Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Kristian Rekstad

A Modeling Environment in the Cloud for Education

Master's thesis in Computer Science
Supervisor: Hallvard Trætteberg
June 2021

NTNU
Norwegian University of
Science and Technology

Kristian Rekstad

A Modeling Environment in the Cloud for Education

Master's thesis in Computer Science
Supervisor: Hallvard Trætteberg
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

Software engineering has an approach called Model-Driven Development (MDD). This is taught to students in higher education. The approach is reliant on tools, and one such tool is the Eclipse Modeling Framework (EMF). While EMF can be used to teach students about MDD, it is unpopular because of its ties to the Eclipse Integrated Development Environment (IDE), causing students to resist learning MDD. Cloud based alternatives exist for the Eclipse IDE, like Gitpod with VSCode, which provide benefits in an educational organization. However, the EMF tools used in education are not available in these alternatives. This thesis tries to enable the cloud based alternatives to support EMF.

The thesis' approach is based on Design Science Research, where a design is created and a software artifact is implemented. The design draws inspiration from the Language Server Protocol (LSP) and Graphical Language Server Platform (GLSP), protocols for text and diagram editing. These protocols already work in VSCode.

The result is a Tree Editor extension for VSCode. EMF models can be edited as trees. This extension uses a three component design: a generic tree editor user interface, a VSCode extension, and an EMF specific server. The extension and server communicate with a newly designed protocol, the Tree Language Server Protocol (TLSP).

The resulting implementation can be built upon to work with EMF modeling in the cloud. The TLSP protocol and software architecture can be used by other tools that need tree editors, that aim to support multiple IDEs as well. A wider adoption of TLSP in IDEs will make migrations of tree editors to other IDEs easy. Regardless, the design provides a reusable server for EMF, which can ease migrations of EMF to other IDEs.

Sammendrag

Programvareutvikling har en tilnærming som kalles Model-Dreven Utvikling (MDD). Dette undervises til studenter i høyere utdanning. Tilnærmingen er avhengig av verktøy, og et slikt verktøy er Eclipse Modeling Framework (EMF). Selv om EMF kan brukes for å lære studenter om MDD, er det upopulært på grunn av sin tilknytning til Eclipse Integrated Development Environment (IDE), som gjør at studenter stritter i mot å lære MDD. Skybaserte alternativer til Eclipse IDE eksisterer, som Gitpod med VSCode, og de har nyttige egenskaper for en utdanningsorganisasjon. Verktøyene i EMF finnes derimot ikke for disse alternativene. Denne masteroppgaven prøver å legge til rette for å støtte EMF i de skybaserte alternativene.

Fremgangsmåten i masteroppgaven er basert på Design Science Research, hvor et design blir lagd og en programvare blir utviklet. Designet drar inspirasjon fra Language Server Protocol (LSP) og Graphical Language Server Platform (GLSP), protokoller for tekst- og diagramredigering. Disse protokollene brukes allerede i VSCode.

Resultatet er en utvidelse for VSCode for redigering av tre-strukturer. EMF-modeller kan redigeres som trær. Denne utvidelsen består av tre komponenter: et generisk brukergrensesnitt for tre-redigering, en utvidelse for VSCode, og en EMF-spesifikk tjener (server). Utvidelsen og serveren snakker med en nylig designet protokoll: Tree Language Server Protocol (TLSP).

Den resulterende programvaren kan bygges på videre, for å bruke EMF-modellering i skyen. TLSP-protokollen og programvarearkitekturen kan brukes av også andre verktøy som trenger tre-redigering, og som sikter på å støtte flere IDE-er. En utbredt bruk av TLSP i IDE-er vil gjøre at migrering av tre-redigeringsverktøy til andre IDE-er blir forenklet. Uavhengig av dette, så gir designet en gjenbrukbar server for EMF, som kan forenkle migreringen av EMF til andre IDE-er.

Acknowledgments

Hallvard Trætteberg for being a very helpful supervisor and for interesting discussions.

Norwegian University of Science and Technology (NTNU) for providing access to research papers, my education, and for providing an office to write this thesis.

Dr. Jonas Helming and Maximilian Koegel at EclipseSource for their helpful blog posts. An Dr. Helming in particular, for providing answers about my research at the EclipseCon 2020 conference, and the initial title for the thesis.

CoPCSE-NTNU for this latex document template: https://github.com/COPCSE-NTNU/thesis-NTNU.

All the helpful free software This thesis would be so hard to write, without the aid of LaTeX, VScode, LaTeX Workshop VSCode extension, MiKTeX, Zotero, Calibre, Obsidian, Git and much more. The creators and contributors to these projects don't get enough praise.

Abakus, Online and TIHLDE student organizations for free coffee, and for selling noodles and candy.

Lånekassen for funding my education through loans and scholarships.

My parents, Jenny and Håvard, and my girlfriend, Ingrid M. J. for all the love, support and motivation they give me.

Contents

Abstract . . . iii
Sammendrag . . . v
Acknowledgments . . . vii
Contents . . . ix
Figures . . . xiii
Tables . . . xv
Code Listings . . . xvii
Acronyms . . . xix
Glossary . . . xxi
1 Introduction . . . 1
1.1 Model-Driven Development in Education at NTNU . . . 1
1.2 The Eclipse Ecosystem Wants to Run Software in the Cloud . . . 2
1.3 A Pre-project Identified a Need for a Tree Editor . . . 3
1.4 Research Objectives . . . 5
1.4.1 Problem . . . 5
1.4.2 Scope . . . 5
1.4.3 Objectives . . . 5
1.5 Thesis Structure . . . 6
2 Background . . . 9
2.1 Conceptual Modeling and Model-Driven Development . . . 9
2.2 Model-Driven Development at NTNU in the Course TDT4250 . . . 10
2.3 Eclipse Modeling Framework Editors for Ecore . . . 11
2.3.1 Sample Reflective Ecore Model Editor . . . 12
2.3.2 EMF Forms Ecore Editor . . . 13
2.4 Introduction to Tree Structures . . . 14
2.5 Master-Detail Tree Editor . . . 16
2.6 An Overview of EMF: Ecore Metamodel, XMI Serialization and GenModel for Code Generation . . . 16
2.7 Visual Studio Code and Theia . . . 18
2.7.1 Visual Studio Code . . . 18
2.7.2 Theia . . . 19
2.8 Visual Studio Code's Custom Editor API . . . 20
2.9 Language Server Protocol Architecture . . . 21
2.9.1 Base Protocol . . . 22

2.9.2 Language Server Protocol . . . 23
2.10 JSON-RPC . . . 24
2.11 Cloud and Gitpod . . . 24
2.12 Eclipse Modeling Framework in the Cloud . . . 25
2.12.1 EMF.Cloud . . . 25
2.12.1.1 Model Server . . . 26
2.12.1.2 Theia Tree Editor . . . 26
2.12.1.3 Coffee Editor . . . 27
2.12.2 Graphical Language Server Platform (GLSP) . . . 27
2.12.3 Other Tools by the Eclipse Ecosystem . . . 29
2.12.3.1 JSON-Forms . . . 29
2.12.3.2 CrossEcore . . . 29
2.13 Pre-project Results . . . 30
2.13.1 Research Questions . . . 30
2.13.2 Stakeholders . . . 30
2.13.3 Software Requirements . . . 31
2.13.4 Architecture and Protocol for a Solution . . . 33
2.13.4.1 Architecture . . . 33
2.13.4.2 Protocol . . . 35
3 Method . . . 37
3.1 Design Science Research . . . 37
3.1.1 The General Design Cycle . . . 38
3.1.2 Methodology . . . 40
3.2 Requirements Engineering . . . 41
3.2.1 Stakeholder Discussion . . . 42
3.2.2 Requirements Extraction . . . 42
3.2.3 Source Code Analysis of Similar Projects . . . 43
3.2.4 Use Cases and Prototyping . . . 43
3.2.5 Agile Requirements . . . 43
3.3 Development Methodologies . . . 44
3.3.1 Agile . . . 45
3.3.2 Iterative Development . . . 45
3.3.3 Lean and Minimum Viable Product . . . 46
3.3.4 Tracer Bullets . . . 46
3.3.5 Domain-Driven Design . . . 47
3.3.6 Test-Driven Development . . . 48
3.3.7 Prototyping . . . 48
3.4 Evaluation . . . 48
3.4.1 Software Artifact . . . 49
3.4.2 Open Source Viability . . . 49
4 Results . . . 51
4.1 Software Artifact: Tree Editor Extension for Ecore in Gitpod . . . 52
4.1.1 Custom Editor . . . 52
4.1.2 IDE Commands . . . 54

4.1.3 Genmodel and Model Instance 54
4.1.4 Configuration and Logging 56
4.2 Design Artifact: Tree Document Model 57
4.2.1 Borrowed Terms 57
4.2.2 The Domain Model 58
4.3 Design Artifact: Architecture for Tree Language Server Systems . . . 62
4.3.1 Architecturally Significant Requirements 62
4.3.2 Changes from pre-project 62
4.3.3 System explanation 63
4.3.3.1 Context 63
4.3.3.2 Containers 65
4.3.3.3 Components 67
4.3.3.4 Code 69
4.4 Design Artifact: Tree Language Server Protocol 76
4.4.1 Activation 76
4.4.2 User Actions 78
4.4.3 Property Editing 78
4.4.4 Tree Editing 81
4.5 Open Source Project: Measures Taken for Viability and Maintainability 82
4.5.1 Code Availability 82
4.5.2 Documentation 82
4.5.3 Automation 82
4.5.4 Licensing 83
4.5.5 Code 83
4.5.6 Issue Tracking 84
5 Evaluation 85
5.1 Use Case Completeness Evaluation of Tree Editor Extension 85
5.1.1 Test Case Details 87
5.2 Qualitative Evaluation of the Tree Editor Extension 90
5.3 Qualitative Software Architecture Evaluation 92
5.3.1 Reusable Components for Related Migrations 92
5.3.2 Components for Migrating EMF to Other IDEs 93
5.4 Evaluation of Open Source Project Viability 93
5.4.1 Project Evaluation 93
5.4.2 Readme Evaluation 94
6 Discussion 97
6.1 VSCode as an EMF Tree Editor in the Cloud 97
6.2 Reuse of EMF java code 98
6.3 Creating a Tree Editor for VSCode Requires Substantial Effort 99
6.4 Designing a Standardized Tree Language Server Protocol 100
6.5 Limitations 102
7 Conclusion 105
7.1 Future Work 106

Bibliography **107**
A Tree Editor Functional Requirements from Pre-project **113**
B Pre-project Data Structure Code **117**

Figures

2.1 Screenshots of the Sample Reflective Ecore Model Editor in Eclipse IDE. . . . 13
2.2 EMF Forms Ecore Editor . . . 14
2.3 Tree Structure Visualizations . . . 15
2.4 VSCode User Interface . . . 19
2.5 Theia User Interface . . . 20
2.6 The Language Server Protocol Benefits . . . 21
2.7 LSP Protocol Design . . . 22
2.8 Class Hierarchy of Theia Tree Editor Nodes . . . 27
2.9 GLSP Overview . . . 28
2.10 JSON-Forms Example . . . 30
2.11 Tree Editor Architecture . . . 34

3.1 Design Science Research Process Model . . . 39
3.2 Layered Architecture . . . 47

4.1 Overview of Results . . . 51
4.2 Tree Editor Extension installed in Gitpod . . . 52
4.3 Tree Editor Extension showing studies.ecore . . . 53
4.4 Tree Editor Extension Custom Commands . . . 54
4.5 Tree Editor Extension showing studies.genmodel . . . 55
4.6 Tree Editor Extension showing a dynamic instance . . . 56
4.7 Tree Editor Extension with configuration and logging . . . 57
4.8 System context diagram for Gitpod . . . 64
4.9 Gitpod container diagram . . . 66
4.10 Gitpod deployment diagram . . . 68
4.11 Ecore Tree Editor component diagram . . . 69
4.12 Tree Editor Frontend class diagram . . . 71
4.13 Tree Editor Extension class diagram . . . 73
4.14 Tree Language Server class diagram . . . 75
4.15 Protocol Sequence Diagram of Start/Stop and Document Opening . . . 77
4.16 Protocol Sequence Diagram of Action Triggering . . . 78
4.17 Protocol Sequence Diagram of Property Form . . . 80
4.18 Protocol Sequence Diagram of Tree Changes . . . 81

Tables

5.1 Use Case Evaluation of the Tree Editor Extension 87
5.2 Open Source Evaluation of the Project 94
5.3 Open Source Evaluation of the Readme File 95

A.1 Functional requirements for a master-detail Tree editor with property sheet. 114

Code Listings

2.1 A Request Message Example 23
2.2 JSON-RPC examples copied from [28]. 24
2.3 GLSP Server Interface 28
4.1 TreeDocument 58
4.2 TreeRoot 59
4.3 TreeNode 60
4.4 Action 60
4.5 ActionConfiguration 60
4.6 ActionEvent 61
4.7 HierarchyConfiguration 61
B.1 Pre-project Tree Data Structure 117
B.2 Pre-project Available Actions Data Structure 118
B.3 Pre-project Action Data Structure 118
B.4 Pre-project Hierarchy Schema Data Structure 118

Acronyms

API Application Programming Interface. 17, 18, 20, 21, 27, 30, 34, 45, 62, 74, 89, 93, 97–99, 102

ASR Architecturally Significant Requirement. 62

DSL Domain-specific language. 1, 5, 11

EMF Eclipse Modeling Framework. iii, 1–6, 10–12, 16–18, 26–28, 30–32, 35, 36, 42, 49, 57, 58, 61, 62, 67, 68, 74, 76, 88–93, 97–103, 105, 106

GLSP Graphical Language Server Platform. 4, 27, 28, 90, 91, 101, 105

IDE Integrated Development Environment. 4–6, 12, 18, 20, 21, 25–27, 33–35, 49, 52, 58, 63, 65–67, 69, 70, 76, 83, 88, 90–93, 97, 99–101, 105

LSP Language Server Protocol. 4, 19, 21–24, 28, 36, 72, 74, 76, 77, 83, 88, 91, 92, 101, 105

MDD Model-Driven Development. 1, 2, 5, 6, 9, 10, 16, 17, 31, 42, 47, 49, 100, 106

NTNU Norges Teknisk-naturvitenskapelige Universitet. 1, 2, 5, 10, 11, 31

OCL Object Constraint Language. 11, 31, 89, 93

RPC Remote Procedure Call. 21, 23, 24

TLSP Tree Language Server Protocol. iii, v, 57, 62, 67, 69, 72, 74, 76–78, 80, 81, 88–90, 92, 93, 99, 101, 102, 105, 106

XMI XML Metadata Interchange. 12, 13, 16–18, 33, 54, 89

XML Extensible Markup Language. 17

Glossary

artifact The term *artifact* means something artificial or human created, instead of something occurring in nature [44, p. 6].. 37, 38

cloud Remote data centers that provide computing as a service. Commonly used by businesses to provide web infrastructure. 2, 3, 5, 6, 12, 25–27, 49, 90, 97, 100, 105

domain A phenomena in the real world or area of interest that must be analyzed to solve a problem. A domain is often abstracted to consist of entities, relations, processes and rules. 1

Eclipse Che A cloud based or self hosted workspace and IDE for software development. It is based on Kubernetes and Theia. 97

Eclipse IDE An IDE by the Eclipse Foundation. Originally created by IBM. It is based on a plugin architecture using OSGi, and is written in Java. 1–6, 10–14, 17, 18, 21, 30–32, 42, 43, 49, 53, 76, 86, 88–91, 98, 100, 101, 103, 105, 106

Ecore The EMF core model. A metamodel similar to UML Class Diagrams. 1–4, 10–14, 16–18, 30–36, 42, 43, 49, 53, 54, 57, 58, 60, 62, 88, 89, 91, 93, 98

Electron A desktop application runtime for javascript, based on Chromium. 19

git A free program to version software code. Users create a git repository in a folder, and then track and version all the files inside that folder. 63

GitHub A website for software project management and source code sharing. Based on the Source-control Management (SCM) software called “git” . 2, 19, 26, 63, 82, 84, 95

Gitpod A cloud based workspace and IDE for software development. It is based on Docker, Kubernetes and Theia. 2, 6, 18, 20, 25, 26, 31, 32, 34, 49, 52, 63, 65, 85, 97, 101, 105

JSON Javascript Object Notation. A serialization format for object structures. 17, 24, 30, 35, 76

JSON-RPC Remote Procedure Call (RPC) protocol using Javascript Object Notation (JSON) serialization. It allows a process to execute functions in another process and obtain the results. 21, 23–25, 35, 36, 70, 74, 76, 91

NodeJS A javascript interpreter for desktop, based on the Chromium V8 javascript engine. It also includes some desktop APIs like filesystem access. 18, 72

open source The source code for a software is available; not just for inspection, but for re-use and modification. 2, 4, 6, 7, 12, 13, 18, 19, 34, 43, 44, 49, 51, 82, 83, 85, 94, 95, 98, 100, 105

REST Represential State Transfer (REST). A paradigm for creating HTTP APIs, centered around resources. 27, 35, 36, 62, 99, 102

TDT4250 Advanced Software Design. A course at NTNU. It runs during the autumn, and teaches computer science students concepts like MDD, code generation, DSL and dynamic component based systems. 1–5, 10, 16, 31, 42, 49, 53, 85, 90, 103

Theia An IDE for software development. Theia is accessible in a web browser and as a desktop application. The implementation reuses much of VSCode's internals. Managed by the Eclipse Foundation. 3, 4, 6, 18, 20, 26–29, 32, 34, 35, 52, 65, 67, 85, 90, 97, 101, 106

TypeScript A programming language developed by Microsoft. It is a superset of the Javascript programming language, and adds static typing. TypeScript code is compiled to javascript, and can then run in a web browser or NodeJS. 35, 58, 59, 69

UML Unified Modeling Language. A common modeling language for creating diagrams such as Class Diagrams. It is standardized by Object Management Group. 3, 12, 16, 17, 69

VSCode Visual Studio Code. An IDE for software development. The full name is Visual Studio Code. Managed by Microsoft. 2–4, 6, 18–21, 26–29, 32, 34, 35, 52–54, 56–58, 62, 65–67, 69, 70, 72, 74, 76, 83, 85, 86, 90, 92, 93, 97–101, 105, 106

WebSocket A two-way communication protocol over TCP sockets made available for web browsers. It allows for a persistent and reusable connection which can send multiple messages, unlike regular HTTP requests. Commonly used to avoid polling over HTTP, or live updates of a website. 27, 35, 36

Chapter 1

Introduction

1.1 Model-Driven Development in Education at NTNU

In a world that becomes more digital for each day, there is a large need for software development. Software is often created by writing code using programming languages that compile down to computer instructions. Developers write the code based on a set of requirements, and change it when the requirements change.

One alternative approach to software development, is Model-Driven Development (MDD). This approach has the developers create models of their domain, and this model drives the rest of the software development. The code is usually generated from the model. If the software requirements change, the model is updated first, and the code is re-generated. The model itself is often one or more artifacts in the software project, expressed in a modeling language. Modeling simplifies the domain by using abstraction, and reduces the world down to the entities, relations, procedures (or other abstractions) that are needed to solve the relevant problems.

The MDD approach is taught at Norges Teknisk-naturvitenskapelige Universitet (NTNU). The course is named *TDT4250 Advanced Software Design*. A modeling language called *Ecore* is used in TDT4250. This language comes from the Eclipse Modeling Framework (EMF). The models can generate java code, and can extend the Eclipse IDE as a plugin. The plugin lets a user enter data for a model instance by using Eclipse IDE as a user interface. Students also learn to create Domain-specific languages (DSLs) with an Ecore model as its core.

Eclipse IDE is required to work with EMF modeling. It has editors for Ecore, code generation and model validation. There are two main types of editors: hierarchical tree editors and graphical diagram editors. There are also multiple different implementation on the tree editors, based on different underlying frameworks.

The reliance on Eclipse IDE is a problem for students. Students don't like to work in Eclipse, because of various issues with usability, errors or stability [1]. If a student wants to use EMF afterwards in their job, they would have to use Eclipse IDE, and also convince their team to do it as well. Some students see EMF as being too Eclipse IDE related, as well, and incorrectly see it as a tool for only developing Eclipse plugins. This results in students resisting to learn EMF, and also MDD by implication, because no EMF alternative is taught.

NTNU wants to move from Eclipse IDE to VSCode running in a web browser. This is a recent decision, and mainly for the course in Object Oriented Programming with java. Some of the reasoning behind the change, is to avoid installation issues from Eclipse IDE, and to ease online collaboration through GitHub and publication of assignments. VSCode is an advanced text editor that has increased in popularity in the recent years. It is based on web technologies, but normally runs as a desktop application. A website and service called *Gitpod* allows running VSCode in a web browser, and connect it to a workspace based on a GitHub repository. The workspace has the project files, software development kits and other tools already installed and running in a remote machine in the cloud. This avoids all installation on a student's machine.

For TDT4250 to follow suit and move to Gitpod, the Ecore editors would have to be available in VSCode as well. The current situation is that there are no Ecore editors for VSCode. There are also no known MDD frameworks for VSCode that integrates with the other curriculum of TDT4250 either, as alternatives to EMF.

1.2 The Eclipse Ecosystem Wants to Run Software in the Cloud

The Eclipse Modeling Framework is powered by open source software and an ecosystem of developers. The framework has many tools and software libraries available, contributed by various developers and organizations. These developers and organizations, is what this thesis nicknames the *Eclipse Ecosystem*. Some prominent actors are the organizations *TypeFox*, *EclipseSource* (with Dr. Jonas Helming and Maximilian Koegel), *Obeo* and *RedHat* [1]. For example, *Gitpod* is developed by TypeFox, and one of the Eclipse IDE tree editors for Ecore is created by EclipseSource [2, 3].

Cloud is becoming more popular, and the Eclipse Ecosystem is heading there. When something *runs in the cloud*, it really means that it runs on rented computers in a data center somewhere outside of the organization. Running in the cloud is a win for developers and organizations, because they don't need to take care of their own hardware. And scaling up to more computers is as easy as clicking a button, or

often happens automatically with load balancing technology. No more purchasing of hardware and configuring it. When developers "embrace" the cloud, it also means working more with web technologies and less with desktop applications.

To use EMF in the cloud, the Eclipse ecosystem has started to create new tools. Most of the tools are related to **running** EMF-based software, but not **developing** it. There are *some* advances to developing in the cloud, with *Gitpod* and the VSCode re-implementation *Theia*, but neither have tools for EMF.

1.3 A Pre-project Identified a Need for a Tree Editor

This masters thesis is preceded by a pre-project thesis. This work happened during the Autumn of 2020, the semester before this masters thesis. The results were presented in [1]. The project began by identifying what to build. The need for EMF editing in the cloud was known, but not how to do it or if it was even possible.

The pre-project identified a need for a web-based tree editor for working with EMF. Early plans were to create a diagram editor, inspired by *UML Class Diagrams* and the Eclipse IDE diagram editor for Ecore named *Ecore Tools* (based on *Sirius* by aforementioned Obeo) [1]. During an online conference for the Eclipse ecosystem, EclipseCon 2020, it became clear that EclipseSource was already working on this [4]. However, based on the author's experience as a former student of TDT4250, most of the work with Ecore happened in a tree structure[1] editor with a property sheet. This kind of editor has what is known as a *master-detail layout*, where the tree is a master view, and the property sheet is the details of the current selection in the tree. No actor in the Eclipse ecosystem was working on such a tree editor for Ecore models for VSCode. Preliminary searches online did not find such an editor created by anyone outside the Eclipse ecosystem either.

Initial requirements for a tree editor were chosen. The period of work was constrained to the pre-project and master's thesis, which is from August 2020 to June 2021. This constraint made it a goal to reduce the amount of unnecessary work and reduce re-implementation of existing solutions. For example, the Eclipse Modeling Framework is big, with many years worth of experience ingrained in its implementation details. Therefore, a non-functional requirement emerged: **the editor should re-use as much of the existing EMF java code as possible.**

Another non-functional requirement was that **it should run inside VSCode as an extension**. Gitpod was at the time was using Theia as the editor, which was compatible with VSCode extensions [1]. Theia has two extension mechanisms,

[1] *Tree structure* here means the hierarchical parent-child structure, perhaps better known from file system folders and file browsers.

but only the VSCode extension mechanism could be installed during runtime by students [1]. Because a goal was to use the Gitpod service for TDT4250, this compatibility was needed.

The third non-functional requirement was that **the project should be open source and designed to live longer than the period of work.** A goal is to include all or most of the functionality already present in Eclipse IDE, which was estimated to be more work than what was possible to do during the pre-project and master's thesis. Therefore, the development will need to be taken over by someone else afterwards. Either the Eclipse ecosystem, or a master's thesis by another student. An open source project needs some additional care if it wants to succeed. For the Eclipse ecosystem to handle it, the software should have a compatible license, and not copy or use code with incompatible licenses. The code should also be well structured, documented and easy to contribute to for others.

The initial, unrefined functional requirement was that **VSCode should be able to view, edit and save Ecore models and model instances in ".ecore" and ".xmi" files.** The pre-project did further work to refine this functional requirement into multiple smaller requirements, and discovered many new ones, by requirements extraction [1, p. 47, 48]. As noted in the discussion in [1, p. 51], the list of functional requirements was not complete.

Related software architectures and protocols were analyzed. Because the EMF tooling had to move to VSCode now, it is plausible that it will need to move to another IDE later in the future. The pre-project explored protocols related to this, like Language Server Protocol (LSP) and Graphical Language Server Platform (GLSP).

The pre-project used prototypes to verify the feasibility of the architecture. The main issues solved in the pre-project were related to hypothetical design choices and feasibility. It tried to answer if and how java could be executed from the VSCode extension, to reuse EMF code. The pre-project also looked for a good data model to support editing of any tree structure, while providing a user interface with high usability and constraints [1, p. 24, 25].

More work was needed in order to evaluate the pre-project solution. No complete editor was produced during the pre-project. It only proved the possibility of creating a custom frontend, and that extensions could run java programs. The entire protocol was unexplored, and no real EMF data was ever loaded into the frontend. This master's thesis will pick up on these results and try develop them further into a usable solution. It also aims to create an open source repository that is viable and suitable for further development by the Eclipse ecosystem and other master students.

1.4 Research Objectives

1.4.1 Problem

Problem definition How can students use the Eclipse Modeling Framework (EMF) in a cloud based Integrated Development Environment (IDE) in order to learn Model-Driven Development (MDD) as part of the course TDT4250, *without* using the Eclipse IDE?

Value

1. Students may be more motivated to learn MDD if they do not need to use Eclipse IDE, and do not perceive the MDD framework (EMF) as Eclipse IDE-specific or only for deploying to the Eclipse IDE [1, p. 2]. Few to no other courses at NTNU target Eclipse IDE as the deployment/target platform, and Kuzniarz and Martins [5] found that students resist learning when the technology and skills are not used in other courses. Students also dislike or have problems with Eclipse IDE itself, and feedback collected from teaching students in 2015 by Jordi Cabot [6] found that much of the complaints were about installation issues and problems with the tools, not problems with MDD as a concept.
2. By moving EMF from Eclipse IDE to other Integrated Development Environment (IDE), the value of the framework itself may increase, as adoption of EMF does not imply adoption or use of Eclipse IDE. Industry may use the framework for modeling, without requiring the developers to use Eclipse IDE. A problem for MDD adoption in general is low impact on personal career needs, identified by Jon Whittle *et al.* [7].

1.4.2 Scope

There are many different activities in MDD, and the course TDT4250 investigates related areas such as creating Domain-specific language (DSL) and custom code generation templates. Tackling all the aspects of MDD and TDT4250 in this thesis is unrealistic, because of the size of work.
The scope is therefore limited to enabling the **creation and editing of EMF model files**. This thesis is also limited in scope in terms of how far a design for a solution is developed and instantiated as code. The aim is to prove feasibility of such a design and instantiation, but not to create a fully functional editor with all the features needed to do modeling. Instead, **a software foundation will be established**, where the design and principles can guide *other developers'* further development towards a complete solution.

1.4.3 Objectives

There are three objectives for this thesis.

Objective 1: EMF Modeling in the Cloud The first objective is to design a solution to enable Model-Driven Development (MDD) with the Eclipse Modeling Framework (EMF) in a cloud based IDE. Gitpod with VSCode is chosen as the IDE. A solution should be able to support all the modeling tasks needed to teach MDD, and Eclipse IDE should not need to be installed on a student's computer.

Objective 2: Open Source project The artifact should exist longer than the period of work for the master's thesis, and be developed further by contributors other than this thesis' author. The artifact should be in a open source project, to fit in with the expectations of the Eclipse ecosystem, current trends and expectations of students.

Objective 3: An architecture to enable future related IDE migrations This may not be the last time EMF will be used in a new IDE. And other tools and frameworks may need the same type of migration, as cloud adoption increases. Therefore, the solution should apply a software architecture that allows easier migration to another IDE, by providing more reuseable and "higher abstraction level" components than what EMF currently has. The software architecture should also provide guidance or be directly applicable to other tools that work in a similar fashion. A bonus objective is if this design can provide instantiated components (artifacts) that are reuseable outside of EMF.

1.5 Thesis Structure

The thesis starts by introducing the core problems and context in Chapter 1.

Next, a substantial amount of background material is presented in Chapter 2. This introduces Model-Driven Development and how it's practically done in an education context. Existing tools for working with EMF are presented, as they are prior art and crucial for a solution's design. Some terminology and theory is presented for trees and tree editors. Then EMF is described in more detail. After that, the VSCode and Theia IDEs are introduced, as they are central to the solution. Some existing protocols are presented, because they solve analogous problems to what this thesis identifies. A section will then follow, describing how the Eclipse ecosystem is itself targeting the cloud, and some relevant tools they provide. Lastly, the findings of the pre-project are presented. The pre-project laid a lot of the initial foundations for this master's thesis.

The next chapter details the method used to develop a solution. It explains the overarching methodology of Design Science Research, and the finer grained methods used for requirements engineering, development and artifact evaluation.

The following results section presents a developed software artifact, a software

architecture and a related protocol. It also explains the efforts done to increase the likelyhood of success as a open source project.

After the results, a chapter of evaluation follows. This sees how valuable, fit and complete the designs are, as well as how well the project follows open source project guidelines.

A discussion chapter argues for the implications of the results and evaluation, and the final chapter concludes this thesis.

Chapter 2

Background

This background section will explain some of the concepts, approaches, technologies and software architectures required to understand this thesis. The findings from the pre-project in [1] will also be presented in more detail than the introduction, as the findings are central to this thesis. Lastly, a section on open source software project management follows, as they shape many of the choices made in the implementation of a solution.

2.1 Conceptual Modeling and Model-Driven Development

Rationale Model-Driven Development (MDD) is the approach to software development which this thesis aims to support. Therefore, and understanding of MDD is beneficial, in order to see how an editor should work.

Modeling and abstraction The core of MDD is the model. The model is a human created construct, formed through humans working together to discuss and refine a problem domain until they reach a consensus of what abstractions help them solve the relevant problems [8, p. 154]. Humans perceive the world (and problem domain) as many different phenomena, and conceptual modeling is the act of trying to describe these at some level of abstraction [9, p. 1, 408]. The model is assumed to resemble the phenomena and work the same way, and yet be simpler than the real world [9, p. 414]. Abstraction means to find something common in different observations of a phenomena, and *generalize* their features, *classify* coherent clusters of objects and *aggregate* concepts into more complex ones [8, p. 1]. The model will never describe every aspect of the world perfectly, but can *reduce* the world down to relevant aspects, and easily *map* between model elements and real world phenomena [8, p. 1-2].

Modeling languages In order to describe the model, a *language* is used. To realize the benefits of MDD, a *formal language* is used. The language can be textual or graphical, or both, and imposes a formally defined syntax on the modeler [8,

p. 13].

Modeling tools The advantage of using a formal language is that it can be parsed and understood by software tools, as well as humans. The tools can validate the model according to the syntax, and to specific rules for the domain. Tools can also generate code, or execute the model itself. The model can be transformed into other models, or text or graphics [8, p. 8].

Model-Driven Development The central idea of Model-Driven Development is that the model is the source of truth that *drives* the rest of the engineering and development [8, p. 9]. There is not a separate model for analysis and for design, but a single one for both [10, p. 49]. The software code becomes an expression of the model itself, and changes to the code often happen as the result of changes to the model [10, p. 49]. Because the model and the software are so directly related, the MDD approach is heavily reliant on tools to automate the tasks of validation and code generation. The formal language may also sacrifice some of its human readability in order to be understood by tools [9, p. 232]. To solve this, one can use other tools that interpret, transform or present models in other ways [9, p. 233]. This increases the reliance on tools for MDD even more, including visual editors.

2.2 Model-Driven Development at NTNU in the Course TDT4250

Rationale Because the target audience of the software solution (tree editor) are students at NTNU, it is helpful to know how they work with Model-Driven Development. Their use cases are the ones being solved, meaning the solution must be made with this context in mind.

MDD at NTNU To do Model-Driven Development effectively, tools should be used. In the course "*TDT4250 Advanced Software Design*"[1] at NTNU, the chosen tools are in the Eclipse Modeling Framework (EMF) [11]. This includes the modeling language Ecore, visual editors in Eclipse IDE, model validation logic, the code generator named "GenModel"[2] (generator model), and more. EMF is a battle-tested technology also used in certain industries, and is well integrated with the Eclipse IDE. The course TDT4250 also uses Eclipse IDE as a case study for other software design concepts, such as modularity (plugin architecture) and dynamic systems (OSGi), and custom Domain-specific languages which automatically work with Eclipse IDE. EMF is relevant for most or all of those concepts.

[1] Course description is available at https://www.ntnu.edu/studies/courses/TDT4250#tab=omEmnet.

[2] The code generator is actually named "codegen", but users only see the configuration model called "GenModel".

Development methodology Students are taught a methodology or approach for how to do modeling. They start by specifying a problem space, for example bookkeeping an organization of employees or the courses in NTNU, and then abstract the problem into a model. The initial model is externalized as Ecore by using a tree editor in Eclipse IDE.

Then an *model instance* is made, based on the model, and filled with example data from the domain. This model instance is used to test and verify that the model is appropriate for the problem space. Adjustments are made to the model to accommodate any problems with the model instance.

Then validations can be created for the model, by one or both of the following approaches: writing Object Constraint Language (OCL) into model annotations, or marking the model element with an annotation and implementing it as java code. OCL is a Domain-specific language for navigating models and evaluating expressions, and the Eclipse IDE can detect annotations with OCL and evaluate them against the Ecore model. The other option, writing java code, requires the student to first create a new *genmodel* file from the model (by using a menu in Eclipse IDE), generating a java code project from the model, and then writing validation logic into the generated code. For the java code to be picked up, Eclipse IDE can start a new instance which installs the generated code as a plugin [12].

Next up, when the model is deemed sufficient, and the most important validations are in place, the student can try to create a user interface. One of several choices here is to create an *Eclipse IDE plugin*. EMF provides code generation for utilities used to integrate the model into an editor for Eclipse IDE. The student uses the genmodel to create these, and tweaks the code if wanted. Then everything is installed into Eclipse IDE by launching a new Eclipse IDE instance with the code installed as a plugin.

Lastly, the user interface can be tested. The student creates a new model instance file, enters some example data from the domain, and runs validation logic.

Lecture materials The steps mentioned in the methodology above are available online in [12–15]. This is an advantage, because they can by used used in this master's thesis as a basis for creating evaluations and acceptance criteria.

2.3 Eclipse Modeling Framework Editors for Ecore

Rationale These editors are the ones being re-implemented in cloud-based IDEs. Understanding their functionality and workings is important, as these editors shape the work of this thesis. The functionalities provided are assumed highly usable and good, because they are the result of many years of work and experience. They will be re-implemented in a solution.

Multiple editors When editing Ecore models in Eclipse IDE, there are different editors to pick from. Usually, Ecore models and model instances are saved as XML Metadata Interchange (XMI), which is a standardized serialization format based on XML. The Ecore models have the file extension `.ecore` while model instances either have `.xmi` or a custom extension for the model, specified by the modeler (e.g. `.organization` or `.courses`). The GenModel has `.genmodel` as file extension. However, Ecore models are rarely (if ever) edited as XML. Instead, the files are loaded and presented in a tree structure editor or diagram editor. These editors are specialized for Ecore, and can understand the model.

The diagram based editors use a notation that is based on UML Class Diagrams, with boxes, labels and arrows. Which editor to use can often be a personal preference. They are all functionally equivalent, with regards to modeling. The next subsections will describe the most common tree editors in more detail.

2.3.1 Sample Reflective Ecore Model Editor

The "*Sample Reflective Ecore Model Editor*" is one of the main Ecore editors in Eclipse IDE. A screenshot of the editor is shown in Figure 2.1. The model instances can be edited in a *reflective* editor (without the user first generating java code and installing an Eclipse IDE plugin). Here, reflective means that the editor uses a metamodel (see Section 2.6) for the model instance, and tries to infer the tree structure from containment relationships.

This editor can open both Ecore models and model instances. A screenshot of a model opened in the editor is shown in Figure 2.1a, and a model instance in Figure 2.1b.

This editor is open source[3], and the editor is itself originally generated by a genmodel [1, p. 10].

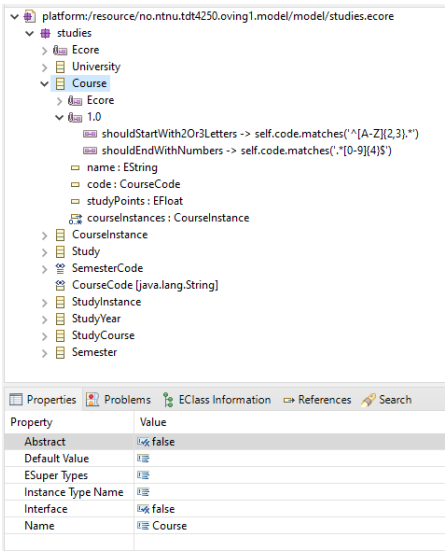
This editor internally uses a java class called `ReflectiveItemProvider`[4] from the `org.eclipse.emf.edit` EMF package, to extract text labels and infer icons for the tree view [1, p. 10].

For Ecore models (with `.ecore` file extension, not model instances), it uses an `EcoreItemProviderAdapterFactory`[5] to get labels and icons [16].

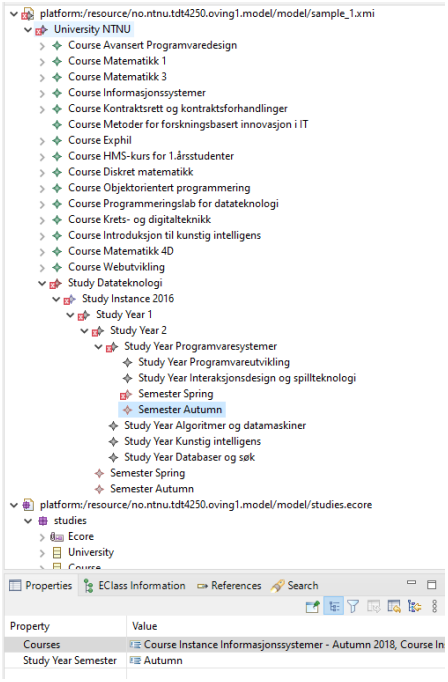
[3] Sample Reflective editor source: `https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.editor`.

[4] `ReflectiveItemProvider` source code: `https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.edit/src/org/eclipse/emf/edit/provider/ReflectiveItemProvider.java`

[5] `EcoreItemProviderAdapterFactory` source code: `https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.edit/src/org/eclipse/emf/ecore/provider/EcoreItemProviderAdapterFactory.java`



(a) A model opened in the editor.



(b) A *dynamic instance* (XMI file) opened in the editor.

Figure 2.1: Screenshots of the Sample Reflective Ecore Model Editor in Eclipse IDE.

These "item providers" are especially interesting, because they could be reused in a new editor.

2.3.2 EMF Forms Ecore Editor

The *EMF Forms Ecore Editor* is a newer editor than the Sample Reflective editor, and uses EMF Forms[6] as the technology to provide a user interface [3]. This editor is open source[7]. A screenshot of the editor is shown in Figure 2.2.

This editor is implemented as a generic editor for all Ecore model instances, and two subclasses that are specialized for Ecore and GenModel [3]. The generic editor is called *Generic XMI Editor* in Eclipse IDE, and the Ecore specific editor is called *Ecore Editor*.

The biggest difference compared to the Sample Reflective editor, is how the user

[6]More info about EMF Forms here: `https://www.eclipse.org/ecp/emfforms/index.html`.

[7]*EMF Forms* source code: `https://git.eclipse.org/c/emfclient/org.eclipse.emf.ecp.core.git/tree/bundles/org.eclipse.emfforms.editor.ecore`.

interface looks, and that the property sheet is customized based on a *view model file*. Customizing en editor based on a view model or UI schema is a general technique, also seen in JSON-Forms in Section 2.12.3.1. The Sample Reflective editor uses Eclipse IDE's built in property panel. In the EMF Forms editor, the properties are also grouped into *standard* and *advanced*.

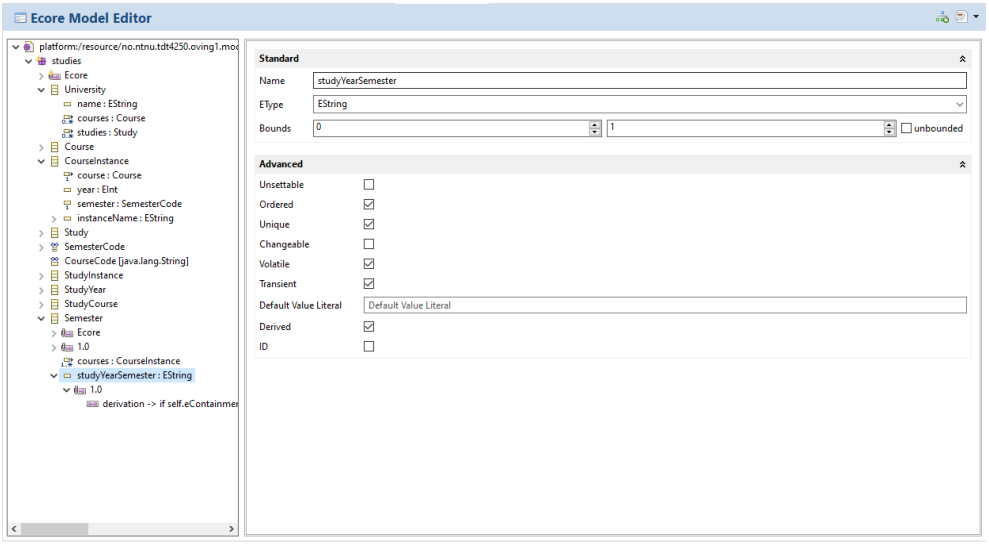


Figure 2.2: A screenshot of a model in the EMF Forms based Ecore Editor.

2.4 Introduction to Tree Structures

Rationale Because the editors center around a tree structure, a clear understanding of trees is helpful.

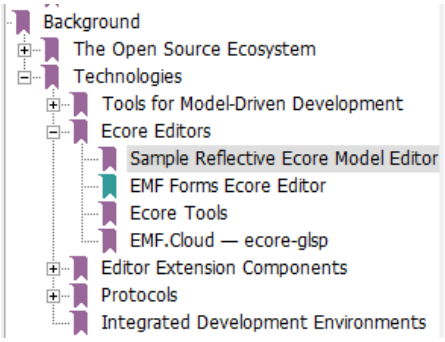
Trees A *tree* is a data structure. The tree is composed of *nodes*, and one node is designated as the *root node* node or *tree root*. Each node can have zero or more *children* nodes, and one *parent* node. The root node does not have a parent. When representing the tree as code, it is possible to omit either the parent or child relationship in a node, making the parent or child implicit. The relationship can still be found, by *traversing* the tree. Traversing means to visit every node it the tree by following the parent or child relationships. Nodes that are children of the same parent are called *siblings*, and parents of parents are called *grandparents*.

Visualizing trees There are many ways to present trees to humans. Two common approaches are *hierarchy* and *diagram*.

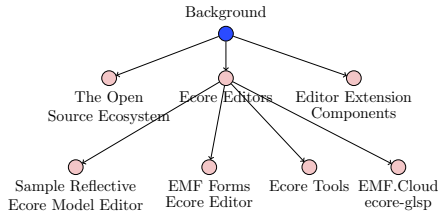
In a hierarchy, the parent is presented as a row, and its children on separate rows below (see Figure 2.3a). The children are often indented as well, and possibly

connected with dots or lines to the parent.

In a diagram, nodes are often displayed as a circle or box (see Figure 2.3b). The parent is displayed above its children, and the children are aligned on the same row. The parent-child relationship is shown as a line or arrow, connecting the parent to the child.



(a) A tree visualized as a hierarchy. The top node is the root.



(b) A tree visualized as a diagram. The blue node at the top is the root.

Figure 2.3: A tree visualized as a hierarchy and diagram. The labels are section titles of [1], as an example.

Nodes The tree is more useful when the nodes have properties. The minimum property is children or parent. But a useful property is a name, label or id, with regards to presenting the tree to a human. There may be properties on the relationships between a node and its children, but these may be hard to present visually in hierarchy-type visualizations. For a diagram type visualization, the properties may be presented as labels on the edge.

Mapping to trees A data structure can be mapped to a tree if it has separate objects with a references, containment or aggregation relationship. The references can not be circular (where a node has a child which is also a parent or grandparent etc.). There can be different ways to map to a tree, depending on what properties are used (or not used). The labels can also come from various object properties, be derived from them or combine multiple properties into one label.

Editing a tree Common operations on trees either modify the structure, or modify the properties of a node. Structural modifications can be to add a new child, to delete a child, or to move a child from one parent to another. Nodes can be copied, and pasted on the same parent or other parents, or themselves. Less common operations are inserting a new node between a parent and child, turning the latter into a grandchild. Likewise, a node can be removed, merging its children into its parent, making them effectively siblings to the removed node.

2.5 Master-Detail Tree Editor

Rationale The tree editors use a layout pattern called *master-detail*.

Description As the name *Tree Editor* implies, they are used to edit a tree. There are mainly two different things that can be edited: the parent-child relationships and the node's properties. The user interfaces for the tree editors in Section 2.3 use a pattern called *master-detail*. This means the user interface is composed of two parts: a *master view* and a *detail view*.

Master view The tree structure is shown as a hierarchy in the master view. It is common for the master view to be positioned to the left of a detail view, or above it. The user interacts with the master view to add, remove and select nodes. Adding a new child to a parent is done here.

Detail view When a node is selected, its properties are displayed in the detail view. It is common for the detail view to be positioned to the right of a master view, or below it. The detail view is usually a *input form* or tabular (rows and cells) structure. The user usually enters text, numbers, ticks checkboxes and opens selection dialogues from the detail view.

2.6 An Overview of EMF: Ecore Metamodel, XMI Serialization and GenModel for Code Generation

Rationale The Eclipse Modeling Framework (EMF) is the Model-Driven Development framework used in TDT4250. The tree editor will modify Ecore models, so it helps to understand the concepts and names used in the Ecore metamodel. It is also useful to know the different tools and components in EMF, because the tree editor intends to reuse as much of them as possible internally, to save development effort.

Eclipse Modeling Framework The Eclipse Modeling Framework (EMF) is a part of the Eclipse Modeling project from the Eclipse Foundation. It is a framework and code generation facility that lets developers define models. The models can be java code, XML Metadata Interchange (XMI) or UML, and the other two can be generated [17, p. 14]. This framework may be chosen as the tools for doing Model-Driven Development (see Section 2.1). In EMF, the models are expressed with the Ecore modeling language. This modeling language is similar to UML Class Diagrams, in terms of the concepts and what it can express [17, p. 16]. The real world data that could fit inside a specific model is called a *model instance*.
The framework was made to take use of the editing capabilities and utility of the Eclipse IDE [17]. This means that there is much tooling and integration for EMF

with Eclipse IDE. For example, EMF can generate a plugin to edit model instances in Eclipse IDE.

Ecore metamodel The modeling language in EMF is Ecore. A *metamodel* is the model of a model. This means that Ecore is the metamodel for all models expressed using . Ecore is itself modeled in Ecore, so it is its own metamodel.

Model concepts The main concepts used in Ecore to model, are `EClass`, `EAttribute`, `EReference` and `EDataType`[8]. These are distinct objects with names, properties and inheritance, like in object oriented programming. As for the metamodel, `EClass`, `EAttribute` and `EReference` are all extending `ENamedElement`, which defines their `name` property [17].
When modeling, `EClass` is used to create java classes. The `EAttribute` and `EReference` are used to model class properties, like member variables. An `EAttribute` defines a property, such as e.g. *age* or *address*, while `EReference` defines a reference/association to another `EClass`, e.g. *parent* or *order*. The `EAttribute` has a attribute type, the `EDataType`, which can be e.g. `EInt` or `EString` [17].
Java class methods are modeled with another concept, the `EOperation`. Lastly, everything in the model lives inside an `EPackage`, which represents a java package (or other kind of code module). There are more concepts in Ecore, but many are only used internally as part of the metamodel, to represent Ecore itself.

XMI serialization When an Ecore model is written as a text file, it needs *serialization*. The official format for serializing Ecore is XML Metadata Interchange (XMI). This format is based on Extensible Markup Language (XML). The file extension is usually `.ecore`. Model instances can also be serialized as XMI, and have custom file extensions or `.xmi`. It is also possible to serialize Ecore to other formats, like JSON, using third party tools.

EMF runtime java API The java code generated by EMF will by default extend a set of java classes defined by EMF. Instead of a generated `EClass` extending `java.lang.Object`, it extends `EObject`. And instead of using an `ArrayList`, a collection in Ecore will use a `EList`. When creating a new instance, the class constructor is not used, but a Factory instance on the generated `EPackage` for the model.

All of these framework java-classes are the EMF java Application Programming Interface (API). They provide much of the power, flexibility, reflection and metamodeling capabilities of EMF in java. For example, a program can work with a EMF model without knowing the code beforehand, by using the reflection API to retrieve names and properties of a model object.

[8]The name Ecore comes from EMF Core, and the 'E' prefix for `EClass` etc. come from Ecore.

The API also provides utilities for working with the model. There are APIs for listing the children of an `EObject`, getting a human representation of it, and for modifying and observing state changes. Another important API is the `ResourceSet` and `Resource`, used to read and save models to serialized XMI files.

GenModel code generation Code generation is an important part of EMF. The generator can be configured with its own generator model, nicknamed the *GenModel*. This model holds options for how the code will be named, what templates should write the code, if the code can use the EMF APIs, and more. This model is itself an Ecore model, and has an `.genmodel` file extension [17, p. 28].

The generator can also produce more than just a java representation of the model. A test suite can be generated, with an empty test skeleton for the generated code. It can also generate utilities for creating model editors, in what is called the *.edit* java package. The name ".edit" is appended to the original package name. This has *ItemProvider* classes which helps an editor to find the human representations, properties, child objects, and to notify on changes.

Another utility is related to the Eclipse IDE, which is the *.editor* java package. This holds key classes for integrating with Eclipse IDE, making it a custom editor. For example, custom actions, project wizards, eclipse plugin logic is part of this.

Custom code The generated code must usually be modified by a developer. This can be to fill in the implementation of a `EOperation`, or tweak some behavior. The generated code has a `@Generated` java annotation, which the developer changes to prevent the code generator from overwriting the method body.

2.7 Visual Studio Code and Theia

The two IDEs relevant for this thesis are Visual Studio Code (VSCode) and Theia. Both are available as editors in Gitpod as cloud based IDEs.

2.7.1 Visual Studio Code

VSCode is a very popular open source IDE created by Microsoft [18]. A screenshot is shown in Figure 2.4. It uses web technologies like javascript, NodeJS and Electron to provide an advanced text editor and tools for programming on a desktop. Originally made only for desktop, VSCode was later adapted to also work in a browser when GitHub[9] launched Codespaces [19]. VSCode is extensible, and allows third party developers to create extensions. These are distributed from Microsoft's extension store: Visual Studio Marketplace[10].

[9] GitHub is owned by Microsoft.

[10] Marketplace website: `https://marketplace.visualstudio.com/vscode`.

Programming languages One common use of extensions is to support new programming languages. The text editor in VSCode is a generic text editor component called *Monaco* [20]. This same text editor is used for all programming languages. For the text editor to know the keywords, suggestions and other specifics of a programming language, the extension uses a standardized protocol to inform Monaco. This protocol is called the Language Server Protocol, and is described in Section 2.9.

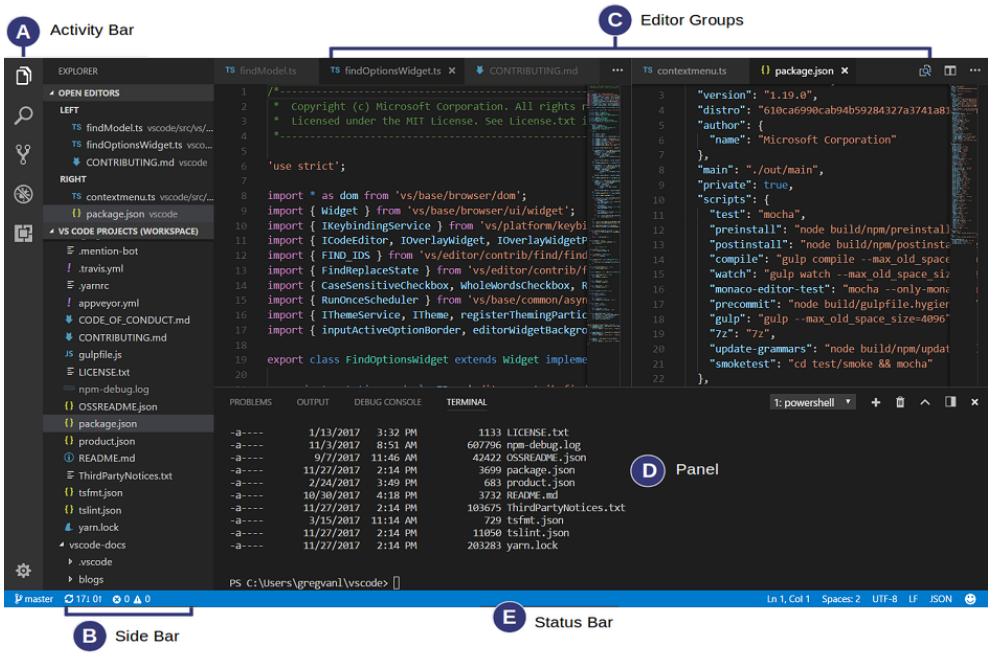


Figure 2.4: The VSCode user interface, annotation with the different components (A-E).

2.7.2 Theia

Theia is based on the open source components from VSCode, without a proprietary component that Microsoft added for telemetry. Theia is managed by the Eclipse Foundation under the *Cloud Development project* (see Section 2.12), and was created to be web based from the start (before Codespaces launched, when VSCode was desktop only). A screenshot is shown in Figure 2.5. The main uses of Theia are workspace services like Gitpod and Eclipse Che, but it is also intended to be a "web based version" of the Eclipse Rich Client Platform. This means tools can create their own distribution of Theia, where they are deeply integrated [21].

Extensions Theia can load extensions using the same Application Programming Interface (API) as VSCode. Theia calls these "Theia Plugins". Another way to extend Theia is using "Theia Extensions". These have full control over the IDEs, and

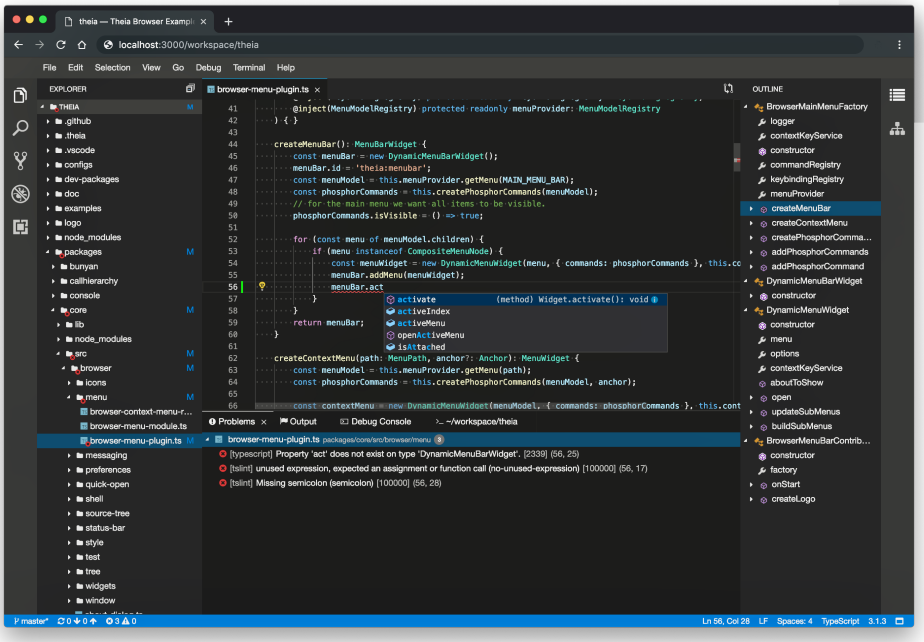


Figure 2.5: The Theia user interface.

can modify practically anything. Installing a Theia Extension requires the user to perform a full compilation of Theia itself [22]. A Theia Plugin (or VSCode extension) however, can be installed at runtime. Because of licensing issues with Microsoft and the Visual Studio Marketplace, Theia Plugins are instead hosted at a independent marketplace called *OpenVSX* [23].

2.8 Visual Studio Code's Custom Editor API

A VSCode extension is allowed to use a set of Application Programming Interfaces provided by VSCode. One such API is the *Custom Editor API*. This allows an extension developer to create **custom editors other than text editors**. This could be diagrams, pictures, graphs, or **trees**, for example. The developer has the full freedom of a web browser, as they are given their own isolated frame. Normally, an extension cannot modify the user interface outside of the provided APIs. This is in contrast to inside the provided `WebView`, where the developer has to *create and manage* the entire user interface. In addition to a user facing `WebView`, the developer must create their own document model. By default, VSCode uses a document model for text documents, with selections, edits, versions and more. The `CustomDocument` only has a uri pointing to the file. Another central part is the `CustomEditorProvider`, with a few methods to fill in, like opening, undoing and saving a document.

2.9 Language Server Protocol Architecture

Goal There are many programming languages, and many Integrated Development Environments. Traditionally, every IDE would have a special integration for every language it supported. Extracting tokens, keywords, providing auto completion, code formatting and so on. This leads to a lot of rework every time a new IDE comes around, and duplication of work every time a new programming language is supported. Essentially, every m number of IDEs that support an n number of programming languages result in $m \times n$ different integrations. This is illustrated in the left side of Figure 2.6.

A solution to this $m \times n$ problem is the *Language Server Protocol* (LSP). If instead, every IDE has a generic text editor for all languages, they only need to support the LSP. Once an editor "talks" LSP, it can support **all programming languages** that have a LSP *language server*. Likewise, a programming language only needs to develop **one language server**, and it supports all IDEs that use LSP [24]. This is shown in the right side of Figure 2.6.

This protocol was created by Microsoft, and is in use today on VSCode. Many IDEs and text editors have adopted it afterwards, like Eclipse IDE (with LSP4E), Atom, Vim, Sublime Text, Spyder and more, via both official and unofficial plugins to these IDEs [25]. The protocol is quite extensive, and defines approximately 40 different requests with corresponding responses, 20 notification types, in addition to data structures needed to support all of these [26].

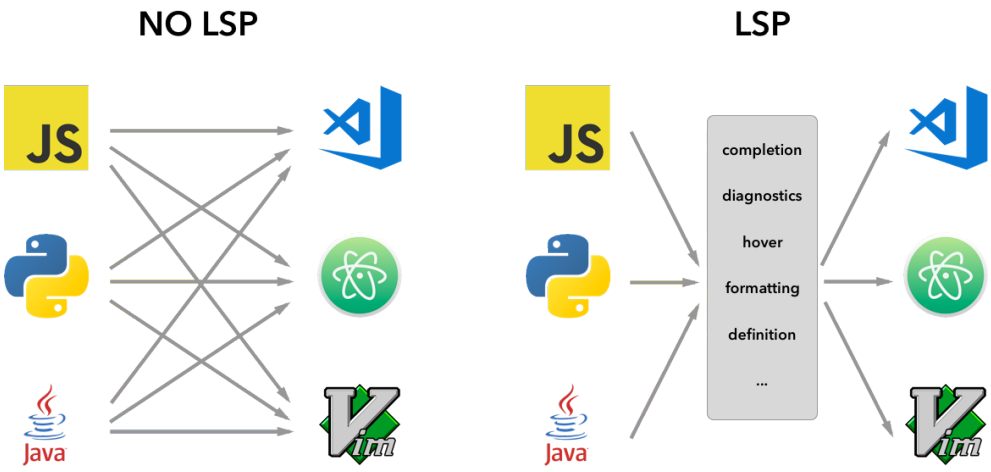


Figure 2.6: The benefits of using the LSP. The left side shows all the integrations (as arrows) required for 3 languages (javascript, python, java) and 3 editors (VSCode, Atom, Vim), without the LSP. The right side shows how the LSP can reduce the amount of work by unifying the common elements of programming language editors into a standard protocol. Figure copied from Microsoft [27].

Protocol The Language Server Protocol is based on a *Base Protocol*. This Base Protocol is similar to HTTP, in that it has a *header* section and a *content* section. The content section contains Remote Procedure Calls (RPCs), using a protocol called JSON-RPC. This is shown in Figure 2.7.

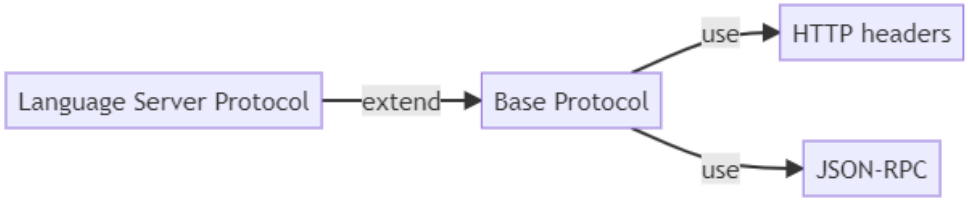


Figure 2.7: The Language Server Protocol protocol extends a Base Protocol with JSON-RPC content.

2.9.1 Base Protocol

All communication in LSP uses concepts from the Base Protocol. This protocol has a header and content section, as mentioned above. Conceptually, the protocol assumes there is one *client* and one *server* which communicates. Note that the server can also initiate requests to the client. In addition, the Base Protocol defines specific types of messages: *Request Message*, *Response Message*, *Notification Message*, and *$ Notifications and Requests* [26].

Header The header is comparable to a HTTP header, with key-value pairs separated by colon, and a line break for each new pair. The currently supported header keys are `Content-Length` and `Content-Type`. The `Content-Length` specifies how many bytes the content is [26].

Content The content section contains the actual message data, like requests and responses. This section follows the JSON-RPC protocol, described later in Section 2.10 [26].

Request and Response A Request Message describes a request from a client to the server. This must have an ID, a method name (for Remote Procedure Call (RPC)) and parameter values for the method. When a client sends a Request, it means that the server should execute the given method with the given parameters. The server must then respond with the results of the execution in a Response Message. This Response must have the id of the originating Request, as well as the results or an error [26].

An example of a Request is shown in Code listing 2.1. It is the `textDocument/signatureHelp` method, specifying a `textDocument` and `position` with parameter values for the `textDocument/signatureHelp` method call.

Code listing 2.1: A Request Message Example

```
Content-Length: 201

{
    "jsonrpc":"2.0",
    "id":"1",
    "method":"textDocument/signatureHelp",
    "params": {
        "textDocument": { "uri": "file:/..." },
        "position": { "line": 5, "character": 3 },
    }
}
```

Notification A Notification Message is more like an event. It does not have an ID, and does not get a Response Message in return. The Notification, like the Request, specifies a method and parameter values [26].

$ Notifications and Requests If a Notification or Request has a `$/` at the start of the method name, it is an optional and protocol implementation-specific message. Not all clients and servers handle these messages. A notification can be ignored, and a request must be answered with a specific error, if the message is not implemented.

2.9.2 Language Server Protocol

The Language Server Protocol (LSP) defines JSON-RPC requests, response and notification messages that are sent in the Base Protocol. These are specified as method names and parameter values, as well as semantics and rules related to the sequences, responses to, and content of these messages. LSP also defines a set of JSON data structures, which are used in the messages as parameter values and response types [26]. The protocol is versioned, where `3.16` is the current version.

The LSP defines many messages, related to these categories:

- Window
- Telemetry
- Client
- Workspace
- Text Synchronization
- Diagnostics
- Language Features

The most important category is Language Features, which define Requests such as: completion, hover, signature help, references, code action, formatting, rename, and more. The full list is available in the *LSP Specification* [26].

2.10 JSON-RPC

JSON-RPC is a stateless and lightweight protocol for doing Remote Procedure Calls (RPC). It works over any transport mechanism that can send and receive text. The data in JSON-RPC is sent as JSON, an object structure serialization format originally from javascript [28].

RPC is a technique to start a procedure on a remote server, as the name suggests. A procedure is synonymous with a function or method in programming. In JSON-RPC, they are called by specifying the name and the parameters in a *Request object*. A request object must have the properties `jsonrpc` and `method`, and `id` if it is not a notification. It may have `params`. Notifications, as explained for LSP, do not need a response. The *Response object* must have the properties `jsonrpc`, `id`, and either `result` or `error`. The result is the return value for the called procedure. The error is an *Error object*, with `code`, `message` and `data` [28].

The request parameters in `params` are either a list of positional parameters, or an object with named parameters. If there are no parameters, they can be omitted [28]. An example of several JSON-RPC messages are shown in Code listing 2.2. The arrows indicate the direction.

Code listing 2.2: JSON-RPC examples copied from [28].

```
Syntax:
--> data sent to Server
<-- data sent to Client

RPC call with positional parameters:
--> {"jsonrpc": "2.0", "method": "subtract", "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}

RPC call with named parameters:
--> {"jsonrpc": "2.0", "method": "subtract",
     "params": {"subtrahend": 23, "minuend": 42}, "id": 3}
<-- {"jsonrpc": "2.0", "result": 19, "id": 3}

Notifications:
--> {"jsonrpc": "2.0", "method": "update", "params": [1,2,3,4,5]}
--> {"jsonrpc": "2.0", "method": "foobar"}
```

2.11 Cloud and Gitpod

Cloud The cloud is the term used for rented computing power and data storage in data centers owned by third parties. This is in contrast to in-house or on-premise servers. An advantage of running software in the cloud is that maintenance of hardware is outsourced. If a hard drive or processor breaks down, it is the cloud vendor's responsibility to fix, and to provide failover mechanisms so a customer is not impacted.

Another advantage is the ability to scale up or down instantly on demand. If a on-premise server is overloaded, the organization has to purchase more servers and configure them. Just the shipping of hardware alone takes more time than requesting more compute power from a cloud provider. The cloud providers usually have so large data centers that they never "run out", as long as a customer is able to pay for it. Some of the best known cloud providers today are Amazon with Amazon Web Services, Google with Google Cloud, and Microsoft with Azure.

Gitpod Gitpod is a cloud based Integrated Development Environment (IDE). It is provided as a service, or it can be self hosted. The idea behind Gitpod is that a developer does not need to install the tools on their own machine. Instead, a machine is provisioned at a cloud provider, and any tools are installed there. The developer interfaces with this machine through a web based IDE. For Gitpod, the default IDE is Theia. The source code is downloaded from an online source code host, such as GitHub, and into a workspace on the provisioned machine.

2.12 Eclipse Modeling Framework in the Cloud

The Eclipse Cloud Development project As mentioned in the introduction (Section 1.2), the Eclipse ecosystem is interested in running software in the cloud. This means that they have spent the last few years creating tools to support cloud oriented deployments for software built on Eclipse Modeling Framework. The Eclipse Foundation has an umbrella project called *Eclipse Cloud Development*. In the Eclipse Foundation, a project is not a single codebase, but rather a home for frameworks, tools and components. Under this umbrella exists projects like *EMF.Cloud*, *Eclipse Che*, *Eclipse GLSP*, *Eclipse Theia*, *Eclipse OpenVSX* and more [29].

2.12.1 EMF.Cloud

The Eclipse ecosystem found it suitable to create a new project under this umbrella, and called it *Eclipse EMF.Cloud*. The description for Eclipse EMF.Cloud starts with the following:

> "*Eclipse EMF.cloud comprises a set of components that facilitate and simplify the adoption of the Eclipse Modeling Framework (EMF) in cloud-based applications.*
> *[...]*
> *As a consequence, by its nature, EMF.cloud is open to any software project that aims to address the challenges and specific requirements of using any aspect of EMF in a browser-based setting or cloud deployment.*"
> — Smith [30]

EMF.Cloud software The components provided by EMF.Cloud are still in active development. Most of them center around building a modeling environment in

Theia, for existing EMF models. The example case that is used is a “Coffee brewing model”. Because much of the work targets Theia, the Eclipse ecosystem uses Theia Extensions. This means they can not be used in Gitpod, because the IDE has to be replaced with their customized Theia. However, much of the work here is still relevant, as components to use in a VSCode extension, and as design to draw inspiration from.

The EMF.Cloud project currently provides these components, according to [31]:

- modelserver
- modelserver-theia
- model-validation
- coffee.editor
- ecore-glsp
- theia-tree-editor
- json-forms-property-view
- modelserver-glsp-integration
- emf-jackson

The most relevant components for this thesis are detailed in the following subsections.

2.12.1.1 Model Server

The EMF.Cloud Model Server provides a web server for working with EMF models. While EMF already support model loading, manipulation and serialization in the EMF runtime API, this server exposes these to the web. It does so by providing a REST API for working with models, and WebSocket channels for subscribing to change events. The Model Server also manages a “shared editing domain” for the loaded models, and changes models using EMF `Commands` [32].

This Model Server is already used in other EMF.Cloud components, like the coffee editor and ecore-glsp [33, 34].

2.12.1.2 Theia Tree Editor

Theia Tree Editor is a framework for creating master-detail tree editors [32]. It uses the Theia extension mechanism, and uses core components of Theia itself [1]. This hinders reuse in other IDEs like VSCode. However, the data structures and configuration schemas used in the Theia Tree Editor are good sources for design inspiration. A diagram of its `Node` interface (for tree nodes) is shown in Figure 2.8[11]

[11] Node source code: `https://github.com/eclipse-emfcloud/theia-tree-editor/blob/3da9d6a3c58cad140c228408b92a554fe5dd1b41/theia-tree-editor/src/browser/interfaces.ts#L30`.

[12].

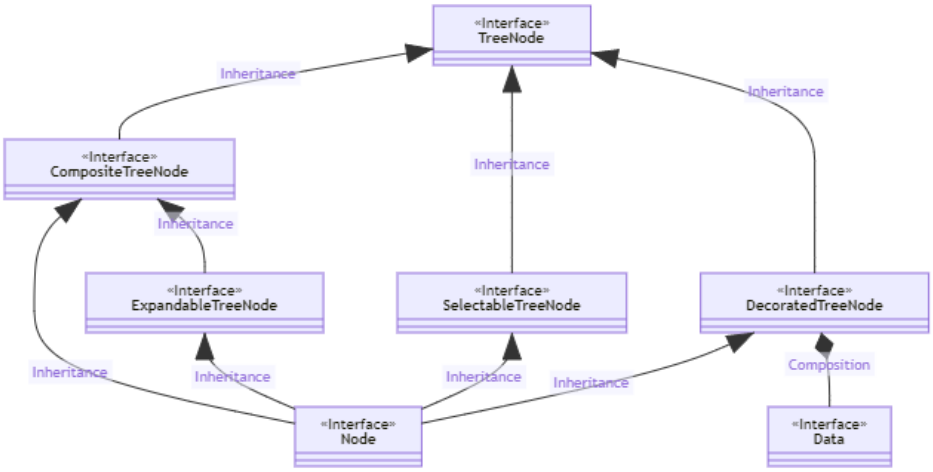


Figure 2.8: A class hierarchy of the tree nodes in Theia Tree Editor. Class properties and methods are not shown. The `Node` interface extends several different interfaces, picking up various properties from each. Only the `Node` interface is in the Theia Tree Editor library itself. The other interfaces are in the core Theia codebase. Adopted from "Figure 2.7" in [1, p. 15]

2.12.1.3 Coffee Editor

The Coffee Editor is an example application, trying to demonstrate the use of EMF.Cloud components in a real cloud deployment. This editor uses Theia, JSON-Forms, GLSP, a code generator, and the Model Server [32]. This editor is interesting because it applies the technologies, demonstrating their use, purpose and value. It also demonstrates the use of a Model Server shared among multiple editing components, like the GLSP and Theia Tree Editor working on the same backing coffee EMF model instances.

2.12.2 Graphical Language Server Platform (GLSP)

This is another project under the Eclipse Cloud Development project. The Graphical Language Server Platform (GLSP) is a framework for building diagram editors in the web. The editors can either be standalone or integrated into Theia and VSCode. The GLSP defines its own Language Server Protocol (LSP) for diagrams [35]. A figure from the official website is shown in Figure 2.9.

[12] SelectableTreeNode source: `https://github.com/eclipse-theia/theia/blob/af9b883dd929c79c1593bf4bd526df11600e21cf/packages/core/src/browser/tree/tree-selection.ts#L109`.

This is a good source of design inspiration, because it both works with EMF models, and it applies the Language Server Protocol architecture to a new domain other than text editing.

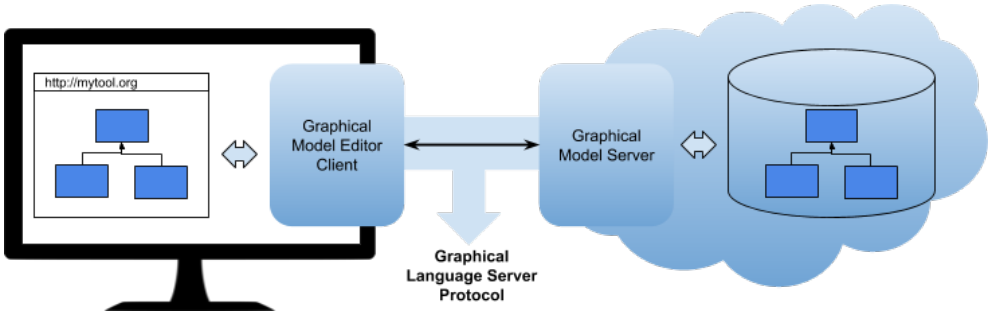


Figure 2.9: A diagram of GLSP. A web based diagram editor shows the diagram obtained from a Graphical Model Editor Client. This client talks to a graphical model server over the Graphical Language Server Protocol. Copied from [35].

GLSP Protocol The LSP-based protocol has a very small "surface area". The protocol uses the same Base Protocol as defined in LSP (see Section 2.9). The java implementation also uses the same libraries as Eclipse's Java LSP server: the LSP4J jsonrpc library. The server side of the GLSP protocol is shown in Code listing 2.3. Compare this to LSP, which has about 40 Requests and 20 Notifications — this GLSP protocol only has two requests and one Notification. Instead of defining many different methods, the "meat" of the protocol is inside the `process` method. The `ActionMessage` holds an `Action`, which is an abstract class with a `kind` field. This `Action` is extended (subclassed) to about 20 different versions. These have names like `FitToScreenAction`, `CenterAction`, `SelectAction`, `SaveModelAction` [36]. The GLSP client and server implementations rely on *action handlers*. The Grphical Modeling Editor Client (usually inside VSCode or Theia) can decide if an `ActionMessage` from the diagram viewer should be forwarded to the Graphical Model Server or not. Sometimes the action can be performed entirely inside the client. The same applies for messages from the server, which can be forwarded to the diagram editor or stop in the Graphical Model Editor Client [37].

Code listing 2.3: GLSP Server java interface. Copied from [38].

```
package org.eclipse.glsp.server.jsonrpc;

import java.util.concurrent.CompletableFuture;

import org.eclipse.glsp.server.actions.ActionMessage;
import org.eclipse.glsp.server.protocol.GLSPServer;
import org.eclipse.glsp.server.protocol.InitializeParameters;
import org.eclipse.lsp4j.jsonrpc.services.JsonNotification;
import org.eclipse.lsp4j.jsonrpc.services.JsonRequest;
```

```
public interface GLSPJsonrpcServer extends GLSPServer<GLSPJsonrpcClient> {
   @Override
   @JsonRequest
   CompletableFuture<Boolean> initialize(InitializeParameters params);

   @Override
   @JsonNotification
   void process(ActionMessage message);

   @Override
   @JsonRequest
   CompletableFuture<Boolean> shutdown();
}
```

2.12.3 Other Tools by the Eclipse Ecosystem

Not all the efforts by the Eclipse ecosystem are made under the Eclipse Foundation's management. Some projects exist outside this, in code repositories belonging to individuals and organizations that work with EMF. Some of the most relevant software projects are described below.

2.12.3.1 JSON-Forms

The purpose of JSON-Forms is to easily create user interfaces for data entry in the web, using HTML forms. A screenshot of such a form is shown in Figure 2.10. JSON-Forms is a project by the EclipseSource organization. This is the same organization that created the EMF Forms based Ecore editor for Eclipse IDE, described in Section 2.3.2.

JSON-Forms is using the same core approach as EMF Forms, where the view is described in a declarative fashion with a *UI schema*. This schema describes a data entry form. It describes the input fields, their labels, what data they effect and the grouping of view elements [39].

In addition to a UI schema, a form using JSON-Forms needs a *JSON schema*, which describes the types, structure and validation rules for the underlying data. Together, these two schemas are enough for JSON-Forms to render and edit a JSON data object in a user interface.

2.12.3.2 CrossEcore

CrossEcore is a project by Simon Schwichtenberg with the aim of cross platform code generation using EMF. It targets the programming languages C#, TypeScript, JavaScript, and Swift. CrossEcore also implements the EMF runtime API for these languages, as well as an Object Constraint Language (OCL) compiler [40].

This project is relevant because it can generate TypeScript code from Ecore models, and has also experimented with creating online editors [41].

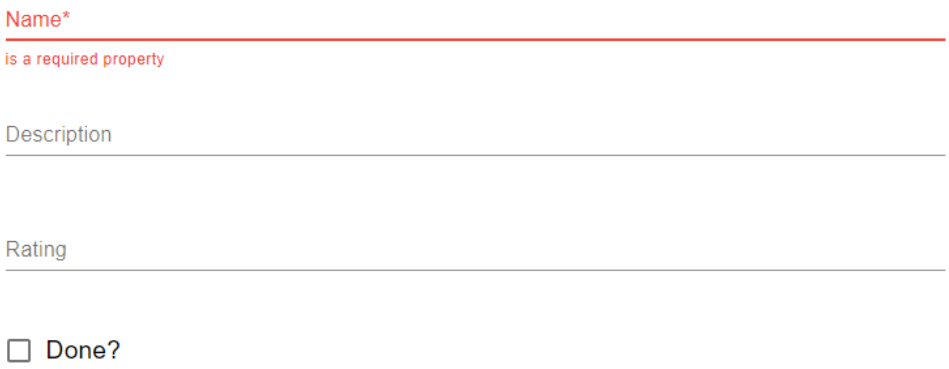


Figure 2.10: Example of a form rendered with JSON-Forms. Adopted from [39].

2.13 Pre-project Results

2.13.1 Research Questions

The pre-project started by asking the following research question:

> *How can we modernize Model-Driven Development Frameworks to appeal to the next generation of software developers, using recent developments in cloud IDEs?* [1, p. 3]

After answering this question, the pre-project narrowed down to the following research question:

> *How can we design an Ecore master-detail tree editor that works in both VSCode and Theia, while reusing existing tools for Ecore such as codegen and validation?* [1, p. 24]

A set of five related sub-questions were also posed, and subsequently answered. These questions set the context for a solution, and the stakeholders, constraints and requirements that would be needed. The following subsections (2.13.2 to 2.13.4) will present the results of the pre-project that are the basis of work done in this thesis.

2.13.2 Stakeholders

A stakeholder is someone affected by or interested in the solution. It can be an organization or people [42, p. 52]. The pre-project identified the key stakeholders in [1, p. 3] to be:

- Kristian Rekstad (author). Goal: increase adoption of MDD by students and industry. Has to design and develop the initial solution.
- Hallvard Trætteberg (supervisor). Goal: teach students the concepts of MDD in TDT4250. Wants to use Gitpod and EMF for student assignments.

- Teachers/Lecturers that use EMF. Goal: teach students. Have to present the tree editor, use it and support students that ask for help.
- Students. Goal: learn useful technologies and pass courses like TDT4250 to get a grade. Will have to use EMF if they study Computer Science with the "Software Engineering" specialization at NTNU.
- Industry professional using EMF to do MDD. Goal: develop software for a business/client. May want to use EMF without using Eclipse IDE, for personal reasons or organization policy.
- Eclipse Foundation. Goal: foster a community of developers and provide open source software. The maintainers of EMF.
- Eclipse ecosystem developers. Goal: contribute to Eclipse Foundation projects. May possibly have to maintain and further develop this (or a derivative) solution if this project succeeds and they embrace it.
- Developers of third party VSCode extensions that use tree editors. Goal: provide a high quality editor for their specific problem domain. Could use the architecture, protocol and frontends of this solution, if this solution is high enough quality, architected to be reusable and partially independent of EMF, and reuse will reduce their design and/or development time.

2.13.3 Software Requirements

Requirement engineering approach The pre-project tried to establish the software requirements for a tree editor. A literature review failed to find related works that listed the requirements for a tree editor. The literature review also failed to find related works for modeling in the cloud with the purpose of creating Ecore models. The related works either *deployed* Ecore models to the cloud, were textual editors, or did not use Ecore [1, p. 3].

Without literature to suggest requirements, and without users to test on (except the author and supervisor), the best option was to analyze the existing tree editors in Eclipse IDE. Common modeling tasks were performed (see Section 2.2), and detected functionality was recorded. The result gave an initial list of functional requirements, but not a complete one. However, by following an agile approach instead of waterfall, this list does not need to be complete[13]. More requirements will emerge naturally as work progresses. Still, having a good overview of the requirements is needed to correctly decide a software architecture, because of "architecturally significant requirements" that affect the architecture [42, p. 291].

Constraints A constraint is a restriction on the available choices for a solution [43, p. 7]. The most important constraint discovered was that the tree editor must be a VSCode extension. There is an alternative extension mechanism for

[13]Agile values working software over extensive documentation, thus spending time on creating a working solution is better than a "worthless" list of everything a solution *could have done*.

Theia, which was deemed incompatible with Gitpod[14] [1, p. 38].

Functional requirements A functional requirement specifies *what* a solution must do, such as supported features [43, p. 7]. The pre-project identified several functional requirements for a tree editor in the cloud. The full list of functional requirements, with id, requirement and description can be found in Appendix A. The list can be summarized as follows:

- Provide a master-detail tree editor in VSCode and Theia (Gitpod) by using an extension mechanism of the IDE.
- The tree editor must show nodes with labels and icons as a hierarchy.
- Allow selecting a node in the tree editor by clicking it.
- Provide a property sheet for the selected node in the tree editor.
- Provide an action bar with actions that can be dynamically specified by a backend server.
- Child nodes can be hidden or shown in the tree by a user.
- The tree editor and property sheet must update when the underlying model changes in the server.
- The action bar shows appropriate actions based on the selected node.
- The tree editor must allow creation of new nodes.
- The tree editor must allow deleting nodes.

Some more important requirements were implicit, and not defined in the list. This was not intentional, and an evidence to the list's non-completeness. Some of the implicit requirements are explicitly defined as follows:

- The editor must handle Ecore models.
- The editor must handle model instances from XMI files.
- The tree structure must be based on containment properties in the Ecore model.
- The editor must provide a command in the IDE to create a new Ecore file with the minimum XMI needed for a valid empty model.
- Tree nodes can be moved to new parents by drag-and-drop by the user.
- The drag-and-drop can not let the user drop a node on a parent that cannot contain the node as a child.
- Saving a model will serialize it as XMI to a file on disk.
- An action in the action bar must be added to run *model validation*.
- An action in the action bar must be added to run *code generation*.
- The editor shall show multiple tree roots when there are related model files. Opening a Ecore file shall also show any genmodel file. A model instance shall also include a root for the Ecore model in the same editor.
- A user can open more than one unique Ecore model at the same time, in

[14] Gitpod can use Theia as its editor frontend, but the user is not allowed to recompile and upload a new version of Theia. The alternative extension mechanism, *Theia Extensions* needs a full recompilation of Theia [1, p. 38]. However, VSCode extensions can be installed during runtime, also in Theia in Gitpod.

separate “tabs” in the editor.
- Any modification to the model must support undo and redo.

Non-functional requirements A non-functional requirement specifies characteristics or properties of the solution [43, p. 7]. Most of the non-functional requirements are grounded in empirical evidence like what the Eclipse ecosystem and web development ecosystems are currently doing. A non-formal list of the non-functional requirements is as follows:

- Compatibility with a code editor in Gitpod.
- Use a permissive open source license.
- Avoid software dependencies that are closed-source or use restrictive licenses.
- Use a distributed architecture with components reusable in other IDEs, inspired by architectures already in use by similar solutions [1, p. 24].
- Configurability of user-facing options. Choices of colors, fonts, file system paths and similar should be possible to change [1, p. 24].
- Configurability of mapping of Ecore models to trees. Which containment references to use as children, and custom logic for labels should be user-specifiable [1, p. 24].
- Localize the user interface in English.
- Flexibility and extensibility in the protocol to the server, allowing custom messages [1, p. 24].

2.13.4 Architecture and Protocol for a Solution

A specific software architecture was proposed. It had a goal to solve the requirements for: software reuse, Theia and VSCode compatibility, tree hierarchy editing, and a solution that could be transferable to other tree domains and editors. Additionally, the protocol would try to stay close to related solutions, as they are empirically tested and familiar to developers in the Eclipse ecosystem. By creating prototypes, major “blockers” or risk factors of the proposed architecture was tested and proven non-problematic, such as creating a custom frontend and running java programs [1, p. 38-46].

The presented architecture is a suggestion, but is not validated or implemented. It may need changes, and implementing it requires a substantial effort.

2.13.4.1 Architecture

The tree editor shall be a VSCode extension, and use the available Application Programming Interfaces for VSCode extensions. No dependencies to Theia shall be introduced. To view the tree structure as a hierarchy, the *Custom Editor API* in VSCode must be used (see Section 2.8). A custom editor can present the editor

inside VSCode as a *WebView*, meaning a custom webpage free to render anything, isolated from the rest of VSCode.

Components The editor thus comprises 4 components: the tree editor WebView ("**editor frontend**"), the VSCode extension integration ("**extension**"), a *Tree Language Server* ("**TLS**") and the *EMF.Cloud ModelServer* ("**ModelServer**") [1, p. 48, 49]. An illustration is shown in Figure 2.11

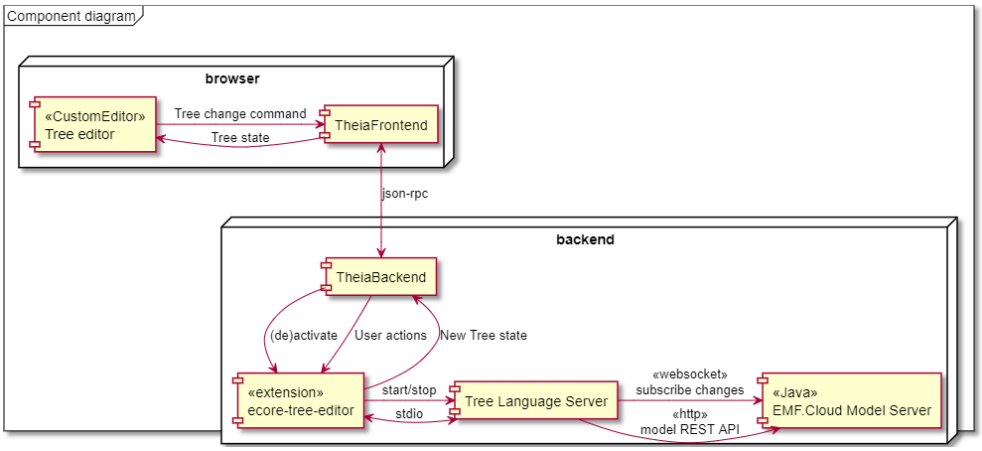


Figure 2.11: A suggested architecture for a tree editor. Copied from Figure 5.15 [1, p. 49]. The diagram is based on Theia, and the JSON-RPC between TheiaFrontend and TheiaBackend happens behind the scenes, and is therefore not relevant to discuss.

Editor frontend The editor frontend must be a web application that renders the tree as HTML, and provides interactivity with javascript. It communicates to the extension using *messages* containing JSON[15]. The editor frontend will send messages that are *commands* with the changes or actions a user triggered. The extension will send messages with the new tree state to be shown.

Extension The extension is the main artifact which a user will install into VSCode or Theia. This must be implemented with the TypeScript language or javascript. The extension will bundle the compiled code for the editor frontend, TLS and ModelServer inside it. The extension is responsible for integrating with the IDE, so that model files are opened in the custom editor, and handles commands triggered by the user in the IDE (such as actions to create a new file, or saving a model). The extension will start and stop the TLS process, and communicate over a JSON-RPC protocol or REST plus WebSocket. The extension and TLS can either use TCP sockets or standard in/out as the transport.

[15]This is a constraint imposed by the WebView API in VSCode.

TLS A Tree Language Server (TLS) will contain specific knowledge about Ecore and EMF. The choice of programming language is unconstrained. The main purpose of the TLS is informing the extension of tree state, provide configuration for the tree editor and extension, and receive commands to modify models. The TLS must be able to communicate using the same protocol as the extension (JSON-RPC or REST and WebSocket), and either do so over standard in/out, or listen to incoming TCP sockets. The TLS will perform commands to modify EMF models by using as much re-use of existing code and frameworks as possible. The TLS will start the ModelServer and communicate with it to read and modify models. The ModelServer is a main part in the strategy to re-use existing code.

ModelServer This is a component already made by EclipseSource in java. The ModelServer exposes REST endpoints for working with EMF models, such as listing models, reading models, and changing models with the EMF Commands framework. Changes to a model are exposed as WebSocket endpoints which can be "subscribed" to.

2.13.4.2 Protocol

The communication between the extension and the TLS should follow a defined protocol. The protocol will contain the data structures and message formats to send, the serialization standard to use for messages, and define any required order for messages.

Base protcol The pre-project did not progress far enough to formally define this protocol, and did not implement it (or an alternative) either. However, it specified a starting point and some rules. The protocol should draw inspiration from the Language Server Protocol, and use the "Base Protocol" defined in LSP (see Section 2.9.1). The Base Protocol has two parts: a HTTP header section, and a JSON-RPC content section [1, p. 17, 18]. The content section will specify the remote procedures to be called, and contain the responses with data, such as tree structures, success or failure status, and errors.

Data structures This protocol can use the data structures to contain generic tree structures, proposed in Code listing 5.3 in [1, p. 43, 44] (added as Code listing B.1 in Appendix B). The protocol can not contain any specific references to EMF, except as values in the generic data structures. The central properties in the tree strucure was *name*, *type*, *id* and *children*. The *type* would for Ecore be one of for example: *EClass*, *EPackage*, *EReference* and so on.

The action bar should be populated based on a "action schema" data structure (Code listing 5.4 and 5.5 in [1, p. 45], see Code listing B.2 and Code listing B.3 respectively in Appendix B), passed from the TLS to the extension in the protocol.

The hierarchy should be constrained by using a "hierarchy schema" (Code listing 5.6 in [1, p. 45], see Code listing B.4 in Appendix B). This would whitelist the allowed children for a node based on the *type* property. This would be passed from the TLS to the extension.

Chapter 3

Method

The method used will try to achieve the project objectives with correct results, and avoid or lower risks for project failure.

Pre-project The pre-project that came before this master's thesis, in [1], is regarded as a part of the methodology. It did the initial steps of problem identification, building, and prototyping a solution.

Software project Alongside this thesis, **a software project** will be created, which is developed by the author as part of the method. A substantial amount of time is dedicated to this project.

General failure criteria The project is a failure if the results are invalid, or cannot be realized into a real solution, or are so low quality that the project does not receive further development. The project is also a failure if it does not provide any value for its stakeholders.

Method overview The following sections describe the key elements to the method. There is an overarching approach, called Design Science Research. It has 6 phases, from problem identification, to development, to evaluation and communication. There is no methodology given by Design Science Research for executing the development phase. Therefore, a method for this phase must be crafted from experience and existing practice. The development phase consists of requirements engineering methods, and software development methods.

3.1 Design Science Research

Design Science Research in information systems is a methodology for creating new knowledge by designing, building and evaluating software artifacts. It may not be as widely known as "the scientific method" is, and is therefore explained in more detail.

Design *Design* in information systems is an iterative process and a resulting software artifact. A software artifact is to be built to solve problems for humans, and evaluated to prove it solves the problems [44, p. 2].

Research *Research* is an activity that adds new knowledge and understanding about something. Research should be systematical and use data to answer questions, solve problems and provide understanding [44, p. 2, 3].

Design Science Research *Design Science Research* is an approach to research where knowledge is created by design. It is defined by Alan Hevner and Samir Chatterjee [44, p. 5] as follows:

> *"Design science research is a research paradigm in which a designer answers questions relevant to human problems via the creation of innovative artifacts, thereby contributing new knowledge to the body of scientific evidence. The designed artifacts are both useful and fundamental in understanding that problem."*

The end goal of a Design Science Research project is to create information technology artifacts, that improve exiting solutions or solve a problem for the first time [44, p. 6]. A similar methodology may also be known under the name *Design and Creation*, as presented by Oates [45, p. 108]. According to Alan Hevner and Samir Chatterjee [44], the artifacts are generally classified as *constructs*, *models*, *methods*, *instantiations* or *better design theories*[1]. A very important aspect of Design Science Research is *evaluation* of the artifact. The evaluation is the process that uncovers new knowledge, and separates the process from routine design [44, p. 7]. There are many aspects that could be evaluated, but the aspects that *should* be evaluated are those that are related to the reason for creating the artifact in the first place; the aspects related to the research objectives [45, p. 115].

3.1.1 The General Design Cycle

Design Cycle Problem solving by design can follow a general design cycle. This is a circular and iterative process. The reasoning that occurs in a design cycle, and the knowledge generated during a cycle, is illustrated in Figure 3.1 [44, p. 26].

Awareness of Problem The process begins by becoming aware of a problem or opportunities, in the context of humans or an organization. A proposal for what could be solved is made explicit.

[1] This thesis aims to produce an instantiation: an implemented or prototype system. The thesis also seeks to advance on *better design theories*, with regards to software architecture and protocol design.

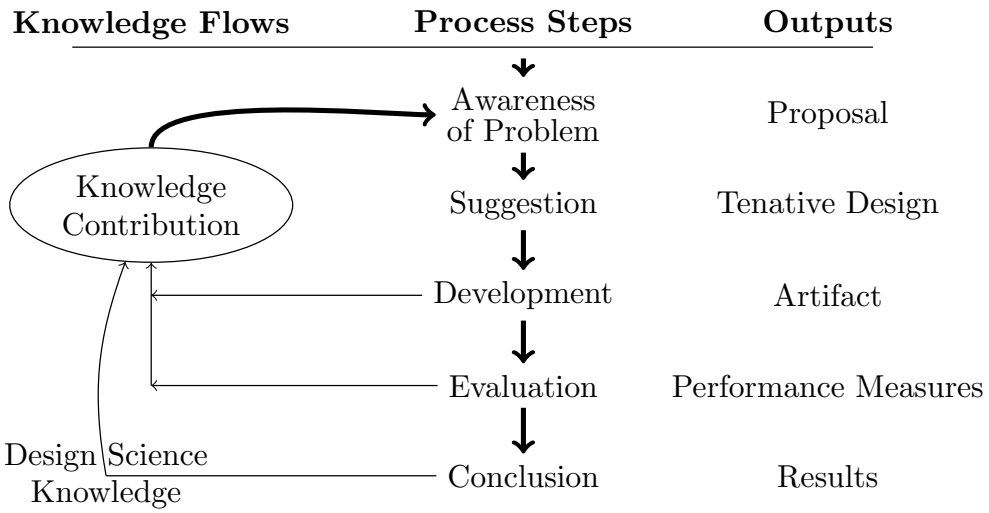


Figure 3.1: Design Science Research Process Model. The general process followed by Design Science Research. Design begins with awareness of a problem, and progresses through a suggestion for a solution, to development, evaluation and a conclusion. The stages produce different outputs, shown in the right column. After the conclusion, new knowledge is contributed. There is also knowledge produced by development and evaluation, nicknamed "circumscription". This knowledge is fed back into a new round of suggestion [46, p. 11-13]. (Adopted from Figure 3 in Vijay Vaishnavi *et al.* [46, p. 11])

Suggestion Then, a suggestion phase begins, where existing knowledge and theories are applied, as well as creativity, to create a tentative design that fits the proposal. This design could be flawed or incorrect, which is why it is important to realize the design, to detect issues.

Development The development phase will build a solution or prototype, aiming to fulfill the suggested design. This phase will uncover problems, inconsistencies, new learning about the problem, and other related knowledge. That knowledge is useful for creating a new and improved design. The quality of the *implementation* of the artifact does not need to be novel, as it is the *design* which is interesting [46, p. 12].

Evaluation After an artifact is created, the evaluation phase will measure the artifact. The measurements originate from the initial proposal, which holds the criteria for success. This phase may also discover new knowledge, which can be used later to create a new and improved design [46, p. 13].

Conclusion Finally, the conclusion phase will consolidate the results. The knowledge gained from the results will either be "firm" or "loose ends". Vijay Vaishnavi *et al.* [46, p. 13] describes this as the following:

> "*Not only are the results of the effort consolidated and "written up" at this phase, but the knowledge gained in the effort is frequently categorized as either "firm" — facts that have been learned and can be repeatedly applied or behavior that can be repeatedly invoked — or as "loose ends" — anomalous behavior that defies explanation and may well serve as the subject of further research.*"

3.1.2 Methodology

Based on the understanding of design science research, and the steps of a design science research process (Figure 3.1), a six step Design Science Research Methodology has been made by Alan Hevner and Samir Chatterjee [44, p. 28-30]. This methodology forms the skeleton of this thesis. The six steps of the methodology are the following:

1. *Problem identification and motivation.*
2. *Define the objectives for a solution.*
3. *Design and development.*
4. *Demonstration.*
5. *Evaluation.*
6. *Communication.*

1. Problem identification and motivation The specific research problem must be defined. The definition is used to develop the artifact which solves the problem.

The value of a solution to the problem should be justified as well. If the value of the solution is justified, it can motivate the researcher and the thesis' audience to pursue the solution and accept the results [44, p. 28, 29].

2. Define the objectives for a solution The objectives should be inferred from the problem definition, and the author's knowledge of what is possible and feasible. The objectives can be how much better a new solution should be (quantitative), or a description of how a new artifact would solve problems that are currently unsolved (qualitative) [44, p. 29].

3. Design and development Create an artifact to solve the problem and fulfill the objectives. The artifact can be one of the five classes listed in Section 3.1 (constructs, models etc.). The desired functionality and architecture is determined, and the actual artifact is created [44, p. 29].

4. Demonstration The artifact is demonstrated, to solve instances of the identified problem. This could be experiments, simulations, case studies etc. [44, p. 30].

5. Evaluation The artifact is observed and evaluated to measure how well it solves the identified problem. This can be done by comparing the results of the demonstration to the objectives of an ideal solution. There are many different ways to evaluate an artifact, and the correct approach should be decided based on the nature of the identified problem. After evaluation, the researcher can go back to step 3 to improve the design. If there is not enough time, resources or a need to do so, the process moves to step 6 instead [44, p. 30].

6. Communication The process must be communicated to other researchers and relevant audiences. This communication includes: the problem and its importance, the artifact and its utility, the rigor of the artifact design, and the effectiveness of the design [44, p. 30].[2]

3.2 Requirements Engineering

In Design Science Research, there is little guidance for how to execute the actual *design and development* phase. However, the software engineering field has many approaches and ideas for how to do this.

The design of the artifact starts by gathering requirements. These specify the concrete behaviors of the artifact; both the behaviors required for achieving the research objectives in Section 1.4, and those required for a highly usable and valuable solution.

[2]This thesis is a central part of this communication.

The identified requirements are both part of the design, and influence the design. Most requirements are formed from existing knowledge of the background theory. They are also discovered as the solution is developed. The following subsections 3.2.1-3.2.5 describe the key inputs for the software requirement engineering process.

3.2.1 Stakeholder Discussion

Discussion with stakeholders reveal many requirements, use cases and needs. The two key types of stakeholders here are EMF experts and TDT4250 students. The supervisor, Hallvard Trætteberg, fills the role as both a EMF expert and lecturer of TDT4250. The author, Kristian Rekstad, fills the role as a TDT4250 student.

Dialogue questions include "What features are required to model with EMF?" and "What features would you like to see in a new solution?", as well as "Which features are missing from the existing solutions?".

The same questions can be asked both before realizing a solution, and underway as the response to prototypes and current progress of an unfinished solution.

3.2.2 Requirements Extraction

Based on existing editors The existing tree editors in Eclipse IDE already implement every feature needed to do Model-Driven Development with the Eclipse Modeling Framework. Therefore, they are excellent sources of requirements. Especially the Sample Reflective Ecore Editor (Section 2.3.1) and EMF Forms Ecore Editor (Section 2.3.2).

No official requirements lists The pre-study failed to find any related research detailing requirements for a tree editor [1, p. 3]. No design documents or requirements specifications were found for the Ecore editors in Eclipse IDE either.

Use cases and requirement detection Therefore, the approach became to extract the requirements from the Ecore tree editors. The extraction is done by following use cases of modeling, as described in Section 2.2. When a new functional requirement is discovered through use, it is recorded in a list.

Shortcomings This approach will find many of the required and "obvious" requirements. However, hidden functionality and expert level functionality is not guaranteed to be found. The rationale is that this functionality is not needed (yet) anyways, as the goal is to fulfil the common use cases that students have when learning MDD.

There is also a risk that the user which is recording the requirements fail to detect functionality. Some functionality can be so obvious or "second nature" that the user is oblivious to it. Such functionality *should* become apparent later, however, when the solution is developed and tested. Any big omissions will prevent the use cases from succeeding.

3.2.3 Source Code Analysis of Similar Projects

Open source editors Because the tree editors for Ecore in Eclipse IDE are open source, it is possible to read and analyze the source code. Finding the main classes responsible for editor functionality, and analyzing their method names, initialization procedures and method calls, may expose requirements. This approach may also detect some of the more hidden functionalities, and the more "internal" functional requirements.

Architecture and software re-use Another advantage is that the internal architecture and patterns are exposed, which can be used to influence the artifact design. This may increase familiarity with the design for the Eclipse ecosystem, aiding the open source goals of this project. It also highlights the opportunities for software re-use, when familiar code, classes, interfaces, design patterns or software libraries are used.

Shortcomings Source code analysis is dependent on analyzing the correct source code files. If they are not found, this will fail. This also requires the source code to have some level of quality and readability to be useful for someone not already invested in that editor code base. The software architecture and design patterns used will matter too, in case functionality is hidden, dispersed or not clearly visible from the source code.

3.2.4 Use Cases and Prototyping

Creating realistic use cases based on Section 2.2, and executing them with early versions or prototypes will detect missing requirements. This is because a user will be blocked from progressing if a critical functionality is missing.

3.2.5 Agile Requirements

Agile Core values during this thesis' requirements engineering process come from *Agile*. Agile is a counterpart to the Waterfall process[3]

[3]In waterfall, software is designed, developed and tested in very separate stages. All the requirements are collected, before any design or development begins. An early mistake will not be discovered until the very end of the process. Changes to requirements require a restart of the project phases.

Change over Plan It embraces the fact that requirements change during the design and development, and thus favors **responding to change** over following a plan [47]. Requirements will change as they are discovered, refined and better understood later on.

Software over Documentation Another key value is that Agile prefers **working software** over comprehensive documentation [47]. This means that a small, working software artifact is more valuable than a large, complete and consistent list of software requirements and design, without any working software to show for it.

Impact of agile on requirements engineering The result is that the method here will start by collecting *some* requirements, by using the previously described inputs. When there are enough requirements to sufficiently solve the known use cases, design and development can begin. There is no goal to create a complete list at the start. The requirements are also changed, and new ones added, during the design and development.

3.3 Development Methodologies

The case for software development is the same as with requirements engineering: Design Science Research has little guidance. And again, the software engineering field has the answers.

The goal for the development methodology is to **create the right solution**, which solves the identified problem and fulfils the software requirements. The methodology also aims to **avoid or reduce risks** for project failure, by tackling it as early as possible. Research often deviates from routine design here, by going for the risks first instead of delaying or hiding them, as this may lead to new knowledge [45, p. 114].

Another goal for the development process is to create “good”, high quality software, so the project can be accepted by the open source developer ecosystem for further development and maintenance. Bad code or a bad design may result in a full rewrite by the next interested developer, or the developer may try to contribute but find it hard and give up.

Development methodology will not follow one strict practice, but rather piece together many different practices and values, which have lead to good results in the author’s past.

3.3.1 Agile

Development will follow agile values and principles, as described in *Manifesto for Agile Software Development* [47] and *Principles behind the Agile Manifesto* [48]. This means readjusting plans, rapid feedback from stakeholders, and software that works underway in development.

Agile development As there are many unknown factors in development, such as third party components and services to comply and integrate with, and unknown and hard to use APIs, the plans and designs may change. As with software requirements, the data structures, algorithms and design in the software solution will have to change as the developer learns the systems and problem space better. **Responding to change** will be valued more than following a plan here. Also, **working software** is more valuable than extensive documentation, meaning that code comments, tests, design specifications and diagrams will be given less effort than code, particularly if done up-front before the code. The alternative is that this documentation is made, but the code for it quickly proves itself impossible to make, or there is not enough time to implement it, leaving only useless documentation as the result. This also ties in to **simplicity and maximizing work-not-done**.

Regular reflection will be used weekly or bi-weekly, to assess if the process can be more effective. Sometimes tools and technologies may seem like a good fit for the development, but instead wastes more time than the developer productivity provided. Retrospective analysis of the development progress will try to detect this, and then expose if bad approaches are used. If so, these will be removed or replaced if possible.

Stakeholder involvement is important as well. The development will have a stakeholder as the developer (the author), which knows how the artifact will be used. Additionally, the supervisor will see a demo during development, to provide feedback and help prioritize the next steps.

3.3.2 Iterative Development

The software system will be developed iteratively. This means the components will be implemented up to a threshold of functionality, and executed to evaluate the behavior. The evaluation is not a formal and rigorous one, but rather informal and aims to quickly confirm if the software has the correct behavior. Then the components are developed some more, in a loop until the project ends.

The components are also developed incrementally. It also means a component will be worked on until it reaches *some* functionality, and then the next component will be worked on until it is on par. No component is developed to completion while the others are not started on.

3.3.3 Lean and Minimum Viable Product

Lean development is a set of principles inspired from Lean Manufacturing (for automobiles and such) [49].

Eliminate waste A core principle is to **eliminate waste**. This is also seen in Agile, as maximizing work-not-done. What Lean regards as waste is any work and output that does not have value. This means avoiding: unnecessary code and functionality (things deemed "could be nice to have"), unfinished work in progress (code and features not completed), defects and poor quality (do it well, do it once).

Build quality in Another principle is **building quality in**. Important business logic should have automated unit tests (however not all code needs tests[4]) Tedious and repetitive tasks should be automated, for example with scripts and tools.

Create knowledge A third principle is to **create knowledge**. Code will have comments where needed[5]. Documentation will explain the software at a high level.

Deliver fast The last principle applied is to **deliver fast**. While there are no students to use the artifact now, the focus is still on creating a functioning solution as early as possible. This is done by prioritizing the requirements in an order to get a *minimum viable product* (MVP). This is a solution that is just barely usable. The goal is to get it into the hands of users as quickly as possible, because this creates valuable feedback.

3.3.4 Tracer Bullets

Tracer bullets Using Tracer Bullets in code is a metaphor from *The Pragmatic Programmer* [50]. When using a machine gun, the operator does not precisely calculate where to shoot ahead of time. Instead, tracer bullets are occasionally loaded into the gun, which glow up and give visual feedback to the operator as to where the bullets travel.

Tracer code The same idea applied to coding means that uncertainty and risk is not dealt with by heavy upfront planning. A software solution likely has to interface with many unknown parts, adding risk for each one. The developer creates "tracer code", which has the goal of quickly connecting all the components and parts of the system, without adding lots of functionality. Stubs and empty code

[4] A project with high uncertainty and changing requirements may find tests a hinder as they have to up updated all the time. This results in extra work and rework.

[5] Not all comments are good comments. Code with side effects, strange design choices or hard to read implementation may need clarification by using comments.

can be used so the code compiles, as long as all the actual components in the final design are integrated and running. If any problems are discovered, then adjust and redesign as necessary [50].

The tracer code is real code, not a prototype. Therefore, it should be made with proper quality. It just lacks functionality. With a working skeleton system, more functionality can be added on top.

3.3.5 Domain-Driven Design

Domain-driven design is an approach to domain modeling in software. The goal is to create a domain model in software that uses the same language a domain expert uses, to create better, more evolvable and understandable code. If the domain has for example "trees" and "nodes", the code will also use trees and nodes as names, making stakeholder discussion straight forward. The code will accurately model the domain, increasing understanding and capturing the details of the business logic [10]. The model is represented as normal, executable code. There is no specific file representation, and no MDD framework involved. This is not a MDD method — it is a object oriented coding and naming method.
Another element from Domain-driven design is to use a layered software architecture. The domain model is isolated from the rest of the code. This makes it easy to see and reason about the behavior of the code, in terms of business logic [10, p. 69]. The alternative is that business logic is spread around the system, partially in the user interface components, persistence logic, and so on. The layers makes every aspect of the program more cohesive and makes interpretation of the designs easier [10, p. 69].

Practically, the domain model will be in its own layer, and the user interface will be in its own layer. The user interface will depend on the domain model. The domain model will be unaware of any user interface. An illustration is shown in Figure 3.2.

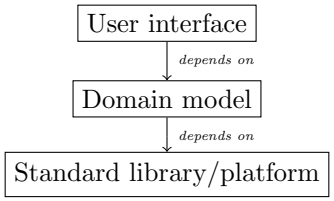


Figure 3.2: Layered architecture. The components above depends on the components below, but not vice versa.

3.3.6 Test-Driven Development

Test-driven development is about creating automated tests before writing the implementation [51, p. 105]. It will be used sparingly, for cases where the behavior is complex and important to get right. The developer will have to judge when it is needed, based on expected complexity, requirements and behavior for a unit of code. This may be especially relevant for some business logic in the domain model. Writing the tests first also creates a more testable design [51, p. 106].

The benefit of automated tests is confidence in the code against bugs. It also helps for when other developers join the project, as they can be confident about making changes without breaking existing code. A goal is to have other contributors develop this project further, and therefore avoiding "legacy code" is a good thing. The author of *Working Effectively with Legacy Code* writes:

> "Legacy code is somebody else's code. But in programmer-speak, the term means much more than that. [...]
>
> In the industry, *legacy code* is often used as a slang term for difficult-to-change code that we don't understand. [...]
>
> To me, *legacy code* is simply code without tests. [...]
>
> Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."
>
> — Feathers and Martin [52]

3.3.7 Prototyping

Prototyping will be used to create simple implementations when there is uncertainty of the design and big risks. A prototype will create learning by coding in the real environment, and then prototype code is discarded afterwards. A prototype can test feasibility and reveal good and bad sides of a design, quickly and cheaply.

The main bulk of prototyping has already been performed, as part of the pre-project in [1].

3.4 Evaluation

This section will describe how the evaluations for the artifacts were made. The evaluations try to test for value in the solution design, by comparing how well the artifacts solve the identified problem.

3.4.1 Software Artifact

The built artifact is evaluated according to the Design Science Research methodology in Section 3.1.2.

Assumptions The functionality of the original Eclipse IDE editors for Ecore is assumed to be correct and useful for students. The functionality is also required, in order to effectively use EMF for Model-Driven Development (MDD).

Demonstration goal Therefore, a demonstration should show the presence of the original functionality from Eclipse IDE in the new artifact. To do this, the artifact will be used to complete *use cases*, based on the modeling approach used in TDT4250 (see Section 2.2).

Additionally, a goal is to not use the Eclipse IDE, and a goal is to perform the use cases in a cloud based IDE, in this instance Gitpod.

Evaluation of demonstration The evaluation[6] will be a list of tests with modeling actions from Section 2.2. A test is successful if the tester (the author) can perform the action, and without using Eclipse IDE and also doing it in Gitpod.

3.4.2 Open Source Viability

Evaluation goal A goal of this thesis is that the artifact's source code is developed further, by either master students, the Eclipse ecosystem, or other contributors with interest in EMF or tree editors. The strategy to solve this is by making the source code open source.

Therefore, the source code will be evaluated to indicate how fit it is to be an open source project.

Test criteria To test how fit the project is, a checklist is synthesized from online guides for open source projects. The sources are highly reputable, such as the Eclipse Foundation, the GitHub community, and sites endorsed by these. The criteria will check for presence of elements or properties of the project, and succeed if it is present. A qualitative evaluation will proceed, to conclude the test results.

[6]The evaluation is classified as *ex ante* and *artificial*, for those familiar with the approach by Sonnenberg and vom Brocke [53] and [54].

Chapter 4

Results

This chapter will present the outcomes of the design and development phase. First, the artifact is presented from a user's point of view, showcasing functionality with screenshots. Then, the artifact's design will be presented from an architectural view, and the protocol will be presented last.

The results will also present the measures taken for making the open source project viable, and something a developer community may want to develop and maintain further.

Evaluation of the results is presented later, in Chapter 5.

An overview of the results can be previewed in the informal Figure 4.1.

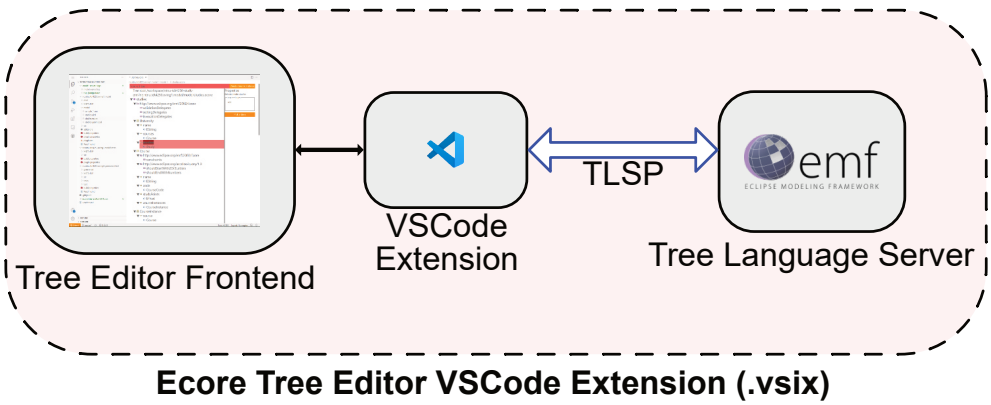


Figure 4.1: Informal diagram of the results, showing the main components.

4.1 Software Artifact: Tree Editor Extension for Ecore in Gitpod

This is of interest for a stakeholder, and someone aiming to do further research on this design.
The output of the development phase is an artifact which is a VSCode extension. The artifact is a `.vsix` file, and can be installed in a Gitpod workspace. The following results are from Gitpod using VSCode as the editor frontend, not Theia[1]. One way to install it[2], is to upload the `.vsix` file to the workspace, right clicking on it and selecting "Install Extension VSIX". When installed in the IDE, it is shown in the extensions panel as *Ecore Tree-editor*, shown in Figure 4.2.

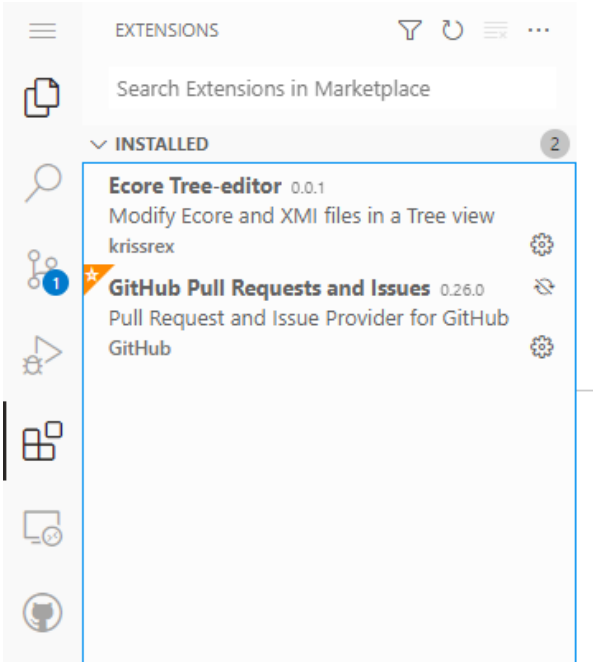


Figure 4.2: The extension is installed as *Ecore Tree-editor* in Gitpod with VSCode.

4.1.1 Custom Editor

This extension adds a new Custom Editor, which is automatically opened when the user opens a `.ecore`, `.genmodel` or `.xmi` file. The model file is loaded and transformed by the extension, and presented as a tree to the user.

[1] VSCode is the default for Gitpod, instead of Theia [19, 55].
[2] The "best" way is to publish the extension to OpenVSX, and search for it in the extensions panel.

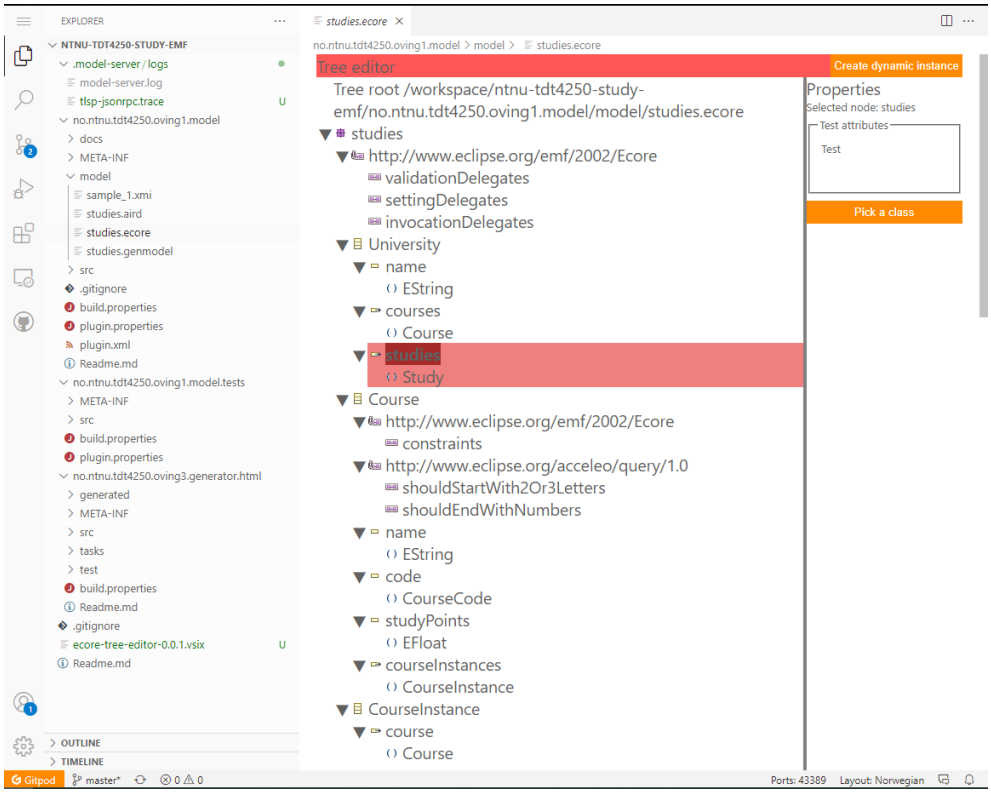


Figure 4.3: The Tree Editor Extension has opened a Custom Editor for the `studies.ecore` file.

Example model An Ecore model made in TDT4250 in 2019 has been used as an example to demonstrate the artifact. The `.ecore` file is opened in Figure 4.3. This figure shows three columns, from left to right: the default VSCode file explorer, the custom editor's master layout (tree structure), and the custom editor's detail layout (properties sheet).

Action bar There is also a red action bar at the top, with an orange action button to create a new dynamic instance. The action buttons shown will vary, depending on what the selected node is. The orange color of the button is coming from the color theme of the VSCode editor. With a dark theme, this action button could be blue, for example.

Master layout The master layout can show multiple roots. In this document, single root is shown for the "studies.ecore" file. The root node is a "studies" package, with children displayed below. This node has a label, "studies", and a specific icon indicating it is a package — the purple box with a cross. The icons used are the same ones used in Eclipse IDE for the Sample Reflective Ecore Editor (see Section 2.3.1), and depend on the type of node.

Clicking the black triangle next to a node will collapse it, hiding its children and rotating the triangle 90 degrees counter-clockwise.
Inside the master layout, a node is selected in dark red, with the label “studies”. Its child node “Study” is also highlighted, in a lighter red. (Note that the colors of selected nodes were arbitrarily chosen during development, and could be changed to give more contrast with the node’s label.) A node can be selected by clicking on it, and holding `ctrl` will add to the selection, allowing multiple nodes to be selected.
Dragging a node in the hierarchy and dropping it on a node, should change this node’s parent. Right clicking a node will open a context menu, with the possible children nodes to add. Dropping a node on an invalid parent will be prevented, by using a hierarchy schema, and indicated by changing the mouse cursor to a “forbidden” icon[3].

Detail layout The detail layout has a property sheet, currently showing a unfinished example form. This layout should use the *JSON-Forms* library to render properties, based on the node’s properties and a *UI schema* for that node type.

4.1.2 IDE Commands

The extension also provides Commands to VSCode. These are actions that can be invoked at any time. The student can invoke them from the Command Palette[4] by typing “Ecore” or another part of the command’s name. A screenshot is shown in Figure 4.4, with a command to create a new model file. This file will have the minimum XMI contents required for a blank model.

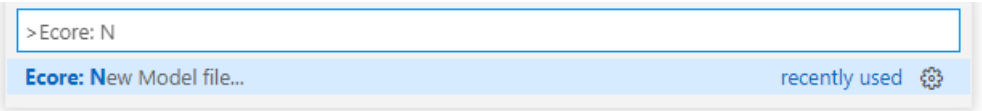


Figure 4.4: The Tree Editor Extension adds custom commands to the Command Palette. One of them is shown here, named *Ecore: New Model file...*.

4.1.3 Genmodel and Model Instance

The editor can open a `.genmodel` file or a `.xmi` model dynamic instance file as well. The genmodel is shown in Figure 4.5, and the dynamic instance in Figure 4.6. This will show two roots in the editor, as the original `.ecore` model is related to the opened file. One root is the genmodel (or model dynamic instance), and the other root is the study model.
The GenModel editor is not specialized, so it renders the tree as any other Ecore model.

[3]Note that drag-and-drop and node creation are not currently implemented, only accounted for by the design, by using a hierarchy schema.

[4]Press `F1`, or `ctrl + shift + P` (command on mac), or `Menu → View → Command palette`

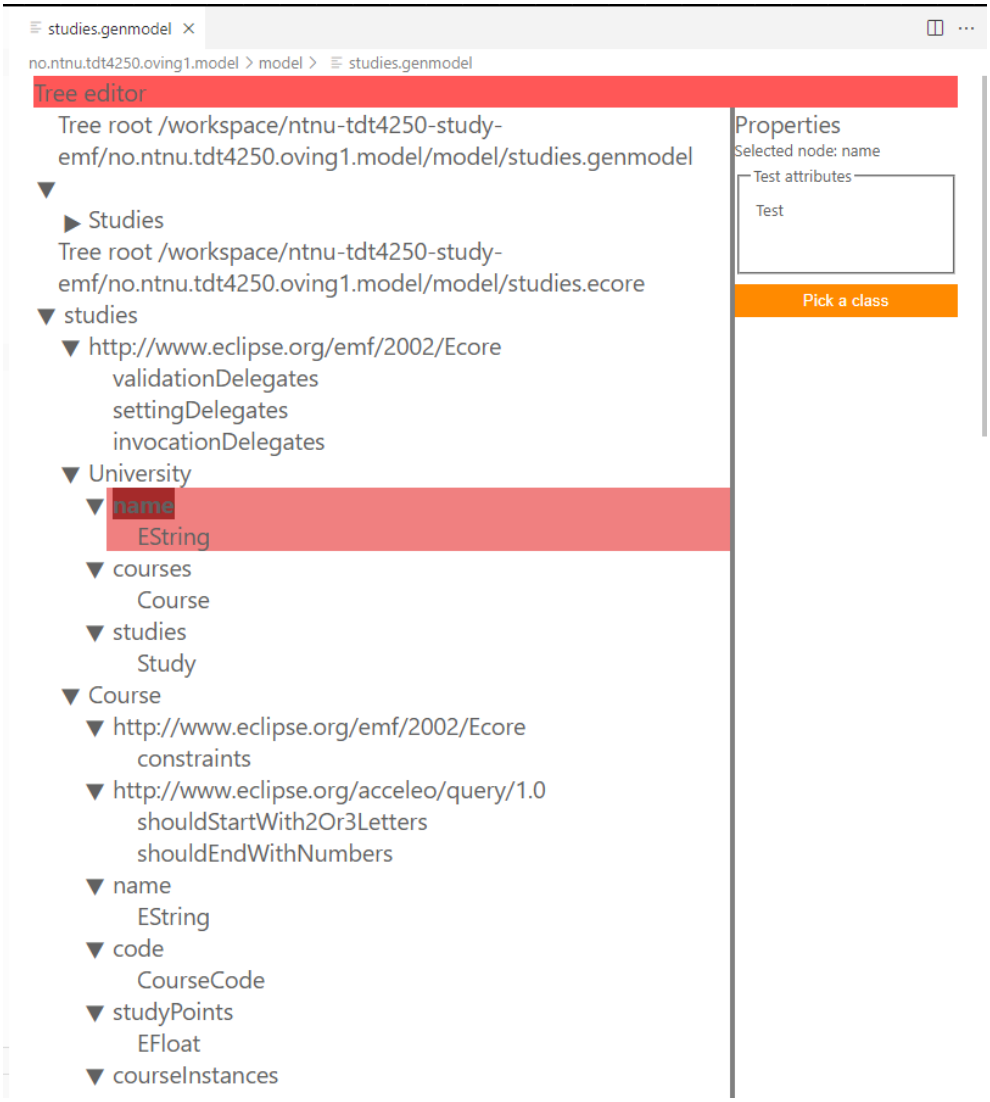


Figure 4.5: The Tree Editor Extension with the `studies.genmodel` file open. It has two roots, the GenModel and the model. The GenModel is collapsed/hidden at the "Studies" node.

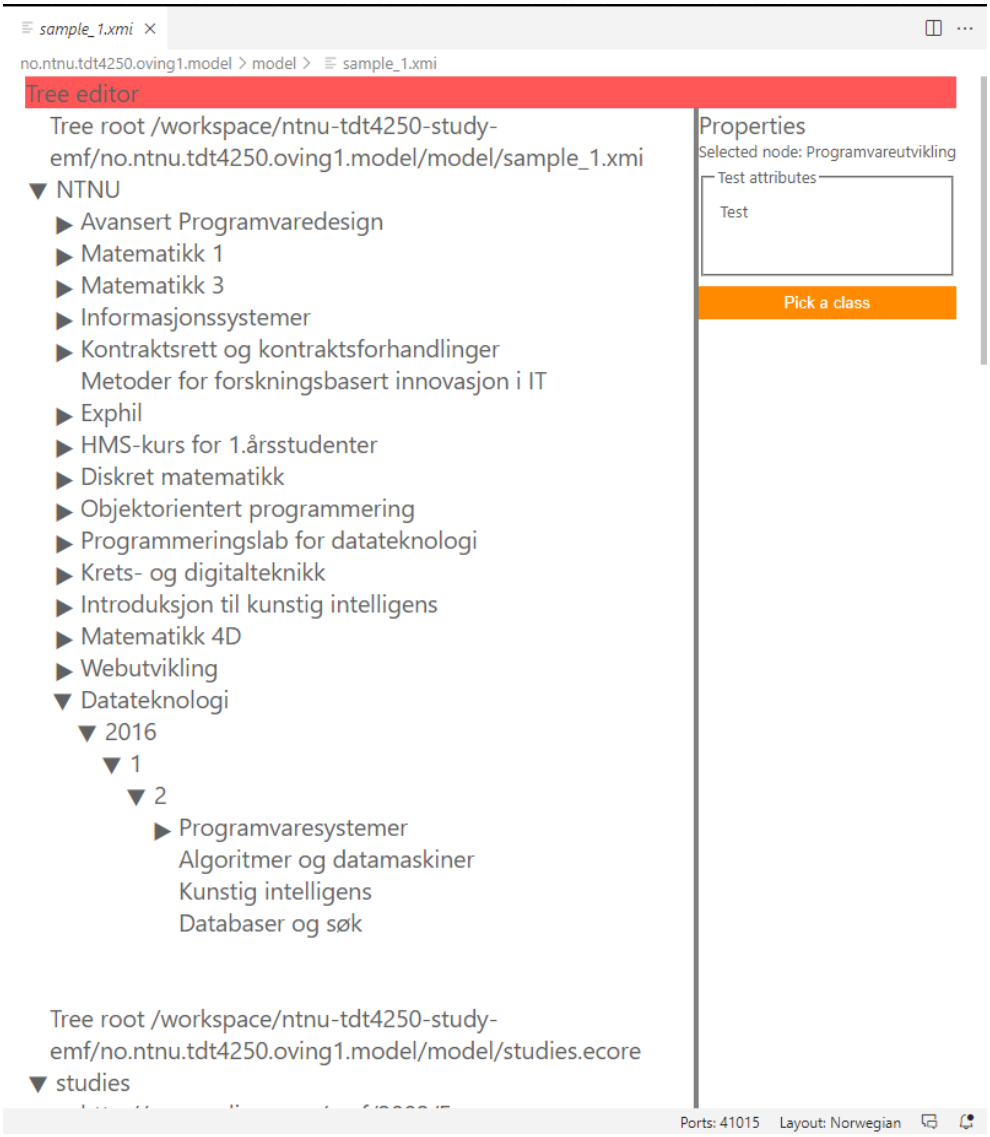


Figure 4.6: The Tree Editor Extension with the `sample_1.xmi` dynamic instance open. This is data that conforms to the model defined in `studies.ecore`.

4.1.4 Configuration and Logging

The extension has configuration options that a user can set. One such option is the logging level, a threshold to hide log messages in the log panel. The extension can also log internal events and messages to a Output panel in VSCode, for the user to debug and identify extension errors. This is mostly useful for extension developers, not students. But it works as an example of using configuration options. It is identified that using configurations will be needed. The configuration and output panel are shown in Figure 4.7.

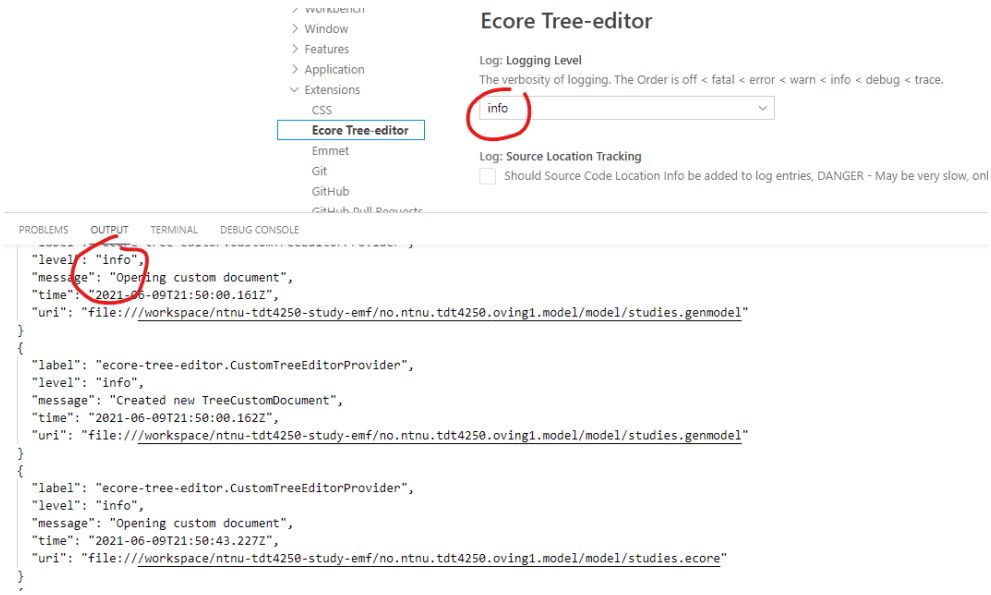


Figure 4.7: The Tree Editor Extension adds configuration options to the VSCode settings menu, shown in the top right. The extension also adds log outputs to a Output panel. The figure is annotated with two red circles. The upper circle is indicating the configuration option to filter the output based on log level. The lower circle is highlighting that same log level from a message in the output panel.

4.2 Design Artifact: Tree Document Model

A central design result is the constructs used to represent trees in an editor. These constructs are referred to as the **domain model** or simply **model** in this section (not to be confused with an model). This representation is what the frontend presents to a user, and what the extension and server is using to communicate over Tree Language Server Protocol (TLSP). Knowing this model is essential for communicating how the design works. This is because it is used as the "ubiquitous language" formed by Domain-driven design (see Section 3.3.5 and [10]). Note that **the domain model is generic for all trees and unaware of EMF**. This is because it aims to be a domain model for the TLSP, reuseable for other use cases than EMF editing. This section will explain where some of the names come from, and what these data structures look like.

4.2.1 Borrowed Terms

Trees When words like tree, node, root and children are used, they refer to the concepts for tree structures described in Section 2.4. A tree has exactly one root. The root can have nodes as children, and these can have children again. A node has a name and an icon that can represent it in a hierarchical tree structure.

Icon An icon is a visual representation — a picture, illustration, symbol — that represents some information about a node. An icon can show how one node differs from another, like what type it is, or it can show if a node is invalid or not.

DataUri This is a more technical term. The trees will be displayed on the web, with icons. A way to store icons as text is using the HTML data-uri scheme. It is just text, but has a semantic meaning. When set as the image source in a web browser, it will be displayed as a picture. A data-uri starts with a prefix which specifies the scheme, the content type and the encoding, for example: ``. Then, it is followed by the encoded image.

Document Because this is for an editor, the concept of documents are borrowed from VSCode. A document is the editor's representation of a file that the user wants to modify. When a editor window is open in an IDE, this editor shows a single document. The document can be opened, modified, saved, renamed and so on.

4.2.2 The Domain Model

The domain model is specified in this thesis by using TypeScript, but it can be translated to other languages, such as java.

A brief summary of the more exotic features of TypeScript may be helpful for the reader: Note that the '?' means optional or nullable. TypeScript can also "alias" types, meaning a new type can be defined by simply renaming an existing type. This can put more meaning into types like `string` and `number`, especially when they are reused in multiple places. Some builtin interfaces are used, like `Array` (a list) and `Record` (an object, or dictionary/map-like structure with keys and values).

The domain model will now be presented. It consists of the following elements: `TreeDocument`, `TreeRoot`, `TreeNode`, `NodeIcon`, `IconConfiguration`, `HierarchyConfiguration`, `Action`, `ActionEvent`, `ActionConfiguration` and `EditorState`. It also defines the following aliases for strings: `ActionId`, `IconDataUri`, `NodeId` and `NodeType`. These concepts are explained below.

TreeDocument The main data structure is the `TreeDocument`, in Code listing 4.1. This holds a list of `TreeRoots`. A document can have multiple roots, because there can be related trees. For example in EMF, the `.genmodel` file has a related `.ecore` file. Opening the GenModel would also show the Ecore model, in an editor with two roots.

Code listing 4.1: TreeDocument TypeScript code.

```
interface TreeDocument {
```

```
  roots: Array<TreeRoot>;
}
```

TreeRoot The `TreeRoot` in Code listing 4.2 holds references to the tree's root node. It also holds the configurations for how the nodes should be given icons, what actions a user can perform on the nodes, and what a valid node hierarchy looks like. The actions, hierarchy and icons use the `type` property of `TreeNode` to enforce this. The `TreeRoot` has an id as well, to separate it from other roots. This id must be unique inside the TreeDocument.

A `TreeRoot` does not require a root node, for example in the case the `TreeRoot` was just created or the node was deleted.

Code listing 4.2: TreeRoot TypeScript code.

```
interface TreeRoot {
  id: string;
  rootNode?: TreeNode;
  actions: ActionConfiguration;
  hierarchy: HierarchyConfiguration;
  icons?: IconConfiguration;
}
```

TreeNode For representing the nodes themselves, there is the `TreeNode` in Code listing 4.3. It has an id that is unique in the `TreeDocument`. The id is a string, but aliased to `NodeId` in TypeScript.

Next, the `TreeNode` has a `type`, which is very important. This type is a string, for example "EClass" or "EAttribute", and decides the icon, the allowed child nodes, and the possible actions a user can perform on this node. The string is aliased as `NodeType` in the model.

The `name` is what shows up in the hierarchical tree structure when presented to the user. It also reflects a property the user can edit. The name can for example be "MyClass", "Organization" or "NTNU".

To help inform the user what a node represents, the `documentation` property can hold a help string. The user interface could show this on hover, or when a node is selected in a designated help area.

The `TreeNode` holds instances of other `TreeNodes` in the `children`-property. This is what enables the tree structure to be represented.

Sometimes, a node can be special, for example invalid. To indicate this, the optional `iconOverride` can specify a new icon instead of the one from the `TreeRoot`'s

icon configuration.

The last property is the `EditorState`. This has the properties `selected` and `collapsed`, used to hold presentation information about the node. Being collapsed means that the children are not shown.

Code listing 4.3: TreeNode TypeScript code.

```
interface TreeNode {
  id: NodeId;
  type: NodeType;
  name?: string;
  documentation?: string;
  children: Array<TreeNode>;
  iconOverride?: IconDataUri | NodeIcon;
  editorState?: EditorState;
}
```

Action As mentioned, a user can perform actions. These can be validating an Ecore model, creating a new dynamic instance, and so on. This is represented by an `Action` in Code listing 4.4. The purpose of the `Action` is to show the user something they can perform, but only contain enough information so it can be sent back to the Tree Language Server to be performed there. Essentially, an `Action` is like a reference to a procedure on the server.

The action has an `id`, which is unique in the `TreeRoot`. This is what the server uses to know which procedure should be executed. The `name` and optional `icon` are for presenting the `Action` to the user.

Code listing 4.4: Action TypeScript code.

```
interface Action {
  id: ActionId;
  name: string;
  icon?: IconDataUri;
}
```

ActionConfiguration The list of all `Actions` live under the `TreeRoot`, inside the `ActionConfiguration`. The intention is that the user is presented with an action bar, or other list of actions, which can change depending on the selected `TreeNode`. This `ActionConfiguration` in Code listing 4.5 also specifies what actions are always shown in such an action bar, in the `defaultActionbarActions`.

The mapping of `ActionIds` to lists of `NodeTypes` in `nodeActions` is used when a node is selected. Each id can be examined to see if it supports the given `NodeType`. The type is supported if it is present in that `ActionId`'s list.

Code listing 4.5: ActionConfiguration TypeScript code.

```
interface ActionConfiguration {
```

```
  availableActions: Array<Action>;
  defaultActionbarActions?: Array<ActionId>;
  nodeActions?: Record<ActionId, Array<NodeType>>;
}
```

ActionEvent When the user triggers an action from the frontend, it is sent as an `ActionEvent` to the server. The `ActionEvent` is shown in Code listing 4.6. To reference which `Action` was triggered, the `ActionId` is set in the `action` property. The server may also want to know what `TreeRoot` the selected node was in, at the time of triggering the action. Because actions can operate on specific nodes, like creating a new dynamic instance in EMF, the currently selected `TreeNode`'s id are set as the `targetNodes`.

Code listing 4.6: ActionEvent TypeScript code.

```
interface ActionEvent {
  targetNodes?: Array<NodeId>;
  action: ActionId;
  targetRoot: TreeRoot;
}
```

NodeIcon and IconDataUri A `TreeNode` and an `Action` can have icons. There are also icons in the `TreeRoot`'s `IconConfiguration`. A single image is specified as `IconDataUri`, which is just an alias to a string type.
However, for more complex icons like the nodes', an editor may want to layer or alter the icons. It could be to add multiplicity information, validity state, or other variants of an icon. This is supported through composition with the `NodeIcon`. It defines a list of `IconDataUris`, which are drawn from bottom to top, stacked on each other.

HierarchyConfiguration The final element presented is the `HierarchyConfiguration` in Code listing 4.7. The `roots` specify what is allowed to be a `TreeRoot`'s `rootNode`. For example, "EPackage" could be such a `NodeType`.

The `allowedChildren` specifies a mapping between a parent's `NodeType` and its possible children's `NodeType`. This is designed with node creation an drag-and-drop in mind. A mapping could for example be "EClass" to "EAttribute", "EAnnotation" and "EReference"[5].

Code listing 4.7: HierarchyConfiguration TypeScript code.

```
interface HierarchyConfiguration {
  roots: Array<NodeType>;
  allowedChildren: Record<NodeType, Array<NodeType>>;
}
```

[5]This mapping is an example, and not complete.

4.3 Design Artifact: Architecture for Tree Language Server Systems

The software architecture may be of interest to developers of tree editors and similar kinds of VSCode extensions. There is potential to directly reuse components from this design as software dependencies/libraries in similar projects.

4.3.1 Architecturally Significant Requirements

The high level software architecture for the tree editor was shaped mainly by three Architecturally Significant Requirements (ASRs). The editor must be a VSCode extension, the tree viewer must use the VSCode Custom Editor API, and the extension must reuse EMF java code and the EMF.Cloud Model Server. This results in a system of three main components: a *Tree editor frontend*, a *Tree editor extension* and a *Tree Language Server*.
Another ASR is that the underlying document may change, and the rest of the system must respond and be updated. Therefore, a bi-directional communication between components is established, and an event driven architecture is used. The communication is isolated and standardized in a protocol, called the *Tree Language Server Protocol* (TLSP). This protocol is presented in detail in Section 4.4.

4.3.2 Changes from pre-project

From the pre-project, the high level architecture (see Figure 2.11) changed mainly by avoiding the EMF.Cloud Model Server as a separate running process, and is now embedded inside the Tree Language Server [1, p. 49].

On the code level, only the extension code and `TreeDocument` domain model are similar. The `TreeDocument` has changed to accommodate multiple roots, by introducing the *TreeRoot* and moving the *ArchitectureSchema* and *IconConfiguration* as children of these root, instead of the *TreeDocument*. This allows configurations to be different on a per-root basis, which is needed when for example opening both a GenModel and Ecore model in the same document.

The frontend is different from the pre-project prototype by using the *Vue.js* framework, and implementing actual features. The pre-project was only an example viewer, not communicating with the extension.

For the EMF.Cloud Model Server, this is now bundled inside the Tree Language Server, instead of a separate component. The pre-project used REST to communicate, while now it happens in java by using the classes of the EMF.Cloud Model Server, then relaying the answers over TLSP. The pre-project did not implement any real logic inside the server either.

4.3.3 System explanation

This section will explain the software architecture through a series of diagrams called the *C4 Model* (Context, Container, Component, Code) [56]. This is a top-down approach where one "zooms" in on the system components, so the wider context is clear.

4.3.3.1 Context

At a high level, the system has students interacting with Gitpod. The developed system runs inside Gitpod as an extension. This is illustrated in Figure 4.8. The student use Gitpod as a development environment, where they use the IDE, change files in a Workspace, and runs programs in a terminal. Gitpod uses git to retrieve the student's code from GitHub, and pushes changes back to it.
When a Student wants to install an extension to the IDE in Gitpod, it can either let the Student upload an extension file, or search the publicly available extensions in the OpenVSX extension registry. The instantiated artifact from this thesis could be uploaded to OpenVSX.

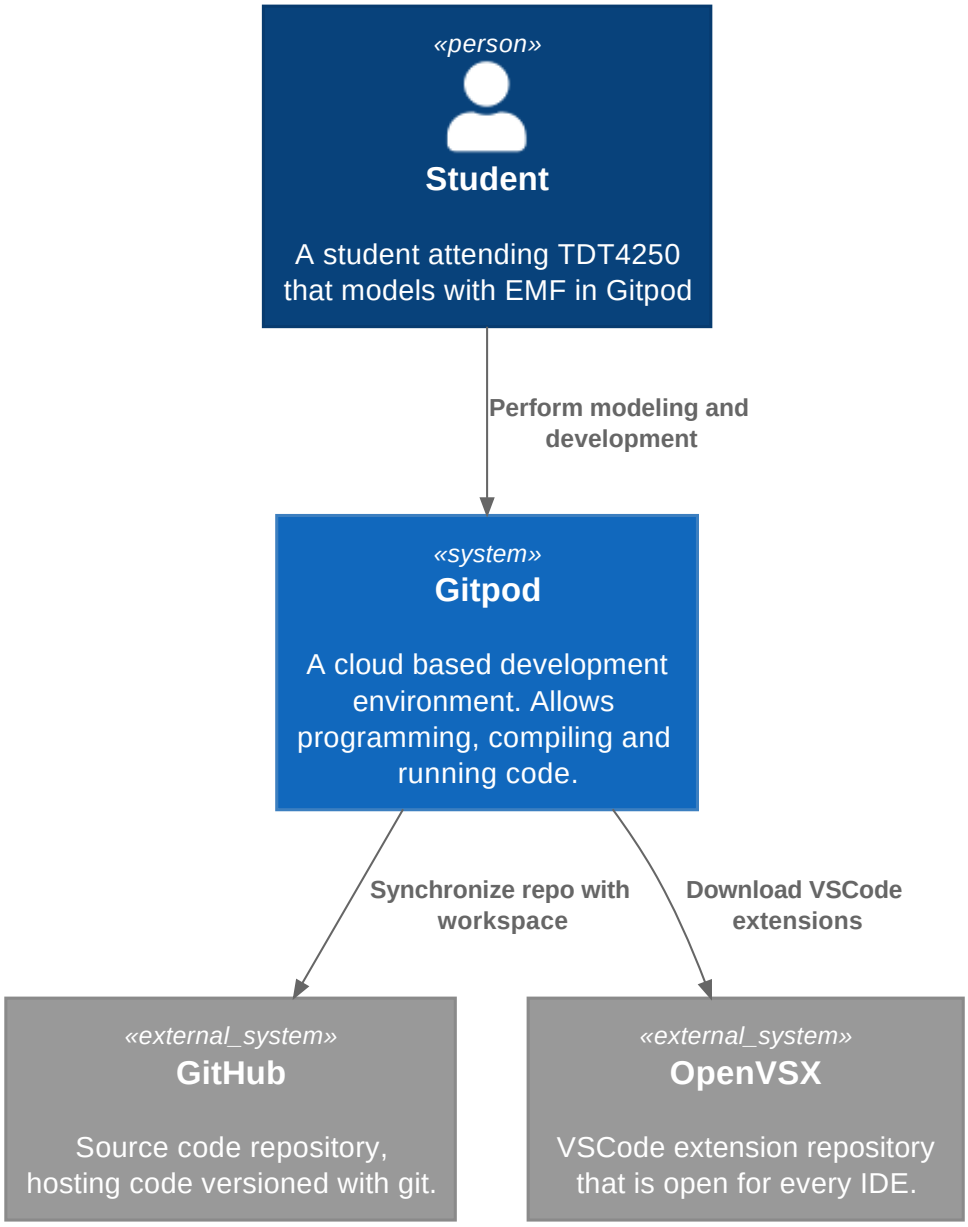


Figure 4.8: A system context diagram for Gitpod. The extension will run inside the Gitpod service, used by a student to do modeling and developing. Gitpod uses git to synchronize code with GitHub. The extensions in Gitpod are downloaded from a service called OpenVSX.

4.3.3.2 Containers

Inside the Gitpod system, there is a IDE, the *Ecore Tree Editor Extension* from this thesis, and the Workspace. This is shown in Figure 4.9. The IDE can be Theia or VSCode. This IDE is responsible for providing the user interface to the student. It also has the responsibility of installing and activating the extension. The extension runs in the environment provided by the Workspace. For example, the operating system and the available programs are provided by the Workspace, as well as the student's project files. If the extension wants to run a java program, the Workspace must have a Java Runtime installed.

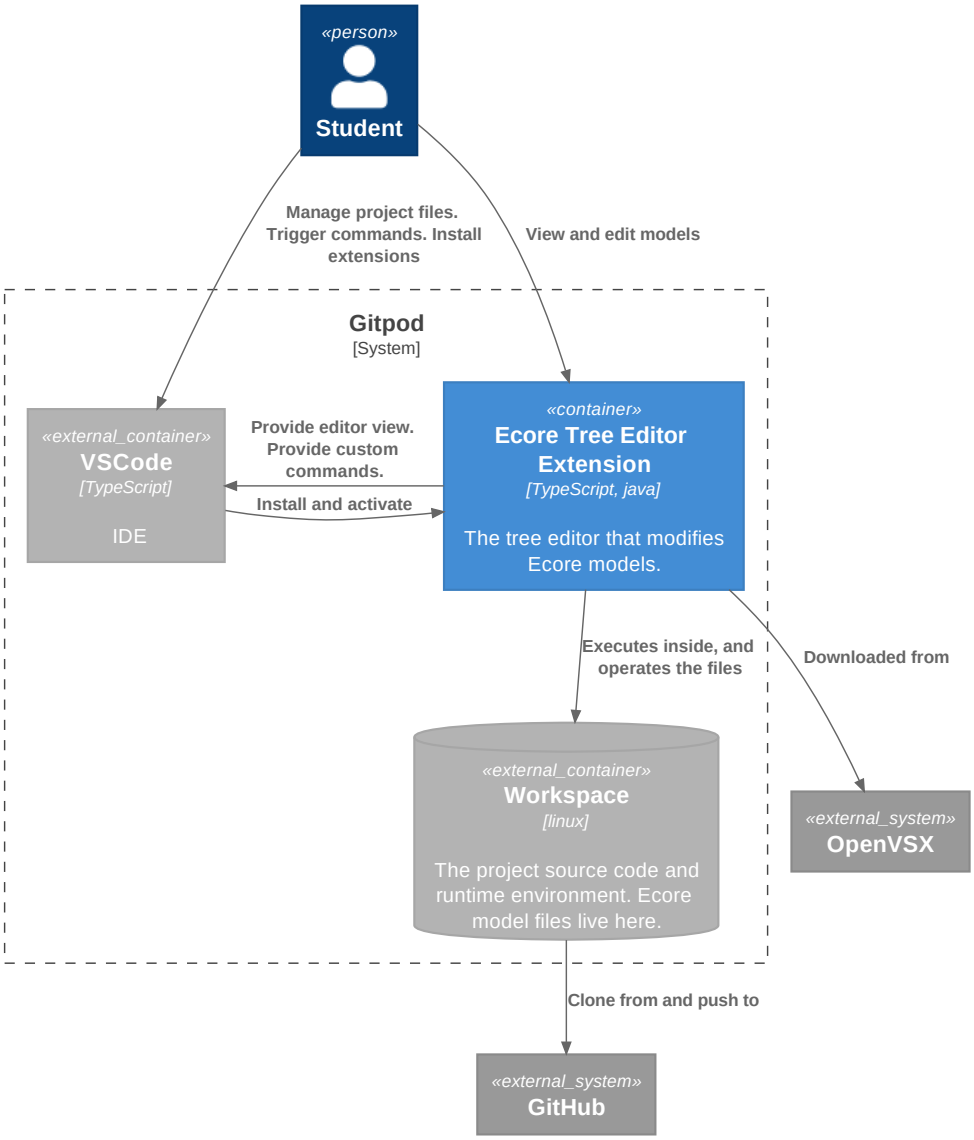


Figure 4.9: Container diagram for gitpod. The Gitpod system from Figure 4.8 is expanded to show its internal components. The IDE used by Gitpod is VSCode. The student will interact with VSCode, and install the Ecore Tree Editor Extension created from this thesis. This extension will also provide a user interface, which the student uses for modeling. This extension reads files from the Gitpod workspace, and uses the runtime provided by the workspace such as a Java Runtime Environment.

4.3.3.3 Components

The Ecore Tree Editor Extension itself consists of three main components. It is the *Tree editor extension* (or simply "extension"), which integrates with VSCode or Theia. Then there is the *Tree editor frontend* (or "frontend"), which provides a user interface with the hierarchical tree structure, labels and icons to the student. The last component is the *Tree Language Server* (TLS, or "server"), a java based server with knowledge about EMF and the EMF.Cloud Model Server. This is shown in a deployment diagram in Figure 4.10.

The Tree editor extension and Tree Language Server talk together using a protocol named Tree Language Server Protocol (TLSP). This protocol is another design artifact from this thesis, and is described later in Section 4.4. This protocol knows nothing about EMF, and the same with the Tree editor frontend. These two parts of the design only work on generic tree structures, as described in Section 2.4.

The Tree editor extension is the component responsible for providing both the Tree editor frontend, and the Tree Language Server. It also knows about EMF, because it registers the `ecore`, `genmodel` and `xmi` file extensions to VSCode and Theia. The custom Commands from the IDE's Command Palette are provided by this Tree editor extension as well.

Any changes to the student's model files are saved to disk by the Tree Language Server. The Tree editor frontend is close to stateless, and the Tree editor extension only bridges the Tree editor frontend and the TLS.

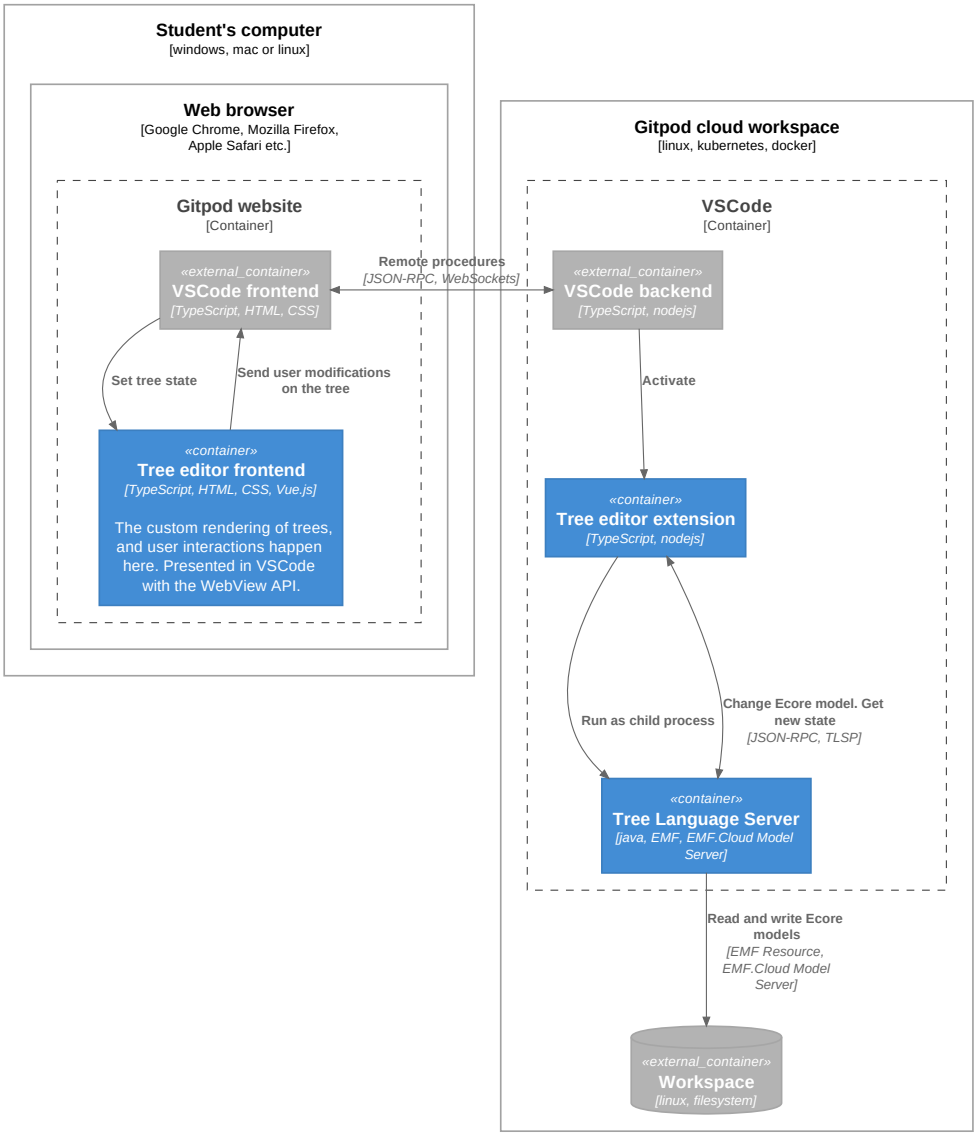


Figure 4.10: Deployment diagram of Gitpod. The student will use their computer to load the Gitpod website. The Gitpod service will start a computer in a cloud provider, to create a cloud workspace. The student only loads the VSCode frontend and Tree editor frontend into their browser. VSCode has a backend which runs inside the Workspace, and communicates to the frontend over WebSockets, using JSON-RPC. The VSCode backend will activate the Tree editor extension, which in turn will start a Tree Language Server. This Tree Language Server runs java, and reuses the EMF tooling. The Tree editor extension communicates to the Tree Language Server over a well defined protocol, where it asks to read model files, and execute commands to change the models. The Tree Language Server uses the Workspace to read and write `.ecore` files.

4.3.3.4 Code

Modules At a code module level, the Ecore Tree Editor extension is made of 5 modules. The extension, frontend, server, and two shared libraries: *Tree Document model* and *VSCode and Webview RPC* (or simply RPC library). The frontend and extension both use the Tree Document model and the RPC library. This is shown in Figure 4.11. All the modules are coded with TypeScript, except the server, which uses Java.

The server also "uses" the Tree Document model, but by re-implementing[6] it in java. The Tree Document model module is the result of using Domain Driven Design and a layered architecture (by Evans [10]). It encapsulates the concepts and business logic related to editing tree structures.

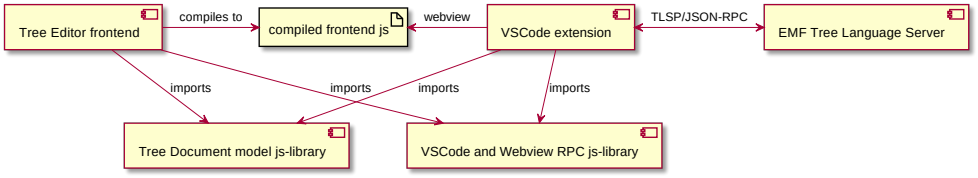


Figure 4.11: Component diagram of the Ecore Tree Editor. The for the the extension is organized in 5 separate modules. The main module is the VSCode extension. This extension bundles the compiled frontend javascript artifact, and the compiled EMF Tree Language Server java jar-file. The Tree DOcument model js-library is the layer with the domain model for tree editors. It is used in both the frontend and the extension.

Classes Three UML class diagrams are presented, one each for the frontend, extension and server. These are not complete, meaning some classes, properties, methods and relationships are not shown. This is intentional, to increase the clarity, comprehension and the essence of the diagrams.

Frontend classes A diagram of the frontend is shown in Figure 4.12. The execution environment for this is a web browser frame, meaning it has access to the HTML DOM[7]. It has a view layer using a framework called *Vue.js*. The frontend's state (`TreeDocument`) is held in a state storage called "Store", using a library called *Vuex*. This state can only be changed through explicit mutations and actions. This is so the store can intercept changes, and send them to the server via the extension, over the Tree Language Server Protocol. The frontend talks to the extension using a `VSCode` interface, where the actual VSCode IDE injects an implementation. A mocked version (`MockVSCode`) is provided as an implementation when testing and developing the frontend outside of the VSCode IDE. Two

[6]Not ideal, but no good transpiling (programming language translating) software was found in a reasonable amount of time, to automate this.

[7]Document Object Model, which is how a web browser represents web pages.

classes help the communication between the extension and frontend: `TreeEditorWebview` and `VscodeExtension`. These utilize a JSON-RPC-like protocol, over the `VSCode` method called `postMessage` and the javascript `Window`'s `addEventListener`. The `FormEditor` view is intended to use the JSON-Forms library, but this view is not finished.

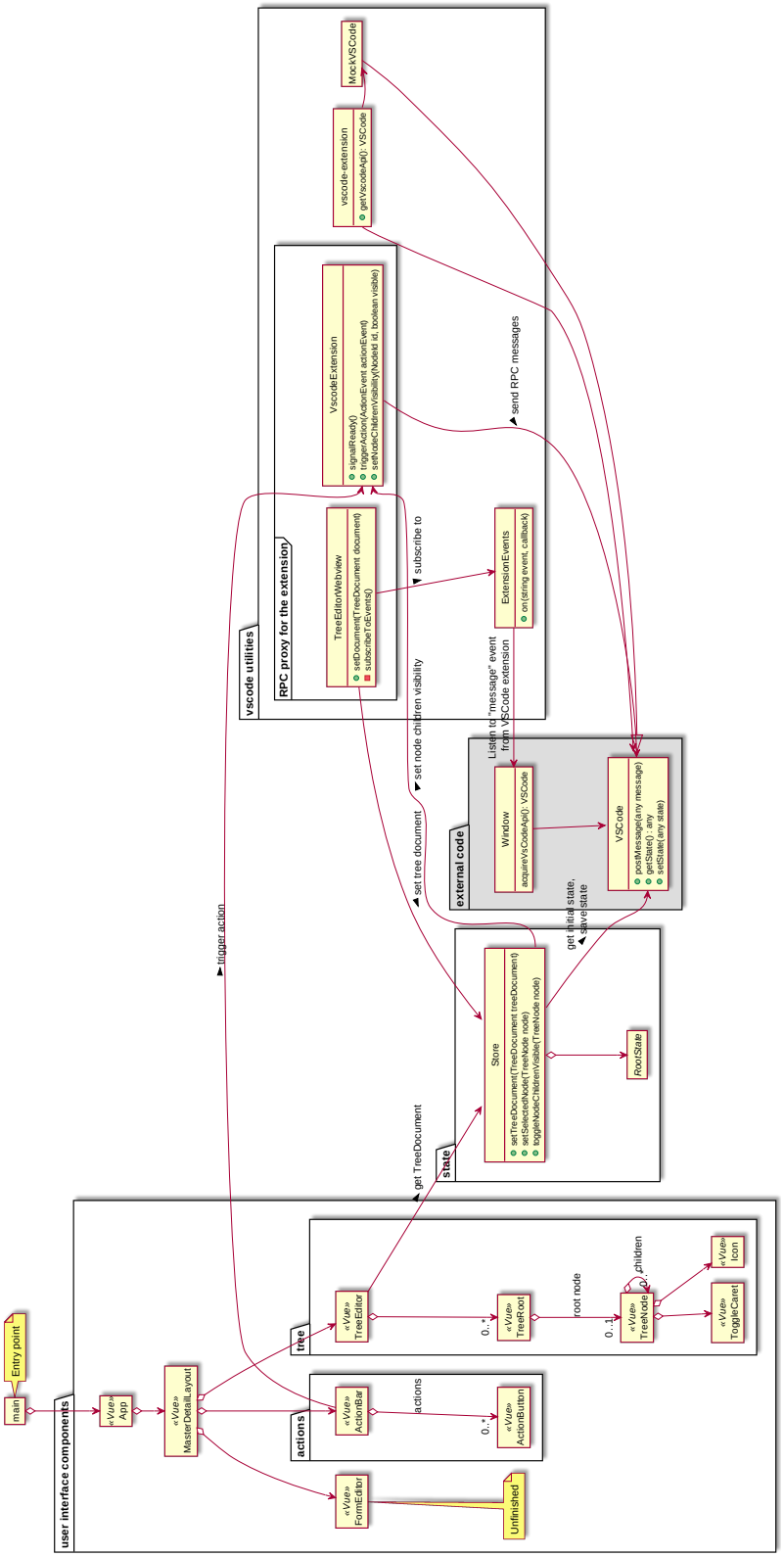


Figure 4.12: Class diagram of the Tree Editor Frontend component.

Extension classes A diagram of the VSCode extension is shown in Figure 4.13. The execution environment for this is NodeJS. The `extension` file is activated by VSCode when particular triggers specified in the extension's `package.json` manifest occur. One such trigger is opening a `.ecore` file. The `extension` then registers commands and the custom editor. The `CustomTreeEditorProvider` is asked by VSCode to create a document and editor for the `.ecore` file. The editor uses the compiled outputs of the frontend, and puts it inside a `WebView`. A `WebView` is an isolated execution context provided by VSCode (analogous to an `IFrame` in HTML), where a custom user interface can be shown. An extension is otherwise not allowed to modify the user interface in VSCode.

The extension also starts a java process with the executable `.jar` file for the server. It then attaches to the *standard in* and *standard out* as communication channels for the TLSP. The communication and protocol parsing uses the `vscode-jsonrpc` library from Microsoft, also used in the official LSP implementation for VSCode.

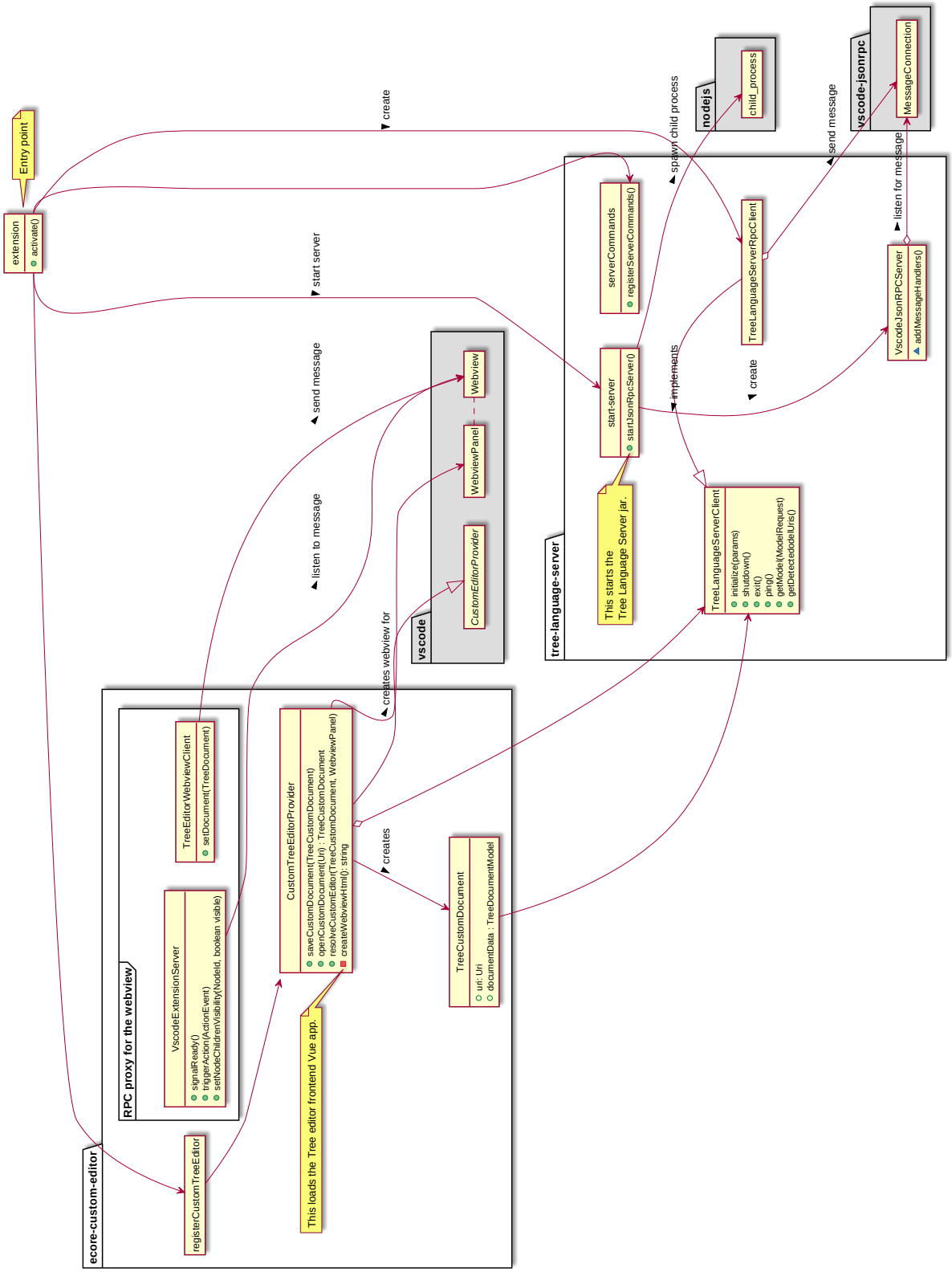


Figure 4.13: Class diagram of the Tree Editor Extension component.

Server classes A diagram of the server is shown in Figure 4.14. The TLSP server for EMF starts a JSON-RPC server listening to *standard in* and *standard out*, to communicate with the extension. The protocol is defined with two annotated java interfaces: `Server` and `Client`. The `Server` represent this server itself, while the `Client` is the VSCode extension side. An implementation of the `Server` interface is central, as it does the actual logic in the Tree Language Server Protocol. The `ServerImpl` and `Client` are handed to a `Launcher`, which comes from the *LSP4J* project. This is an Eclipse Foundation project which provides a Java LSP. Here, the JSON-RPC component is standalone, and reused here, with the TLSP as protocol instead of LSP.

The `ServerImpl` delegates most of the work to an `EmfTreeModelController`, which in turn delegates to the EMF.Cloud Model Server or a `EcoreToTreeDocumentMapper`. The latter uses the EMF runtime API and `ReflectiveItemProvider` from the `.edit` EMF package. The `EcoreToTreeDocumentMapper` maps a EMF `Resource` to a `TreeDocument` data structure, compatible with the one in the Tree Document model library component for javascript.

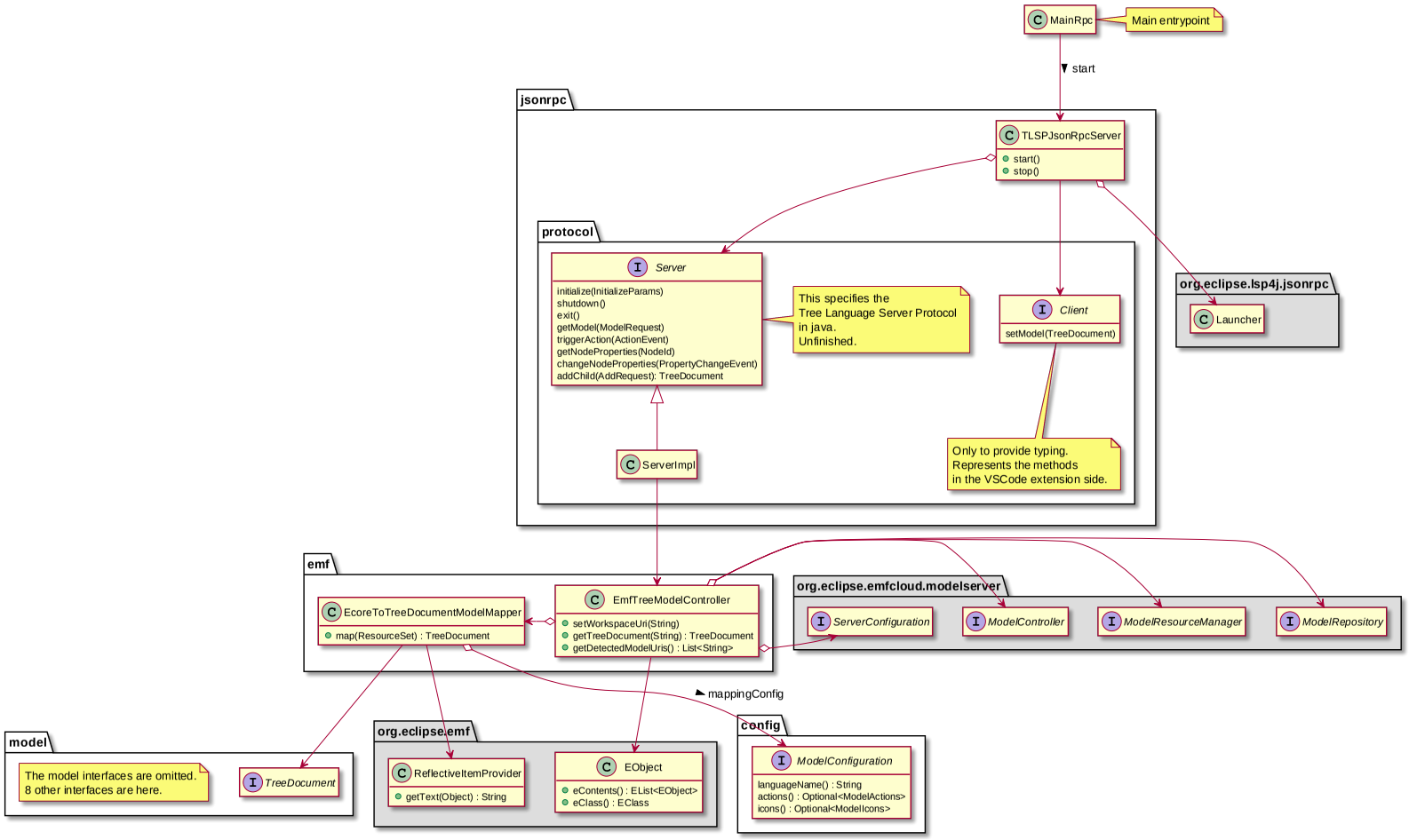


Figure 4.14: Class diagram of the Tree Language Server component.

4.4 Design Artifact: Tree Language Server Protocol

All communication between the Tree Language Server (TLS) and the VSCode extension happens over a protocol. This protocol is part of the artifact design from this thesis, and is called the Tree Language Server Protocol.

Because the original EMF editors are being moved from Eclipse IDE to VSCode, the protocol draws inspiration from Language Server Protocol. If Eclipse IDE already used a LSP-like language server, the migration would be much easier. And since it moved once to VSCode, it may move again later, for example to IntelliJ (or some other IDE).

The TLSP protocol builds on top the the *Base Protocol* described in Section 2.9.1. That means it sends a header section followed by a content section. The content has JSON-RPC data, being requests, responses, errors, and notifications. As a reminder: a request must be responded to with a response or error, while a notification does not get an answer. This means it is a bidirectional communication, where both the extension and the server can initiate a request or notification. The TLSP describes what data structures, method names, parameter values and return values should be present in the JSON-RPC content. Because the protocol uses JSON-RPC to call the remote procedures, all the data must be serializable to JSON.

The following subsections present the orders of procedure calls in the TLSP. The protocol allows for a stateful server, so for example the workspace must be set before a model is loaded. The diagrams use UML sequence diagrams. These have components listed inside boxes at the top, and the timelines as lines coming out below the boxes. The diagrams are read top to bottom. A timeline with a box on it represents a process lifetime inside that component.

4.4.1 Activation

Extension activation and document opening is shown in Figure 4.15. When the extension is activated by VSCode, the server should be started. When the server is ready to listen for TLSP messages, this is indicated by writing a message to an output channel not used for TLSPs[8].

The extension requests *initialize* with any options the server would need. The server responds when it is done, allowing the extension to know when it can send the next command. The workspace path is set to the folder with a student's code.

When a `CustomTreeEditorProvider` in the extension has been asked to open a `.ecore` document, the server is requested to get the model for this document. The

[8]The server uses *standard error* to log and communicate anything that is not TLSP.

server responds, and the tree document model is set on the frontend.

If the extension is asked to stop, it will first send a *shutdown* request to the server, allowing it to respond when ready to stop. Then an *exit* notification is sent, which stops the server and breaks the communication. This shutdown followed by exit procedure is directly inspired by LSP.

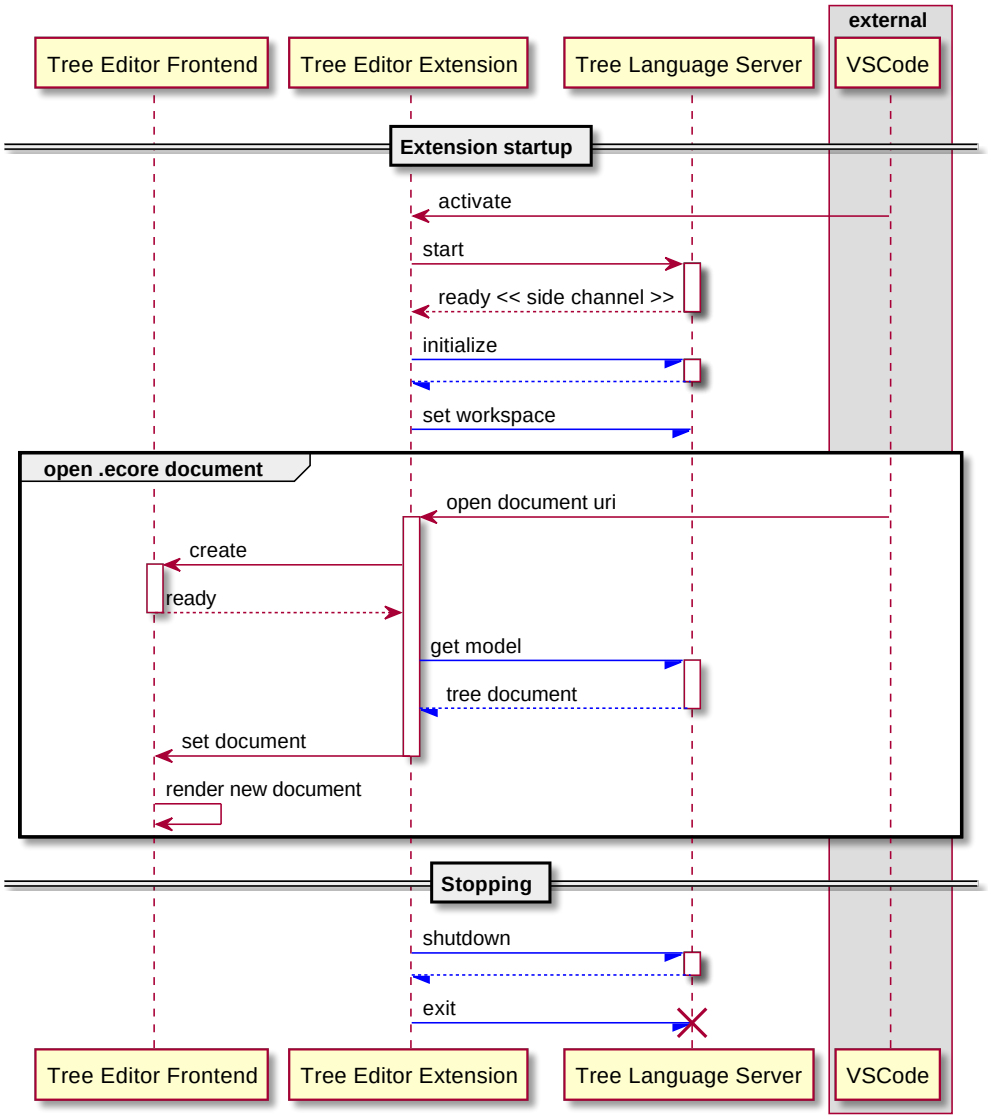


Figure 4.15: Sequence diagram for the protocol when starting and stopping the server. The blue half-arrow (⇀) is part of the Tree Language Server Protocol (TLSP).

4.4.2 User Actions

When a user triggers an action from the frontend, such as validation or code generation, an `ActionEvent` is sent to the server. If this event changes the model, a notification will be sent by the server to update the document state. This is shown in Figure 4.16.

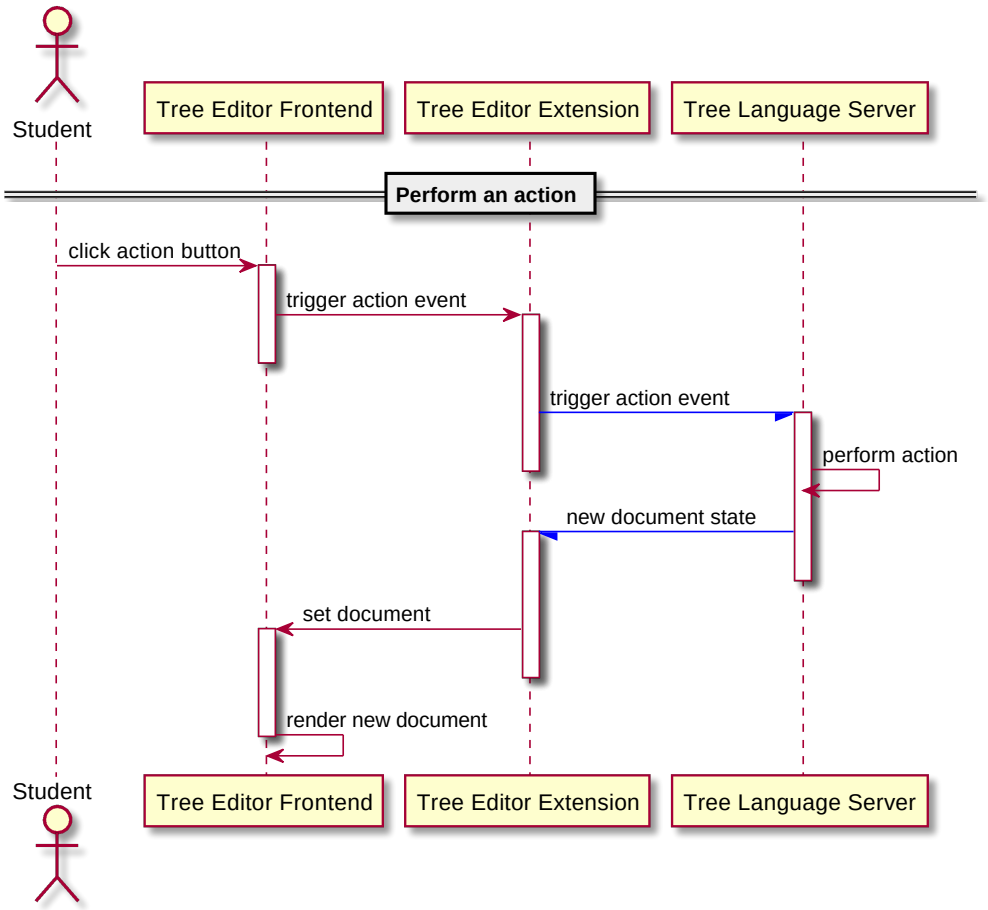


Figure 4.16: Sequence diagram for the protocol when triggering an action. The blue half-arrow (⇀) is part of the Tree Language Server Protocol (TLSP).

4.4.3 Property Editing

When a student wants to modify a model element, they first have to select the corresponding tree node. When the selection changes, the frontend asks the extension for the properties of that node. This request is then sent to the server, where it returns both the node properties and the schema for JSON-Forms to present it. This is shown in Figure 4.17.

Then when the properties of the node are changed, an event is sent to the server indicating the id of the node and the new property values. This is shown in the lower half of Figure 4.17.

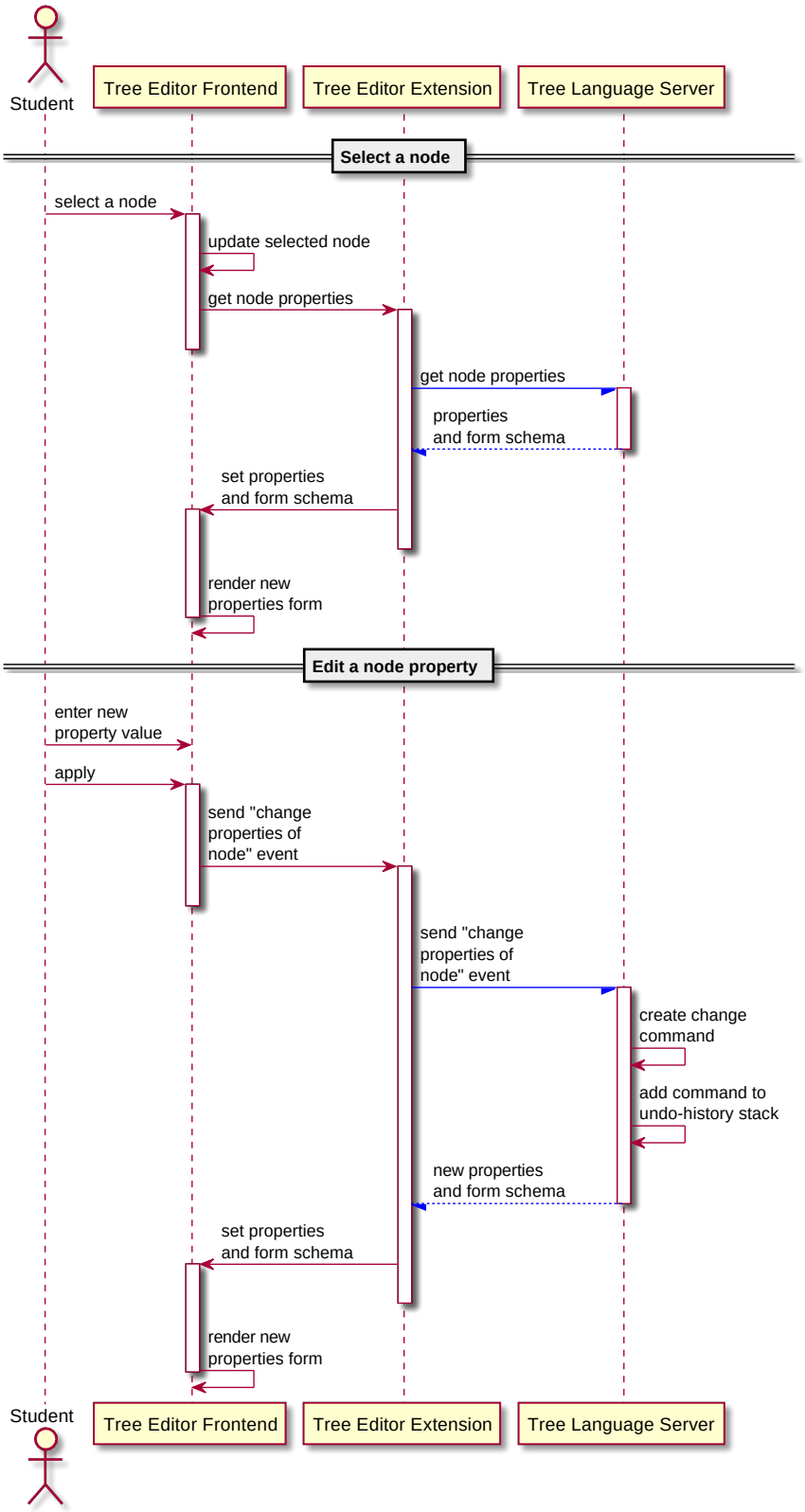


Figure 4.17: Sequence diagram for the protocol when editing a node property. The blue half-arrow (⇀) is part of the Tree Language Server Protocol (TLSP).

4.4.4 Tree Editing

Editing the tree hierarchy by adding children is done by first selecting the child node's type. This can be presented using the `HierarchySchema`, which is already sent on the `TreeDocument` when the model was loaded. When a student selects the node to add a child on, and the type of child node, this is sent via the extension towards the server. The server should create a `Command` from the `.edit` framework, in order to have a undo/redo history. The new document state is returned afterwards. This is shown in Figure 4.18.
Other structural edits, like deleting nodes and moving children to a new parent have a similar sequence, although not shown for brevity.

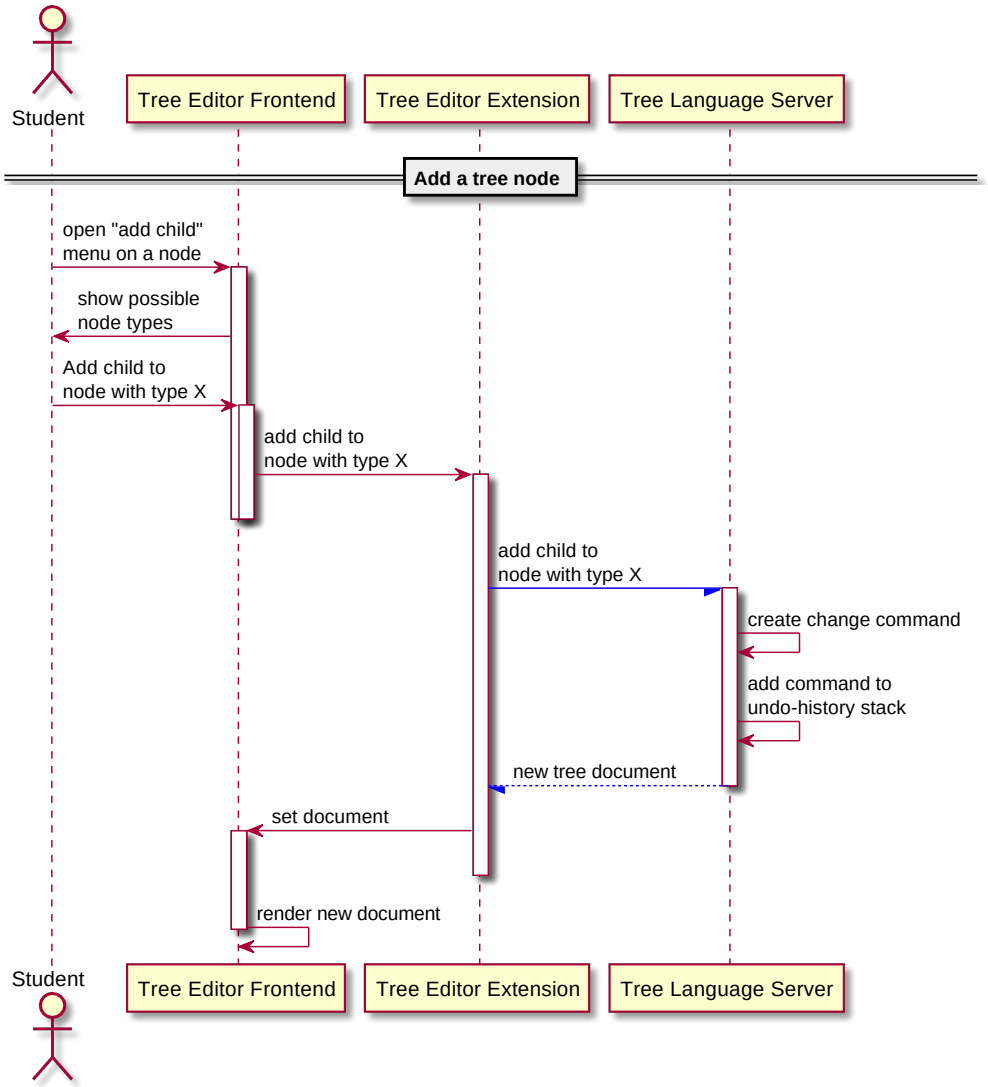


Figure 4.18: Sequence diagram for the protocol when adding a child node. The blue half-arrow (⇀) is part of the Tree Language Server Protocol (TLSP).

4.5 Open Source Project: Measures Taken for Viability and Maintainability

This section will describe the measures taken in order to make the project[9] viable and maintainable as an open source project.

4.5.1 Code Availability

Possibly the most important part of open source, is available source code. The project is hosted[10] on a public website for collaboration on open source software: GitHub.
Also important, is the project visibility being *public*, not private.
The project has the supervisor added as a contributor, in case one project maintainer is unavailable.

4.5.2 Documentation

Readme The main project has a "Readme" file with an overview of the project's components.
The components named "tree-document-model-js", "tree-editor-frontend", "vscode-ecore-tree-editor-extension" and "vscode-webview-tree-editor-rpc" have a Readme.
The "model-server" component does not have a Readme.

All the readme files are either very minimal, or the default Readme from a project generator.

Source code All the modules contain some comments inside the source code. Not all the source code is documented, only where the author deemed it necessary. A code base search[11] returned that 58 files of 169 files had comments, with a total of 128 comments.

4.5.3 Automation

Package manager A package manager is used for installing dependencies and compiling each module individually. For the TypeScript modules, `npm` (Node Package Manager) is used, and dependencies are tracked in a `package.json`. For the java module, `mvn` (Apache Maven) is used, and dependencies are tracked in a `pom.xml`.

[9] The results report on the project in version `59b722c117`, available at `https://github.com/krissrex/tdt4900-master-thesis-ecore-tree-editor/tree/59b722c117346dcc53da16275819e0d5952f0d05`.

[10] Project source: `https://github.com/krissrex/tdt4900-master-thesis-ecore-tree-editor`.

[11] `ag -stats -c -ignore-dir dist '\Q/**\E\s' .`

Build Build scripts using `bash` are provided, that compile the modules (using npm or mvn) and copy the outputs to the correct path. They also build in the correct order, regarding inter-module dependencies.

IDE configuration Files are added to automatically configure a contributor's IDE, if they use VSCode for the TypeScript modules and IntelliJ for the java module. When using VSCode, a list of recommended extensions is provided as well, which can be automatically installed. There are *Tasks* defined for VSCode that can trigger the different npm builds, and *Run configurations* to start the modules.

CI/CD There is no Continuous Integration (CI) and Continuous Deployment (CD) configured. This can be added later when needed; for 1 developer it is overhead.

4.5.4 Licensing

Module license The modules use the MIT license[12]. It is a very simple and permissive license, compatible with open source, and commonly used. The licenses are not in separate files or the readme. They are instead mentioned in the `package.json` and `pom.xml` files.

Copied code Some code is copied from other sources. The original license has been included in these cases. No code is copied from incompatible or strict licenses that contradict MIT.

Third party dependencies No proprietary dependencies are used[13] , and none with incompatible or intrusive licenses.

4.5.5 Code

Code style The code uses readable names and small files. The programming languages (TypeScript and Java) are common, especially in this context. The code is formatted with automatic code formatters[14], ensuring a consistent style.

Dependencies The dependencies and libraries used are common and in some cases official, in this context. Effort has been put into using the same dependencies as related works (such as LSP and EMF.Cloud projects).

[12]https://opensource.org/licenses/MIT
[13]TypeScript modules were scanned with: `npx license-checker --production`.
[14]Prettier and IntelliJ format the code.

4.5.6 Issue Tracking

An issue tracker is available on GitHub. A user is required, but signup is free. A discussion forum is available as well on GitHub. There is no Wiki, but it is easy to create one on GitHub if demand arises.

Chapter 5

Evaluation

This chapter uses the Design Science Research methodology's evaluations on the previous chapter's results. The evaluations investigate how much of the needed functionality is actually supported. They also present some prescriptive design theories, as design science evaluations are expected to produce knowledge.

The first two evaluations look at the tree editor itself, compared to the goal of modeling in the cloud and Objective 1. The third evaluation investigates the software architecture with regards to Objective 3. The last evaluation examines the project from an open source contributor's point of view, using a checklist with best practices. This stems from Objective 2.

5.1 Use Case Completeness Evaluation of Tree Editor Extension

Introduction The design was evaluated for completeness based on a list of modeling actions. The modeling actions stem from the modeling process used in TDT4250, in Section 2.2.
This evaluation was performed by the author, not in a class of TDT4250 students. Evaluation was based on an artifact at the proof-of-concept stage.

Setup To perform the evaluation, a Gitpod workspace was created with VSCode as the editor of choice. A Gitpod user can choose between Theia and VSCode in their preferences before creating the workspace. VSCode was used because it is the default, and an issue in Theia gave error messages when uploading extensions. The message indicated a problem with the current version of theia, instead of the extension.

A Gitpod Workspace using `https://github.com/krissrex/ntnu-tdt4250-study-emf` was used. This contains the model from a 2019 run of TDT4250.

The Ecore Tree Editor extension was build locally on the author's machine. The resulting `.vsix` extension installer was uploaded to the workspace. Right clicking on this file inside VSCode allowed installation, by selecting "Install Extension VSIX".

The existing model files reside in the `no.ntnu.tdt4250.oving1.model/model` directory.

Results The list is shown in Table 5.1. Each row is a test case with a unique ID. The result is presented in the "Supported?" and "Requires Eclipse IDE" columns. The optimal result in "Supported?" is "YES", meaning the design both supported the case and the artifact implemented it. Next best is "Yes", meaning a clear approach can be seen by the author to develop the artifact further using the existing design's constructs and models, to support the case. A case with "No" indicates that the design needs to change. If a case cannot confidently be answered, the result is "Unknown". A related paragraph for that test should explain why it is unknown. If "Requires Eclipse IDE" is "Yes", the case would need Eclipse IDE to be solved, requiring design changes to the artifact.

ID	Use Case	Supported? [NA/No/Unknown/ Yes/YES]	Requires Eclipse IDE
1	Create new .ecore model file	YES	No
2	View Ecore model by opening the .ecore file	YES	No
3	Create an EPackage, EClass and EAttributes and EReferences	Yes	No
4	Change the properties of the package, class and attributes	Yes	No
5	Create a new dynamic instance file from an EClass	Yes	No
6	Enter dynamic instance data	Yes	No
7	Change the .ecore model by adding a EAttribute to the EClass	Yes	No

Table 5.1 continued from previous page

ID	Use Case	Supported? [NA/No/Unknown/ Yes/YES]	Requires Eclipse IDE
8	Open the dynamic instance, confirm if it is marked as invalid because the new attribute is not filled in.	Unknown	Yes
9	Open the .ecore model file, and add a new validation to the EClass as an EAnnotation. Use the java validation kind, not OCL.	Yes	No
10	Create a .genmodel file based on the .ecore file.	Unknown	No
11	Generate java project with the model code	Unknown	Yes
12	Write a validation in the java code	NA	No
13	Load the model code into the IDE, to use the validation	No	Yes
14	Edit the model and run the custom validation	Yes*	Yes
15	Generate a user interface or editor plugin	No	Yes

Table 5.1: Use Case-evaluation of the Tree Editor Extension design. Based on modeling in TDT4250. Each row and ID is a step in the modeling process. The evaluation result is in the "Supported?" and "Requires Eclipse IDE" columns. For tasks that are not applicable to the modeling environment, the NA (Not Applicable) is chosen. The 'Yes' in "Supported?" indicates that the design should support it, possibly with further development, but not major redesign. The all-capital 'YES' means the developed artifact has demonstrated it.

5.1.1 Test Case Details

The individual results will now be discussed in more detail. Each paragraph is denoted with the test ID in Table 5.1. The knowledge presented from the cases are interior to the design, and provide prescriptive design theories for this context

and scope [53].

1 The design can do this with Commands in the extension. The command should result in a message over TLSP with a file path to create the model at. The server should then use the EMF.Cloud Model Server to create the file, giving it a empty Ecore model as argument.

2 The editor should have a formal and available set of requirements for what it must do in order to comply with the TLSP. This can help when creating new tree editor frontends for other IDEs. If the TLSP should be reused to solve the $m \times n$ problem (see Section 2.9), then the protocol must also be used in other IDEs with the same kind of generic frontend.

The mapping of Ecore elements to nodes should use the EMF `ItemProvider` mechanisms. For the actual Ecore metamodel and GenModel, a specific `ItemProvider` should be used, instead of the *reflective* variant. The reflective variant creates poor labels, from a usability standpoint, and do not match the ones seen in Eclipse IDE.

3 Creating a new node should be done using the `HierarchySchema` construct to constrain the user interface (with regards to child types), and then use a specific command in TLSP to perform the creation. Commands to the TLSP must include enough information to identify the document, model root and node to alter.

The existing design does not use versions for `TreeCustomDocument`, but this may be required in a new design iteration. The LSP uses versioned documents, which can ensure actions and problems are done and reported respectively against the correct document version, where only the “current” version is of interest.

The editor frontend should be as stateless as possible. The frontend can store (cache) the tree as internal state, but should not be responsible for changing the state, as it will cause synchronization issues because the state also exists in the extension and the server. Using the server for all state changes also ensures that invalid states can be prevented.
Also, when state changes are done in the frontend or extension, there is a risk that domain specific knowledge (about EMF) is captured there, which should instead be in the server. Only the server is the truly reusable component, and must contain all such domain specific knowledge.

4 Changing the properties should use the form in the detail view. This form may require completions for certain form fields, such as references to names of other model elements (for example an EReference can have autocompletion for the type property, suggesting EClass names). The TLSP should be queried for property completions, given the node ID, an appropriate property identifier and the current

contents of the property to complete.

The property changes must be stored immediately in the extension (as a "dirty" document or similar), before being applied to the model. This is because the editor frontend is transient, and lost if the user changes editor tabs, having to be recreated from the extension's state and the frontend's cache. Storing the property changes in the extension allows the use of "dirty" (or unsaved) documents, as opposed to the frontend cache where it risks being lost.

5 The `Action` construct should be sent via TLSP, with data that indicates a dynamic instance must be created. This also needs a file name. The server could call back the extension over TLSP to request a filename, where the extension prompts the user.

6 Entering custom instance data should be similar to editing the Ecore model. A want for customizing the label logic and the "tree mapping rules" has been identified, but is not designed for. (For example, a student would use a combination of two fields as the label, such as `firstname + '_' + lastname`.) A next design should accommodate for such flexibility, using configuration files or augmenting model files in the user's project. (The GenModel is an augmenting model file). Alternatively the model could be altered with annotations that specify the label and tree mapping rules.

7 Changing the model should also notify the extension about changes, so the dynamic instance editor tab is able to update its state.

8 It is unknown how to handle a changing model from a model instance's perspective. Some suggestions are that the user server tries to correct the model instance, or the user has to edit the instance using a text editor (XMI). Such corrective behavior would require Eclipse IDE for now, if Eclipse IDE even changes the model instance in this case (this is unknown to the author).

9 The java validation annotations are only text, and defer the job to the code generator. They are easily added as child nodes. The OCL variant should not be hard to support either, as the OCL evaluation would be performed in the server. Doing OCL evaluation in the frontend may need the EMF runtime API to be present, because the OCL evaluator traverses objects using the metamodel. However, the javascript based EMF implementation for cross-ecore (see Section 2.12) was not bug free, based on an attempt to use it in this design.

10 The GenModel needs to know about the Ecore model, because it augments or decorates it. A new file is not necessarily empty; it may have some minimal data, like a XMI structure with a reference to the ecore model file. This may need some

custom logic, possibly contained in a java library used in Eclipse IDE. However, the exact details are unknown because they were not researched, due to being out of scope (time constraints). A new design should reuse the GenModel file creation logic, to create a new genmodel file.

11 The `Action` construct should be sent over TLSP to trigger code generation. However, the server implementation to trigger code generation is unknown. It is assumed that it exists in a reusable java library used by Eclipse IDE, but there is a possibility that it relies on Eclipse IDE internals.

12 Not applicable, as a text editor and java extension does this.

13 When creating a validation in java, the model needs to be compiled and loaded into the server. The current design does not have a mechanism for loading a user created java project. The extension should provide a Command to load the model into the server. The server should be notified with TLSP, in a new design possibly using a custom `Message` construct, specifying an action and parameters the same way GLSP does in its `ActionMessage` construct (see Section 2.12.2). This is because loading a java project is very specific to EMF, and not a general concept for all tree structure editing. Using a custom Message and a handler in the server will avoid polluting TLSP with EMF concepts.

14 Running the custom validation is trivial, just send an `Action` over TLSP. However, this case is marked with an asterisk, because running the validation is only trivial if the server *already has the validation code loaded*. So task 14 depends on 13; if 13 cannot be solved then validation cannot be performed using custom code.

15 The code generator for GenModel does not create a web based editor yet. It creates an Eclipse IDE plugin and editor. A new design should include a code generator that can create a cloud based editor. This could use Theia, and possibly this Tree Editor Extension itself. Theia seems intended for this type of use.
Alternatively, it could use the *Theia Tree Editor* (see Section 2.12.1.2) which the "coffee-editor" example uses, because this deployment can compile Theia. However, that approach can not run in VSCode.

5.2 Qualitative Evaluation of the Tree Editor Extension

The current design (*models* and *constructs*) achieves the basic goals of modeling in a cloud based IDE. Applying the architecture of a general editor and domain specific server seems to fit the problem of tree structure editors.

The developed artifact (*instantiation*) is **not** ready to be used by students of TDT4250, as much of the design is not implemented. Some important features are not yet

supported by the design, mainly related to code generation and validation. They are not proven to be infeasible, but are unexplored.

Specialized editors Another unexplored area is the special adaptation of editors to GenModel and Ecore themselves. The GenModel presented in Eclipse IDE has many options that are not present in the actual file itself. This may mean that there are default options presented in the Eclipse IDE, and only saved to the genmodel file if changed. The feasibility of a customized editor is unexplored.

High development cost An important observation is that creating a Tree Editor frontend from scratch requires many "basic" features to be manually made. Keyboard shortcuts and right-click context menus are not present by default. For example, undoing the text entered in a property form field must also be supported explicitly by the design; the IDE does not provide it for free. The amount of work may be substantial, to reach a high level of quality and useability.

Another factor increasing the development effort, is that this design requires developers to work with two separate domains in mind: the EMF and the generic Tree Editing. Care must be taken to not "bleed" domain concepts and names into the other components, across bounded domain contexts. Every feature of editing Ecore models must first be mapped to a generic Tree Editor concept.

Conflicting protocol design paradigms When designing the protocol, two different paradigms of protocol designs were encountered. One is the LSP approach, where many methods and capabilities are explicit in the protocol. The other is the GLSP approach, where the protocol has approximately 3 methods in total, and all the logic is encapsulated in a `ActionMessage`. The server will perform a `handler` lookup for the message, and thus move much of the "surface area" of the protocol to internal code and handlers. Essentially, the GLSP approach is wrapping the JSON-RPC data inside another layer of JSON-RPC data. One observed advantage of this is the forwarding of JSON-RPC calls: the GLSP approach only needs to forward a `ActionMessage` and it is done. The LSP has to "unpack and re-call" the forwarded message. One potential cause for the difference is that forwarding is much more present in GLSP: it also uses a Custom Editor frontend. Thus messages are relayed from the frontend via the extension and to the server. In LSP, the editor is natively supported by the IDE.
The design of this thesis' artifact chose the LSP approach, of a large and explicit protocol. Without evidence, it is assumed that this approach is easier for *other* developers when they want to implement a server for a domain other than EMF.

5.3 Qualitative Software Architecture Evaluation

The software architecture has been designed to keep EMF out of the tree editor frontend and TLSP protocol. The server has been designed to function independently of the IDE, when it comes to EMF editing. The server also reuses software from EMF to avoid reimplementation.

5.3.1 Reusable Components for Related Migrations

Frontend The frontend instantiation can be further developed to become a high quality component. This should provide enough utility and value that other VSCode extension developers use it when they need to implement tree editors.
Currently, the tree editor frontend is not good enough for this. It is missing a context menu (right click), keyboard navigation, drag-and-drop of nodes, navigation from the VSCode problems menu and log[1]. The current styling is visually unpleasing, and the user experience is poor. None of these are unsolvable problems, but do require substantial effort. That is where the value of a single, reuseable and standard tree editor lies: avoiding the effort when it has already been done.

Protocol The protocol is designed independent of EMF, on the same ideas of the editor frontend. When the protocol is specified enough, so it is near complete and highly functional, different IDEs can start to support it natively. The benefits of LSP lies in this native IDE support: the text editors are already made and high quality; the language server implementer only needs to create one component. Currently, the protocol is underdeveloped. The specified methods have matching functionality in the editor, but do not go further in "fear" of specifying invalid or unusable methods.

The protocol does not have an official specification document or website. The LSP has this, which serves as the authoritative source for the specification, along with Microsoft's implementation for VSCode. A protocol should be specified in clear, unambiguous language, and have a clear indication of who the authors and maintainers of the specification is. A protocol should also be versioned and have a changelog. The current design only specifies the protocol in source code.

Component distribution The current open source project is not very reuseable. While the components have a design and architecture to support reusability, they are not published to artifact repositories like maven central and npm. This is crucial for reusability in other extension developers' projects, as copying code or downloading compiled binaries is deemed "dirty", wrong, and likely to become outdated.

[1]Text editors can navigate to a specific file and line when the user clicks a specific URI. These are used to link to stack trace source locations, warnings and problems.

5.3.2 Components for Migrating EMF to Other IDEs

Reusable server The server is developed independent of VSCode. The only external dependencies are the Java 11 Runtime and a client for the Tree Language Server Protocol. The server should contain all the logic related to EMF. The current instantiation has some knowledge inside the VSCode extension, like the file types and VSCode Command names. This is because they must be known before extension activation. A future and more complete first party integration of TLSP into could move this knowledge to the protocol instead. Regardless, it is a small amount of EMF knowledge, compared to what the server holds. Transition to other IDE will be easy, if they support TLSP and can run the server component.

EMF code reuse The server is successfully able to reuse parts of the EMF framework. The EMF runtime API, the Ecore metamodel, the `ReflectiveItemProvider` and the EMF.Cloud Model Server are all used. A server should reuse other components as well, like the code generator, OCL compiler/interpreter, validation framework and so on. These are not reused in the instantiation yet, but because of time constraints. They are not re-implemented either, which is the "worst case scenario".

5.4 Evaluation of Open Source Project Viability

5.4.1 Project Evaluation

The software project created as part of this thesis has been evaluated according to Section 3.4.2. The results of the evaluation are shown in Table 5.2. Some results are clarified further below, using the test requirement's ID as the paragraph name.

OS8 The project has a Changelog.md file for the VSCode extension, but not for the other components or the project as a whole.

OS10 There is no official plan. But the project could potentially be promoted using EclipseCon 2021, or through email to related stakeholders such as EclipseSource members (like Dr. Jonas Helming).

OS11 The author is the committed person, but the duration of commitment cannot be guaranteed to be long enough.

OS18 Not all the code is commented, but the (subjectively) required parts are.

ID	Requirement	Present [No/Yes]
OS1	Open source license	Yes
OS2	Readme	Yes
OS3	Contributing guidelines	No
OS4	Code of conduct	No
OS5	LICENSE file	No
OS6	NOTICE file	No
OS7	Easy to remember project name	Yes
OS8	Changelog file	No*
OS9	No sensitive material in the commit history (passwords etc)	Yes
OS10	You have a marketing plan for announcing and promoting the project	No*
OS11	A person is committed to managing community interactions (issues, pull requests)	Yes*
OS12	Consistent code conventions and clear function/method-/variable names	Yes
OS13	Wiki or documentation website	No
OS14	GitHub Issue templates for bugs and feature requests	No
OS15	GitHub Pull Request templates	No
OS16	Semantic release versioning (using `major.minor.patch`, like `1.4.1`)	Yes
OS17	Code dependencies (and transitive dependencies) do not use GPL or Sun BCLA	Yes
OS18	Commented and documented code, explaining intentions and edge cases	Yes*

Table 5.2: Evaluation of the project as open source. The requirements are sourced from Mike Linksvayer *et al.* [57], Danny Guo *et al.* [58], Beaton [59] and Wayne Beaton *et al.* [60]. The results are in the "Present" column. A result with an asterisk (*) is explained further in the text.

5.4.2 Readme Evaluation

The root level "Readme" file is very important. It is the face of the project to the world. Many of the open source evaluation elements are related to the Readme. Therefore, they have been condensed to a separate evaluation. Note that this project has multiple modules, and therefore multiple Readme files. This evaluation only looks at the top level, project wide Readme. The results of the evaluation are shown in Table 5.3. Some results are clarified further below, using the test requirement's ID as the paragraph name.

OSR3 The Readme does not have the description, but the GitHub project has a description shown in the side of the webpage of the project.

OSR5 The Readme does not have a project background story, but the GitHub project has a description explaining this is a result of a masters thesis.

ID	Requirement	Present [No/Yes]
OSR1	The project has a name.	Yes
OSR2	Badges or icons (e.g. CI build status, npm version, open-vsix store page, vscode store page)	No
OSR3	Description of what the project does.	No*
OSR4	List of suported features.	No
OSR5	Project background story.	No*
OSR6	How can a user use this project?	No
OSR7	Where can a user get more help?	No
OSR8	Is the project ready for use?	No
OSR9	Feature roadmap	No
OSR10	Contributing description.	No
OSR11	Getting started with contributing.	No
OSR12	Authors and acknowledgement	No
OSR13	Project license	No
OS14	Project status (e.g. active, lost interest, discontinued, deprecated, looking for new owner, migrated)	No

Table 5.3: Evaluation of an open source project's Readme. The requirements are sourced from Mike Linksvayer *et al.* [57], Danny Guo *et al.* [58], Beaton [59] and Wayne Beaton *et al.* [60]. The results are in the "Present" column. A result with an asterisk (*) is explained further in the text.

Chapter 6

Discussion

The discussion will look at the implications of the results and their evaluations in a bigger picture. Arguments will be presented based on these results, eventually leading to the conclusions in the next chapter.

6.1 VSCode as an EMF Tree Editor in the Cloud

When moving the Eclipse Modeling Framework to the cloud, VSCode is a suitable IDE for this. It can run as a cloud IDE in Gitpod. VSCode also provides enough mechanisms to build a tree editor, without unreasonable amounts of effort.

VSCode is a better choice than Theia While Theia is a product of the Eclipse ecosystem's efforts, it remains a deployment target rather than a standalone IDE. Theia is replaced in Gitpod with VSCode as the default IDE. The developers struggle to keep feature parity with VSCode, as they have to catch up whenever VSCode moves forward, while at the same time managing their other tools and components[1]. The Custom Editor API for example, came much later to Theia[2]. Even if Theia have more tools from the Eclipse ecosystem for deploying EMF models, these mostly rely on the Theia Extension mechanism, incompatible with Gitpod[3].

VSCode is extensible enough VSCode does not provide tools for creating tree editors. However, the extension API provided by VSCode has enough features to build a tree editor. The different file extensions can be associated with a customized document editor. There are APIs to perform Commands, such as requesting the extension to create a new file, or some other arbitrary action. Developed extensions can be distributed and installed without extra work, bureaucracy or

[1]The main developers behind Theia are the same as those behind Gitpod and Eclipse Che, meaning they are spreading their efforts thinly.

[2]The API was added in March 2021 to Theia. VSCode released it to the public in March 2020.

[3]Deploying a Theia Extension requires compilation of Theia, and replacing the entire IDE in Gitpod. Remember that a Theia *Plugin* is equivalent to the VSCode extension concept. A Theia Extension is different and not compatible at all with VSCode.

fees/costs. This makes VSCode a good candidate for extending with EMF editor capabilities.

VSCode extensions can show a tree editor The Custom Editor API from VSCode allows an extension to freely render an editor using web technologies like HTML, javascript and CSS. This is enough flexibility to create a custom tree editor to show hierarchical tree structures with labels and icons, and a property sheet. The tree editors in Eclipse IDE can be re-implemented on a functional level as a Custom Editor in VSCode.

VSCode extensions can run compiled programs The Eclipse Modeling Framework relies heavily on java as it exists now. Because VSCode can run compiled programs such as executable `.jar` files, the existing EMF code can be ran under VSCode. However, this code needs to communicate across processes to integrate it into a VSCode extension. VSCode itself has no way of reaching inside the process, but can use standard mechanisms like streams (*stdin/stdout*), sockets, and HTTP requests. This makes it possible to avoid re-implementing all of EMF for a javascript runtime.

6.2 Reuse of EMF java code

Using a server component in the tree editor allows implementing it with java. This in turn opens for reuse of the existing EMF java runtime API, related EMF libraries and the components used in Eclipse IDE.

Java server that reuses EMF Because VSCode extensions can run compiled programs, and not only javascript extension code, java can be used. This in turn opens for using the official EMF runtime API, and the official Ecore metamodel implementation. These allow reflective access to any EMF models. It also saves effort, because it reuses existing code. A contributing open source developer already familiar with EMF will also find it intuitive to contribute on such a server component. It also removes the risk of using third party Eclipse Modeling Framework (EMF) runtime API implementations, like CrossEcore, which are not as battle tested as the java one[4].

Additionally, it lets the server implementation use any EMF tools already developed for java, like solutions for storage, change management, change-transactions and so on.

The EMF.Cloud Model Server can be embeded in the server Instead of adding another process, another communication step, and another components to man-

[4]This solution tried using CrossEcore-generated TypeScript in the editor frontend, but scrapped it because the underlying library and code generator had bugs.

age, the EMF.Cloud Model Server can be used from java. The server is itself made with java, and has an extensible design. The design implementation in this thesis demonstrated that this kind of use is possible. Instead of having a REST API library control the EMF.Cloud Model Server, the internally used components are extracted and put behind the server's TLSP implementation. These components implement several editing-related features, like scanning for al model files, loading EMF `Resources` and other useful features[5].

6.3 Creating a Tree Editor for VSCode Requires Substantial Effort

Even if VSCode is *able* to support a tree editor for EMF, the editor itself has to be created. With the current situation, this is a lot of effort.

Many functional requirements No good source can provide all the functional requirements needed for a EMF tree editor, in a clear and concise manner. Existing literature does not provide it, so it must be extracted. And because this thesis did not have it as an objective, little effort was spent on formally capturing and describing the requirements. The thesis did however attempt to find requirements as an input to the design process, by simulating use cases. The discovery is that the amount of requirements is high, and it is hard to separate between necessities and nice-to-haves. A proper effort to create *the* tree editor component for all tree editing uses, other than EMF, needs to do the requirements engineering more formally. Especially when the functionality needs to be replicated for other IDEs if they are to support TLSP.

Little comes out of the box Regarding the development of a tree editor in VSCode, even the most basic features must be implemented. For example, a text input field in a browser will normally let a user right click and select "undo", or press `ctrl Z` to undo. In a VSCode custom editor, this is not there. No context menu shows up at all when right clicking. The reasoning might be that the Custom Editor is a blank slate, for any kind of editor. But as a developer, one would prefer if an existing framework could provide most of the common and trivial functionality. Luckily, the custom editor is still able to use any javascript libraries and frameworks that target a web browser. If any existing components can provide a context menu or framework for building editors, they can be used. A catch is that they have to conform to the stateless and remote-controlled nature of the Custom Editor's WebView.

High usability demands high effort Because so little comes out of the box, many rudimentary features that are assumed and expected by a user will have

[5]It provides `EditingDomains`, and processes EMF `Commands` It also provides UI Schemas for models, to use with JSON-Forms.

to be implemented. The existing Eclipse IDE and its tools for EMF were disliked partially for problems with the tools. A tree editor in the cloud might not be judged on any "lighter" terms by a student, and can be perceived equally bad or worse. Installation issues and problems with the cloud based editor need to be minimized, to provide a good user experience. And the functionality may have to match the existing high standards of quality that are seen elsewhere on the web and IDE space. Neglecting this may just create another inferior editor, which works against the purpose of increasing MDD and EMF adoption.

Open sourcing requires investment before payoff Publishing a project online is not "open sourcing" it. And developers will not flock to the project as free manpower. As evident in the evaluation, a lot of work has to be done on the Readme and project documentation. Open source projects almost have to be "sold" to developers, like commercial products. The Readme must clearly convey the purpose, the novelty and usefulness, the functionality and guide the contribution setup for the open source project. The project also needs a good, memorable name. Then it has to be marketed to interested developers, so they can discover it. And once contributors come, the project needs a person to manage it, processing any new issues and feature requests, reviewing incoming contribution code and merging pull requests. Only after all this, comes the direct payoffs for the project itself.

However, there is another benefit of being open source. Even if no one wants to contribute, the project can be used as inspiration, code can be copied or "scavenged for parts" into other projects. This project did this itself, by taking the Base Protocol jsonrpc implementation from the open source LSP4J and VSCode LSP projects.

6.4 Designing a Standardized Tree Language Server Protocol

When EMF first is in the process of moving from Eclipse IDE to VSCode, choices to make another future migration easier may pay off. Standardizing on a protocol for all cloud based tree editors, and also isolating all EMF editing to a single server component can support this.

Tree editors can use a standardized protocol The design work in this thesis has shown that EMF can be mapped to a generic tree structure. This tree structure is free of all EMF concepts, and thus other domains with tree structures[6] could map to the same generic tree structure, and immediately fit into the developed tree editor frontend. This tree editor frontend is also demonstrated to be instructed about domain specifics (EMF) using a generic tree editor protocol:

[6]For example HTML, XML, JSON and file systems.

TLSP. Standardizing on such a protocol can give the same benefits seen in LSP in solving the $m \times n$ language–IDE problem, and more out-of-the-box support for a language when moving to a new IDE which supports LSP. Eventually, with good adoption, such a protocol could reduce migrations of EMF to different IDEs to become trivial.

EMF may need to migrate IDE again New IDEs show up. Trends change. VSCode itself launched in 2015. Eclipse IDE launched in 2001, while EMF[7] launched in 2003. While it has been 14 years between Eclipse IDE and VSCode, it was only 2 years between VSCode and Theia. When IDEs become more component-based and reuseable, the available IDEs in the market may increase[8]. Gitpod itself changed from Theia to VSCode. If a solution had implemented EMF editing in Gitpod a year earlier, tightly coupled to Theia, it would already need to transition to VSCode. There are also non-cloud IDEs where EMF could be integrated. For example, IntelliJ is a popular IDEs for Java, without any EMFs support. A premise for this migration, however, is that the other IDEs can implement TLSP and a tree editor frontend.

LSP has a suitable Base Protocol While the Language Server Protocol is designed for text editing, its Base Protocol is generic and reusable. This Base Protocol can be specialized to support a tree editor protocol. A big win for this protocol is that it is proven to work for both LSP and GLSP, for text and diagrams respectively, **and** it has readily available components to create new software. The extension in this design reused the `vscode-jsonrpc` component, directly from the LSP project. The server used the java version of this, from the `Eclipse LSP4J` project. Getting started with the Base Protocol has essentially no upfront development cost. If the design is similarly component-based and LSP independent for the other programming languages supported by LSP, the Base Protocol may already be available as a dependency/library in other programming languages too.

Another advantage of using the Base Protocol, is that it is bi-directional. This allows the server to initiate a request to the client. When working with EMF, the model can be changed by the server or other editors (like a diagram editor). This bi-directional protocol then allows the server to notify any editor of changes.

Tree editors can use TLSP The standardized protocol can be created on the Base Protocol and become analogous to LSP for trees. The design created in this thesis demonstrates that this is feasible. Thus, tree editors can standardize around the Tree Language Server Protocol (TLSP). The current protocol design may need changes, but it should be sufficient as a starting point.

[7] The earliest version the author could find in the Eclipse EMF release page was v1.0.2 in 2003.

[8] A parallel is seen in web browsers, where Google's open source Chromium also is the basis for Microsoft Edge, Vivaldi, Brave, and others.

TLSP design creates reusable software Regardless of a wider adoption of TLSP or not, its design when applied to EMF tree editing will create more reusable software. The editor frontend can be reused by an unrelated tree editing project. The EMF TLSP server can be reused as an alternative to EMF.Cloud Model Server. If the server contains the core of EMF editing, the external interface (TLSP) could be replaced if not needed[9].

6.5 Limitations

Research is not always perfect, and creative design is hard to do in an objective manner. This section addresses some limitations, shortcomings and uncertainties of this thesis.

An unufinished instantiation leaves risk in the design Because development takes longer time than what is allocated for a master's thesis, a substantial amount the protocol design is left on a theoretical design stage. Many challenges and constrains appear first when the design is implemented, imposed by the surrounding frameworks, APIs and tools used to create the software. Although the author is confident in this thesis' use case evaluation, the results cannot be guaranteed with the utmost certainty before they are all implemented. Regardless, the overarching design and protocol should still be valid and contribute new knowledge. Most obstacles encountered from here on, are assumed to be possible to work around with minor changes to the protocol.

Agile development causes a protocol to emerge slowly Doing an Agile approach means creating software that works. It also means postponing work, instead of having a lot of unfinished work in progress. This is good for delivering working software to customers and users, but not for developing a full and complete protocol specification. While an agile approach can always back up the validity and usefulness of protocol elements with a working implementation, it may leave unidentified risks for later, contrary to how research aims to tackle risk early. It can be argued, however, that Design Science Research uses the build-evaluate cycle exactly to produce correct knowledge, instead of drafts and unproven hypotheses.

Lacking contact with stakeholders and Eclipse ecosystem The developers of tools like Theia Tree Editor, ecore-glsp, JSON-Forms etc. may have deeper insights and knowledge than the author. They may be able to suggest better, more pragmatic and effective design solutions. They can also verify or disprove, or at least indicate, the need for generic tree editors and protocols for EMF. The author did not actively communicate with the Eclipse ecosystem, outside of two private chats during EclipseCon 2020. There has also not been contact with other students of

[9] Just like the implemented server did with EMF.Cloud Model Server's REST API.

TDT4250. The main reason is that the course is taught in the autumn, while the master thesis is done during the spring semester. However, the author has been a student of TDT4250 and talked to other classmates during the semester for their opinion on EMF and Eclipse IDE.

Design Science Research methodologies can cause ad-hoc evaluations The literature on Design Science Research says little on how to evaluate. There are only abstract, high level recommendations to the forms of evaluation, such as action research, case study, simulation and so on. This can cause a researcher to create an evaluation which is ultimately positive of their view, or disregards important aspects. The author has attempted to keep the evaluations related to the original research goals, and grounded in existing practice. Regardless, the weakness of unguided evaluation design is there.

Chapter 7

Conclusion

This thesis started by finding a way to enable modeling using Eclipse Modeling Framework in the cloud, to solve Objective 1. A pre-project suggested creating a tree editor for Gitpod, by creating a VSCode extension. This thesis then designed an extension which used a software architecture and protocol that drew inspiration from the Language Server Protocol and Graphical Language Server Platform designs. The thesis also presented a way to implement a tree editor with this architecture, and a method which extracts functional requirements from the existing EMF tree editors in Eclipse IDE through use cases. The architecture is a three component system: a generic tree editor frontend, a VSCode extension, and an EMF-specific server. This architecture also entailed using a protocol, dubbed the Tree Language Server Protocol (TLSP), between the VSCode extension and the server. This design was implemented up to the point it could successfully render EMF models in Gitpod with VSCode.

The successful design and implementation of this architecture means it can be the basis of a new sibling protocol to LSP and GLSP: the TLSP. This can be used by other tools that undergo a similar migration to the cloud. This design also created encapsulated and reusable components. It can also ease future migration of EMF to another Integrated Development Environment, because the EMF logic is contained in a reusable server, independent of VSCode. This means it solves Objective 3: An architecture to enable future related IDE migrations.

Lastly, this thesis produced an open source project for the editor and TLSP. However, open sourcing software requires more effort than what is currently done to attract contributions. Gaining traction online for open source is similar to product marketing. The amount of effort required to create a viable solution for cloud based tree editing in VSCode is substantial. External contributions may be required for completing the implementation.

7.1 Future Work

This thesis has uncovered some new potential areas of research.

Theia or VSCode as a deployment platform The GenModel and code generator can target Eclipse IDE as a deployment platform. The model gets a plugin generated, and an editor in Eclipse IDE for model instances. A similar approach can be interesting for deploying a tree editor for model instances, but using Theia or VSCode instead of Eclipse IDE. These could potentially reuse the TLSP as well. Doing this would increase the value of EMF, and further illustrate the values of Model-Driven Development and code generation.

Collaborative modeling Modeling is a collaborative task. With the move to cloud, and with the increased amount of work from home[1], collaboration can move online as well. This is already normal for things like Google Docs, and Jetbrains just added "Code With Me". Obeo is also developing this for their cloud based Sirius modeling tool. When all the editing is done through a protocol like TLSP, the data could be redirected to multiple clients, meaning multiple students' computers.

Completing the EMF editor A lot of the remaining work is routine design, not research. But some remaining parts may be more challenging, such as getting the GenModel working and specializing the editor to properly show a `.genmodel` file like in Eclipse IDE. There are also challenges to loading the generated code back into the server, for validations and custom `ItemProviders`. Completing this will make the editor more useful, and also move it closer to a solution suited for industry use, beyond just education.

[1] Due to covid-19.

Bibliography

[1] K. Rekstad, "A Modeling Environment in the Cloud for Education," Preproject, Norwegian University of Science and Technology, Trondheim, Norway, Dec. 9, 2020, 81 pp. [Online]. Available: https://github.com/krissrex/ntnu-tdt4501-preproject-article/releases/download/v0.1-29/thesis.pdf.

[2] Typefox. "TypeFox - Smart Tools For Smart People." (), [Online]. Available: https://www.typefox.io/ (visited on 12/09/2020).

[3] Eclipsesource. "EMF Forms Editors," EclipseSource. (Feb. 2016), [Online]. Available: https://eclipsesource.com/blogs/tutorials/emf-forms-editors/ (visited on 11/11/2020).

[4] Jonas Helming, "Ecore tools in the cloud - behind the scenes," presented at the EclipseCon 2020 (https://www.youtube.com/watch?v=YQyaCR_V5zc), Oct. 22, 2020. [Online]. Available: https://www.eclipsecon.org/2020/sessions/ecore-tools-cloud-behind-scenes (visited on 05/29/2021).

[5] L. Kuzniarz and L. E. G. Martins, "Teaching Model-Driven Software Development: A Pilot Study," in *Proceedings of the 2016 ITiCSE Working Group Reports*, ser. ITiCSE '16, New York, NY, USA: Association for Computing Machinery, Jul. 9, 2016, pp. 45–56, ISBN: 978-1-4503-4882-9. DOI: 10.1145/3024906.3024909. [Online]. Available: https://doi.org/10.1145/3024906.3024909 (visited on 11/26/2020).

[6] Jordi Cabot. "I failed to convince my students about code-generation," Modeling Languages. (Feb. 9, 2015), [Online]. Available: https://modeling-languages.com/failed-convince-students-benefits-code-generation/ (visited on 12/04/2020).

[7] Jon Whittle, John Hitchinson, Mark Rouncefield, Håkan Burden, and Rogard Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering," 2015. DOI: 10.1007/s10270-015-0487-8.

[8] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, ser. Synthesis Lectures on Software Engineering 1. San Rafael, Calif.: Morgan & Claypool, 2012, 166 pp., ISBN: 978-1-60845-882-0.

[9] J. Krogstie, *Model-Based Development and Evolution of Information Systems: A Quality Approach*. New York: Springer, 2012, ISBN: 978-1-4471-2935-6.

[10] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004, 529 pp., ISBN: 978-0-321-12521-7.

[11] Hallvard Trætteberg and Jon Espen Ingvaldsen. "EMF - TDT4250 - NTNU Wiki." (Jun. 26, 2017), [Online]. Available: https://www.ntnu.no/wiki/display/tdt4250/EMF (visited on 05/30/2021).

[12] Hallvard Trætteberg. "Constraints and validation - TDT4250 - NTNU Wiki." (Apr. 9, 2020), [Online]. Available: https://www.ntnu.no/wiki/display/tdt4250/Constraints+and+validation (visited on 05/30/2021).

[13] Hallvard Trætteberg. "EMF step-by-step - TDT4250 - NTNU Wiki." (Jun. 15, 2017), [Online]. Available: https://www.ntnu.no/wiki/display/tdt4250/EMF+step-by-step (visited on 05/30/2021).

[14] Hallvard Trætteberg. "Editing Ecore model instances - TDT4250 - NTNU Wiki." (Jun. 25, 2017), [Online]. Available: https://www.ntnu.no/wiki/display/tdt4250/Editing+Ecore+model+instances (visited on 05/30/2021).

[15] Hallvard Trætteberg. "Genmodel - TDT4250 - NTNU Wiki." (Jun. 26, 2017), [Online]. Available: https://www.ntnu.no/wiki/display/tdt4250/Genmodel (visited on 05/30/2021).

[16] Ed Merks. "EcoreEditor.java." in collab. with IBM. (May 6, 2021), [Online]. Available: https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.editor/src/org/eclipse/emf/ecore/presentation/EcoreEditor.java (visited on 05/30/2021).

[17] Ed Merks, Frank Budinsky, Marcelo Paternostro, and Dave Steinberg, *EMF: Eclipse Modeling Framework*, 2nd ed., Erich Gamma, Lee Nackman, and John Wiegand, Eds., ser. The Eclipse Series. Upper Saddle River, NJ: Addison-Wesley, 2009, 704 pp., ISBN: 978-0-321-33188-5.

[18] StackOverflow. "Stack Overflow Developer Survey 2019," Stack Overflow. (2019), [Online]. Available: https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019 (visited on 12/07/2020).

[19] Sven Efftinge. "Product Roadmap, Q1/2021," Notion. (), [Online]. Available: https://www.notion.so/Product-Roadmap-b9b5eac0a15147ac8d2dd25cf0519203#1ddc3df582b14349b9d0ebb194e7af94 (visited on 06/13/2021).

[20] Benjamin Pasero and G. Van Liew, *Source Code Organization*, in *Visual Studio Code Wiki*, 1f6491a, Microsoft, Oct. 5, 2020. [Online]. Available: https://github.com/microsoft/vscode/wiki/Source-Code-Organization (visited on 10/05/2020).

[21] J. Helming and M. Koegel. "The Eclipse Theia IDE vs. VS Code," EclipseSource. (Dec. 6, 2019), [Online]. Available: https://eclipsesource.com/blogs/2019/12/06/the-eclipse-theia-ide-vs-vs-code/ (visited on 10/05/2020).

[22] J. Helming and M. Koegel. "How to add extensions and plugins to Eclipse Theia," EclipseSource. (Oct. 17, 2019), [Online]. Available: https://eclipsesource.com/blogs/2019/10/17/how-to-add-extensions-and-plugins-to-eclipse-theia/ (visited on 12/06/2020).

[23] Sven Efftinge and Miro Spönemann. "Open VSX." (Apr. 9, 2020), [Online]. Available: https://www.gitpod.io/blog/open-vsx/ (visited on 12/06/2020).

[24] Microsoft. "Overview," LSP/LSIF. (), [Online]. Available: https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/ (visited on 12/07/2020).

[25] Microsoft. "Tools supporting the LSP." (), [Online]. Available: https://microsoft.github.io/language-server-protocol/implementors/tools/ (visited on 06/15/2021).

[26] Microsoft. "Language Server Protocol Specification - 3.16." (Jun. 4, 2021), [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/specification-3-16/ (visited on 06/15/2021).

[27] Microsoft. "Language Server Extension Guide," Language Server Extension Guide. (Sep. 10, 2020), [Online]. Available: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide (visited on 09/23/2020).

[28] JSON-RPC Working Group. "JSON-RPC 2.0 Specification." (Mar. 26, 2010), [Online]. Available: https://www.jsonrpc.org/specification (visited on 09/23/2020).

[29] W. Beaton. "Eclipse Cloud Development," projects.eclipse.org. (Oct. 28, 2014), [Online]. Available: https://projects.eclipse.org/projects/ecd (visited on 06/15/2021).

[30] C. Smith. "Eclipse EMF.cloud," projects.eclipse.org. (Sep. 18, 2019), [Online]. Available: https://projects.eclipse.org/projects/ecd.emfcloud (visited on 06/15/2021).

[31] Tobias Ortmayr, Eugen Neufeld, and Camille Letavernier, *Eclipse-emfcloud/emfcloud*, version 06e077d, eclipse-emfcloud, Jun. 14, 2021. [Online]. Available: https://github.com/eclipse-emfcloud/emfcloud (visited on 06/16/2021).

[32] E. Foundation. "EMF.cloud." (), [Online]. Available: https://www.eclipse.org/emfcloud/ (visited on 06/15/2021).

[33] Eugen Neufeld, *Eclipse-emfcloud/coffee-editor*, version ddeac18, eclipse-emfcloud, May 26, 2021. [Online]. Available: https://github.com/eclipse-emfcloud/coffee-editor/blob/573830104d9ad541568e6b46f8b3359e8dd0948d/backend/plugins/org.eclipse.emfcloud.coffee.modelserver.app/META-INF/MANIFEST.MF (visited on 06/16/2021).

[34] Nina Doschek, *Eclipse-emfcloud/ecore-glsp*, version 03fd9c5, eclipse-emfcloud, May 17, 2021. [Online]. Available: https://github.com/eclipse-emfcloud/ecore-glsp/blob/cbb4efff9351a43414c47fba25bad1b9630d9a67/client/packages/theia-ecore/package.json#L23 (visited on 06/16/2021).

[35] Eclipse Foundation. "GLSP," GLSP. (2020), [Online]. Available: https://www.eclipse.org/glsp/ (visited on 09/29/2020).

[36] Tobias Ortmayr, Philip Langer, Martin Fleck, Camille Letavernier, Nina Doschek, Lucal Koehler, and Johannes Faltermeier, *Eclipse-glsp/glsp-server actions*, version db6dac6, eclipse-glsp, Jun. 16, 2021. [Online]. Available: https://github.com/eclipse-glsp/glsp-server/tree/63a99f86e40c81a5ddd1b08970a1e374e8c79259/plugins/org.eclipse.glsp.server/src/org/eclipse/glsp/server/actions (visited on 06/16/2021).

[37] Tobias Ortmayr, *Eclipse-glsp/glsp-vscode-integration*, version 53fa808, eclipse-glsp, Jun. 11, 2021. [Online]. Available: https://github.com/eclipse-glsp/glsp-vscode-integration/blob/0fe499f6abc246e1c2cb0a31edf333c7525b13d4/packages/vscode-integration/src/glsp-webview.ts#L185 (visited on 06/16/2021).

[38] Philip Langer, *Eclipse-glsp/glsp-server*, eclipse-glsp, Jun. 16, 2021. [Online]. Available: https://github.com/eclipse-glsp/glsp-server/blob/63a99f86e40c81a5ddd1b08970a1e374e8c79259/plugins/org.eclipse.glsp.server/src/org/eclipse/glsp/server/jsonrpc/GLSPJsonrpcServer.java (visited on 06/16/2021).

[39] EclipseSource. "What is JSON Forms? | JSON Forms." (), [Online]. Available: https://jsonforms.io/docs (visited on 06/16/2021).

[40] Simon Schwichtenberg, *Crossecore/ecore-typescript*, version 2e7b04a, CrossEcore, Apr. 30, 2021. [Online]. Available: https://github.com/crossecore/ecore-typescript (visited on 06/16/2021).

[41] Simon Schwichtenberg. "CrossEcore," GitHub. (), [Online]. Available: https://github.com/crossecore (visited on 06/16/2021).

[42] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed, ser. SEI Series in Software Engineering. Upper Saddle River, NJ: Addison-Wesley, 2013, 589 pp., ISBN: 978-0-321-81573-6.

[43] K. E. Wiegers and J. Beatty, *Software Requirements*, Third edition. Redmond, Washington: Microsoft Press, s division of Microsoft Corporation, 2013, 637 pp., ISBN: 978-0-7356-7966-5.

[44] Alan Hevner and Samir Chatterjee, *Design Research in Information Systems*, ser. Integrated Series in Information Systems. Springer, Boston, MA, 2010, vol. 22, ISBN: 978-1-4419-5653-8. [Online]. Available: https://doi.org/10.1007/978-1-4419-5653-8.

[45] B. J. Oates, *Researching Information Systems and Computing*. London ; Thousand Oaks, Calif: SAGE Publications, 2006, 341 pp., ISBN: 978-1-4129-0223-6 978-1-4129-0224-3.

[46] Vijay Vaishnavi, William Lewis Kuechler, and Stacie Petter, Eds., *Design Science Research in Information Systems*, in collab. with Gerard De Leoz, Jun. 30, 2019. [Online]. Available: `http://www.desrist.org/design-research-in-information-systems/` (visited on 02/02/2021).

[47] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. "Manifesto for Agile Software Development." (2001), [Online]. Available: `https://agilemanifesto.org/` (visited on 06/04/2021).

[48] "Principles behind the Agile Manifesto." (), [Online]. Available: `https://agilemanifesto.org/principles.html` (visited on 06/04/2021).

[49] Rachelle Lynn. "Guiding Principles of Lean Development," Planview. (), [Online]. Available: `https://www.planview.com/resources/articles/lkdc-principles-lean-development/` (visited on 06/05/2021).

[50] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Reading, Mass: Addison-Wesley, 2000, 321 pp., ISBN: 978-0-201-61622-4.

[51] H. Kniberg, M. Cohn, and J. Sutherland, *Scrum and XP from the Trenches: How We Do Scrum*. C4Media, 2015, ISBN: 978-1-4303-2264-1.

[52] M. C. Feathers and R. C. Martin, *Working Effectively with Legacy Code*, ser. Robert C. Martin Series. Upper Saddle River, NJ: Prentice Hall Professional Technical Reference, 2005, 434 pp., ISBN: 978-0-13-117705-5.

[53] C. Sonnenberg and J. vom Brocke, "Evaluations in the Science of the Artificial – Reconsidering the Build-Evaluate Pattern in Design Science Research," in *Design Science Research in Information Systems. Advances in Theory and Practice*, K. Peffers, M. Rothenberger, and B. Kuechler, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 381–397, ISBN: 978-3-642-29863-9.

[54] J. Venable, J. Pries-Heje, and R. Baskerville, "A Comprehensive Framework for Evaluation in Design Science Research," in *Design Science Research in Information Systems. Advances in Theory and Practice*, K. Peffers, M. Rothenberger, and B. Kuechler, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 423–438, ISBN: 978-3-642-29863-9.

[55] George Tsiolis. "Menu entry "About Gitpod" for Theia is missing an image · Issue #3989 · gitpod-io/gitpod," GitHub. (Apr. 2019), [Online]. Available: `https://github.com/gitpod-io/gitpod/issues/3989#issuecomment-822246441` (visited on 06/09/2021).

[56] Simon Brown. "The C4 model for visualising software architecture." (), [Online]. Available: https://c4model.com (visited on 06/07/2021).

[57] Mike Linksvayer, Mike McQuaid, Justin Kenyon, Katrin Leinweber, Sophie Shepherd, John Mueller, Emil Laine, and Andrew Lee. "Starting an Open Source Project," Open Source Guides. (Dec. 16, 2020), [Online]. Available: https://opensource.guide/starting-a-project/ (visited on 05/20/2021).

[58] Danny Guo, Haroen Viaene, David Kleuker, Eric Berry, and Alex Falcon. "Make a README," Make a README. (Aug. 20, 2020), [Online]. Available: https://www.makeareadme.com (visited on 05/20/2021).

[59] W. Beaton. "Third Party Content Licenses," The Eclipse Foundation. (Oct. 19, 2017), [Online]. Available: https://www.eclipse.org/legal/licenses.php#approved (visited on 06/13/2021).

[60] Wayne Beaton, Fabian Steeg, Denis Roy, Dave Carver, Ed Merks, Bjorn Freeman-Benson, Anne Jacko, Ian Skerrett, Mike Milinkovich, John Arthorne, and Eclipse Foundation, Inc. "Eclipse Project Handbook | The Eclipse Foundation," Eclipse Foundation Project Handbook. (2020), [Online]. Available: https://www.eclipse.org/projects/handbook/ (visited on 05/20/2021).

Appendix A

Tree Editor Functional Requirements from Pre-project

The following Table A.1 is copied from the results section in the pre-project, at [1, p. 47-48]. It presents a non-complete list of functional requirements for a tree editor.

Table A.1: Functional requirements for a master-detail Tree editor with property sheet.

ID	Requirement	Description
		The software must use an extension mechanism to provide a custom editor for trees.
FR1	Provide an interactive Tree Editor in VSCode and Theia (Gitpod)	A textual representation is not sufficient. The tree comprises a hierarchy of nodes and their child nodes.
FR2	Provide an interactive Property sheet in VSCode and Theia (Gitpod)	The software must use an extension mechanism to provide a custom property sheet for tree nodes. The property sheet needs to be synchronized with the selected node in the tree editor.
FR3	Provide an action bar with dynamically provided actions in VSCode and Theia (Gitpod).	The action bar should have actions that are specified by a backend Tree Language Server.
FR4	The Tree must view nodes with labels and icons.	Every node should have a default icon that depends on its node "type". Every node should have a name that is read from the node data.
FR5	Tree nodes with children can toggle the visibility of children by user interaction.	An icon or symbol will show if a node has children. If the user interacts with this icon, e.g. a click, all the children will toggle their visibility on/off.
FR6	The Tree and Property views update automatically when the underlying model changes.	Subscribe to change notifications from the Model Server, in the Tree Language Server.
FR7	The Action Bar updates when the tree selection changes.	Show the available actions for the newly selected node.

Table A.1 continued from previous page

ID	Requirement	Description
FR8	Support creation of new nodes.	
FR9	Support deletion of existing nodes.	
FR10	Support selecting a node.	

Appendix B

Pre-project Data Structure Code

The data structure for containing a tree, designed in the pre-project during prototype number two, is shown in Code listing B.1.

The structure for Actions are shown in Code listing B.2 and Code listing B.3.

The structure for defining a node hierarchy is shown in Code listing B.4.

Code listing B.1: Javascript code from the WebView for a data model describing the tree nodes. This listing is copied from "Code listing 5.3" in [1, p.43, 44].

```
// Icon could be configured as default + optional overrides

var exampleTree = {
  type: "root",
  children: [
    {
      type: "EResource",
      name: "MyEcore.ecore",
      icon: "",
      id: "1",
      properties: [],
      children: [
        {
          type: "EPackage",
          name: "my-ecore",
          icon: "",
          id: "2",
          properties: [],
          children: [
            {
              type: "EClass",
              name: "Person",
              icon: "",
              id: "3",
              state: {
                selected: true,
                valid: false,
                dirty: true,
              },
              properties: [],
              children: [
```

```
                {
                  type: "EAttribute",
                  name: "age",
                  icon: "",
                  id: "4",
                  properties: [
                    {
                      name: "Value",
                      value: 25,
                      label: "This␣is␣JSON-Forms␣territory",
                    },
                  ],
                },
              ],
            },
            {
              type: "EClass",
              name: "MyOtherClass",
              icon: "",
              id: "4",
              properties: [],
              children: [],
            },
          ],
        },
      ],
    },
  ],
};
```

Code listing B.2: Javascript code from the WebView to specify the available actions. This listing is copied from "Code listing 5.4" in [1, p.45].

```
var exampleAvailableActions = [
  { id: 0, name: "Create␣dynamic␣instance..." },
  { id: 1, name: "Validate" },
  { id: 2, name: "Create␣genmodel..." },
];
```

Code listing B.3: Javascript code from the WebView to specify default actions and per-node actions. This listing is copied from "Code listing 5.5" in [1, p.45].

```
var exampleDefaultActions = [1];
var exampleActionSchema = {
  EResource: [2],
  EClass: [0],
};
```

Code listing B.4: Javascript code from the WebView to specify what children a node type can have. This listing is copied from "Code listing 5.6" in [1, p.45].

```
/** Could be generated from Ecore using EReferences present in metamodel. */
var exampleHierarchySchema = {
  root: ["EResource"],
  EResource: ["EPackage"],
  EPackage: ["EClass", "EDataType", "EEnum"],
  EClass: ["EAttribute", "EReference", "EOperation", "EAnnotation"],
  EAttribute: ["EAnnotation"],
```

```
};
```

Kristian Rekstad

A Modeling Environment in the Cloud for Education

NTNU
Norwegian University of
Science and Technology