understand (Figure 21).

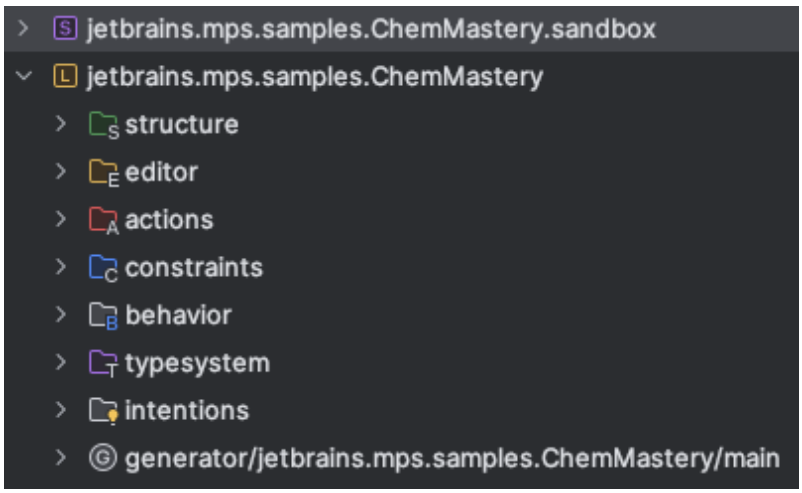

Figure 21: Structure of an MPS language definition

The structure and editor definitions are most important to the language. The structure definition consists of so-called “concepts”, which represent the different structural elements of the language. Each concept can define properties, possible child elements and references to other concepts (Figure 22).

[1] https://cogniterra.org/course/28

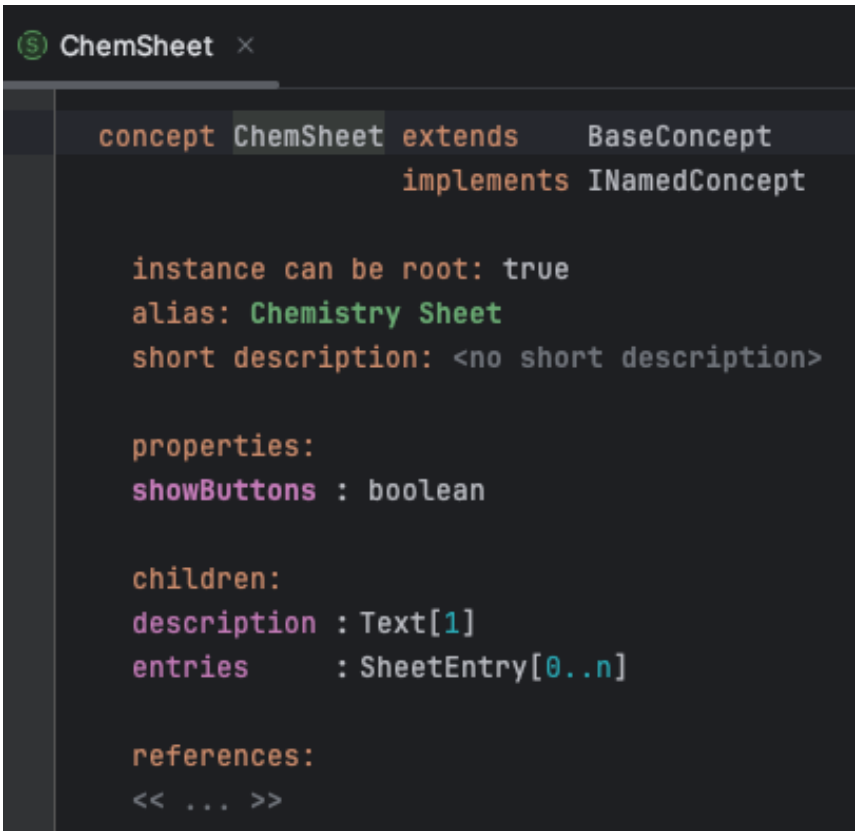

Figure 22: Example of an MPS concept definition

Each language concept has its own editor. The editor defines how a language concept is presented to the user and how the language user provides information to create an instance of the concept. The editor can use textual or graphical elements for language users to provide the required information. However, the resulting editor always includes some graphical elements and is never just plain text. Figure 23 shows an example of an editor definition that uses Java Swing elements. An example of the resulting editor presented to the language user can be seen in Figure 24.

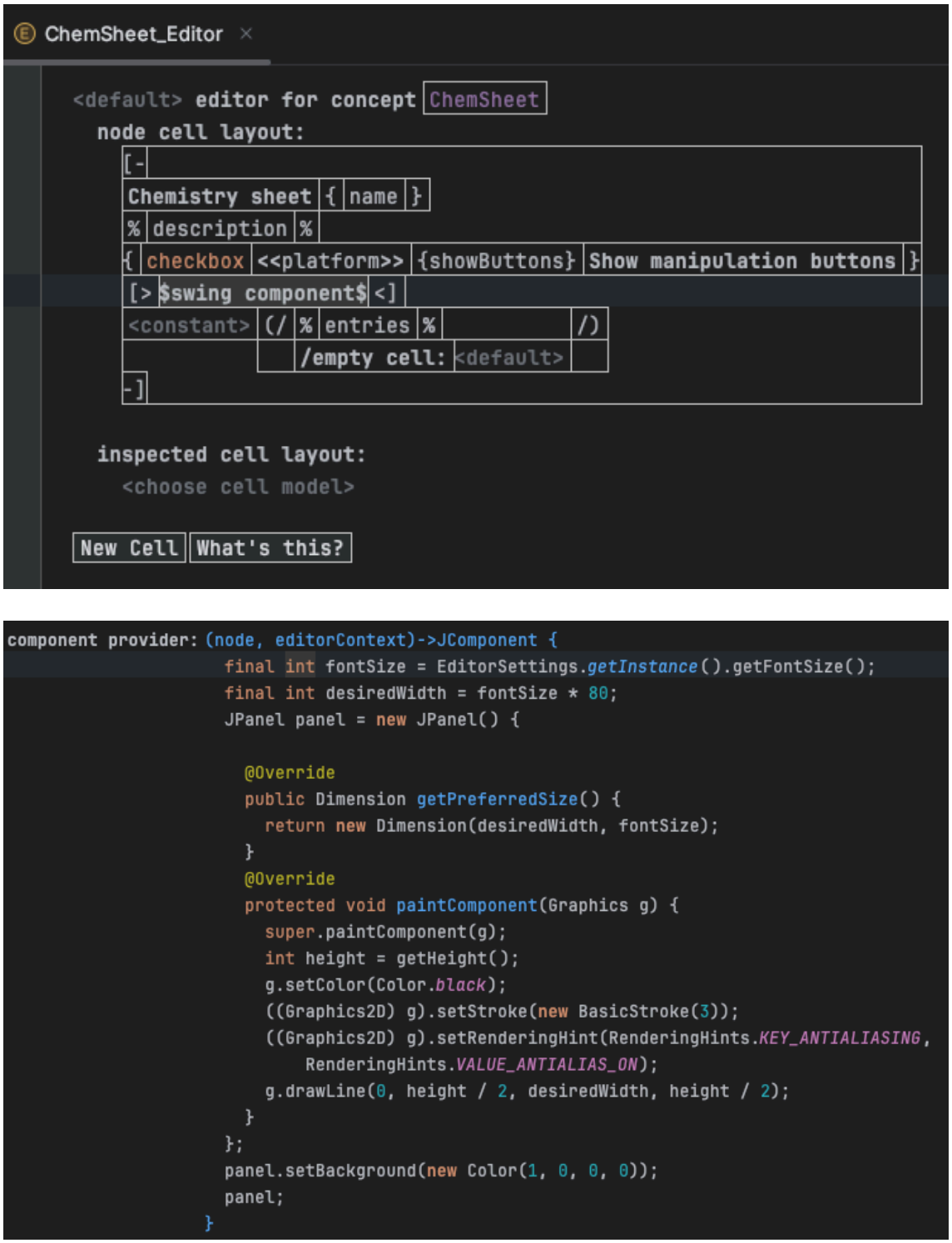

Figure 23: MPS editor definition using graphical elements

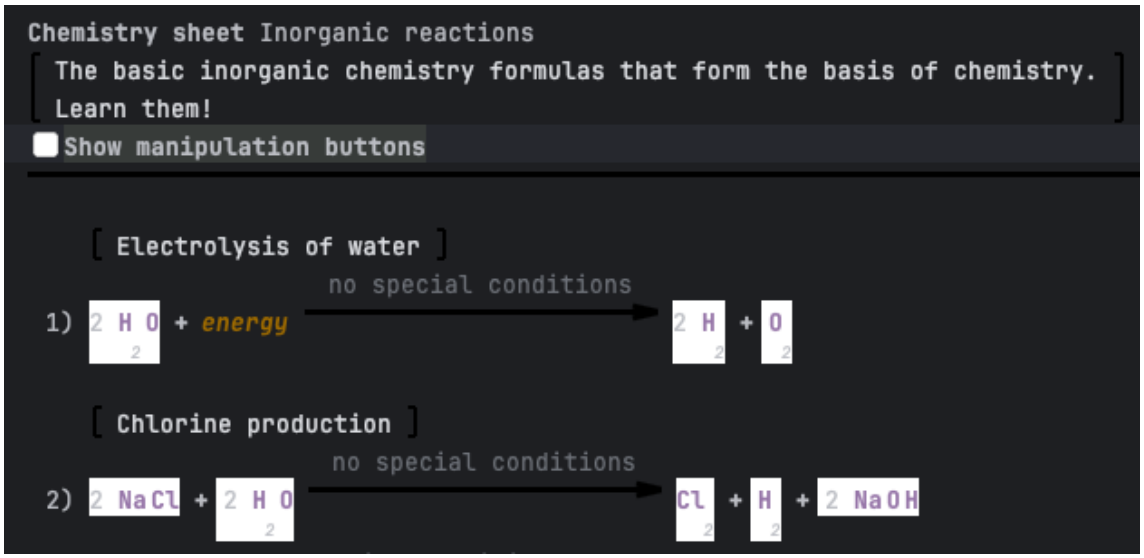

Figure 24: Example of an MPS editor presented to the language user

The resulting MPS plugin can be installed in the MPS IDE itself and used there in a solution project. The plugin could also be installed in a 2023 version of IntelliJ. However, the ChemMastery language did not appear anywhere in IntelliJ.

B.2: Langium

A basic example of a Langium project can be generated using Yeoman[1]. The project can be generated with a basic setup of a VSCode extension, a CLI and the language itself.

The grammar definition in Listing 9 is a bit easier to understand for beginners than the Grammar-Kit grammar (see Appendix B.5), but that is mainly because Langium does not include configuration for a program structure interface.

```
grammar TestLang

entry Model:
    (persons+=Person | greetings+=Greeting)*;

Person:
    'person' name=ID;

Greeting:
    'Hello' person=[Person:ID] '!';

hidden terminal WS: /\s+/;
terminal ID: /[_a-zA-Z][\w_]*/;
terminal INT returns number: /[0-9]+/;
terminal STRING: /"(\\.|[^"\\])*"|'(\\.|[^'\\])*'/;

hidden terminal ML_COMMENT: /\/\*[\s\S]*?\*\//;
hidden terminal SL_COMMENT: /\/\/[^\n\r]*/;
```

Listing 9: Langium grammar of the generation getting started example

For a VSCode extension, there is also some configuration for syntax highlighting and comment & bracket tokens. Depending on the language, other extensions of the basic

[1] https://yeoman.io/

setup, e.g. for validators, are required, but the basic setup remains and provides the language server and VSCode extension separately.

B.3: IntelliJ LSP

Setting up an LSP integration via IntelliJ's LSP support requires little effort. Figure 25 shows the file structure required for the setup. IntelliJ calls the `CMLLspServerSupportProvider` when a file is opened. If the file has the correct file extension, it is responsible for starting the language server. The `CMLLspServerDescriptor` contains the start command for the language server and is passed to IntelliJ's server starter by the `CMLLspServerSupportProvider`.

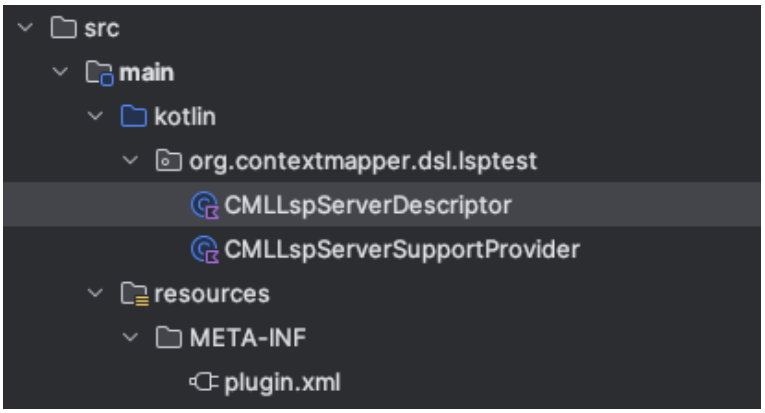

Figure 25: File structure of the basic IntelliJ LSP setup

Finally, a single entry in the `plugin.xml` completes the LSP setup. In a test with the existing CML language server, features like autocomplete worked, but others, like syntax highlighting and code folding, did not.

B.4: LSP4IJ

To assess LSP4IJ's capabilities, its UI config was used over its plugin API. Figure 26 shows the configuration necessary to start the language server.

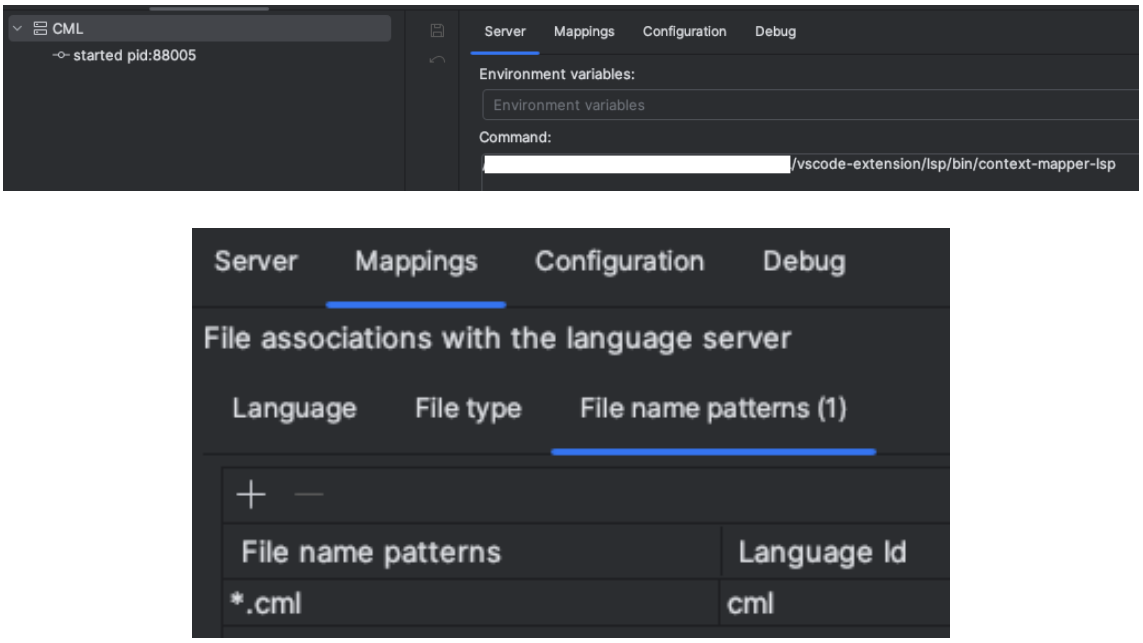

Figure 26: LSP4IJ test UI config

The test showed the same results as with IntelliJ's LSP support.

B.5: Native Integration

The tutorial from JetBrains' documentation [21] guides through the major editor features. For the "Simple" language, which is basically a key-value mapping, the tutorial guides through the creation of 48 classes and a lexer and parser definition. The tutorial uses JFlex[1] to generate the lexer and Grammar-Kit[2] to generate the parser.

While the implementation of editor features can be understood relatively quickly, the JFlex and Grammar-Kit definitions require more knowledge, especially when the grammar is more complicated than a simple key-value mapping. Listing 10 shows the grammar definition for the Simple language. The basic grammar is based on the Parsing Expression Grammar[3] and extended with global and rule attributes [30]. In addition to the grammar itself, Grammar-Kit also includes configuration for the PSI integration of the generated parser.

```
{
  parserClass="ch.streckeisen.intellijtestplugin.parser.SimpleParser"
  extends="com.intellij.extapi.psi.ASTWrapperPsiElement"
  psiClassPrefix="Simple"
  psiImplClassSuffix="Impl"
  psiPackage="ch.streckeisen.intellijtestplugin.psi"
  psiImplPackage="ch.streckeisen.intellijtestplugin.psi.impl"

elementTypeHolderClass="ch.streckeisen.intellijtestplugin.psi.SimpleTypes"

elementTypeClass="ch.streckeisen.intellijtestplugin.psi.SimpleElementType"
  tokenTypeClass="ch.streckeisen.intellijtestplugin.psi.SimpleTokenType"

psiImplUtilClass="ch.streckeisen.intellijtestplugin.psi.impl.SimplePsiImplUtil"
}
simpleFile ::= item_*
private item_ ::= (property|COMMENT|CRLF)
property ::= (KEY? SEPARATOR VALUE?) | KEY {
  pin=3
  recoverWhile="recover_property"

mixin="ch.streckeisen.intellijtestplugin.reference.SimpleNamedElementImpl"

implements="ch.streckeisen.intellijtestplugin.reference.SimpleNamedElement"
  methods=[getKey getValue getName setName getNameIdentifier
getPresentation]
}
private recover_property ::= !(KEY|SEPARATOR|COMMENT)
```

Listing 10: Grammar definition of the Simple language from the custom language plugin tutorial

[1] https://www.jflex.de/

[2] https://github.com/JetBrains/Grammar-Kit

[3] https://en.wikipedia.org/wiki/Parsing_expression_grammar

C: Detailed Technology Evaluation

This appendix contains the detailed utility analysis performed during the technology evaluation.

C.1: JetBrains MPS

Future Proofing

Score 3

Weighted Score 9

Reasoning

MPS is maintained by JetBrains and receives regular updates. It is unlikely that it will be discontinued anytime soon.

Ease of use

Score 1

Weighted Score 2

Reasoning

According to JetBrains, "(…) there are easier tasks in the world than learning MPS." [15] A Stack Overflow post [31] also confirms that. When one already understands how a DSL is created, MPS concepts are certainly easier to understand, but even then, there is a learning curve because projectional DSLs just work differently. For languages that are a bit more complicated than, e.g. the chemmastery sample language (included in the MPS IDE), an understanding of the MPS DSL is required.

Documentation

Score 2

Weighted Score 4

Reasoning

A Stack Overflow post [31] mentioned that the documentation is challenging for beginners, at least in 2010. The documentation [15] seems to have improved since then. The documentation as of March 2025 is quite extensive and includes a link to a getting started course[1] that explains the core concepts of MPS quite well.

Feature Support

Score 1

Weighted Score 3

Reasoning

[1] https://cogniterra.org/course/28

MPS does have equivalents for all defined requirements but in a very different way. Projectional editing cannot read or modify a `.cml` file created with the VSCode extension of Context Mapper. Using MPS would be a breaking change to all existing CML definitions.

Licence

Score 3

Weighted Score 9

Reasoning

MPS is licensed under Apache 2.0.

IDE Compatibility

Score 1

Weighted Score 3

Reasoning

MPS is a JetBrains product and, therefore, only usable with JetBrains products. Language plugins are MPS plugins intended for use in the MPS IDE. With the “MPS Core” plugin, MPS language plugins can also be installed in IntelliJ. However, as of March 2025, the “MPS Core” plugin does not yet support the 2024 versions of IntelliJ [32].

Reusability

Score 1

Weighted Score 2

Reasoning

Using MPS requires a complete rewrite of Context Mapper.

C.2: Langium

Future Proofing

Score 2

Weighted Score 6

Reasoning

Langium gets frequent updates, and version 3.3.0 (November 2024) marked the milestone of Langium becoming a mature project [33]. Langium was created by TypeFox[1] and had about 20 active contributors in 2024, with the main contributions made by about five people [34]. At this point, there are no signs of difficulties in maintaining the project.

[1] https://www.typefox.io

Ease of use

Score	2
Weighted Score	4

Reasoning

The tutorial in the Langium documentation [17] shows that creating a language server is straightforward. The creation of advanced languages will require studying the Langium grammar language.

Documentation

Score	3
Weighted Score	6

Reasoning

The documentation [17] is clear, covers all major features, and includes a complete description of the grammar language and API. There is also a tutorial on getting started and a more advanced example that guides through the creation of validators, generators, and a VSCode extension.

Feature Support

Score	3
Weighted Score	9

Reasoning

Langium leverages LSP. While the documentation [17] does not list supported LSP capabilities, it indicates that all editor requirements are supported. Features that are not supported by LSP can be added through native integration.

Licence

Score	3
Weighted Score	9

Reasoning

Langium is licensed under the MIT[1] licence. According to the Apache 3rd Party Licence Policy [13], software under the MIT licence may be included in an Apache 2.0 project without restrictions.

IDE Compatibility

Score	3
Weighted Score	9

Reasoning

By leveraging LSP, langium supports a wide range of IDEs, including VSCode and IntelliJ.

[1] https://opensource.org/licence/mit

Reusability

Score 1

Weighted Score 2

Reasoning

Langium is written in TypeScript. Though it is possible to still use the existing Context Mapper code by delegating parts of the logic to a separate Java process, that solution would not be ideal. Therefore, a complete rewrite would be possible. The resulting language server can then be reused for different IDEs.

C.3: Rascal

Future Proofing

Score 2

Weighted Score 6

Reasoning

The organisation UseTheSource[1] coordinates contributions to Rascal. In 2024, 5 people actively contributed, with the main contributions made by three people [35].

Ease of use

Score 2

Weighted Score 4

Reasoning

Defining a grammar syntax is not complicated, especially if one is familiar with Haskell, as Rascal is similar to Haskell in syntax. However, the documentation [18] does not include a guide on how to achieve an IDE integration; therefore, it requires deeper knowledge of the Rascal LSP package.

Documentation

Score 1

Weighted Score 2

Reasoning

While the documentation [18] on the Rascal language itself is detailed, there is not much to go on regarding creating a DSL. There is no tutorial on IDE integration, but one page[2] referring to the language server package documentation. The required information has to be pulled from the package API description and examples.

Feature Support

Score 2

Weighted Score 6

[1] http://www.usethesource.io

[2] https://www.rascal-mpl.org/docs/Recipes/BasicProgramming/IDEConstruction

Reasoning

The language server's supported capabilities are not documented, and since creating one requires knowledge of the LSP package, they could not be evaluated in a test. Native integration could enhance missing features, but if too many LSP features are not supported, it could question the use of Rascal in the first place.

Licence

Score	3
Weighted Score	9

Reasoning

Rascal is licensed under BSD-2, which can be used without restrictions in an Apache 2.0 project [13].

IDE Compatibility

Score	3
Weighted Score	9

Reasoning

The documentation [18] explicitly mentions VSCode support. The language server can be integrated into IntelliJ.

Reusability

Score	1
Weighted Score	2

Reasoning

Rascal is its own language and has custom hooks for LSP support, so a rewrite is necessary.

C.4: IntelliJ LSP

Future Proofing

Score	3
Weighted Score	9

Reasoning

It is part of IntelliJ Ultimate and, therefore, maintained by JetBrains. While JetBrains still recommends a native integration, since that supports more features [21], it is unlikely that JetBrains will drop LSP support after just recently adding it.

Ease of use

Score	2
Weighted Score	4

Reasoning

Configuring the language server requires just 2 class implementations and one entry in the plugin.xml [21]. However, debugging problems requires knowledge of LSP.

Documentation

Score 2

Weighted Score 4

Reasoning

While the IntelliJ plugin SDK documentation is quite extensive, there is only one page for LSP integration[1]. The essential parts are documented; information not included in the documentation needs to be pulled from existing plugins using LSP.

Feature Support

Score 2

Weighted Score 6

Reasoning

The IntelliJ LSP supports most editor features from the requirements. Context actions, like e.g. the Context Map generator, can be added through native integration. Code folding tough is not yet supported.

Licence

Score 3

Weighted Score 9

Reasoning

IntelliJ is licensed under Apache 2.0. The same licence applies since LSP support is bundled in the Ultimate version of IntelliJ.

IDE Compatibility

Score 2

Weighted Score 6

Reasoning

LSP is available for IntelliJ Ultimate and VSCode

Reusability

Score 3

Weighted Score 6

Reasoning

The existing language server can be reused. Additional features not supported by LSP can use the same logic as VSCode by properly abstracting IDE/framework specifics.

[1] https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html

C.5: LSP4IJ

Future Proofing

Score 2

Weighted Score 6

Reasoning

LSP4IJ is maintained by the Red Hat developer community and has a small group of active developers [36]. The plugin is relatively new (it was first released in May 2024), so it is unlikely to be discontinued soon.

Ease of use

Score 2

Weighted Score 4

Reasoning

Configuring LSP4IJ via UI is very straightforward. The effort to create a plugin that uses LSP4IJ is comparable to IntelliJ's native LSP support [22]. Debugging problems also requires knowledge of LSP.

Documentation

Score 3

Weighted Score 6

Reasoning

The documentation for LSP4IJ [22] contains a step-by-step guide to creating a plugin based on it (including some pointers for special cases), lists all supported LSP capabilities in detail and explains how to use the plugin UI.

Feature Support

Score 3

Weighted Score 9

Reasoning

LSP4IJ supports all editor requirements. Features not supported by LSP can be added to the plugin implementation.

Licence

Score 2

Weighted Score 6

Reasoning

LSP4IJ is licensed under the EPL-2.0[1] licence. According to the Apache 3rd Party Licence Policy [13], software under the EPL-2.0 licence may be included in binary form, provided that its inclusion is made visible to the user. This condition is fulfilled in the case of an IntelliJ plugin since the dependency is distributed separately, and dependencies on other plugins are visible to users in the plugin marketplace.

IDE Compatibility

Score 3

Weighted Score 9

Reasoning

LSP4IJ is available for IntelliJ Community & Ultimate and LSP is available in VSCode

Reusability

Score 3

Weighted Score 6

Reasoning

The existing language server can be reused. Additional features not supported by LSP can use the same logic as VSCode by properly abstracting IDE/framework specifics.

C.6: Native Integration

Future Proofing

Score 3

Weighted Score 9

Reasoning

The PSI is at the core of custom language plugins in IntelliJ. It is in JetBrains' interest to provide plugin developers with an interface to create custom language plugins. Though JetBrains has an alternative to create language plugins with MPS, it is implausible that JetBrains will enforce projectional editing in the future.

Ease of use

Score 2

Weighted Score 4

Reasoning

[1] https://www.eclipse.org/legal/epl-2.0

The custom language plugin tutorial in the plugin SDK (Software Development Kit) documentation [21] gives a good overview of the relevant extension points for implementing editor features. The concepts explained are easy to understand but require some knowledge of the PSI. While advanced cases may become complicated and require a deeper understanding of how the PSI and editor features work, acquiring the required knowledge should not be too difficult in most cases.

Documentation

Score 2

Weighted Score 4

Reasoning

The IntelliJ plugin documentation is up-to-date but not complete. According to JetBrains “(…), it is not possible to include every feature and Use Case in the documentation. Developing a plugin will sometimes require digging into the IntelliJ Platform code and analysing the example implementations in other plugins” [21]. There is a tutorial for custom language plugins that explains how to integrate them into the different editor features. Additional documentation is provided for every editor feature.

Feature Support

Score 3

Weighted Score 9

Reasoning

A native integration into IntelliJ supports the broadest range of editor features. Therefore, all requirements are supported.

Licence

Score 3

Weighted Score 9

Reasoning

A native integration uses interfaces that are part of the IntelliJ platform, which is licensed under Apache 2.0.

IDE Compatibility

Score 1

Weighted Score 3

Reasoning

A native integration specifically targets IntelliJ. A plugin for VSCode has to be developed separately.

Reusability

Score	2
Weighted Score	4

Reasoning

Reusing parts of the existing implementation should be possible by using the Context Mapper standalone library. Editor features need to be rewritten, though abstracting and reusing the core logic may be possible.

D: Architectural Decisions

This section captures the architectural decisions made during the PoC development. The decision records below follow the MADR (Markdown Architectural Decision Record) format [37], documenting the context and problem description, considered options, the decision outcome, and its consequences per decision.

D.1: IntelliJ Integration Method

D.1.1: Context & Problem Statement

There are different ways in which an IntelliJ custom language plugin can be implemented. Context Mapper already has a language server implementation based on Xtext, which could be reused for the integration.

D.1.2: Considered Options

- Native IntelliJ LSP integration
- LSP4IJ LSP integration
- Native IntelliJ integration

D.1.3: Option Descriptions

See Section 3.3.

D.1.4: Decision Outcome

It was decided to use LSP4IJ for the IntelliJ integration. See Section 3.3 for details.

D.1.5: Consequnces

Using LSP4IJ creates a dependency on a third-party plugin, which means that its maintenance or lack thereof carries more risk than native IntelliJ features.

D.2: Enabling the CML language server to provide semantic tokens

D.2.1: Context & Problem Statement

Important editor features rely on semantic tokens that represent the structure of a file. An example of such an editor feature is syntax highlighting. The current implementation of the CML language server does not provide semantic tokens, meaning syntax highlighting does not work.

D.2.2: Considered Options

- Implementing the `ISemanticHighlightingCalculator` in Xtext
- Re-implementing the language server in Langium
- Falling back to native integration

D.2.3: Option Descriptions

Xtext configuration By implementing the ISemanticHighlightingCalculator, the language server can be configured to return semantic tokens. Unfortunately, the Xtext documentation [3] does not cover this language server extension. Further extensions may also be necessary to make other editor features work, which could prove difficult without documentation.

Re-implementation with Langium Using Langium has the advantage of using a more future-proof framework than Xtext. The framework also proved easy to grasp during preliminary tests. Implementing `AbstractSemanticTokenProvider` supports semantic tokens. As with Xtext, further extensions of the language server may be necessary to support other editor features.

Fallback to native implementation As described in Section 3.3, a native integration offers the broadest range of implementing editor features. However, this would also mean that the IntelliJ and VSCode plugins would no longer have a common technology base.

D.2.4: Decision Outcome

In accordance with the project advisor, it was decided to re-implement the language server with Langium. Since the future of Xtext is unknown, as little effort as possible should go into the existing language server. While a native integration would guarantee a smooth integration of CML into IntelliJ, the LSP approach is still the best solution from a big-picture point of view. However, the project scope must be adjusted as a full re-implementation comes with a significant effort.

D.2.5: Consequnces

As a consequence, the generators have to be re-implemented or adjusted so that they can also be accessed from Node.js. Using Langium also includes the risk that the CML integration in IntelliJ is not as smooth as with a native integration.

E: Implementation details

This appendix includes implementation details for the different components in the language server and the IntelliJ plugin.

E.1: Language Server

In the language server, the registry pattern [38] was applied multiple times where the feature implementation is specific to the `AstNode` type (grammar element representation generated by Langium). The registry pattern ensures loose coupling between the class that acts as an entry point for Langium and its implementation logic. It also makes it easier to add further implementation classes once the language scope is expanded to tactic DDD.

E.1.1: Semantic Token Providers

Figure 27 shows the class diagram of the semantic token provider classes in the PoC.

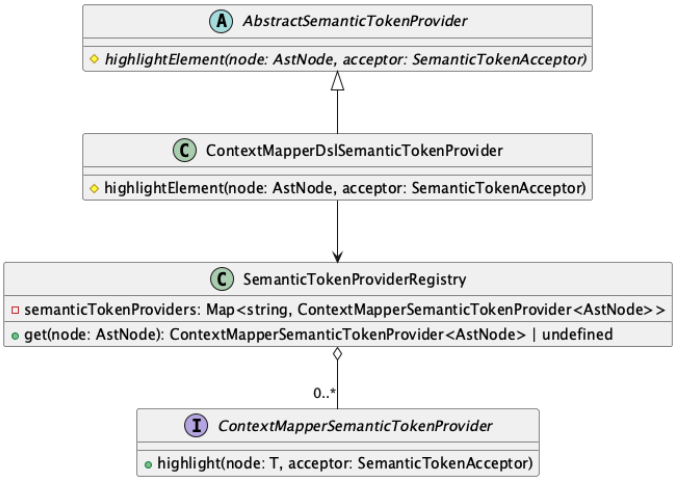

Figure 27: Class diagram of semantic token classes

The responsibilities are divided between the classes as follows:

ContextMapperDslSemanticTokenProvider The class is the entry point for Langium and delegates to the `AstNode`-specific token provider.

SemanticTokenProviderRegistry Keeps track of the available token providers and the `AstNode` type they are responsible for.

ContextMapperSemanticTokenProvider Interface for `AstNode` specific token providers to implement

E.1.2: Reference Resolution

By default, Langium uses top-level elements in a document to resolve references in the AST [17].

This reference resolution behaviour is not enough for Context Mapper since a top-level Bounded Context can also reference a Subdomain, which is nested within a Domain. Therefore, a `ScopeProvider` had to be implemented, adding all nested elements to the resolution scope.

E.1.3: Autocomplete

A custom `CompletionProvider` was implemented to enable the language server to suggest non-alphabetic keywords. Overriding the `filterKeyword` function, which excluded the non-alphabetic keywords, resolved the issue.

While Langium did not initially include all desired keywords in its suggestions, in other cases it did suggest too many elements, such as Bounded Contexts. Langium automatically makes the top-level elements in a document available to other files so that they can import them [17]. However, importing elements from other files was excluded when the decision was made to support only a subset of the CML grammar in this project. Consequently, Langium's default behaviour of "exporting" elements in the top-level scope leads to autocomplete suggestions that are not yet supported. To resolve this, a `ScopeComputation` was created, which does not export anything to the global scope. Once the CML grammar supports imports, discarding the customised scope computation should be possible.

E.1.4: Folding Range Provider

To improve the provided folding ranges, a `FoldingRangeProvider` was created, overriding the Langium default. The implementation logic is a modified version of Langiums `DefaultFoldingRangeProvider`.

To ensure that the first line of a hidden CML element is still visible, a folding range is placed at the very start of the second line of an element block. For comments a modifiction was made, to make hidden comments look like `/*...*/`.

E.1.5: Hover Provider

A subclass of the Langium `MultilineCommentHoverProvider` was created to customise the hover provider behaviour. The subclass implementation is an adaptation of the implementation shared in a GitHub discussion [39]. The documentation texts for the CML keywords are from Context Mapper's current Xtext implementation.

E.1.6: Semantic Validation

Figure 28 shows the class diagram of the semantic validation classes in the PoC.

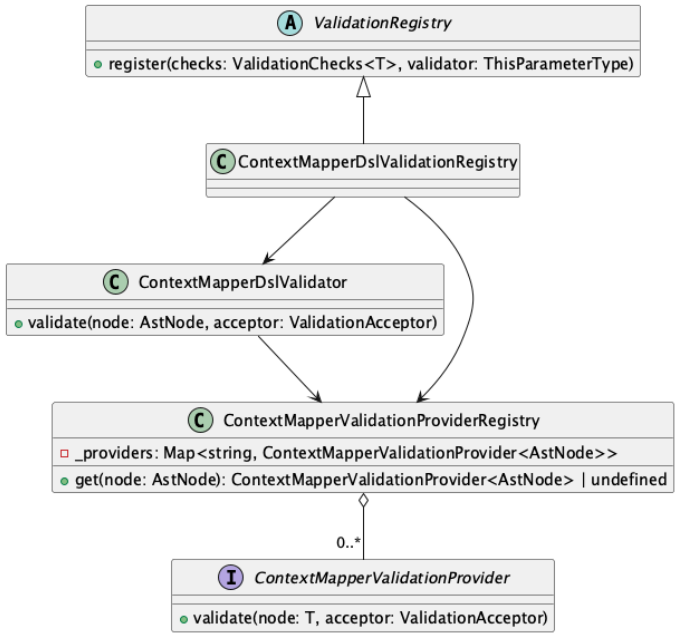

Figure 28: Class diagram of semantic validation classes

The class responsibilities are defined as follows:

ContextMapperDslValidationRegistry Entry point for Langium to get the validator per `AstNode` type

ContextMapperDslValidator Generic validator registered for every `AstNode` type that requires validation. Delegates validation requests to specialised validators.

ContextMapperValidationProviderRegistry Holds the mapping of the `AstNode` type to the responsible specialised validator

ContextMapperValidationProvider Interface for the specialised validators to implement

E.1.7: Document Formatting

Figure 29 shows the class diagram of the applied registry pattern.

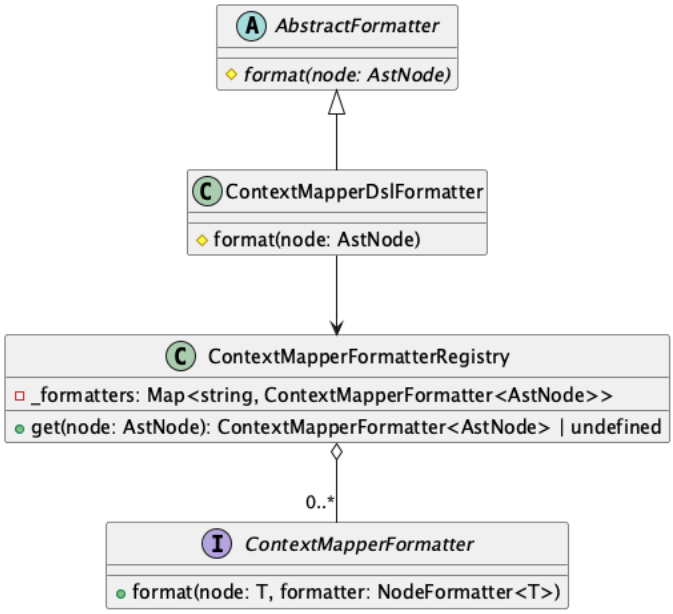

Figure 29: Class diagram of formatter classes

The class responsibilities are defined as follows:

AbstractFormatter Starting point for formatter implementations provided by Langium. Acts as an intermediary between Langium and the custom formatting instructions.

ContextMapperDslFormatter Entry point for Langium to access the Context Mapper formatting instructions. Delegates formatting to the responsible `AstNode` formatter.

ContetxMapperFormatterRegistry Contains the mapping of the `AstNode` type to the responsible formatter

ContextMapperFormatter Generic interface for the Context Mapper `AstNode` formatters to implement

Unlike the semantic token provider, comments are automatically formatted and do not have to be included in the formatter implementation.

E.1.8: Commands & Generators

The entry point for Langium commands is an `ExecuteCommandHandler`, which must be implemented [17]. Figure 30 shows the command/generator setup implementation in the language server.

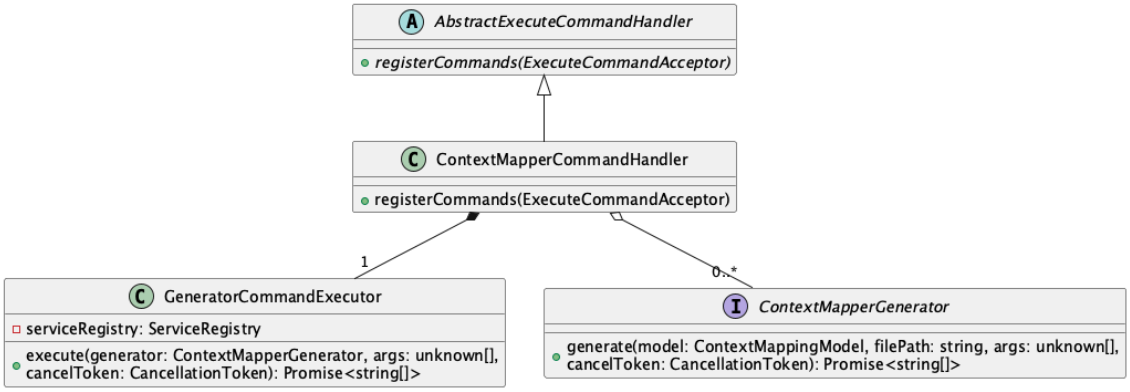

Figure 30: Class diagram of command/generator classes

AbstractExecuteCommandHandler Holds the command registry and provides functions to execute commands

ContextMapperCommandHandler Registers supported commands with their executor functions.

GeneratorCommandExecutor Wrapper class to execute generators. Reads AST model from provided source file and calls generator

ContextMapperGenerator Interface for generators to implement

E.2: IntelliJ Plugin

Below, the LSP4IJ configuration changes and the generator action implementation are documented in more detail.

E.2.1: LSP4IJ Configuration

The following configuration changes for LSP4IJ were applied.

Syntax Highlighting

For the editor to highlight the semantic tokens provided by the language server, a `SemanticTokensColorsProvider` has to be implemented and registered in the `plugin.xml` [22]. The colour provider maps token types and modifiers to IntelliJ's `TextAttributesKey` objects, determining the token's highlighting colour.

Hyperlinking

To resolve the issue of the whole file being displayed as a hyperlink, the LSP4IJ `LSPSemanticTokensStructurelessFileViewProviderFactory` was registered as a file view provider in the `plugin.xml` [22].

Code Folding

To ensure that the folding ranges provided by the language server were applied correctly, the LSP4IJ `LSPFoldingRangeBuilder` had to be registered as the folding builder for CML in the `plugin.xml` [22].

Structure View

For LSP4IJ to populate the structure view, the `LSPDocumentSymbolStructureViewFactory` had to be registered for the CML in the

`plugin.xml` [22]. This configuration change also enabled document breadcrumbs (display the path from the document root to the current cursor position) to work.

E.2.2: Generator Action

IntelliJ provides the "Action" concept to make plugin functionality available to users [21]. Depending on the applied configuration, an action is displayed in IntelliJ's toolbar or as a menu item.

The class diagram below (Figure 31) explains the implementation of the PlantUML generator in the IntelliJ plugin.

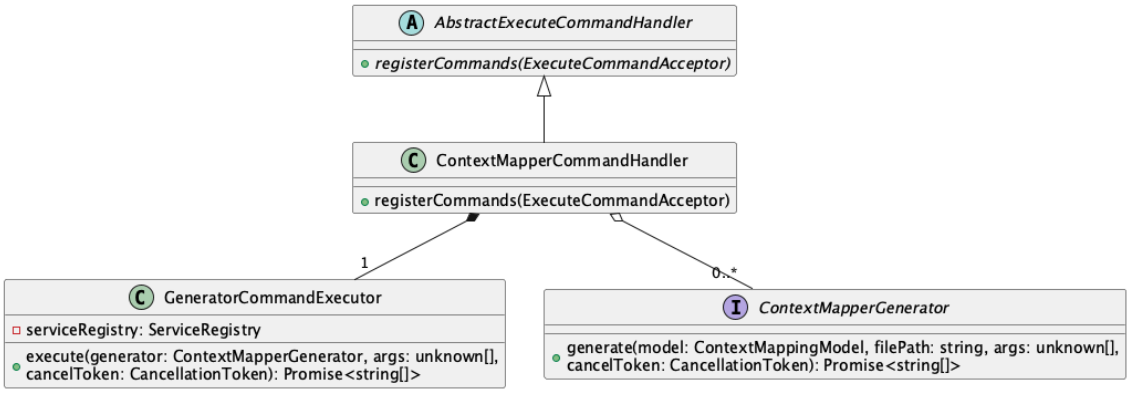

Figure 31: Class diagram of the action/generator classes in the IntelliJ plugin

PlantUMLAction The entry point for IntelliJ to execute the action needs to be registered in the `plugin.xml`.

ContextMapperGenerator Abstracted generator that takes an LSP command, triggers & waits on its execution. While the generator is currently only used for the PlantUML generation, it can also handle other generator commands.

CommandExecutor Class provided by LSP4IJ to trigger command executions in the language server

F: Manual Tests

To verify the correct configuration of LSP4IJ in the IntelliJ plugin, a few manual tests have been executed. The tests are documented below. All tests expect a started IntelliJ Community 2024.3.5 instance with the Context Mapper plugin installed, and the context-mapper-examples[1] project opened.

F.1: Syntax Highlighting

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml	No syntax errors displayed, keywords are highligted	26.05.2025	No syntax errors, highlighting as expected	Pass
2	Create a new CML file and type: `ValueRegister TestRegister {`	A syntax error reports the missing closing brace	26.05.2025	Syntax error appeared	Pass

Table 10: Manual test execution for syntax highlighting feature

F.2: Hyperlinking

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and Ctrl-Hover over the usage of CargoBooking - Context	CargoBooking - Context is displayed as hyperlink	26.05.2025	Hyperlink displayed as expected	Pass
2	Ctrl-Click on the usage of CargoBooking - Context	The caret moved to the definition of CargoBooking - Context	25.06.2025	Caret moved as expected	Pass

Table 11: Manual test execution for hyperlinking feature

[1] https://github.com/ContextMapper/context-mapper-examples

F.3: Occurrence Highlighting

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and place caret in CargoBooking - Context	All occurrences of CargoBooking - Context in the file are highlighted	26.05.2026	All occurrences highlighted	Pass

Table 12: Manual test execution for occurrence highlighting feature

F.4: Autocomplete

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Create a new CML file and type `Bou`	The editor suggests `BoundedContext`	26.05.2025	`BoundedContext` was suggested	Pass

Table 13: Manual test execution for autocomplete feature

F.5: Code Folding

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and collapse the Context Map	`ContextMap DDDSampleMap {...}` is displayed	26.05.2025	Folding worked as expected	Pass
2	Expand the collapsed Context Map	The complete context map is visiable again	26.05.2025	Expand worked as expected	Pass

Table 14: Manual test execution for code folding feature

F.6: Keyword Tooltips

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and hover over the `ContextMap` keyword	A tooltip explaining the Context Map pattern appears	26.05.2025	Tooptip appeared	Pass

Table 15: Manual test execution for keyword tooltip feature

F.7: Structure outline

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and open the structure view	The Context Map and the three Bounded Contexts are outlined	26.05.2025	Context Map and Bounded Contexs were outlined	Pass

Table 16: Manual test execution for structure outline feature

F.8: Find Usages

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and Ctrl-Click on the definition of CargoBooking - Context	The editor reports 4 usages (the definition itself and three others)	26.05.2025	4 usages found	Pass

Table 17: Manual test execution for 'Find Usages' feature

F.9: Document Formatting

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml, add a space after `DDDSampleMap` and execute the “Format Code” action	The added space was removed	26.05.2025	The additional space was removed	Pass

Table 18: Manual test execution for document formatting feature

F.10: PlantUML Generator

ID	Step Description	Expected Results	Executed Date	Actual Result	Status
1	Open file DDD-Sample-Stage-1.cml and right-click in the editor	A Context Mapper menu item appears in the opened popup	26.05.2025	The menu item was dispalyed	Pass
2	Click on the “Generate PlantUML Diagrams” menu item	A src-gen folder with a component diagram is created	26.05.2025	The component diagram was created	Pass

Table 19: Manual test execution for PlantUML generator feature

G: Glossary & List of Acronyms

ADR: Architectural Decision Record

API: Application Programming Interface

ASR: Architecturally Significant Requirement

AST: Abstract Syntax Tree - Data tree containing information about language elements in a file

CI: Continuous Integration

CLI: Command Line Interface

CML: Context Mapping Language

DDD: Domain-Driven Design

DSL: Domain Specific Language - Programming language customised to a specific Domain

IDE: Integrated Development Environment

JSON-RPC: JavaScript Object Notation - Remote Procedure Call

JVM: Java Virtual Machine

LSP: Language Server Protocol - Protocol for communication between development tools and language servers

MADR: Markdown Architectural Decision Record

MPS: Meta Programming System - Language workbench by JetBrains to create custom DSLs

NFR: Non-Functional Requirement

PSI: Program Structure Interface - Interface for IntelliJ plugins to interact with language file contents

PoC: Proof of Concept

QoS: Qualify-of-Service

RegEx: Regular Expression - Expression used to match elements in a text

SDK: Software Development Kit

UML: Unified Modelling Language

VDAD: Value-Driven Analysis & Design - Iterative process aiming to combine value-driven approaches with software engineering practices

H: Bibliography

[1] S. Kapferer, “A Domain-specific Language for Service Decomposition,” Dec. 2018. [Online]. Available: https://eprints.ost.ch/id/eprint/722/1/HS18-MSE-Stefan-Kapferer.pdf

[2] S. Kapferer, O. Zimmermann, and M. Stocker, “Value-Driven Analysis and Design: Applying Domain-Driven Practices in Ethical Software Engineering,” in *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices*, in EuroPLoP '24. New York, NY, USA: Association for Computing Machinery, 2024. doi: 10.1145/3698322.3698332.

[3] Eclipse-Foundation, “Xtext Documentation.” Accessed: Feb. 25, 2025. [Online]. Available: https://eclipse.dev/Xtext/documentation

[4] C. Dietrich, “Call To Action: Secure the future maintenance of Xtext.” Accessed: Feb. 20, 2025. [Online]. Available: https://github.com/eclipse-xtext/xtext/issues/1721

[5] Microsoft, “Language Server Protocol.” Accessed: Feb. 25, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/

[6] Microsoft, “Language Server Sequence.” Accessed: Feb. 25, 2025. [Online]. Available: https://microsoft.github.io/language-server-protocol/overviews/lsp/img/language-server-sequence.png

[7] StackExchange, “Stack Overflow Annual Developer Survey.” [Online]. Available: https://survey.stackoverflow.co/2024/

[8] ContextMapper, “context-mapper-dsl.” Accessed: Feb. 28, 2025. [Online]. Available: https://github.com/ContextMapper/context-mapper-dsl

[9] ContextMapper, “ContextMapper Documentation.” Accessed: Feb. 19, 2025. [Online]. Available: https://contextmapper.org/docs

[10] C. Larman, *Applying UML and patterns : an introduction to object-oriented analysis and design and the unified process*, 2nd ed. Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[11] ContextMapper, “context-mapper-examples.” Accessed: Mar. 04, 2025. [Online]. Available: https://github.com/ContextMapper/context-mapper-examples

[12] “Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model.” [Online]. Available: https://www.iso.org/standard/78176.html

[13] A. S. Foundation, “ASF 3rd Party Licence Policy.” Accessed: Mar. 10, 2025. [Online]. Available: https://www.apache.org/legal/resolved.html

[14] M. Fowler, “Language Workbench.” Accessed: Sep. 09, 2008. [Online]. Available: https://martinfowler.com/bliki/LanguageWorkbench.html

[15] JetBrains, “MPS User's Guide.” Accessed: Mar. 11, 2025. [Online]. Available: https://www.jetbrains.com/help/mps

[16] M. Fowler, “Projectional Editing.” Accessed: Jan. 14, 2008. [Online]. Available: https://martinfowler.com/bliki/ProjectionalEditing.html

[17] Eclipse-Foundation, “Langium Documentation.” Accessed: Mar. 13, 2025. [Online]. Available: https://langium.org/docs

[18] UseTheSource, “Rascal Documentation.” Accessed: Mar. 13, 2025. [Online]. Available: https://www.rascal-mpl.org/docs

[19] Spoofax, “Spoofax.” Accessed: Mar. 12, 2025. [Online]. Available: https://spoofax.dev/

[20] Spoofax, “Spoofax 3.” Accessed: Mar. 12, 2025. [Online]. Available: https://spoofax.dev/spoofax-pie/develop/

[21] JetBrains, “IntelliJ Platform Plugin SDK.” Accessed: 2025. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/welcome.html

[22] RedHat, “LSP4IJ Documentation.” Accessed: 2025. [Online]. Available: https://github.com/redhat-developer/lsp4ij/tree/main/docs

[23] J. B. Kühnapfel, *Scoring und Nutzwertanalysen : Ein Leitfaden Für Die Praxis.*, 1st ed. Wiesbaden: Springer Fachmedien Wiesbaden GmbH, 2021.

[24] O. Zimmermann, “Architectural Significance Test.” Accessed: Oct. 01, 2020. [Online]. Available: https://medium.com/olzzio/architectural-significance-test-9ff17a9b4490

[25] E. Evans, *Domain-driven design : tackling complexity in the heart of software*, 4th prin. Boston: Addison-Wesley, 2004.

[26] S. Brown, “C4 model.” Accessed: May 15, 2025. [Online]. Available: https://c4model.com/

[27] M. Sujew and L. Streckeisen, “Workaround for optional elements in unordered groups.” Accessed: Apr. 14, 2025. [Online]. Available: https://github.com/eclipse-langium/langium/discussions/1903

[28] G. Fontorbe, “Hide non alphabetic tokens.” Accessed: Sep. 30, 2022. [Online]. Available: https://github.com/eclipse-langium/langium/pull/697

[29] R. Cox, “Surviving software dependencies,” *Commun. ACM*, vol. 62, no. 9, pp. 36–43, Aug. 2019, doi: 10.1145/3347446.

[30] JetBrains, “Grammar-Kit.” Accessed: Mar. 24, 2025. [Online]. Available: https://github.com/JetBrains/Grammar-Kit

[31] F. Campagne, “JetBrains Meta Programming System.” Accessed: Mar. 10, 2025. [Online]. Available: https://stackoverflow.com/a/31186463

[32] JetBrains, “MPS Core Versions.” Accessed: Mar. 17, 2025. [Online]. Available: https://plugins.jetbrains.com/plugin/7075-mps-core/versions

[33] Eclipse-Foundation, “Langium Releases.” Accessed: Mar. 13, 2025. [Online]. Available: https://github.com/eclipse-langium/langium/releases

[34] GitHub, “Langium Contributors.” Accessed: Mar. 17, 2025. [Online]. Available: https://github.com/eclipse-langium/langium/graphs/contributors

[35] GitHub, “Rascal Contributors.” Accessed: Mar. 13, 2025. [Online]. Available: https://github.com/usethesource/rascal/graphs/contributors

[36] GitHub, “LSP4IJ Contributors.” Accessed: Mar. 17, 2025. [Online]. Available: https://github.com/redhat-developer/lsp4ij/graphs/contributors

[37] ADR-Organization, “Markdown Architectural Decision Records.” Accessed: Mar. 21, 2025. [Online]. Available: https://adr.github.io/madr/

[38] M. Fowler and D. Rice, *Patterns of enterprise application architecture*. in The Addison Wesley signature series. Boston [etc: Addison-Wesley, 2003.

[39] Y. Daveluy, “Hovers on keywords.” Accessed: Jul. 27, 2024. [Online]. Available: https://github.com/eclipse-langium/langium/discussions/1603

I: List of Figures

J: List of Tables

K: List of Code Listings

Disclaimer

Parts of this thesis were rephrased using the following tools:

- Grammarly[1]
- LanguageTool[2]
- ChatGPT[3]

[1] https://www.grammarly.com/
[2] https://languagetool.org/de
[3] https://chatgpt.com/