A Modeling Environment in the Cloud for Education

Kristian Rekstad

December 9, 2020

Specialization Project for Master's Thesis

Dept. of Computer Science
Norwegian University of Science and Technology

Abstract

Model-Driven Development is a an approach to software engineering. A common framework for Model-Driven Development (MDD) is the Eclipse Modeling Framework. However, students resist learning it because of its use of unpopular technologies. This thesis comes before a master's thesis, and looks at how modeling can be moved to cloud based editors in order to renew the technology stack. At the same time, opportunities for saving effort are found, by reusing existing implementations, architectures and protocols from other tools. This thesis ends by finding the need for a tree based editor, and testing the feasibility of it. Finally, some requirements and an architecture for such an editor are found. These can be further studied and implemented in a master's thesis.

Sammendrag

Model-dreven utvikling (MDU) er en tilnerming innen programvareutvikling. Et vanlig frammeverk for MDU er Eclipse Modeling Framework. Desverre gir studenter motstand når de skal lære dette rammeverket fordi den bruker upopulære teknologier. Denne avhandlingen kommer før en masteroppgave, og ser på hvordan modelering kan flyttes til skybaserte redigeringsprogram for å fornye teknologien som brukes. Samtidig ser den etter muligheter for gjenbruk av eksisterende implementasjoner, arkitekturer og protokoller for å spare arbeidsinnsats. Avhandlingen slutter med å finne et behov for et tre-basert redigeringsverktøy, og tester hvorvidt det er gjennomførbart. Til slutt presenteres funksjonelle krav og en arkitektur for et slikt redigeringsverktøy. Disse kravene kan ses på videre og implementeres i en masteroppgave.

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, and for my education.

Jonas Helming for providing answers about my research, and the initial title for the thesis.

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

Contents

Figures

Tables

Code Listings

Acronyms

API Application Programming Interface. 7, 11, 14, 16, 17, 25, 32, 33, 35, 38–40, 42–44, 47, 49, 52

DSL Domain-specific language. 8, 35

EMF Eclipse Modeling Framework. i, 1–3, 6–8, 14, 17, 21, 23, 25, 27, 29, 35, 44, 53

EMOF Essential Meta Object Facility. 8

GLSP Graphical Language Server Platform. 12, 14, 19–21, 25, 27, 35–37, 42, 52, 53

GWT Google Web Toolkit. 8

IDE Integrated Development Environment. 3, 4, 6, 8, 14, 17, 18, 21, 22, 29, 32, 35, 38, 52, 53

LSP Language Server Protocol. 9, 14, 18–21, 25, 33, 35–37, 52, 53

MDD Model-Driven Development. i, 1–4, 7, 21–24, 48, 52

NTNU Norges Teknisk-naturvitenskapelige Universitet. 3

OMG Object Management Group. 8

RAP Remote Application Platform. 8, 35

RCP Rich Client Platform. 8, 10, 35

RPC Remote Procedure Call. 17

svg Scalable Vector Graphics. 21

XMI XML Metadata Interchange. 8, 9, 11, 12, 16, 22, 25, 44, 47

Glossary

cloud Remote data centers that provide computing as a service. Commonly used by businesses to provide web infrastructure. 2–4, 7, 11, 14, 17, 21–25, 27, 29, 51–53

Eclipse Che A cloud based or self hosted workspace and IDE for software development. It is based on Kubernetes and Theia. 32, 33, 38

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. For more details, see Section 5.2.1.3. 2, 3, 7–13, 21, 23, 27, 35, 52

Eclipse Layout Kernel Eclipse Layout Kernel (ELK) is a system for automatic diagram layout. [**eclipsefoundationEclipseLayoutKernel**] It positions nodes and edges for optimal viewing. 14

Ecore The EMF core model. A metamodel similar to UML Class Diagrams. 1, 3, 6, 8, 9, 11, 12, 16, 17, 22, 23, 25, 27, 29, 44, 49, 53

Electron A desktop application runtime for javascript, based on Chromium. 25, 30, 32

Gitpod A cloud based workspace and IDE for software development. It is based on Docker, Kubernetes and Theia. 3, 32

JSON Javascript Object Notation. A serialization format for object structures. 8, 16, 17, 22, 25, 38, 43, 45, 47

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. 17–19, 21, 25, 33, 38, 42, 45, 49

Kubernetes A container orchestration system created by Google. 7

metamodel In conceptual modeling, the model itself can be modeled. This model of the model is called a meta-model. Ecore is one example of a metamodel. 1, 7, 8

NodeJS A javascript interpreter for desktop, based on the Chromium V8 javascript engine. It also includes some desktop APIs like filesystem access. 25, 37, 39–42

NPM Node Package Manager (NPM). Hosts javascript packages online. Similar to Maven Central, but for javascript. 39

open source The source code for a software is available; not just for inspection, but for re-use and modification. 1, 3, 6, 7, 10–12, 14, 16, 17, 27, 30, 32

React A javascript framework by Facebook for rendering views in HTML. 16

REST Represential State Transfer (REST). A paradigm for creating HTTP APIs, centered around resources. 16, 25, 33, 47, 49, 52

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. 2, 3, 6, 7, 12, 14, 21, 22, 25, 32, 33, 35, 38, 39

UML Unified Modeling Language. A common modeling language for creating diagrams such as Class Diagrams. It is standardized by Object Management Group. 8, 12

VSCode Visual Studio Code. An IDE for software development. The full name is Visual Studio Code. Managed by Microsoft. 2, 7, 14, 21, 22, 29, 30, 32, 33, 36, 40, 43, 45

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. 16, 33, 49

Chapter 1

Introduction

1.1 Context

Software Engineering and Modeling Software engineering is a broad field. It has multiple approaches to methodology, programming languages, practices and paradigms. One approach to developing software systems is to model a domain. The domain is usually a phenomena in the real world which a business is interested in. And the conceptual model is an abstraction of this domain, where essential actors, entities, properties and relationships are formalized.

Model usage This conceptual model can be utilized by the engineers to various degrees. It could simply document the domain in order to help understanding and communication among those working with the domain. Or the model could be the basis of a software system. The software could be generated automatically based on the model, or an interpreting system could execute the model itself.

Model-Driven Development When the model itself is the source of truth for software, the model is *driving* the development. This is called Model-Driven Development (MDD). Changes to the domain result in changes to the model, before changing the system's code. And the code itself is *derived from the model*, often using model-to-text transformations. Oftentimes a *complete* system can not be generated, and a programmer is required to manually code some details or rules of the domain into the generated code.

Education and MDD The concepts and tools used in MDD are being taught to software engineering students. Not all curriculums include MDD, as it is mostly relevant for specializations in Software Development. Development using models is very reliant on tools. One of the tools that students learn is the Eclipse Modeling Framework (EMF) and its metamodel called Ecore. An advantage of EMF is the open source nature, which allows students to look inside its implementation, and

use it for free.[1]

EMF in detail The Eclipse Modeling Framework allows a modeller to create a representation of their domain, and generate java code. The java code has classes for all the modeled entities. It can also generate an *editor*, which is a user facing application. This editor lets a user enter objects from their domain. For example, if the domain was a book store, the model would be *books* with *names* and *authors*. And an object could be *"Harry Potter"* by *"J. K. Rowling"*. The generated editor uses Eclipse IDE as its platform, extending it with an *Eclipse plugin*.

Issues with EMF Despite EMF being a great framework, its tight integration with Eclipse IDE harms the adoption. Some students have a negative attitude against Eclipse IDE, and learning about its architecture feels like wasted time. Kuzniarz and Martins [1] found that students resist when the technology and skills are not used in other courses. A general problem for MDD adoption identified by Jon Whittle *et al.* [2] was a low impact on personal career needs. The students look to the industry, and mostly find mentions of "Cloud, IntelliJ, React, Typescript, Scrum, AI", never Eclipse. They forget that much of the popular technologies are hype driven, and not the only tools used. Brambilla *et al.* [3, p. 21–23] claims that model-driven software engineering has passed the peak of the hype curve, and is now in the "slope of enlightenment". This means MDD is not dead, but rather that we are only now seeing how it is best used.[2] A co-author of [3] tried teaching students in 2015, and collected their feedback. Much of the complaints were about installation issues and problems with the tools. [4]

The proposed solution A platform shift from the Eclipse IDE desktop application to the cloud could alleviate many of the problems. An editor in the cloud would have to be web-based, and could use many of today's modern (and popular) technologies. Another win is that an "editor-as-a-service" does not need installation. As mentioned, this caused many problems in [4]. Editors based on web technology is not unheard of either. One such editor that has become very popular, is Microsoft's VSCode.[3] In fact, the Eclipse Foundation itself has picked up on this trend, and accepted a web-based editor called Theia into their umbrella of projects. This means that the platforms to build a cloud based modeling editor are here. This is not a new thought either, as Lajmi [5] said in 2016: "the next wave of modeling tools will probably be based on cloud IDEs." On the other hand, recreating the entire Eclipse Modeling Framework is no easy task. Therefore, its tooling for validation, serialization etc. should be reused if possible.

[1] Compare this to Matlab+Simulink, which some students learn. It costs 20000 NOK per year for a license.

[2] Compare this to how the hyped Blockchain was supposed to fix every problem, but now we see it doesn't solve much at all.

[3] This thesis is written using VSCode!

Related works Some research related to MDD and the cloud has been done already. In [6] they discuss using an existing model in the cloud. Coulon *et al.* [7] also uses an existing model for their deployment to a cloud IDE. Saini *et al.* [8] discusses modeling in the web, but at an early stage before tools like Sprotty (Section 2.2.3.1) were ready for use. In [9] they create a model editor, but for a language called `UML-RT`.
None of these earlier works look at modeling in the cloud where the goal is to create Ecore models. They either *deploy* Ecore models ([6, 7]), are textual editors ([7, 8]) or not modeling with Ecore ([9]).

1.2 Stakeholders

Kristian Rekstad (author) I believe Model-Driven Development (MDD) has potential, but is being held back by the tools. As a soon-to-be professional developer, I hope that I can leverage the best parts of MDD in my professional career. I will be doing the development of any discovered solution, at least initially. If the solution is "good", it may be handed over to an open source community for further development and maintenance.

Hallvard Trætteberg This project's supervisor and associate professor at the Department of Computer Science at NTNU. He has lectured *TDT4250: Advanced Software Design* and uses EMF there. He has an expert level understanding of Eclipse IDE's internal architecture and design. A solution that can increase students' interest in learning MDD would be of interest. He also seeks to move lecturing over from Eclipse IDE to the cloud and Theia-based Gitpod, if this is possible.

Norwegian University of Science and Technology (NTNU) They educate students in Model-Driven Development, and currently use EMF. Their lectures could use a cloud based modeling editor instead of Eclipse IDE.

Eclipse Foundation They own and maintain the Eclipse Modeling Framework, as well as Theia and Eclipse IDE.

1.3 Research Questions

These are the initial research questions. After more background material is collected, they will be refined in Chapter 3.

RQ1. *How can we modernize Model-Driven Development Frameworks to appeal to the next generation of software developers, using recent developments in cloud IDEs?*

— **RQ1.1.** *What tools are essential to use Model-Driven Development in the cloud?*

— **RQ1.2.** *What are the characteristics of a good cloud based tool for Model-Driven Development?*

— **RQ1.3.** *How can existing modeling tools and software architectures be reused in the new cloud based modeling tool?*

1.4 Scope

This thesis is limited in scope, because it is part of a specialization project. The main purpose of the thesis is to research and explore the background material. This thesis will be followed up by a master's thesis. Therefore, this thesis will only solve a part of the problems found.
The research questions in Section 1.3 will be refined from this exploration. Some software development will be done, but the goal will be determining feasibility and discovering potential problems for a final solution. The development will mainly be exploratory prototyping.

1.5 Overview

An overview of the background material is shown in Figure 1.1, before diving into it. The figure is not optimal with respect to readability, but it illustrates that many of the items are related. Not all relations are shown either, to maintain *some level* of readbility.

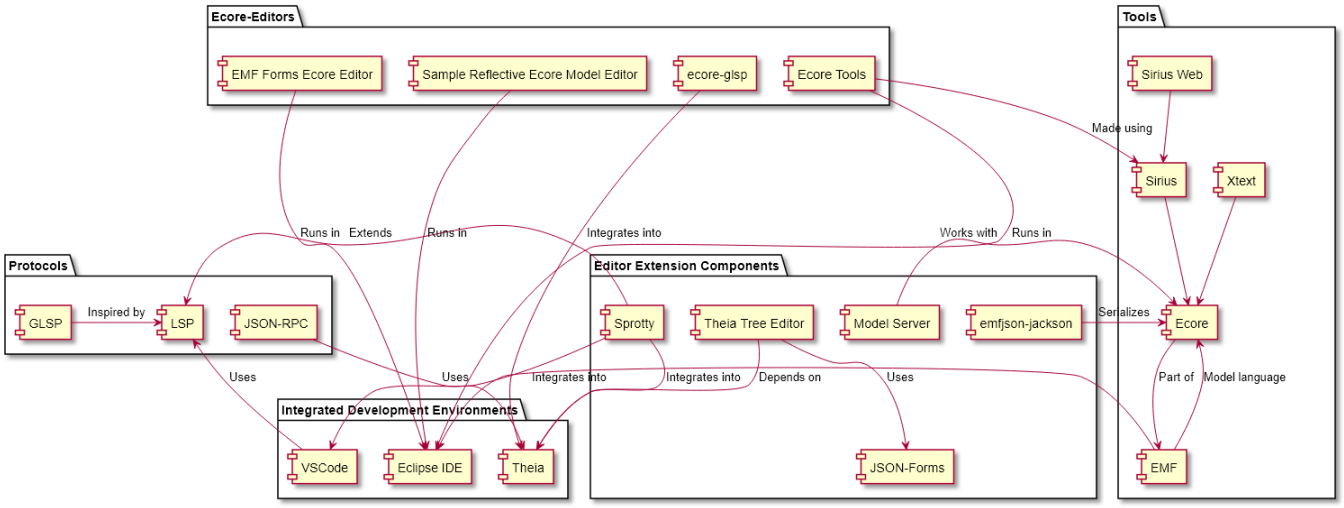


Figure 1.1: An overview of how the background is related.

Chapter 2

Background

2.1 The Open Source Ecosystem

Most of the work related to Eclipse Modeling Framework is done as open source.

2.1.1 Co-opetition

Starting of with a digression An interesting consequence of companies pushing for open standards while providing their own services and tools, is this mix of co-operation and competition: *co-opetition*.
Obeo works to make Sirius great, while providing their proprietary Obeo Designer. At the same time, EclipseSource is competing with its GLSP and their own consulting services. But both of these want to push Eclipse Modeling Framework (EMF) and Ecore forward, by cooperating on its tools and underlying technologies, because these are used at the core of their own products.
Another instance is Typefox with Gitpod, competing with RedHat's Che. However, both contribute to the Integrated Development Environment (IDE) they use: Theia.

2.1.2 Actors

This section will present some of the main actors in the Eclipse Modeling Framework ecosystem. Most of the technologies discussed in Section 2.2 are developed by these actors. Some of the technologies are mentioned by name here, and will be discussed later on.

Eclipse Foundation An independent not-for-profit corporation created in 2004. It aims to serve the Eclipse community, which sprung out from the *Eclipse Project* created by IBM in 2001. [10] Many open source projects are hosted under the Eclipse Foundation.

Obeo Specializes in tailoring open source modeling solutions to clients. [11] Contributes to Eclipse Foundation projects like Sirius, Acceleo, Ecore Tools. [12]

EclipseSource Develops products and tools for engineers. [13] Contributes to Eclipse Foundation projects like GLSP, Sprotty, Theia, EMF Forms, EMF.Cloud. [14]

TypeFox Aims to help businesses and enigneers by providing software tools. [15] The original creator of Theia and Gitpod.

RedHat A provider of enterprise open source solutions. Specializes in linux, cloud and Kubernetes. [16] Creators of Che, and contributes to Theia.

Microsoft The original creators of VSCode, a text editor based on web technology. Microsoft is not a big actor in the modeling world. They are only relevant here because of VSCode. They decide which APIs to support (which Theia may have to implement). Microsoft do not consider the EMF or Theia developers when making choices.

2.2 Technologies

This section will discuss some of the tools, editors, editor components and protocols used in Model-Driven Development. The editor components and protocols are used internally by the tools, and usually not by the MDD users.

2.2.1 Tools for Model-Driven Development

This section will describe the most relevant tools when doing Model-Driven Development (MDD) in the Eclipse Modeling Framework ecosystem.

2.2.1.1 Eclipse Modeling Framework and Ecore

What it is The Eclipse Modeling Framework (EMF) project provides tools for modeling and code generation. EMF consists of three main parts: an *EMF core*, *EMF.edit* and *EMF.codegen*. The core has a metamodel, *Ecore*, and a java runtime with support for change notifications, reflective Application Programming Interface (API) and mechanisms for serializing to the XMI format. EMF.edit contains components for creating model editors, like labels providers and a command framework for undo/redo. The EMF.codegen has code generation facilities which are controlled through a `genmodel` file.[17]

Using it The framework is provided as plugins to the Eclipse IDE. The modeler and user needs to have java installed, and it runs as a desktop application.

The Ecore metamodel Ecore is the metamodel used in the Eclipse Modeling Framework. It is an object oriented model, with concepts like *EClass* and *EAttribute*. The metamodel for Ecore is also Ecore, which means it is defined using itself.[1]

Ecore and standards Ecore is close to the Essential Meta Object Facility (EMOF) specification[2], but is based on MOF Core v1.[3] The EMOF specification is created by Object Management Group (OMG). The result is that Ecore is an alternative to UML Class Diagrams and compatible with other models based on MOF. Hence, an Ecore model could be visualized in a diagram using mostly the same notations as in UML Class Diagrams.

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). XMI is based on XML, and is another standard from OMG. Models created in Ecore can also be serialized to XMI, as can their model instances. Other formats for serialization of Ecore exist, like the more web-friendly JSON format.

Code generation with genmodel The genmodel is a model which augments an existing Ecore model. The serialization format is XMI. Ecore and the genmodel are both created with java code in mind: Ecore maps close to java concepts and datatypes, the genmodel has options for java code generation. The genmodel can also specify which model-to-text generation templates to use. By default it will create an Eclipse IDE plugin, but the target platform can also be Rich Client Platform (RCP)[4], Remote Application Platform (RAP)[5] or Google Web Toolkit (GWT).

2.2.1.2 Xtext

What it is Eclipse Xtext is a framework for creating new programming languages and *domain specific languages*. A language specified with Xtext will get its own Ecore model, which the framework will parse the language into. This means a user of a Domain-specific language (DSL) can write text, and the framework puts specific values of that text into an Ecore model, based on language keywords in the text. A language developer can then modify the user data by programming towards this Ecore model instead of the raw user text. [19]

[1] The Ecore metamodel is available at https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore/model/Ecore.ecore.

[2] https://www.omg.org/spec/MOF/2.4.1/PDF

[3] Ecore implemented MOF, but excluded what it deemed nonessential. Then MOF was inspired by Ecore and split into EMOF and CMOF. [18] They influenced each other. Ecore was an implementation of EMOF before EMOF existed.

[4] This is like the IDE, but without other bundles/plugins.

[5] A JFace renderer for the web

Using it A developer creates a language specification as text, and generates parser, linker, compiler, typechecker, an Ecore model, and a Language Server Protocol (LSP) server. [20] Then, the developer can write any transformation and usage of this Ecore model they want, such as model-to-text transformations or as parameters/data in a software system.

2.2.1.3 Sirius

What it is Eclipse Sirius is a framework for creating domain-specific modeling workbenches. [21] It is an Eclipse IDE addon, which focuses on creating visual editors like interactive diagrams. An illustration of Sirius' internal architecture is shown in Figure 2.1.
Sirius can create editors for an existing Ecore model, but does not rely on code generation. [21] The specification for the workbench is stored in a `.odesign` file. A workbench can have multiple representations of a model, and the representations are stored in a `.aird` file. Both of these files are XMI serializations.

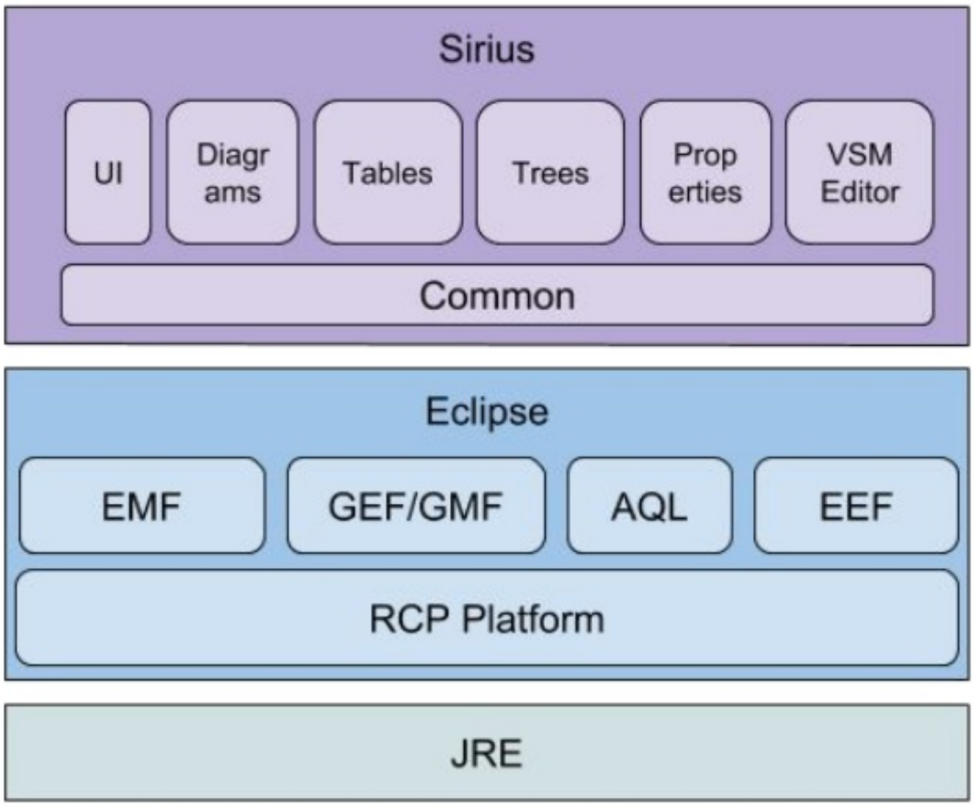


Figure 2.1: The diagram shows the different components in Eclipse Sirius. The components on the top depend on the components below. [22, p. 36].

Using it A developer creates Sirius specifications using the Eclipse IDE. After a workbench is designed, the developer produces an Eclipse *plugin* to distribute the graphical modeling tool. An end-user which edits model instances would install the plugin into their Eclipse IDE.

2.2.1.4 Sirius Web

What it is Sirius Web is an ongoing effort[6] to move the end-user editors of Sirius to the web. The goal is to allow end-users to model with their browsers, instead of the Eclipse IDE. Sirius Web will be open source. An illustration of Sirius Web's internal architecture is shown in Figure 2.2. Note that this is a work in progress, so the actual architecture may deviate once Sirius Web is released. The end-user will only see the *Sirius Web Frontend* (or *Sirius RCP*, which is the old Sirius in Section 2.2.1.3).
Sirius Web does not use Theia, but its representations *can* be shown in Theia. [23]

Using it A developer creates Sirius specifications in *Eclipse Sirius* or *Obeo Designer* as before. Then they deploy the specifications as a Sirius Web editor, instead of an Eclipse plugin.

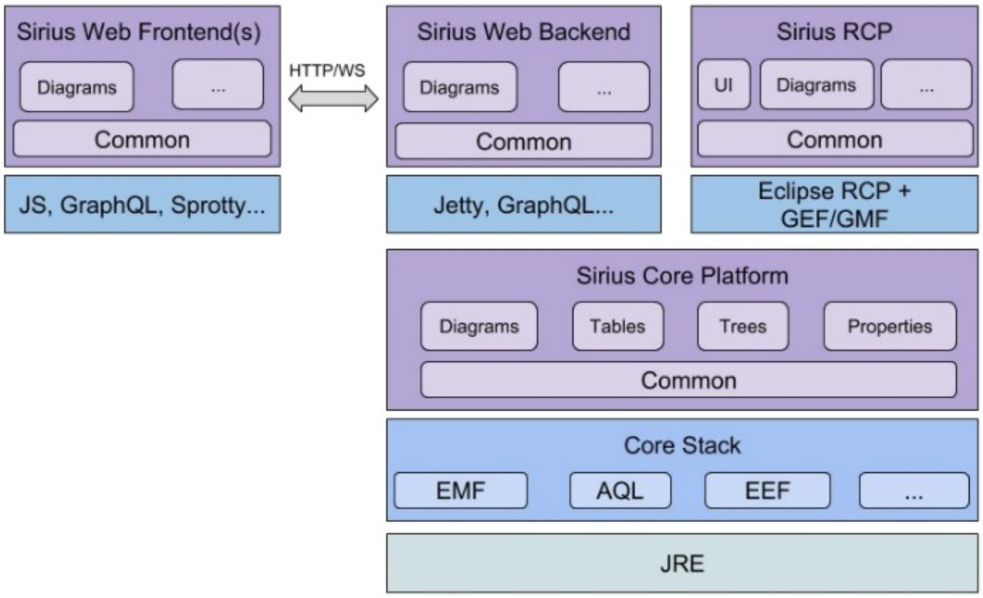
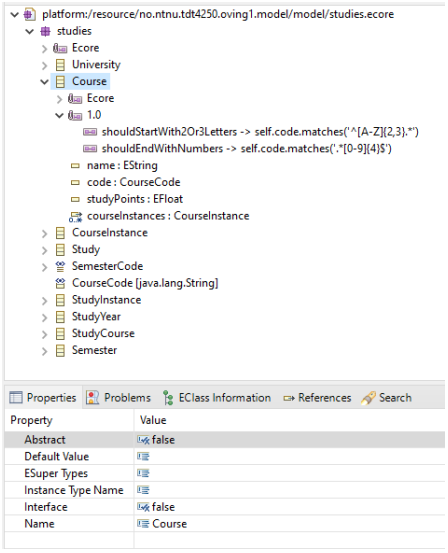
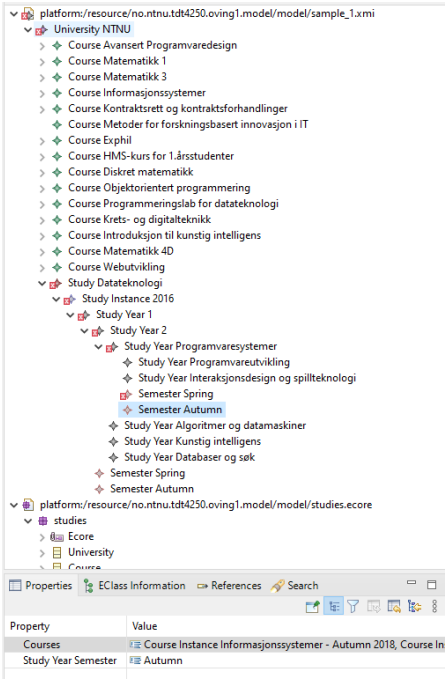


Figure 2.2: The diagram shows the different components that will be in Sirius Web. Components on the top depend on components below. [22, p. 37]

[6] As of 2020. It is supposed to be ready near the start of 2021.



(a) A model opened in the editor.



(b) A *dynamic instance* (XML Metadata Interchange (XMI)) opened in the editor.

Figure 2.3: Screenshots of the Sample Reflective Ecore Model Editor in Eclipse IDE.

2.2.2 Ecore Editors

This section describes some of the existing editors for Ecore models. Most of them are provided in the Eclipse IDE. They can be used to draw inspiration for new cloud editors, and suggest functionalities and requirements that are needed.

2.2.2.1 Sample Reflective Ecore Model Editor

A master-detail tree editor in Eclipse IDE. It uses the *reflective API* of Ecore to convert models into trees. It is open source[7] The source code lives in the org.eclipse.emf.ecore.editor package, (indicating that it was generated by a `genmodel`[8]).
A screenshot of the editor can be seen in Figure 2.3. It supports both models (Figure 2.3a) and model instances (Figure 2.3b).
A relevant piece of the source code is the `ReflectiveItemProvider`[9] from `org.eclipse.emf.edit`. This code converts the model instances to text labels (line 390) and provides their icons (line 380), for display in a tree.

[7] Sample Reflective editor source: https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.ecore.editor.
[8] Some source code also has `@generated` annotations, confirming this.
[9] https://git.eclipse.org/c/emf/org.eclipse.emf.git/tree/plugins/org.eclipse.emf.edit/src/org/eclipse/emf/edit/provider/Refle

2.2.2.2 EMF Forms Ecore Editor

The *EMF Forms Ecore Editor* is based on *EMF Forms*[10], a library to create form editors for Ecore models. It runs in Eclipse IDE and is open source[11]. The editor supports both `.ecore`, `.xmi` and `.genmodel` files. It has a generic editor for all XMI files[12], and specialized editors for ecore[13] and genmodel. [24]
A screenshot can be seen in Figure 2.4.

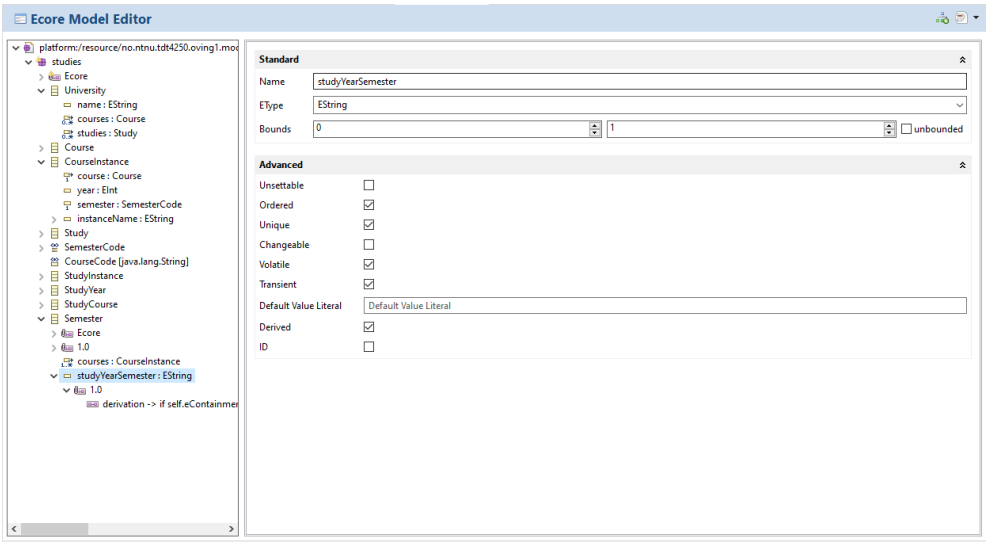


Figure 2.4: A screenshot of a model in the EMF Forms based Ecore Editor.

2.2.2.3 Ecore Tools

The *Ecore Tools* editor is based on Sirius (see Section 2.2.1.3). It displays diagrams as UML Class Diagrams in the Eclipse IDE. The editor is open source[14]
A screenshot of the Ecore Tools editor is shown in Figure 2.5.

2.2.2.4 EMF.Cloud — ecore-glsp

The *ecore-glsp* is an open source[15] Theia *Extension* (Section 5.2.2.3) originally created by EclipseSource. The project was donated to Eclipse EMF.Cloud. It renders Ecore models as diagrams, similar to Ecore Tools.
The extension is made using Graphical Language Server Platform (GLSP) (Section 2.2.4.3) and Sprotty (Section 2.2.3.1). When using the plugin, diagram layouts are persisted to a separate `.enotation` XMI file. [25]

[10] https://eclipsesource.com/blogs/tutorials/emf-forms-editors/
[11] EMF Forms source: https://git.eclipse.org/c/emfclient/org.eclipse.emf.ecp.core.git/tree/bundles/org.eclipse.emfforms.editor
[12] *Generic XMI Editor* in Eclipse IDE.
[13] *Ecore Editor* in Eclipse IDE.
[14] Ecore Tools source: https://git.eclipse.org/r/plugins/gitiles/ecoretools/org.eclipse.ecoretools/.
[15] ecore-glsp source: https://github.com/eclipse-emfcloud/ecore-glsp.

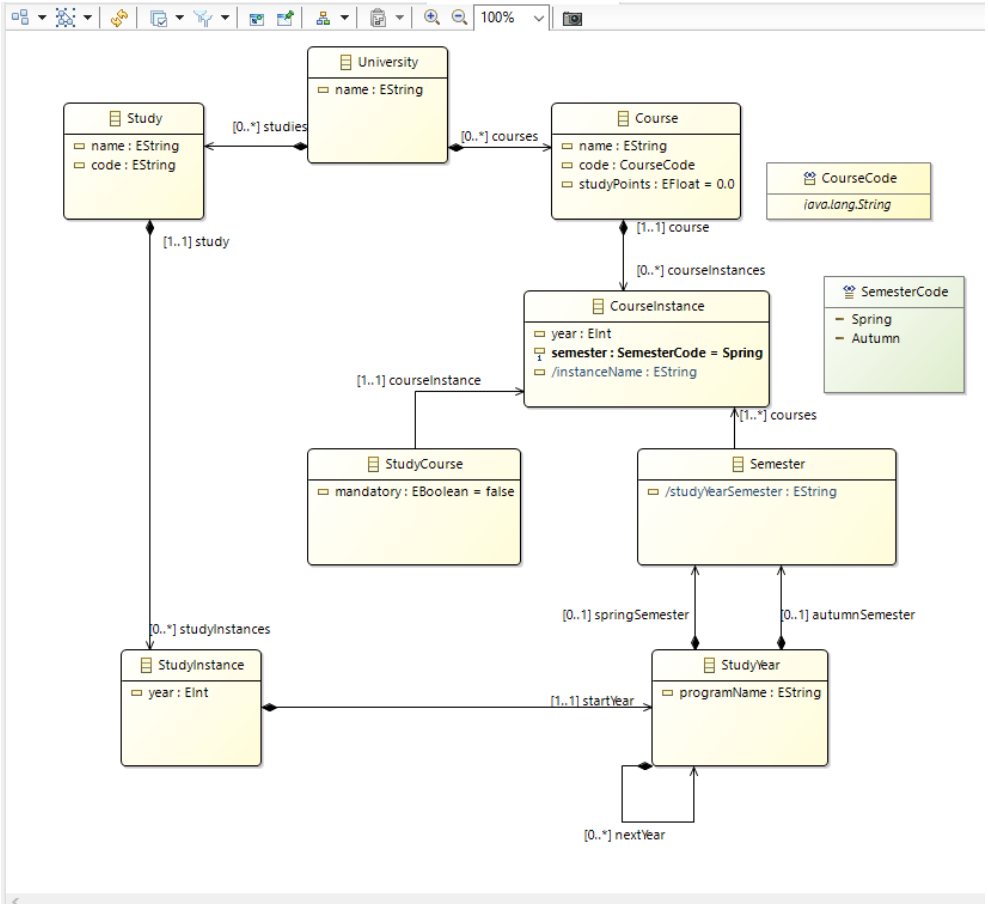


Figure 2.5: Ecore Tools in Eclipse IDE with a model opened.

Although it only supports Theia now, a developer using the GLSP integration for VSCode could possibly adapt it to a VSCode extension (see Section 5.2.2.1). A VSCode integration library is created by EclipseSource[16].
The scope of the ecore-glsp library seems to extend beyond just diagrams. The Github issues[17][18][19] indicate that the developers want to add a tree editor, property view, validation and a model server. The issue for a tree editor suggest using the *Theia Tree Editor* component (see Section 2.2.3.2). However, if they add the Theia Tree Editor, the library *may* lose compatibility with other IDEs than Theia, such as VSCode. Alternatively, the features will only be available in the Theia client.

2.2.3 Editor Extension Components

This section will describe the most relevant components, libraries or widgets for EMF in the cloud, which can be used to create editor extensions.
Some of the libraries belong to *EMF.Cloud*. That is an Eclipse Foundation project umbrella for EMF tools for the cloud[20].

2.2.3.1 Sprotty

Eclipse Sprotty is an open source[21] library to render diagrams in web browsers. It uses Typescript, CSS and svg. Sprotty can animate diagram changes. The architecture is made with LSP in mind, and supports having diagram data sent from a backend. The library is configurable using dependency injection, and allows for adding custom nodes, edges and behaviors. It also provides "glue code" for easy integration with LSP, Theia and Eclipse Layout Kernel. [26] An example of a sprotty diagram is shown in Figure 2.6.
The library is mainly developed by TypeFox and EclipseSource, and managed by the Eclipse Foundation. [27]

2.2.3.2 EMF.Cloud — Theia Tree Editor

The *Theia Tree Editor* is an open source[22] javascript framework for building tree editors in Theia. [29] The framework is targeted at Theia Extensions (see Section 5.2.2.3), and can not probably not be used in VSCode[23]. The main reason is the dependence on tree structures defined in the Theia IDE project itself (see

[16] VSCode glsp integration source: https://github.com/eclipsesource/glsp-vscode-integration.
[17] Tree editor issue: https://github.com/eclipse-emfcloud/ecore-glsp/issues/45.
[18] Model server issue: https://github.com/eclipse-emfcloud/ecore-glsp/issues/28.
[19] Property view issue: https://github.com/eclipse-emfcloud/ecore-glsp/issues/53.
[20] Eclipse EMF.cloud homepage: https://projects.eclipse.org/projects/ecd.emfcloud.
[21] Sprotty source: https://github.com/eclipse/sprotty.
[22] Theia Tree Editor source: https://github.com/eclipse-emfcloud/theia-tree-editor/tree/master/theia-tree-editor.
[23] It *may* be possible, if the VSCode extension pulls in the Theia `core`, however there could be dependencies to other Theia APIs as well.

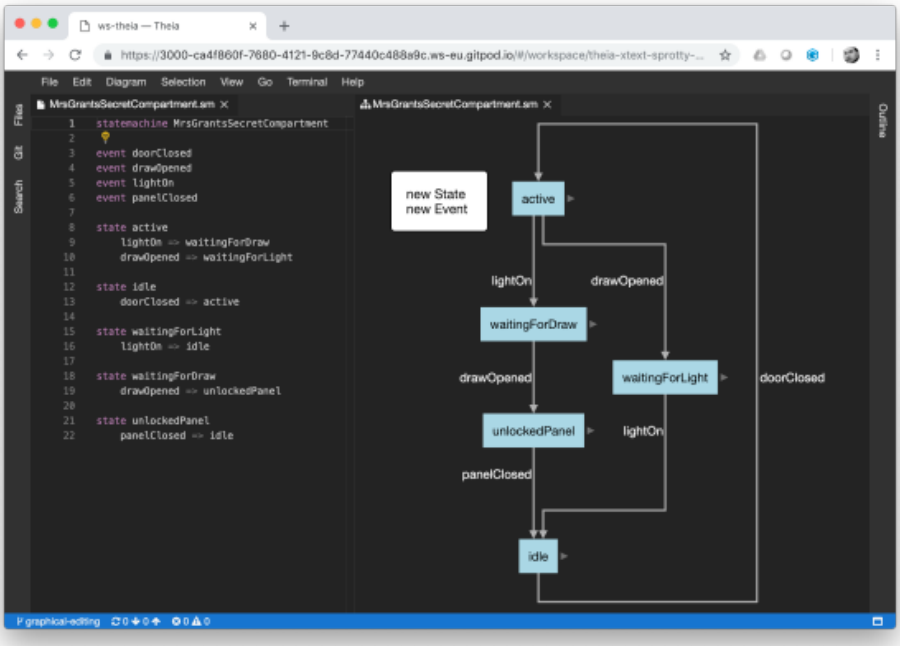


Figure 2.6: This is a screenshot of Theia with a Sprotty diagram on the right side. [28]

Figure 2.7). To render the property form, it uses the *JSON-Forms* library (Section 2.2.3.3). Additionally, it uses React.

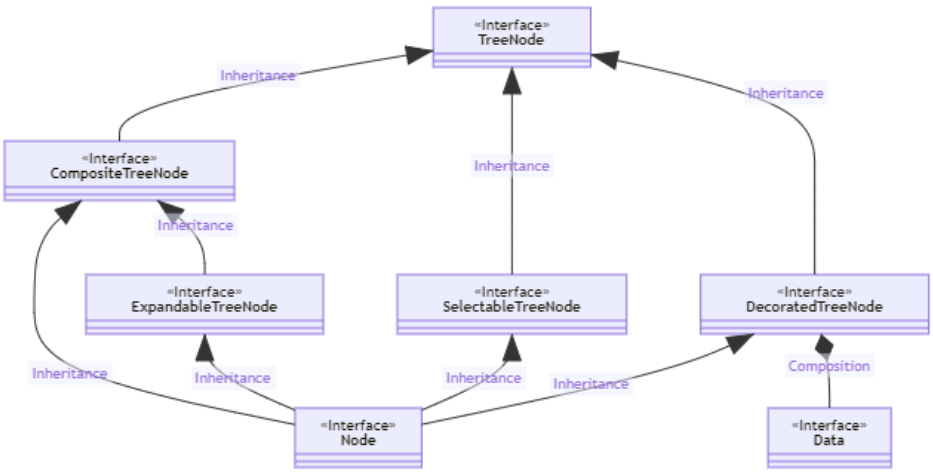


Figure 2.7: The Node class in Theia Tree Editor inherits multiple interfaces, all from the Theia `core` extension. Only `Node` resides in the Theia Tree Editor library itself.

2.2.3.3 JSON-Forms

JSON-Forms is an open source[24] javascript library by EclipseSource for creating HTML form editors. A form is defined by two key elements: a *data/JSON schema* and a *UI schema*. The JSON schema defines the data types and validations for input. The User Interface (UI) schema defines the visible *form controls* (text fields, groups, labels) and binds them to a object property in the JSON schema. [30] For an example of how a form can look like, see Figure 2.8.
A visual editor for JSON-Forms schemas exist at https://github.com/eclipsesource/jsonforms-editor.

2.2.3.4 EMF.Cloud — Model Server

The *Model Server* is an open source[25] java library and standalone web server. It can discover and read Ecore and XMI files in a workspace folder. Then the models can be provided as JSON over a REST API, and changes to the models can be provided via WebSocket subscriptions. [31]
A copy of the standalone `.jar` can be downloaded from a Maven Repository at https://oss.sonatype.org/content/repositories/snapshots/org/eclipse/emfcloud/modelserver/org.eclipse.
SNAPSHOT/. (Note that the standalone seems to embed a coffee model example.

[24] JSON-Forms source: https://github.com/eclipsesource/jsonforms.
[25] Model Server source: https://github.com/eclipse-emfcloud/emfcloud-modelserver.

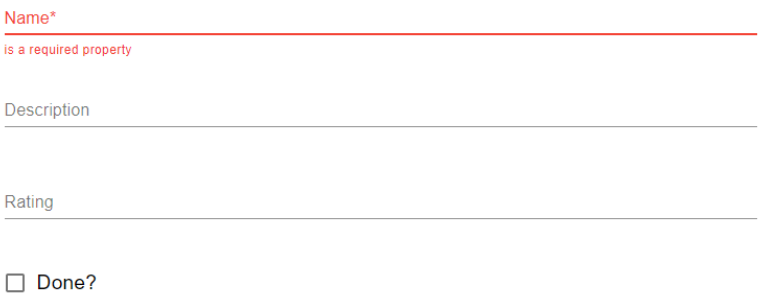


Figure 2.8: An example of a form generated by JSON-Forms. It is displayed in a web browser.

However, the implementation works with all `EObjects` and reads the Ecore metamodel, so the use of the coffee model is not clear). The API provided can also serve *JSON-Schema* and *UI Schema*, which JSON-Forms in Section 2.2.3.3 can consume. The Model Server could be used in an extension or plugin (Section 5.2.2) as a backend child process, provided that the host operating system has a java runtime available.

2.2.3.5 EMF.Cloud — emfjson-jackson

Originally, *emfjson* was a project umbrella for EMF in the cloud. One part of emfjson, the JSON serializer called *emfjson-jackson* was donated to EMF.Cloud. This is an open source[26] java library which uses the Jackson JSON library.[32]

2.2.4 Protocols

This section will describe some of the protocols used by the IDEs in Section 5.2.1 and extensions in Section 2.2.3.

2.2.4.1 JSON-RPC

Description *JSON-RPC* is a stateless Remote Procedure Call (RPC) protocol. That means it allows one process to call *procedures* (methods, subroutines, functions) in another process. The format for serializing the calls' data is JSON. A key design goal is to be a simple protocol. [33]
The protocol does not limit the *transport mechanism* used, so JSON-RPC could be sent inside the same process, across sockets, HTTP or any other form of message passing transport. [33]

Protocol The main definitions are data structures and error codes. JSON-RPC defines a `Request` and `Response` object. [33]

[26] Emfjson-jackson source: https://github.com/eclipse-emfcloud/emfjson-jackson.

The Request must have an `id`, `method` name, `params` data structure and `jsonrpc` version field. A Request without an `id` is a `Notification`, and will not get a Response back. [33]
The Response must have an `id`, `result` or `error` data structure, and a `jsonrpc` version field. [33]

2.2.4.2 Language Server Protocol (LSP)

Goal An IDE often has to support many programming languages. Most of the languages support some common features, such as autocomplete, validation, definitions, references, renaming, selection etc.. The LSP tries to separate language editor clients from language servers. The clients are kept generic, while the language servers know the details for a programming language. [34] This means a IDE developer only needs to create *one* editor, with generic LSP support. And a language developer only needs to create a language server. If the language server adheres to the LSP protocol, any LSP client will automatically support it. This is illustrated in Figure 2.9.

Figure 2.9: The benefits of using LSP. There is no need for re-creating language support for every possible combination of editor and language. [35]

Details The protocol is based on a *Base protocol*, which has a `key-value` *header* and a *content* with JSON-RPC (see Figure 2.10).

The Base protocol The header fields conform to the HTTP specification[27] with regards to structure and formatting.
The Base protocol defines tree different message types that are sent in the JSON-RPC content: *Request*, *Response* and *Notification*. There are also two key rules: every Request must be answered with a Response, and Notification does not need

[27] RFC-7230 at https://tools.ietf.org/html/rfc7230#section-3.2.

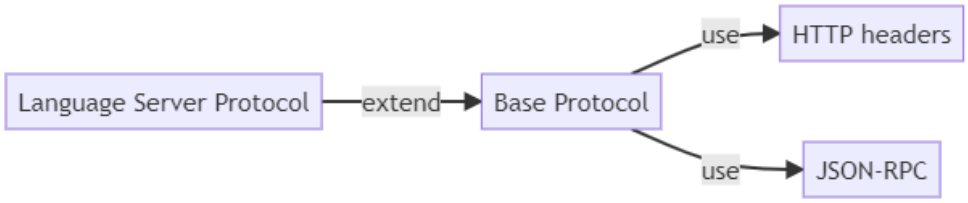


Figure 2.10: The LSP protocol extends a Base protocol that uses HTTP headers and JSON-RPC content.

a Response. The Base protocol provides a list of common error codes, extending the default error codes that JSON-RPC provides. [36]
The Base protocol defines a rule for Request and Notification methods as well: a method starting with `$/` is protocol implementation specific and therefore optional, meaning a client or server can skip implementing it if the method is not suitable[28]. [36]
Lastly, the Base protocol defines two Notifications: `$/progress` and `$/cancelRequest`. [36]

The Language Server Protocol The actual *Language Server Protocol* itself is a large collection of JSON-RPC Request, Response and Notification messages sent over the Base protocol. The protocol assumes that *one* language server is used for *one* language client. [36] This means a language server can not be shared between multiple clients.
The protocol is tailored for textual languages and text documents. It specifies a collection of data structures that should be used. A central data structure is the `TextDocument`. Inside a `TextDocument`, there can be `Locations` in the text, and `Ranges` of text between two `Locations`. Changing a `TextDocument` is done by constructing a `TextDocumentEdit`, which holds a list of text replacements in `TextEdits`, and a pointer to a `TextDocument` version using a `VersionedTextDocumentIdentifier`. A `TextEdit` has `Range` to edit and a string with the new text. [36]
An illustration of the data structures are shown in Figure 2.11. Note that this is only a small subset of the data structures defined in LSP[29].

2.2.4.3 Graphical Language Server Platform (GLSP)

Goal The *Eclipse GLSP* tries to create an analogous protocol to LSP, but for graphical diagram editors. It wants to be a solution for reusable *graphical* language servers as well as generic clients. [37] An illustration of GLSP is shown in Figure 2.12.

[28] The example given is cancellation of asynchronous tasks in a server that is synchronous.
[29] Most of the supported features are described at https://code.visualstudio.com/api/language-extensions/language-server-extension-guide#additional-language-server-features.

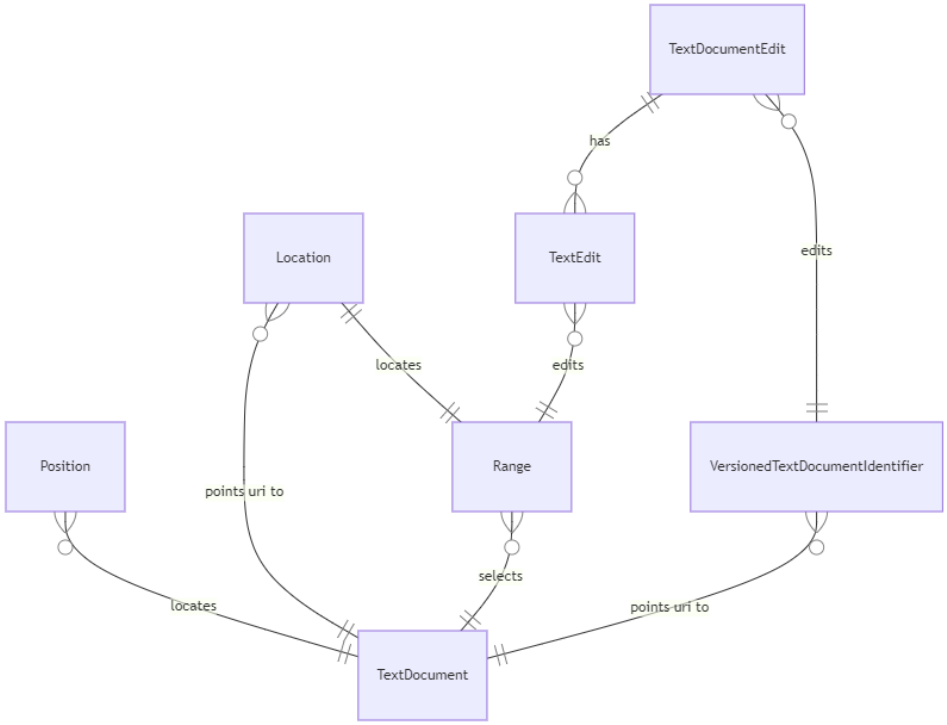


Figure 2.11: Central data structures in LSP.

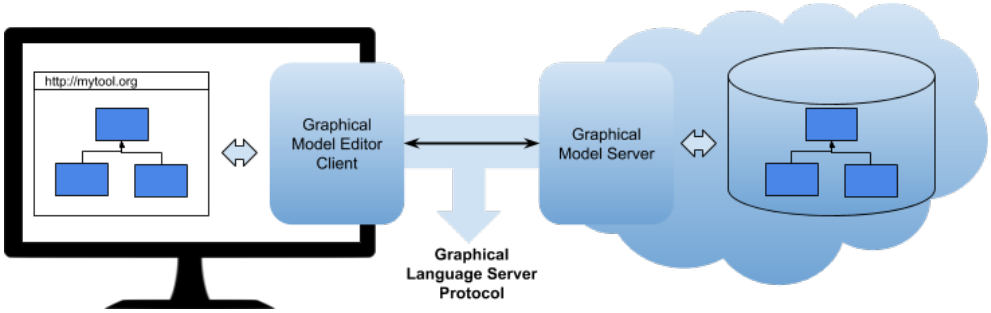


Figure 2.12: An overview of the GLSP. [38]

Details Being a platform, it aims a bit wider than the LSP protocol by providing several implementations. The platform has client and server implementations that speak the GLSP protocol, and integrations into some IDEs. The client deals with rendering and user feedback, for example during drag-and-drop. The server handles the diagram logics, validation and layouting. [37]
The client uses technologies like Sprotty (Section 2.2.3.1), Typescript and svg. The server uses java, and has an existing integration with EMF. [38]

The protocol The protocol is based around client-server communication over JSON-RPC. As with LSP, one server is only responsible for one client. The messages sent are inspired by the protocol used by Sprotty, but extended with user editing capabilities. [39]
A core concept of the protocol is *Action*, which is sent in a *ActionMessage*. All communication happends by sending Actions. Actions are identified by a `kind`, and can be mapped to a concrete Action implementation by this kind. [39] There are four main types of action: *RequestAction*, *ResponseAction*, *RejectAction* and *Operation*. A RequestAction must be answered by a ResponseAction or rejected by a RejectAction. Operations are requests from the client to modify the model. Modifications are always done on the server. [39]
The protocol defines a large collection of Actions and Operations, to cover all needs of a graphical editor[30].

2.2.5 Integrated Development Environments

A short list of relevant IDEs is Eclipse IDE, VSCode and Theia. Eclipse IDE is where modeling has happened traditionally. VSCode is focused on editing text, and not a contender in the MDD space. However, it is extremely popular. [40] Theia is an new editor based on VSCode but adapted for the cloud.

[30] A list of GLSP features are listed at: https://github.com/eclipse-glsp/glsp#features.

Chapter 3

Reviewed Research Questions

The research questions were presented in Section 1.3. Some of them can be answered, at least to some extent, by the background material in Chapter 2.

3.1 The Old Research Questions

Modernizing MDD First, lets look at RQ 1:

> How can we modernize Model-Driven Development Frameworks to appeal to the next generation of software developers, using recent developments in cloud IDEs?

It appears this modernization is already in progess. With the introduction of *Sirius Web* (Section 2.2.1.4) and *ecore-glsp* (Section 2.2.2.4), Ecore editors in cloud based IDEs are coming. In addition, Ecore code generation can already target web browsers using *ecore.js* or *crossecore-typescript*.
The use of more a more popular serialization format, JSON instead of XML/XMI, is being used to *some* extent. Most of the current efforts are still serializing to disk as XMI, using only JSON inside protocols for data transfer. This *could* be an advantage, however, to keep the interoperability between different tools. Most of the developers' work is not done directly in these files, but instead with graphical interfaces. Therefore, this is perhaps not a big issue.
The biggest blocker right now is just implementation: not all features are supported yet. Another concern is targeting only Theia, as is the case for ecore-glsp with its use of Theia Extension (Section 5.2.2.3). This prevents usage in VSCode[1], and also in Gitpod. It could possibly be used in Che, by customizing the IDE used[2].

Essential tools Next up is RQ 1.1:

[1] Although not a cloud editor (yet? https://github.com/features/codespaces), it is very popular. [40]

[2] This may need help from a system administrator. Alternatively, a user can compile their own Docker image with a modified Theia instance, and specify it in a devfile. [41]

> What tools are essential to use Model-Driven Development in the cloud?

The model editor seems like the obvious answer. Two different types of model editor were found. There are tree and property editors, and graphical diagram editors. These should support both the general Ecore model and model instances. Some editors also support the genmodel.
In addition, there is validation, comparison and code generation. Of these, code generation seems most essential, and validation second. The ongoing development in Section 2.2.2 on ecore-glsp also seems to share this view: diagram editor first, then code generation, then validation last[3].
Even though personal experience with EMF relied mainly on a tree editor, little focus has been on implementing Ecore tree editors in the cloud yet. There is the generic Theia Tree Editor (Section 2.2.3.2), but it has not been applied to Ecore yet. A generic tree editor could work with models, model instances and the genmodel, making it a "one solution to rule them all", compared to diagram editors. This is what Eclipse IDE has already, like the Sample Reflective Ecore Model Editor and the EMF Forms based editor.
Validation and code generation tools could be embedded in VSCode extensions as executables, and ran as VSCode commands or from a tree editor. They do not require special user interfaces.

A good tool Perhaps more on the side of *usability*, or *minimum viable product* requirements, is RQ 1.2:

[3]Ecore-glsp already supports genmodel and EMF Codegen: https://github.com/eclipse-emfcloud/ecore-glsp/issues/10.

What are the characteristics of a good cloud based tool for Model-Driven Development?

Its characteristics are about its software architecture and functional requirements. *What does it do*? And *how* does the tool do something?
First off, flexibility seems to be key. Flexibility and extensibility in the protocols used, and configurability with regards to details. It is hard to define everything up front, so some leeway for tool developers is left in. Custom protocol messages; optional functionality. Using dependency injection in order to create an extendible system seems popular in this cohort of developers as well. For example Theia, Sprotty and GLSP all do this.
Second is configuration. It is hard to predict everything, so exposing details as configuration options allows solutions to arise for unknown situations. An example is *java*: a lot of the EMF ecosystem exists as java code, and could be embedded as `.jar` files, but *where is java installed* on the workspace operating system, and which java version? With new environments like Gitpod and Che, some old assumptions do not hold[4].
Third is to augment models instead of extending their serialization format. Most of the tools store some extra data, like layout or options. These pieces of information are stored in files separate from the `.ecore` model, such as `.aird`, `.json` and `.genmodel`. Common is the usage of XMI, but some use JSON.
Fourth is a client-server separation, both because Theia is distributed, but also to have reusable (generic) clients and reusable servers like LSP. Runtime dependencies may dictate how code is organized, for extensions that work across browsers, Electron and NodeJS.

Reusing tools and architectures The last question is RQ 1.3:

> How can existing modeling tools and software architectures be reused in the new cloud based modeling tool?

It seems existing tools that can be run standalone as servers or command line tools can be reused. For example, LSP shows this, by letting language servers execute what they want internally, like existing compilers. Bridging the gap between old programs and new could be done by exposing them over an API, such as REST or JSON-RPC. The Model Server (Section 2.2.3.4) does this, and hides away java and Ecore model parsing behind a REST API.

3.2 Planned Contribution

RQ2. *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?*

— **RQ2.1.** *How can java applications be used in a VSCode extension?*

[4]Example: you assume `java` can be invoked from the OS Path, but what if it is ran as a Docker container instead with `docker run`?

— **RQ2.2.** *How should the editor cooperate with multiple other tools that change the same underlying model?*

— **RQ2.3.** *What data is required for displaying a model as a tree?*

— **RQ2.4.** *How can a tree editor component be generic enough to support arbitrary user actions?*

— **RQ2.5.** *How can the user interface know the model well enough so the user is constrained from creating invalid Ecore trees?*

Chapter 4

Method

How to answer the sub-research questions To answer the sub-research questions RQ 2.1 to RQ 2.5, a design and creation approach will be used. Developing prototypes and testing them in the real environment can quickly remove doubts about feasibility, as long as the prototype works. (In case a prototype fails to demonstrate a working approach, a new approach will have to be designed and tested in a new prototype. If no new design can be made, then the background material needs further research.)

How to answer the main research question To answer the main question, RQ 2, an analysis of existing tools will form the requirements for the editor. The findings from RQ 2.1 to RQ 2.5 will influence the solution, and existing architectures and protocols will also be used as inspiration.
The answer to RQ 2 will be a design, like diagrams and requirements. It will not be a full implemented software solution, because of the limited scope of this pre-master's thesis specialization project. An implementation will instead be explored in the following master's thesis.

Data generation method To find data that can answer the research questions, I will use documentation on the internet from open source software related to modeling. Discovering the related software and documentation will be done by:

- exploring the Eclipse Foundation's project umbrellas for related software
- inspecting the source code of existing Eclipse IDE Ecore editors (discussed in Section 2.2.2)
- searching with internet search engines (Google, GitHub) for EMF related libraries
- viewing presentations from *EclipseCon 2020*[1] about EMF modeling, Sirius Web (Section 2.2.1.4), GLSP

[1] This is a developer conference by Eclipse Foundation. The 2020 conference has an agenda that aligns very well with modeling in the cloud.

- ask questions during[2] or after EclipseCon to presenters or key personalities
- inspecting GitHub organizations that contribute modeling projects, to see if they offer other related projects
- inspecting developer documentation, guides and wiki pages for software, to find related documents
- inspect software dependencies[3] in discovered libraries, to find other related libraries

Data analysis The results will be analyzed qualitative to give a clear view of the state of the art, current developments, possibilities for new development and insight into the architectures and protocols already in use. Together, this will form the basis of a new tool's requirements and design.

[2]The conference happened during this thesis, and was virtual, so this was possible.

[3]Usually listed in files like `pom.xml` and `package.json`

Chapter 5

Results

This section will discuss the contributions from this thesis. Then it dives into some of the data required for a tree editor. Next is a presentation of the prototypes and their results, with regards to the research questions. Finally, a list of requirements and an architecture for a Tree Editor is presented.

5.1 Contribution

This thesis will provide an overview of the developments in cloud IDEs with regards to modeling in EMF. An analysis of the data will show how existing tools and architectures can be applied in a new cloud IDE for Ecore modeling.
Additionally, a foundation for a *software product* will be presented. The foundations will be built further upon in a master's thesis. The foundations are a set of software requirements and an initial architecture for a cloud based Ecore Tree editor inside an existing cloud IDE.

5.2 Extensible Integrated Development Environments

This section will discuss the technologies and mechanisms found during data collection that are relevant for a solution.

5.2.1 Integrated Development Environments

This section will describe the most relevant Integrated Development Environments (IDEs) for Eclipse Modeling Framework and the cloud.

5.2.1.1 Visual Studio Code

What it is Visual Studio Code (VSCode) is a text editor and Integrated Development Environment (IDE) created by Microsoft. The VSCode editor is based on

the open source[1] software (OSS) called *VS Code Project*. [59] The VSCode editor has Microsoft-specific customizations, which makes it slightly different from the OSS project. [42] (This is analogous to the relationship between the open source Chromium and the proprietary Google Chrome.)

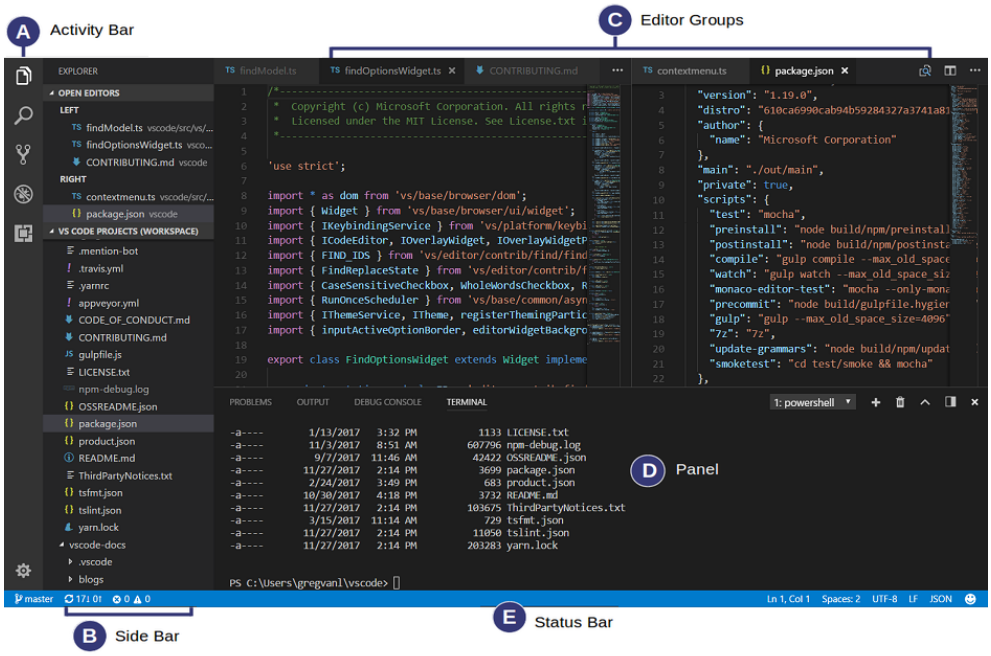


Figure 5.1: The user interface of Visual Studio Code, with annotations for the different components (*A-E*).[43]

Intended use Primarily an editor for textual programming languages. VSCode has no support for diagrams or binary file viewers (like 3D models) out of the box (except pictures like `png` etc.). By default, it supports `Javascript`, `Typescript` and `Markdown`; common technologies related to web development. VSCode can be extended to support other languages and editors, see Section 5.2.2.1.

Architecture VSCode runs as a desktop application, and uses Electron and javascript for its runtime. The architecture is centered on a *core* which loads *extensions*. The core is shown in Figure 5.3, and is split internally into *layers* with different responsibilies. Each layer is again organized based on the target runtime, like *browser* or *Electron*, as seen in Figure 5.2. The actual text editor in VSCode is called *Monaco*, and is available as a standalone package. [44]

[1] Source for VSCode is available at https://github.com/microsoft/vscode

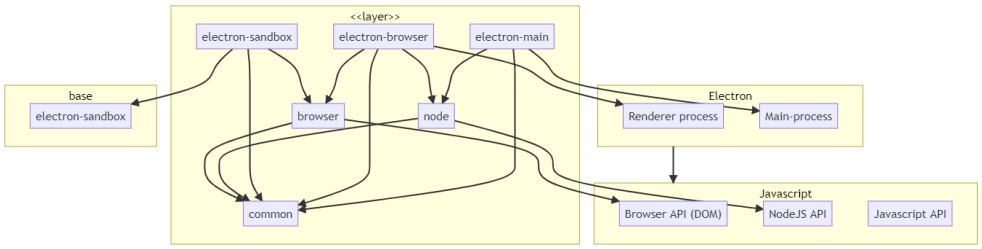


Figure 5.2: The internal architecture of each component in VSCode. The «`layer`» label represents each of the layers in Figure 5.3, like *base* or *platform*. Every layer is organized by runtime dependencies. [44]

Figure 5.3: The internal layers which the VSCode `core` is separated into. [44]

5.2.1.2 Theia

What it is Theia is an editor like VSCode, but designed from the start to work in web browsers. A screenshot is shown in Figure 5.4; notice how it is almost identical to VSCode. The project was originally started by TypeFox, and then donated to the Eclipse Foundation. It is free and fully open source[2]. Theia is based on some of the same components as VSCode, and uses Monaco as well. The main uses of Theia now is as an editor for workspace software like Eclipse Che and Gitpod. Theia supports VSCode extensions, but not all of the APIs in VSCode are implemented yet[3].

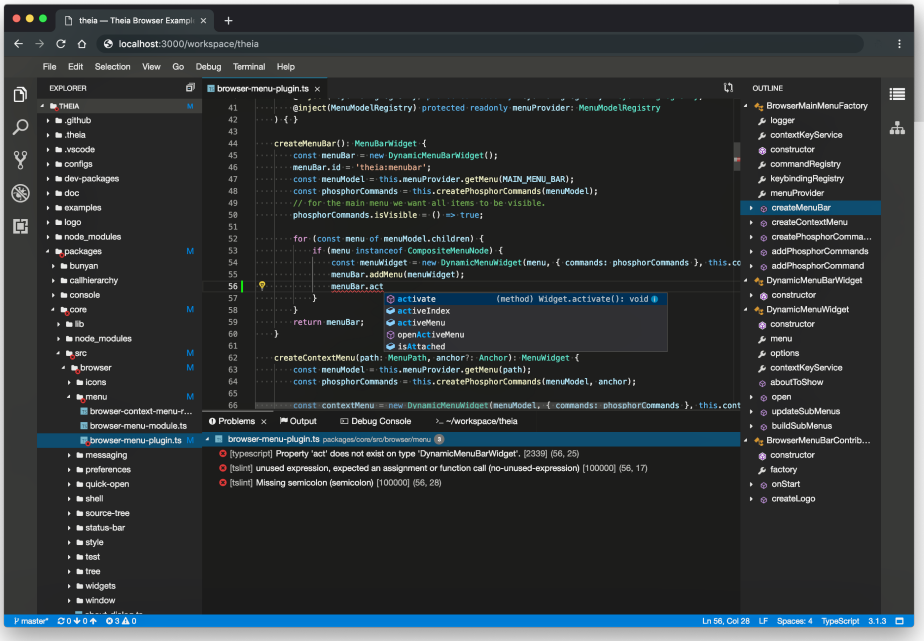


Figure 5.4: A screenshot of the Theia user interface. [45]

Intended use Theia was created to be a framework for creating custom and white-labeled IDEs and tools. Theia itself was not an IDE. [59]

Architecture — runtime The architecture of Theia is similar to VSCode. The biggest difference is in the runtime. Theia is split into a backend and frontend process. The backend *could* run remotely, or locally. The frontend is Electron or a browser. The backend and frontend communicate with JSON-RPC over WebSockets or REST/HTTP. Both the frontend and backend source codes are heavily archi-

[2]Source for Theia is available at https://github.com/eclipse-theia/theia/

[3]A table of API support is provided at https://che-incubator.github.io/vscode-theia-comparator/status.html.

tected around *dependency injection*. The backend runs an `ExpressJs` web server which serves code to the frontend (see Figure 5.5). [46]
The frontend can always assume it has `browser API`, but not `NodeJS/Electron` API. [46]
To support different programming languages, the backend talks over the Language Server Protocol (LSP) with different *language servers*. One backend process serves one workspace/user. Multiple clients can connect, but they will all share the same workspace. [47]

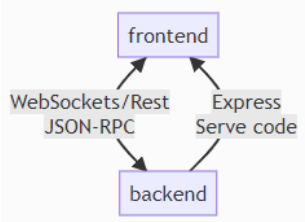


Figure 5.5: Theia serves the frontend with ExpressJs and communicates between the frontend and backend with JSON-RPC.

Architecture — components Theia uses a extension oriented design. This means that almost all functionality is provided as a *Theia Extension* (see Section 5.2.2.3). For example, the APIs for user-installable *Theia Plugins* (see Section 5.2.2.2) are provided as an extension. The *Monaco* editor from VSCode is also provided by an extension. The main components of Theia is shown in Figure 5.6. (Note that this diagram is from 2017 and could be outdated). The components are separated into frontend, backend and external (blue). External resources could be processes for language compilation, package managers, terminal shells etc..
Of particular interest is the `Extensions` component with `Inversify.js`. This is the dependency injection mechanism which loads other extensions into Theia. (Also note that the `Plugin` extension for VSCode extensions is missing in this diagram. That component was added later by RedHat[4].)

Architecture — code organization Every extension in Theia is organized based on the runtime dependencies. This is similar to how VSCode does it. See Figure 5.7. The main difference is `electron-sandbox` in VSCode and `electron-main` in Theia. Every Theia extension can also provide two entrypoints: `frontend` and `backend`. This allows extensions to provide different code for the different parts of Theia (see Section 5.2.2.3).

[4]RedHat chose to use Theia in Eclipse Che, and wanted plugins that could be installed by users during run-time. They opted for the VSCode extension mechanism for this.

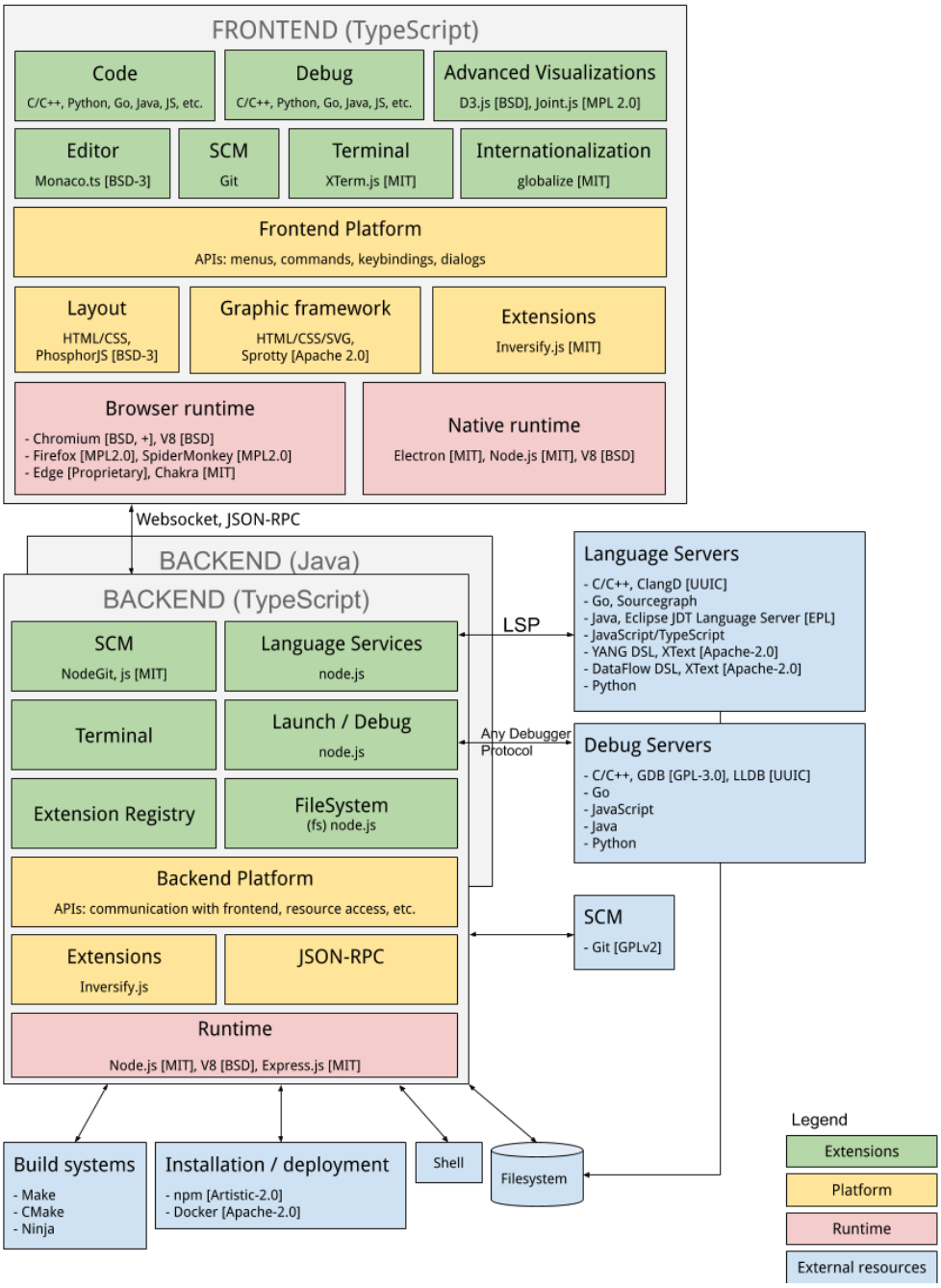


Figure 5.6: The main components in Theia. Some components have third party libraries, languages or runtimes listed under their component name, as examples. The figure is from a design document made in 2017, and could be outdated. (The author is unknown, but assumed to be a TypeFox employee. The document is linked to on the official Theia website). [47]

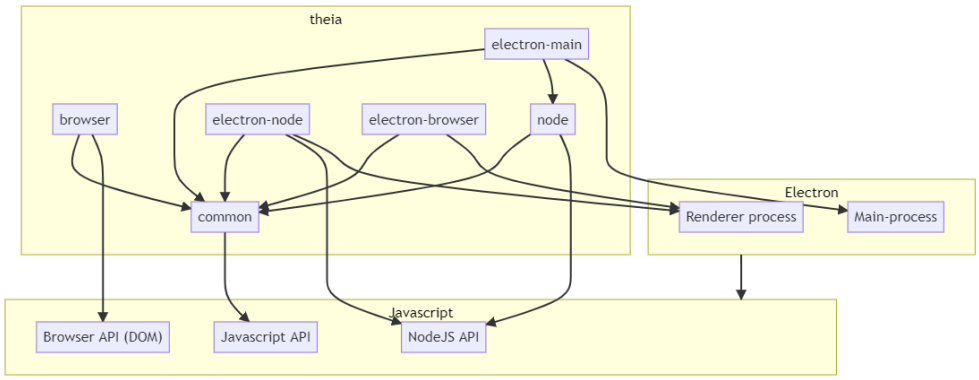


Figure 5.7: The code in Theia extensions are organized based on the dependency on runtime APIs. The box named `theia` represents an extension in Theia. [48]

5.2.1.3 Eclipse IDE

What it is One of the older IDEs. Originally desktop based, but some web frontends have been made (like Remote Application Platform (RAP)). Popular for java development and the main IDE used by EMF. It also serves as a framework for building user interfaces, as in Rich Client Platform (RCP) and generated model editors (from genmodel).

Intended use Eclipse IDE is intended to be used as a IDE and also framework for desktop based editors via RCP. It is very flexible, with support for addons/plugins that extend its capabilities.

Architecture The Eclipse IDE is based on a core platform that loads other plugins and features to extend itself. The core mechanism is OSGi and plugin manifests, along with Eclipse's extension points. OSGi will load plugins and use dependency injection to connect components together. The extension points are predefined areas of the IDE where plugins can add functionality. Plugins can also expose their own contribution points. The plugin manifest specifies where a plugin will contribute, like which extension points should be added to.

5.2.1.4 Coffee Editor IDE

What it is The *Coffee Editor IDE* is an example and reference implementation for a Theia-based IDE. It uses Theia as its core, but adds more functionality through extensions (see Section 5.2.2.3). The Coffee Editor IDE has a graphical editor based on GLSP, a form based editor using JSON-Forms (see Section 2.2.3.3), a tree editor using Theia Tree Editor (see Section 2.2.3.2), textual DSL using XText (see Section 2.2.1.2) and LSP (see Section 2.2.4.2). An illustration of the Coffee Editor IDE is shown in Figure 5.8.

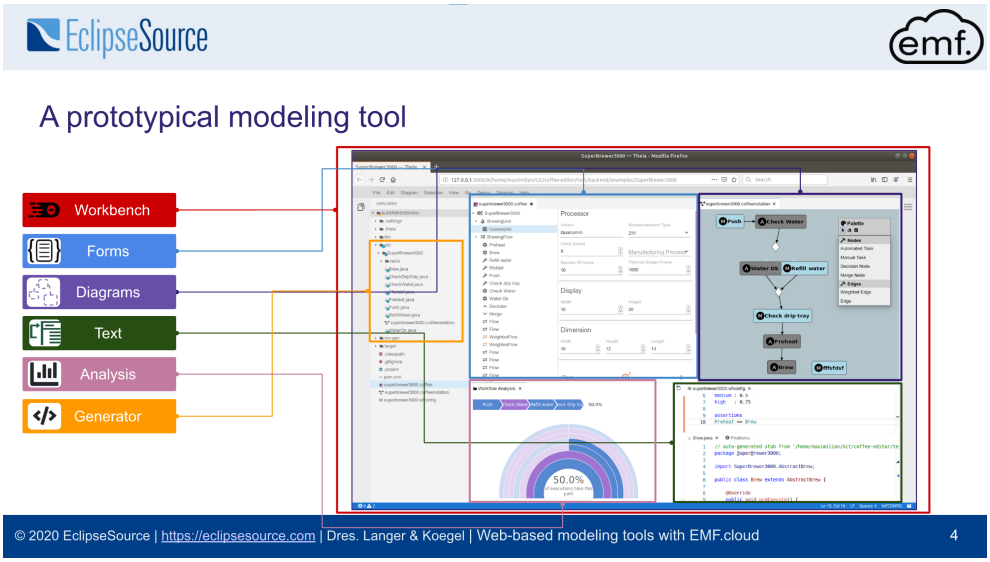


Figure 5.8: An overview of the Coffee Editor IDE components.[49, p. 4]

Intended use The main purpose is to show how existing editor extension components (Section 2.2.3) can be used in a real example. This is not intended to be used by any users.

Architecture Based on Theia with Theia Extensions and a java binary in the backend for "analysis"[5]. The different editors (graphical, form, tree, textual DSL) all communicate to a shared Model Server (Section 2.2.3.4). This is shown in Figure 5.9. How the GLSP and LSP frontend components get their view of the model is by talking through the Theia extension, to a *Connector* in the extension, which then talks to a GLSP/LSP server. This server talks to the shared Model Server (see Section 2.2.3.4) process. This is shown in Figure 5.10.

5.2.2 Editor Extension Mechanisms

This section will describe the extension mechanisms available for the editors in Section 5.2.1.

5.2.2.1 VSCode Extension

Description VSCode extensions are bundles of javascript code and resources in the `.vsix` file format. The extensions use a `package.json` file as a *manifest*, informing VSCode of the *contribution points* used. The manifest also indicates `activation events`, which is *when* an extension will be loaded (they are deactivated by default). [50] The contribution points can be commands, configuration options, programming language support, debuggers, custom editors etc.. [51] An

[5]It does no analysis, it is just a proof-of-concept that sends data and views in in a graph in Theia.

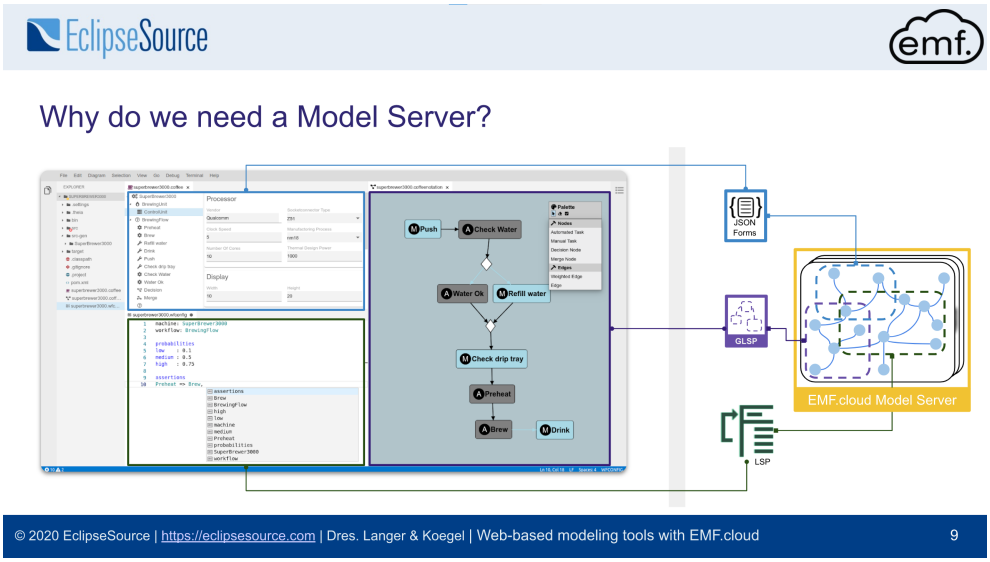
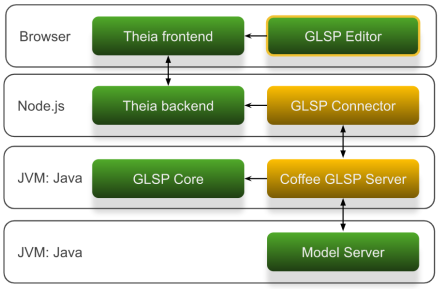
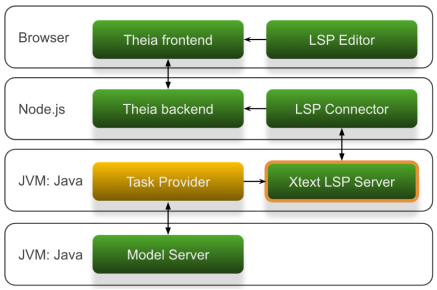


Figure 5.9: The Coffee Editor uses a Model Server to coordinate the different editors. [49, p. 9]



(a) GLSP Editor talks indirectly to the Model Server. [49, p. 17]



(b) LSP Editor talks indirectly to the Model Server. [49, p. 23]

Figure 5.10: How editors in the Coffee Editor IDE are synchronizing their model data by sharing a Model Server.

extension can also contribute views to the VSCode user interface, called the *workbench*, shown in Figure 5.11. [52] The extensions have a main entry point which is called by VSCode in a separate NodeJS process called the *Extension host*. [53] This process isolation improves the stability of VSCode in the case of crashes and hangs in an extension's code.[54]

Restrictions Extensions are not allowed to alter the VSCode user interface directly. [54] All changes must be done via the provided APIs. If one wants to create custom HTML elements, the *WebView* API should be used. This allows displaying HTML and running javascript. However, this WebView is also limited: it must communicate with VSCode using JSON messages, and relies on VSCode to save

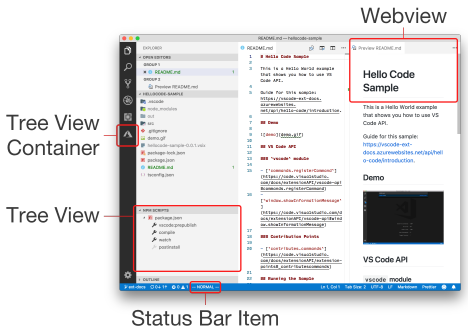


Figure 5.11: VSCode extensions can extend the different parts of the *workbench*. [52]

its state by handing sending it over. [55]

5.2.2.2 Theia Plugin

Description Theia *plugins* are exactly the same as VSCode extensions in Section 5.2.2.1. A Theia Plugin does not need to separate frontend and backend code. The VSCode APIs are transparently separated between the frontend and backend using JSON-RPC. [56] The support for VSCode extensions was added to Theia so Eclipse Che (which uses Theia) could allow users to install extensions without impacting the stability of the IDE. [57] A *Theia Extension* could crash and bring down the entire editor, in contrast to a *Theia Plugin* (VSCode extension) which would only display a warning. Because of licensing issues[6] with Microsoft and its VSCode extension marketplace, all Theia Plugins are hosted on a independent marketplace called *OpenVSX*[7] instead. [58]

Creating a Theia Plugin A developer can follow the same tutorials that VSCode Extensions provide, and use the same APIs. There is no need to do anything special for Theia to run a VSCode Extension. However, only a subset of the APIs are *implemented* yet in Theia. This is because of time constraints, not policy; more APIs are supported as developers find time.

Supported VSCode APIs A list of supported APIs can be found at https://che-incubator.github.io/vscode-theia-comparator/status.html. Most notably for this thesis, is that the `registerCustomEditorProvider`, `CustomEditorProvider` and `createWebviewPanel`[8] are not supported yet. (Progress for Custom editors can be followed in https://github.com/eclipse-theia/theia/issues/6636).

[6] Only Microsoft products can use the marketplace. [58]

[7] https://open-vsx.org

[8] However, this merge says Theia should support it https://github.com/eclipse-theia/theia/pull/3484.

Installing a Theia Plugin Either search for the plugin inside the plugin browser provided by Theia, or drag-and-drop a `.vsix` file into the extension browser. Theia will only search for plugins on OpenVSX, not the Visual Studio Marketplace.

5.2.2.3 Theia Extension

Description Theia Extensions are the building blocks of Theia. They are loaded at launch time, not run time. Dependency injection is used to discover and load extensions. Extensions were the original mechanism added to Theia for customizing its behavior. They are allowed full access to Theia, and can do anything[9]. [57, 59] All the main features of Theia are provided using Theia Extensions.

Installing an extension To install a Theia Extension, a developer has to compile Theia itself. The extension must be available as a package, e.g. on NPM or as a local folder, and added to the `package.json` as a dependency. [60] For an NPM package, the extension's `package.json` should contain "`theia-extension`" in the `keywords`. [61]

5.3 Prototypes

Prototypes are used to answer some of the research sub-questions. First, one or more questions are selected for a prototype. Then a design document is created, which details goals and architecture of the prototype. Next, the implementation is done. Finally, the implementation is executed and evaluated.
This thesis has a total of two prototypes. The first, *prototype 1*, aims to answer RQ 2.1. The second, *prototype 2*, aims to answer RQ 2.2, RQ 2.3, RQ 2.4 and RQ 2.5.

5.3.1 Prototype 1

This prototype tries to answer RQ 2.1:

> How can java applications be used in a VSCode extension?

5.3.1.1 Requirements

- Develop using VSCode extension API (see Section 5.2.2.1). No dependencies to Theia.
- Embed an executable java `.jar` file into the extension
- Install in both VSCode and Theia (Gitpod)
- Execute the `.jar` using a pre-installed java runtime
- Communicate bi-directionally with the started java process from the extension backend, using `standard input/output (stdio)` and over a TCP socket. (Only Theia has the concept of backend and frontend)

[9] Within the scope of browsers and NodeJS.

The *design document* used for implementation is added in Appendix A.

5.3.1.2 Implementation

Project creation An extension project was created using a project generator, *yeoman*, and a project template called *generator-code*. [62]

First step To test if an extension could call *any* executable binary file at all, the first goal was to run the linux `ls` command. By using the NodeJS standard library function `child_process.spawn`, and printing the outputs to a VSCode information popup, this test was created. This intermediate result was a success, and a popup with folder contents was shown, as illustrated in Figure 5.12.

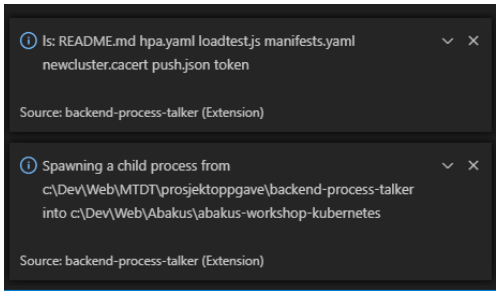


Figure 5.12: VSCode executes a binary executable and displays its output.

Preparing a java jar file Next, a `.jar` file was creted from a java project. This project was a separate one from the VSCode extension. It would create an executable `.jar` which would echo back any text sent to its *standard input*. The executable could also listen to a TCP port on *localhost*, and echo back there. For the `.jar` to be executable, the `main` method was registered in the `META-INF` file as the entry point.

Running the jar in VSCode The code to execute `ls` was modified to run the `.jar` instead. The `.jar` was placed in a (arbitrarily) named folder *lib* in the VSCode extension project root. To get the file path to the `.jar`, a VSCode API was used. A code snipped illustrating this is shown in Code listing 5.1. Triggering the execution of the file was done using a VSCode *command* defined in the prototype, named `Hello World`. Text was sent using standard input/output (`stdin, stdout`). (Another version of the extension was also tried, sending data over TCP to a predefined port on *localhost* (the same machine).)

Code listing 5.1: Typescript code to run a java `.jar` in NodeJS.

```
const command = "java";
const jar = context.asAbsolutePath(path.join("lib", "ecore-tool-process.jar"));
const args: string[] = ["-jar", jar, "stdio"];
const cwd = vscode.workspace.workspaceFolders?.[0]?.uri.fsPath;
```

```
// Start process
const childProcess = child_process.spawn(command, args, {
  cwd,
  windowsHide: true, // hide console windows on Windows
  detached: false, // false = end when parent process ends
});

// Read process output and display it as a popup
childProcess.stdout.on("data", (data: Buffer) => {
  const commandOutput = data.toString("utf-8");
  console.log("Got␣data:", commandOutput);
  vscode.window.showInformationMessage(command + ":\n" + commandOutput);
});

// Send a message to be echoed
childProcess.stdin.write("Hello", (err) => {
  if (err) {
    console.error(err);
  }
  childProcess.stdin.end();
});
```

Bundling the extension The last step is to create a bundle. It is crucial that the bundle has the `.jar` file inside it, otherwise the extension would fail to run it, and this prototype would have a major problem. A tool called `vsce` was installed and used. This tool created a `.vsix` file, which is the extension installer. Theia as VSCode needs this file, as it contains the prototype's code. The `vsce` tool has a `ls` command, which shows all the files it will put into the bundle. This command was executed, and the `.jar` file *was* listed.

Installing the extension Installation in VSCode is done by pressing `ctrl + shift + P` (on windows) and typing `>extensions: Install from vsix`, and then browsing to the file. Installation in Theia is done by opening the *Extensions* panel, and dragging the `.vsix` file into it. The Theia instance used was in Gitpod with the *Theia source code* workspace open[10]

Activating the extension After installation, the Theia *Commands* prompt was opened by pressing `F1` on the keyboard. Then `Hello World` was entered, which is the name of the extension's *command*. Theia proceeded to show the outputs of the `.jar` file in a information popup.

5.3.1.3 Results

Because the extension resulted in a `.vsix` file with a `.jar` inside, and was installed and used in Theia successfully, the first prototype was a success.
The answer to RQ 2.1 is:

[10]https://gitpod.io/#https://github.com/eclipse-theia/theia, requires login.

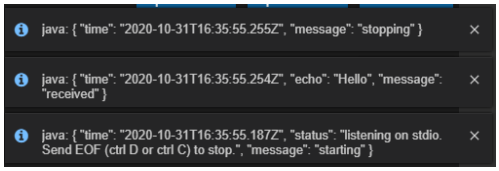


Figure 5.13: Theia is showing information popup windows that contain the outputs of executing a `.jar` file.

> *by bundling the java application as an executable `.jar`, and then starting it as a child process with the NodeJS API `child_process.spawn`. Communication can be done from the Theia backend (or VSCode) over standard input/output or a network socket to a port on localhost.*

This is based on the assumption that the *host operating system has a java runtime installed and available on the path*. However, this runtime *could* be bundled itself. Note that the Theia *frontend* did *not* communicate directly with the child process. All communication was done by the backend and relayed to the frontend over JSON-RPC.

5.3.2 Prototype 2

This prototype tries to answer RQ 2.2, RQ 2.3, RQ 2.4 and RQ 2.5:

> How should the editor cooperate with multiple other tools that change the same underlying model?
> What data is required for displaying a model as a tree?
> How can a tree editor component be generic enough to support arbitrary user actions?
> How can the user interface know the model well enough so the user is constrained from creating invalid Ecore trees?

5.3.2.1 Requirements

Cooperation (RQ 2.2) Cooperation in RQ 2.2 could possibly be solved by channeling any model edits through a Model Server, and listening to this server for change notifications. An example of this was seen in Section 5.2.1.4 with the *Coffee Editor* reference implementation for GLSP (Figure 5.10a).

Tree model (RQ 2.3) By displaying an example tree, the required data will be evident as the view is implemented.

Generic actions (RQ 2.4) By supporting actions through buttons in the user interface, but not hardcoding their clicks to trigger an action, they can be generic. This means a click must only notify something that an *intent* to trigger action `XYZ`

has happened. Implementing a skeleton for this would show what data is required for generic actions.

Constrain tree for validity (RQ 2.5) This is relevant especially for drag-and-drop, where a user should not be able to drop a node onto an "incorrect" parent. Due to time constraints, drag-and-drop will not be implemented now. But the supporting data structure could be sketched out, based on the previous data structures for the Tree model in RQ 2.3.

Here is the list of identified requirements for this prototype:

- Develop using VSCode extension API (see Section 5.2.2.1). No dependencies to Theia.
- Visualize a tree structure in the editor, based on a object structure (e.g. JSON).
- Show labels and icons for nodes in the tree. Labels will be from node *type*, labels from node data.
- Add and remove nodes in the tree from user interaction.
- Property sheet editor that is synchronized with tree selection.
- Visualize both an Ecore meta-model and model instance.
- Use EMF.Cloud Model Server (Section 2.2.3.4) to get the model. Do not read the `.ecore` file from disk in the extension.
- Automatically update the views when the underlying model changes (push not poll)

The *design document* used for implementation is added in Appendix B.

5.3.2.2 Implementation

Setting up the project Similar to prototype 1, a new VSCode Extension project was created. A *standalone* version of the EMF.Cloud Model Server (Section 2.2.3.4) was downloaded[11] Unlike Coffee Editor IDE, this implementation does not use Theia Tree Editor (Section 2.2.3.2), because of its incompatibility with VSCode. A minimal, "homemade" solution was made instead, with CSS, HTML lists (`ol` and `li`) and some javascript.

Connecting to VSCode APIs The editor had to use the `WebView` API, being a non-text editor. The javascript and CSS for a WebView exists in a separate folder without access to the rest of the extension code. This is based on an official Microsoft tutorial called *PawDraw*[12].

[11] Snapshot builds are provided at sonatype: https://oss.sonatype.org/content/repositories/snapshots/org/eclipse/emfcloud/modelserver/org.eclipse.emfcloud.modelserver.example/0.7.0-SNAPSHOT/.

[12] https://github.com/microsoft/vscode-extension-samples/tree/master/custom-editor-sample.

The other piece of the puzzle was a `CustomDocument` and `CustomEditorProvider`. These act as the glue between the `WebView`, VSCode and the extension. Saving and file operations run through these APIs. Most of the handlers on `CustomEditorProvider` were left unimplemented, as only `resolveCustomEditor` was crucial. This is the function that configures the `WebView` for the extension, providing it the tree editor HTML, javascript and CSS etc..

The Tree Editor WebView Inside the `WebView` lives the actual editor user interface. It has security restrictions when it comes to where resources (images etc.) can be loaded from. To test a method for arbitrary icons, an image encoding called *data-uri* was used. It simply encodes the raw image to a Base64 string and prepends some metadata. The result is one long string for one image. A protocol could pass these images for tree node icons instead of referring to file paths. A data structure mapped node types to an icon string, as seen in Code listing 5.2.

Code listing 5.2: Javascript code from the WebView to map icon graphics to tree node types.

```
// Default icons for ECore
var icons = {
  EAnnotation: "...", // truncated
  EAttribute: "...", // truncated
  EClass: "...", // truncated
  EDataType: "...", // truncated
  EEnum: "...", // truncated
  EEnumLiteral: "...", // truncated
  // And then more node types... Truncated.
}
```

To get the data for tree nodes, a sample data structure was created (see Code listing 5.3 for the data). No real EMF or XMI data was read in yet. The fields in the structure were evolved alongside the tree view, adding or restructuring properties when issues arose. A `name` was deemed essential for every node, and moved outside of the `properties`. For determining the *node types*, the Ecore class name was used as the type. (It is uncertain how this works with generics or inheritance, but I assume the type parameters and children could be mapped into the type value, e.g. by appending them). Clicking on a node would store the node as the *selected node*, and trigger an event listener (so the property sheet and action bar could update).

Code listing 5.3: Javascript code from the WebView for a data model describing the tree nodes.

```
// 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: [],
            },
          ],
        },
      ],
    },
  ],
};
```

Form based property editor A simple form was updated with the data of the selected tree node. Changes were sent as notifications to VSCode. These notifications only accept data that can be serialized to JSON. (The communication from the `WebView` to the VSCode Extension *could* possibly be made with JSON-RPC in the future. It was not needed for now).

Action bar An area with buttons was created to trigger arbitrary model actions, such as on-demand validation, genmodel code generation and dynamic instance creation. The actions took their label from an action list, and every action has an *id*. This is shown in Code listing 5.4. Icons could be added in the same way, but was not done. This abstracts actions to only id and label, and then the Tree Language Server could deal with the details of triggering the action.

Code listing 5.4: Javascript code from the WebView to specify the available actions.

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

A list specified which id to always show in the action bar. Another data structure held mappings between node types and lists of actions. This would allow different actions to be available based on the selected tree node. This is shown in Code listing 5.5.

Code listing 5.5: Javascript code from the WebView to specify default actions and per-node actions.

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

Constraining the tree hierarchy No user manipulation of the tree was implemented, due to time constraints. But a data structure that could identify which children are valid for a node type was specified. This is shown in Code listing 5.6. The idea is to consult this mapping when deciding if a child node can be put into a specific parent node, which happens during drag-and-drop and copy-paste operation.

Code listing 5.6: Javascript code from the WebView to specify what children a node type can have.

```
/** 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"],
};
```

Model Server The model server was only partially integrated, due to time constraints. It was started manually, and only a single endpoint was used, which was selecting the *model workspace directory*. This is the directory where the Model

Server will scan for `.ecore` and `.xmi` files. The extension send a request to the Model Server's REST APIs with the workspace folder. A testing command to create a new model file was also implemented. This was a VSCode Command which prompted the user for a filename. Then the extension would send the filename and an example EObject (serialized as JSON) to the Model Server. The Model Server would create a new XMI file when receiving this request.

5.3.2.3 Results

The editor worked, but due to time constraints, some features were dropped. Mainly user editing and model updates were not done, as well as automatic ecore-model-to-tree conversion. A screenshot of the model is shown in Figure 5.14.

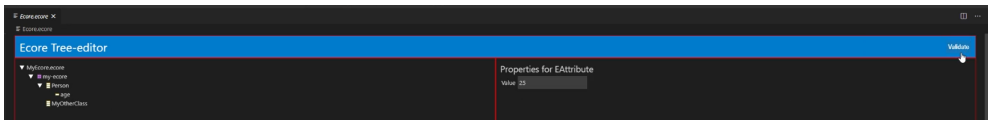


Figure 5.14: A screenshot of the prototype 2 in action. It has three parts: an action bar on the top right, a tree view on the left, and a property editor on the right.

Research questions The answer to RQ 2.2 is:

> *By reading the model data structure from a shared model server, and only editing it by sending commands to this model server.*

The answer to RQ 2.3 is:

> *A nested datastructure of nodes, with labels and node types, as well as properties for each node. See Code listing 5.3 for an example.*

The answer to RQ 2.4 is:

> *By only defining actions as id and label, and sending a schema of default actions and per-node actions to the frontend. Triggering an action will only send a message to the backend Tree Language Server, where this server executes any actual change. See Code listing 5.4 and Code listing 5.5* for examples.

The answer to RQ 2.5 is:

> *By defining a parent-child schema based on node types, that specifies which children are allowed in a parent. See Code listing 5.6 for an example.*

This last answer was not properly tested due to time constraints.

5.4 Tree Editor

This section will present the findings for a *Tree Editor*.

5.4.1 Functional requirements for final editor

Here is a list of identified requirements, in Table 5.1. Note that it may not be complete, with regards to usability and all possible features for MDD tools. More requirements should be specified for a final product, in further works. A good source for requirements are the existing editors mentioned in Section 2.2.2, with regards to a complete solution with good usability.

Table 5.1: 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.

Table 5.1 continued from previous page

ID	Requirement	Description
		An icon or symbol will show if a node has children.
FR5	Tree nodes with children can toggle the visibility of children by user interaction.	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.
FR8	Support creation of new nodes.	
FR9	Support deletion of existing nodes.	
FR10	Support selecting a node.	

5.4.2 Architecture and protocols

A suggestion for an architecture is to use an *VSCode extension* with a *WebView* and *CustomEditor*. Connect the extension to a bundled *Tree Language Server* by using JSON-RPC or REST and WebSocket. The transport mechanism between the extension and the Tree Language Server can be either standard input/output (stdio) or TCP sockets. The Tree Language Server should hold any language dependent details, like mappings from Ecore over to a generic tree structure. Any changes to the model should be relayed from the extension via the Tree Language Server to the Model Server. The extension should only talk about changes in a general manner with the Tree Language Server. These will be converted to Ecore specific operations in the Tree Language Server before being sent to the Model Server over a REST API. A diagram with this architecture is shown in Figure 5.15.

Sharing a Model Server Because multiple extensions might require the same Model Server, an idea (not explored yet) is to have a specific VSCode Extension only for providing this Model Server. Other extensions can then notify a depen-

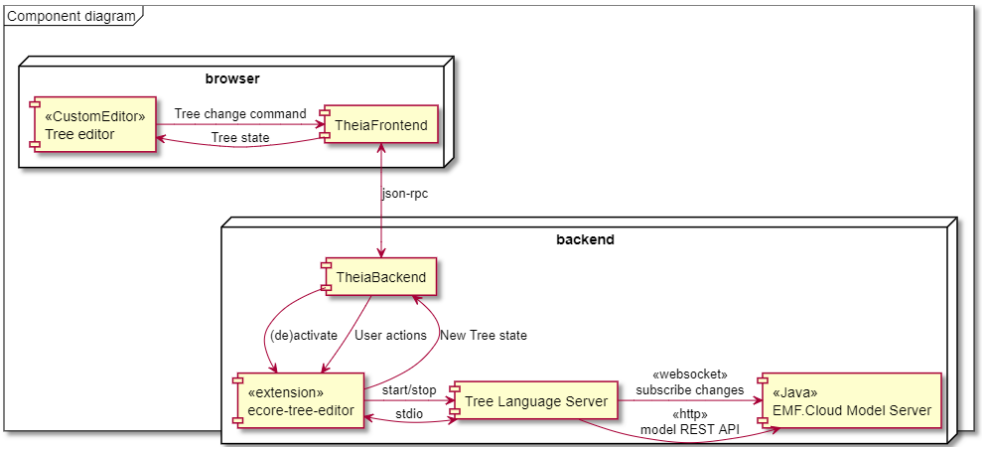


Figure 5.15: A suggested architecture for a tree editor.

dence on this extension, causing it to be installed. This seems supported at least in VSCode. The Model Server extension would then provide the details for connecting to it, to other extensions. It is possible to discover other extensions and get an API from them.

Chapter 6

Evaluation

Because this is a specialization project before a masters thesis, the emphasis has not been on evaluation. Only a brief evaluation follows.
The research questions were answered by the prototypes. Not all questions were answered properly, due to time constraints. This is RQ 2.2 and RQ 2.5.
The main research question, RQ 2 got a partial answer. The list of requirements is not complete, and only lists the most relevant requirements for a cloud based architecture. More is likely needed to get a highly usable product.

Chapter 7

Discussion

Because this is a specialization project before a masters thesis, the emphasis has not been on discussion. Only a brief discussion follows.
The prototypes provided promising results as expected. Realization of a such Tree Editor should be possible. One unexpected finding was that Theia Tree Editor (Section 2.2.3.2) relied on the Theia core, and could not be used in VSCode. This claim that it does not work in VSCode has not been tested. There is still a possibility for it to work, if the dependencies are mostly for data structures and not functionality. Further work could test this in a prototype. In the event that it *does* work, much effort could be saved.
It seems to become a pattern in cloud based IDEs to rely more on generic clients and specific servers. The results strengthens this, by showing that the same pattern used in LSP and GLSP is transferable to a Tree editor.
A Tree editor will only provide the first step towards greater acceptance of MDD in the new generation of developers. There is room for improvement on the code generation side as well. Further work could try to provide genmodel templates that result in ready-made typescript REST APIs, and editors based on Theia using Theia Extension. Some work has already been done with regards to REST, but these code generation templates should be provided by the tool, out of the box.
The actual data structures required for a Tree Language Server protocol are not investigated. The other protocols like LSP and GLSP center around these large collections of ready-made data structures. Further work could be done to identify what commands should be sent between the editor and the backends. The Theia Tree Editor and existing Eclipse IDE editors (Section 2.2.2) could provide a good starting point for collecting data on this.
The requirements for a generic Tree editor client in Section 5.4 are not complete. Further work is needed to find all the requirements. As with the protocol, the Theia Tree Editor and existing Eclipse IDE editors (Section 2.2.2) could provide a good starting point for collecting data on this. No scientific literature containing requirements for a Tree editor were found. The closest result was Karrer and Scacchi [63], which seems to focus on graphs (and interprets *tree* as a graph structure, not a folder-like hierarchical structure as here).

Chapter 8

Conclusion

Because this is a specialization project before a masters thesis, the emphasis has not been on providing a thorough conclusion. Only a brief conclusion follows.
The tools from EMF can be moved to the cloud. Editors of Ecore can be re-implemented to cloud IDEs by using extension mechanisms. Existing tooling for model validation, parsing and generation can be embedded inside those extensions. This removes the need for re-implementing the entire framework, and therefore saves effort.
The need for a Tree Editor was found. Initial prototypes show that it is possible to create. Requirements and an architecture for such a Tree Editor is proposed, drawing inspiration from earlier protocols like LSP and GLSP. Further work is needed to establish the protocol for this Tree Editor, and all the software requirements. Further work is then needed to implement this software solution.
The practice of having a generic client and specific server seems applicable to more editors than just text based editors. This architecture could be the basis of many migrations from editors in desktop IDEs to cloud based IDEs.

Bibliography

[1] 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).

[2] 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.

[3] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice: Second Edition*, 2nd ed. Morgan & Claypool Publishers, 2017, 208 pp., ISBN: 978-1-62705-708-0.

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

[5] A. Lajmi. (Apr. 14, 2016). "Modeling in the Browser: What DSL Forge gives for free," Modeling Languages, [Online]. Available: https://modeling-languages.com/modeling-browser-dsl-forge/ (visited on 11/14/2020).

[6] C. CARRASCAL-MANZANARES, J. S. Cuadrado, and J. de Lara, "Building MDE cloud services with DISTIL," in *International Conference on Model Driven Engineering Languages and Systems*, ser. Model-Driven Engineering on and for the Cloud, vol. 1563, Ottawa, Canada, Sep. 2015, pp. 19–24. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01761670 (visited on 11/26/2020).

[7] F. Coulon, A. Auvolat, B. Combemale, Y.-D. Bromberg, F. Taïani, O. Barais, and N. Plouzeau, "Modular and distributed IDE," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*, ser. SLE 2020, New York, NY, USA: Association for Computing Machinery, Nov. 16, 2020, pp. 270–282, ISBN: 978-1-4503-8176-5. DOI: 10.1145/3426425.3426947. [Online]. Available: https://doi.org/10.1145/3426425.3426947 (visited on 11/26/2020).

[8] R. Saini, S. Bali, and G. Mussbacher, "Towards web collaborative modelling for the user requirements notation using eclipse che and theia IDE," in *Proceedings of the 11th International Workshop on Modelling in Software Engineerings*, ser. MiSE '19, Montreal, Quebec, Canada: IEEE Press, May 26, 2019, pp. 15–18. DOI: 10.1109/MiSE.2019.00010. [Online]. Available: https://doi.org/10.1109/MiSE.2019.00010 (visited on 11/26/2020).

[9] L. Walsh, J. Dingel, and K. Jahed, "Toward client-agnostic hybrid model editor tools as a service," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, ser. MODELS '20, New York, NY, USA: Association for Computing Machinery, Oct. 16, 2020, p. 1, ISBN: 978-1-4503-8135-2. DOI: 10.1145/3417990.3421440. [Online]. Available: https://doi.org/10.1145/3417990.3421440 (visited on 11/26/2020).

[10] M. Milinkovich. (Nov. 20, 2005). "About the Eclipse Foundation | The Eclipse Foundation," [Online]. Available: https://www.eclipse.org/org/ (visited on 12/09/2020).

[11] (). "Obeo into a few words - Obeo," [Online]. Available: https://www.obeo.fr/en/company (visited on 12/09/2020).

[12] (). "Obeo products presentation - Obeo," [Online]. Available: https://www.obeo.fr/en/products (visited on 12/09/2020).

[13] EclipseSource. (). "Services," EclipseSource, [Online]. Available: https://eclipsesource.com/services/ (visited on 12/09/2020).

[14] C. Guindon. (). "Eclipse Membership > EclipseSource | The Eclipse Foundation," [Online]. Available: https://www.eclipse.org/membership/showMember.php?member_id=690 (visited on 12/09/2020).

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

[16] RedHat. (). "The world's open source leader," [Online]. Available: https://www.redhat.com/en (visited on 12/09/2020).

[17] R. Gronback. (). "Eclipse Modeling Project | The Eclipse Foundation," [Online]. Available: https://www.eclipse.org/modeling/emf/ (visited on 10/06/2020).

[18] E. Merks. (Oct. 5, 2007). "Merks' Meanderings: Is EMF going to replace MOF?" Merks' Meanderings, [Online]. Available: https://ed-merks.blogspot.com/2007/10/is-emf-going-to-replace-mof.html (visited on 11/11/2020).

[19] Eclipse Foundation, *Eclipse/xtext*, Eclipse Foundation, Dec. 6, 2020. [Online]. Available: https://github.com/eclipse/xtext (visited on 12/08/2020).

[20] Eclipse Foundation. (). "Xtext - Language Engineering Made Easy!" [Online]. Available: https://www.eclipse.org/Xtext/ (visited on 12/08/2020).

[21] Eclipse Foundation. (). “Sirius - The easiest way to get your own Modeling Tool,” [Online]. Available: https://www.eclipse.org/sirius/ (visited on 12/05/2020).

[22] P.-C. David and Obeo, “[SiriusCon 2018] Sirius Roadmap,” Dec. 11, 2018. [Online]. Available: https://www.slideshare.net/Obeo_corp/siriuscon-2018-sirius-roadmap (visited on 10/06/2020).

[23] Neil Mackenzie and Mélanie Bats. (Sep. 24, 2020). “Sirius Web and XText,” [Online]. Available: https://www.eclipse.org/forums/index.php/t/1105347/ (visited on 10/06/2020).

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

[25] Camille Letavernier, Simon Graband, Aaron R Miller, and Nina Doschek, *Eclipse-emfcloud/ecore-glsp*, version d104af3, eclipse-emfcloud, Sep. 23, 2020. [Online]. Available: https://github.com/eclipse-emfcloud/ecore-glsp (visited on 09/29/2020).

[26] C. Smith. (Aug. 1, 2018). “Eclipse Sprotty,” projects.eclipse.org, [Online]. Available: https://projects.eclipse.org/projects/ecd.sprotty (visited on 12/06/2020).

[27] Eclipse Foundation. (). “Eclipse Sprotty,” projects.eclipse.org, [Online]. Available: https://projects.eclipse.org/projects/ecd.sprotty/who (visited on 12/06/2020).

[28] Eclipse Foundation, *Eclipse/sprotty*, version ab42ddd, Eclipse Foundation, Dec. 6, 2020. [Online]. Available: https://github.com/eclipse/sprotty (visited on 12/06/2020).

[29] *Eclipse-emfcloud/theia-tree-editor*, eclipse-emfcloud, Sep. 17, 2020. [Online]. Available: https://github.com/eclipse-emfcloud/theia-tree-editor (visited on 09/29/2020).

[30] EclipseSource. (). “JSON Forms,” [Online]. Available: https://jsonforms.io/docs/what-is-jsonforms (visited on 12/06/2020).

[31] Eugen Neufeld, Martin Fleck, Tobias Ortmayr, Nina Doschek, Cees Bos, and Camille Letavernier, *Eclipse-emfcloud/emfcloud-modelserver*, version a303e75, eclipse-emfcloud, Dec. 1, 2020. [Online]. Available: https://github.com/eclipse-emfcloud/emfcloud-modelserver (visited on 12/07/2020).

[32] Guillaume Hillairet, *Eclipse-emfcloud/emfjson-jackson*, eclipse-emfcloud, Dec. 1, 2020. [Online]. Available: https://github.com/eclipse-emfcloud/emfjson-jackson (visited on 12/07/2020).

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

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

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

[36] Microsoft. (Jan. 14, 2020). "Language Server Protocol Specification - 3.15," Language Server Protocol Specification - 3.15, [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/specification-current/ (visited on 09/23/2020).

[37] Jonas Helming and Maximilian Koegel. (Nov. 4, 2019). "The Graphical Language Server Platform/Protocol - EclipseSource," EclipseSource, [Online]. Available: https://eclipsesource.com/blogs/2019/11/04/introducing-the-graphical-language-server-protocol-platform-eclipse-glsp/ (visited on 09/29/2020).

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

[39] Philip Langer, Martin Fleck, and Tobias Ortmayr. (Dec. 1, 2020). "Eclipse-glsp/glsp PROTOCOL.md," GitHub, [Online]. Available: https://github.com/eclipse-glsp/glsp (visited on 12/07/2020).

[40] StackOverflow. (2019). "Stack Overflow Developer Survey 2019," Stack Overflow, [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).

[41] Max Shaposhnik, Michal Vala, Lukas Krejci, and Sergii Kabashniuk. (Jan. 8, 2020). "Introduction to Devfile," Introduction to Devfile, [Online]. Available: https://redhat-developer.github.io/devfile/ (visited on 12/07/2020).

[42] *Microsoft/vscode*, Microsoft, Dec. 6, 2020. [Online]. Available: https://github.com/microsoft/vscode (visited on 12/06/2020).

[43] Microsoft. (Oct. 8, 2020). "Visual Studio Code User Interface," [Online]. Available: https://code.visualstudio.com/docs/getstarted/userinterface (visited on 10/29/2020).

[44] 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).

[45] Eclipse Foundation, *Eclipse-theia/theia*, eclipse-theia, Dec. 6, 2020. [Online]. Available: https://github.com/eclipse-theia/theia (visited on 12/06/2020).

[46] Typefox. (). "Architecture Overview," [Online]. Available: https://theia-ide.org/docs/architecture (visited on 12/06/2020).

[47] (2017). "Multi-Language IDE implemented in JS Scope and Architecture," Google Docs, [Online]. Available: https://docs.google.com/document/d/1aodR1LJEF_zu7xBis2MjpHRyv7JKJzW7EWI9XRYCt48/edit?usp=embed_facebook (visited on 09/17/2020).

[48] Anton Kosyakov and Sven Efftinge. (May 2, 2019). "Code Organization," GitHub, [Online]. Available: https://github.com/eclipse-theia/theia/wiki/Code-Organization (visited on 12/06/2020).

[49] Philip Langer and Maximilian Koegel, "Web-based modeling tools with EMF.cloud (published to the web)," presented at the EclipseCon 2020 (Internet (virtual)), Oct. 22, 2020. [Online]. Available: https://docs.google.com/presentation/d/e/2PACX-1vRVnzfSELGwVrZNk5kfSKXtM5Te9hIbzPpnIoi40IEn7QzwFF3pAaXDtq1bg pub?start=false&loop=false&delayms=3000&usp=embed_facebook (visited on 12/08/2020).

[50] Microsoft. (Nov. 6, 2020). "Extension Anatomy," [Online]. Available: https://code.visualstudio.com/api/get-started/extension-anatomy (visited on 12/06/2020).

[51] Microsoft. (Oct. 8, 2020). "Contribution Points," Contribution Points, [Online]. Available: https://code.visualstudio.com/api/references/contribution-points (visited on 10/29/2020).

[52] Microsoft. (Oct. 8, 2020). "Extending Workbench," [Online]. Available: https://code.visualstudio.com/api/extension-capabilities/extending-workbench (visited on 10/29/2020).

[53] Microsoft. (Nov. 6, 2020). "Extension Host," [Online]. Available: https://code.visualstudio.com/api/advanced-topics/extension-host (visited on 12/06/2020).

[54] Microsoft. (Nov. 6, 2020). "Extensions Capabilities Overview," [Online]. Available: https://code.visualstudio.com/api/extension-capabilities/overview (visited on 12/06/2020).

[55] Microsoft. (Nov. 6, 2020). "Webview API," [Online]. Available: https://code.visualstudio.com/api/extension-guides/webview (visited on 10/30/2020).

[56] Paul Maréchal. (Nov. 10, 2020). "Theia Plugin Implementation," GitHub, [Online]. Available: https://github.com/eclipse-theia/theia/wiki/Theia-Plugin-Implementation (visited on 12/06/2020).

[57] J. Helming and M. Koegel. (Oct. 10, 2019). "Eclipse Theia extensions vs. plugins vs. Che-Theia plugins," EclipseSource, [Online]. Available: https://eclipsesource.com/blogs/2019/10/10/eclipse-theia-extensions-vs-plugins-vs-che-theia-plugins/ (visited on 10/05/2020).

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

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

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

[61] Typefox. (). "Authoring Theia Extensions," [Online]. Available: https://theia-ide.org/docs/authoring_extensions (visited on 12/06/2020).

[62] Microsoft. (Oct. 8, 2020). "Your First Extension," [Online]. Available: https://code.visualstudio.com/api/get-started/your-first-extension (visited on 10/30/2020).

[63] A. Karrer and W. Scacchi, "Requirements for an extensible object-oriented tree/graph editor," in *Proceedings of the 3rd Annual ACM SIGGRAPH Symposium on User Interface Software and Technology - UIST '90*, Snowbird, Utah, United States: ACM Press, 1990, pp. 84–91, ISBN: 978-0-89791-410-9. DOI: 10.1145/97924.97934. [Online]. Available: http://portal.acm.org/citation.cfm?doid=97924.97934 (visited on 11/14/2020).

Appendix A

Prototype 1 — Design Document

This is the design document used before implementing prototype 1. The prototype refers to itself as *backend-process-talker*. It refers to the bundled `.jar` file as *Ecore tool jar*, and the process started from this `.jar` as *bundled-process*.

Design Document

What is this?

Context and Scope

Kristian and Hallvard want to know if extensions in VSCode can start and talk to arbitrary background processes. This is because much tooling in MDSE for EMF exists already in the form of java code. Re-using a `.jar` for an extension will simplify the transition of Ecore from EMF in Eclipse into Theia.

This extension is a prototype and will only try to start a bundled `.jar` file and communicate with it.

Goals and non-goals

- Bundle a `.jar` in an extension.
- Start a process from this `.jar`.
- Communicate bi-directionally from the extension with the process. They run on the same machine.
- Run in Theia, install as an VSCode Extension/Theia Plugin.
- Only use VSCode API. No Theia internal API.

> Non-goal: Things that could be goals, but we chose not to do them.

Non-goals:

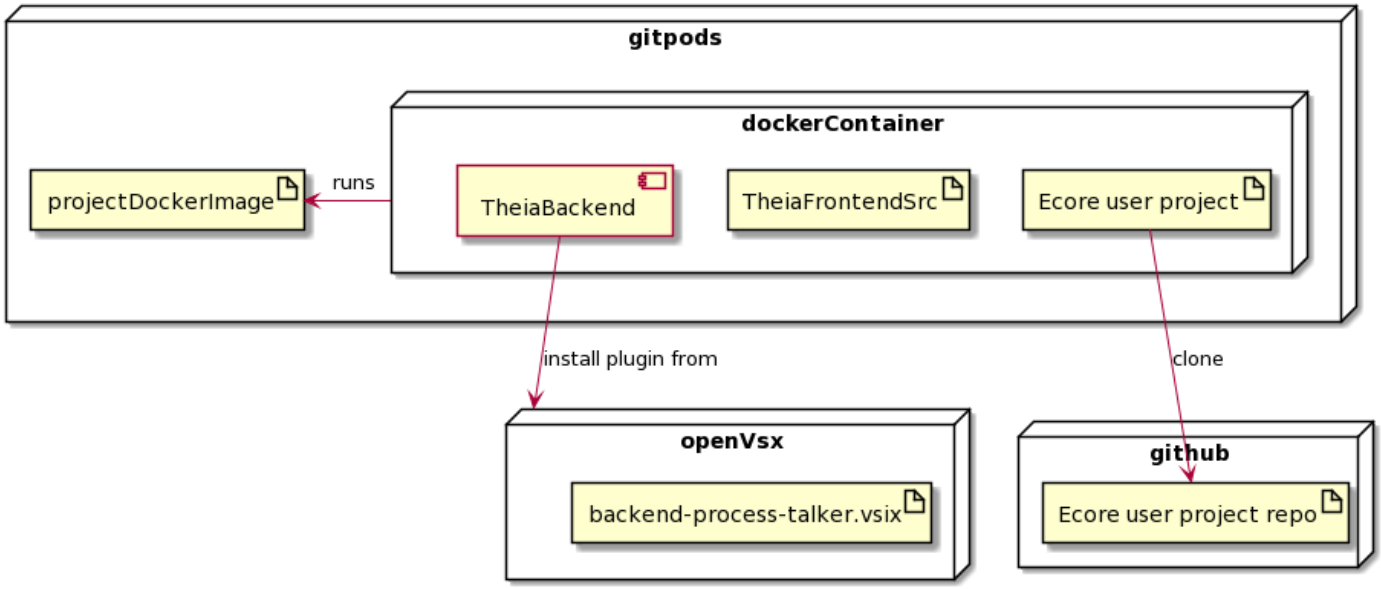
- Start a process automatically when a specific file/view is opened.
- Communicate over LSP.

Design and trade-offs

It will be a VSCode extension project in Typescript.
Using the VSCode extension API you can install the extension during runtime into a Theia instance. This is possibly needed for Gitpods (*assumption*).
A trade-off is that using Theia Extensions, you get full control. Here we are restricted to VSCode API and what subset of it Theia supports.

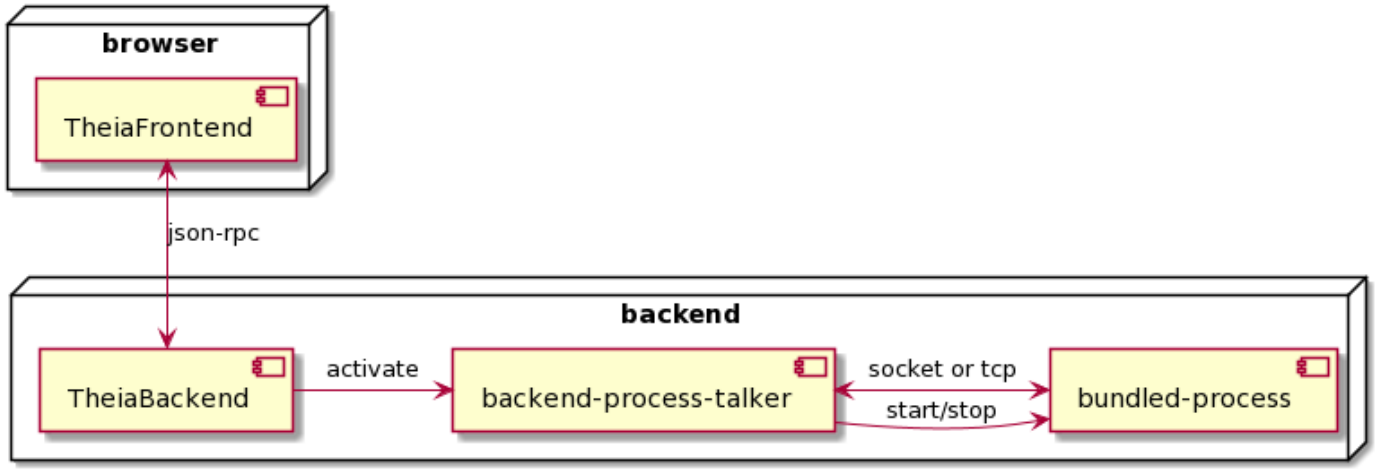
System-context-diagram

Deployment diagram



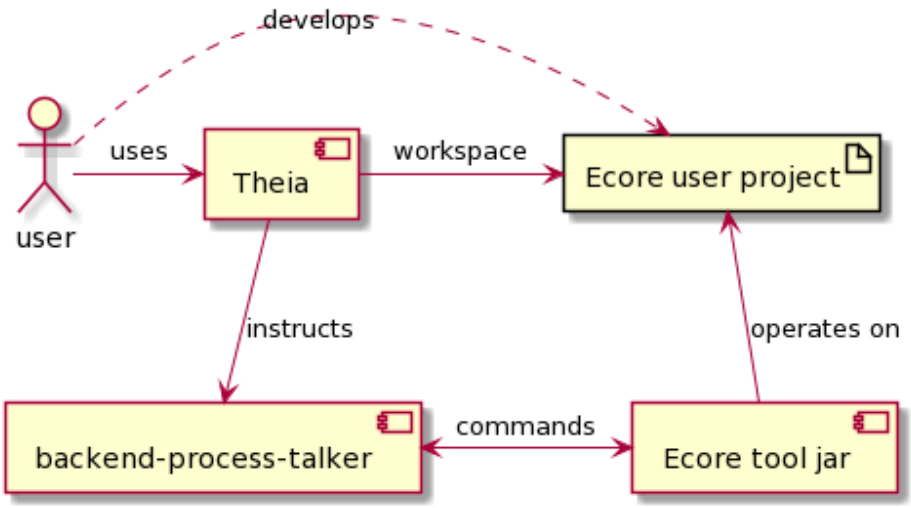
▶ PlantUML code

Component diagram



▶ PlantUML code

Interaction diagram



▶ PlantUML code

Degree of constraint

The project must work in Theia inside Gitpods.
Gitpods is the ultimate target.

It should be an VSCode extension, but if Gitpods can load Theia Extensions, this constraint falls.

Alternatives considered

- Using the terminal in Theia to launch a CLI task that starts the `.jar`.
  - An IDE would be better if it did not require users to pollute their project dependencies with its tooling.
- Using a Theia Extension.
  - This would work, but has the problem that Gitpods deployment might be trickier.
  - Would lose compatibility with VSCode IDE.

Appendix B

Prototype 2 — Design Document

This is the design document used before implementing prototype 2. The prototype refers to itself as *ecore-tree-editor*. It refers to the bundled `.jar` file as *EMF.Cloud Model Server*. The design document refers to a *Tree Language Server*, which is not implemented. To keep the project simple while at an early stage, the Model Server is interacted with directly in the ecore-tree-editor, bypassing the need for a Tree Language Server. This Tree Language Server would evetually be added as a `.jar` file, to use the already provided Model Server java client interfaces.

Design Document

> What is a design document?

Context and Scope

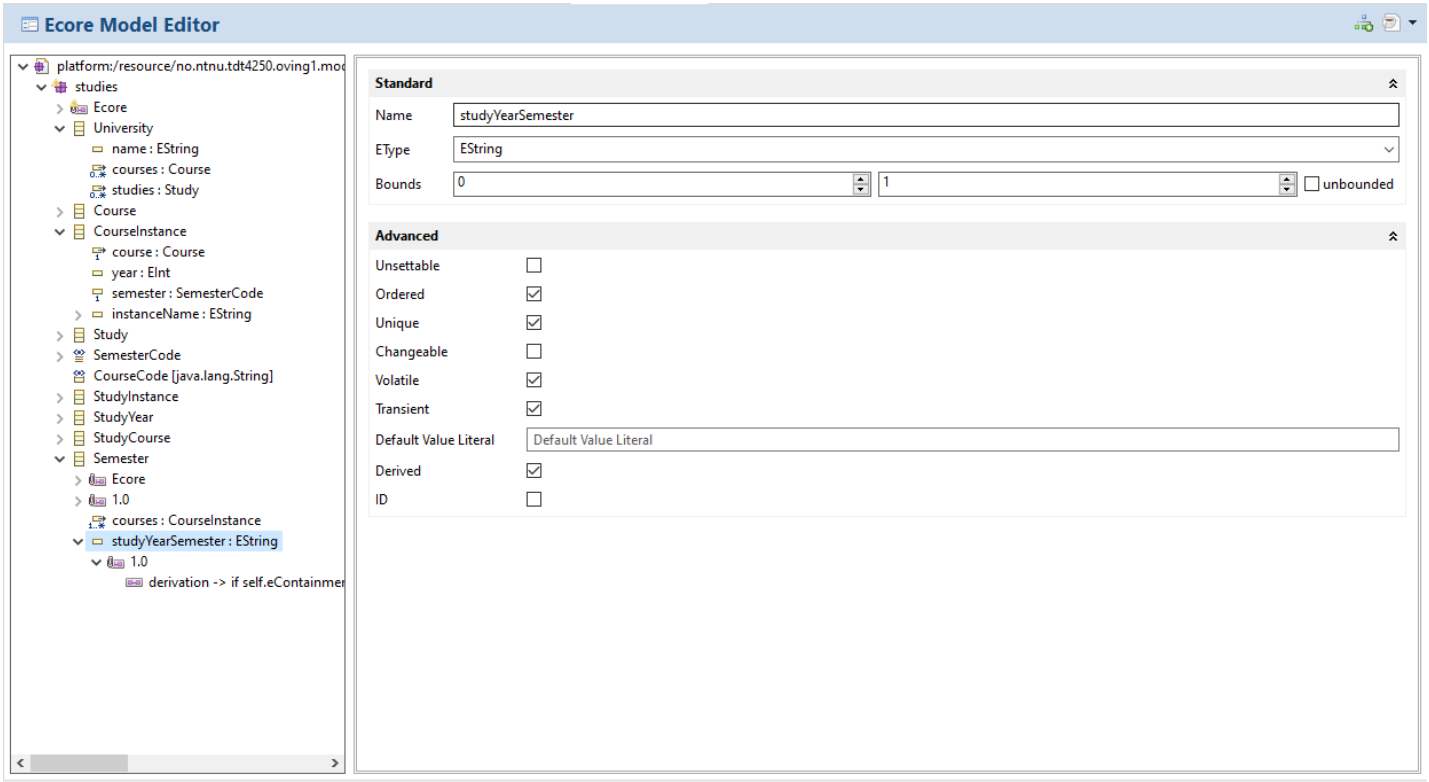
Ecore is a meta model inside the Eclipse Modeling Framework (EMF).
This meta model is used by developers to create domain models when practising Model-Driven Software Engineering (MDSE). An Ecore model can be visualized as a
diagram similar to UML (like EcoreTools/Sirius), as a tree stucture or a raw XML-like file.

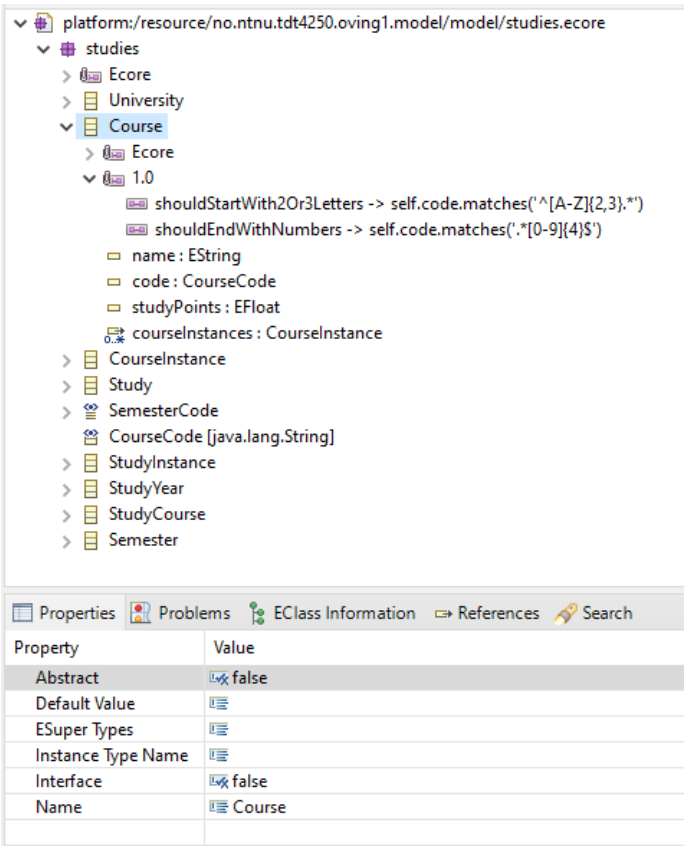
For editing Ecore in Theia, we already have the ecore-glsp. However, this is only a graph editor.
Another common way of editing Ecore is through a tree view + properties panel.

This project aims to implement a tree view editor for Ecore in Theia,
and bundle as a VSCode extension (`.vsix`) that is installable in Gitpods.

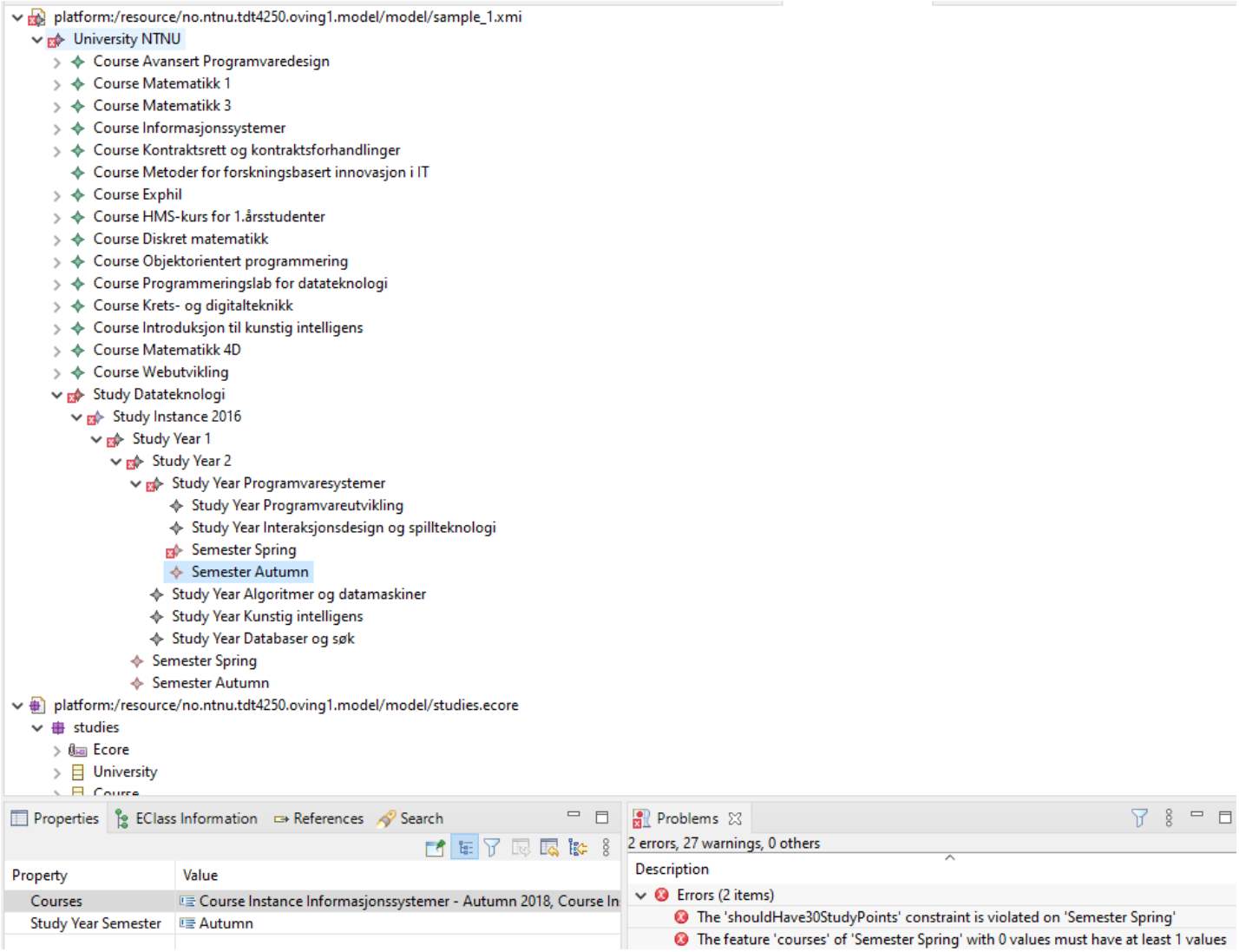
Reference implementations

Inspiration and guidance is drawn from the Eclipse implementations:
Ecore Model Editor and the *Sample Reflective Ecore Model Editor*.





Note that the *Sample Reflective Ecore Model Editor* can also edit model instances (`xmi` -files) and run OCL validations:



Goals and non-goals

- Visualize Ecore models as trees
  - Package, datatype, class, attribute, annotation and annotation-values as nodes
  - Labels: name, icon, datatype
- Add and remove nodes in the tree
- Property sheet editor. Master-detail where tree=master and property=detail.
- Both meta-model (Ecore) and model instance (xmi)
- Automatically update when underlying model changes (push not poll)

> Non-goal: Things that could be goals, but we chose not to do them.

Most of these are non-goals due to time constraints, and could be in-scope for a finalized product.

Non-goals:

- Lazy load children for large models
- Handle dependencies to other models
- Drag-n-drop nodes
- Editable names in tree
- Custom labels (e.g. via OCL) for model instances (xmi)
- Customizable icons for model instances (xmi)

- OCL constraint validation for model instances (xmi)
- Live OCL evaluator for writing queries

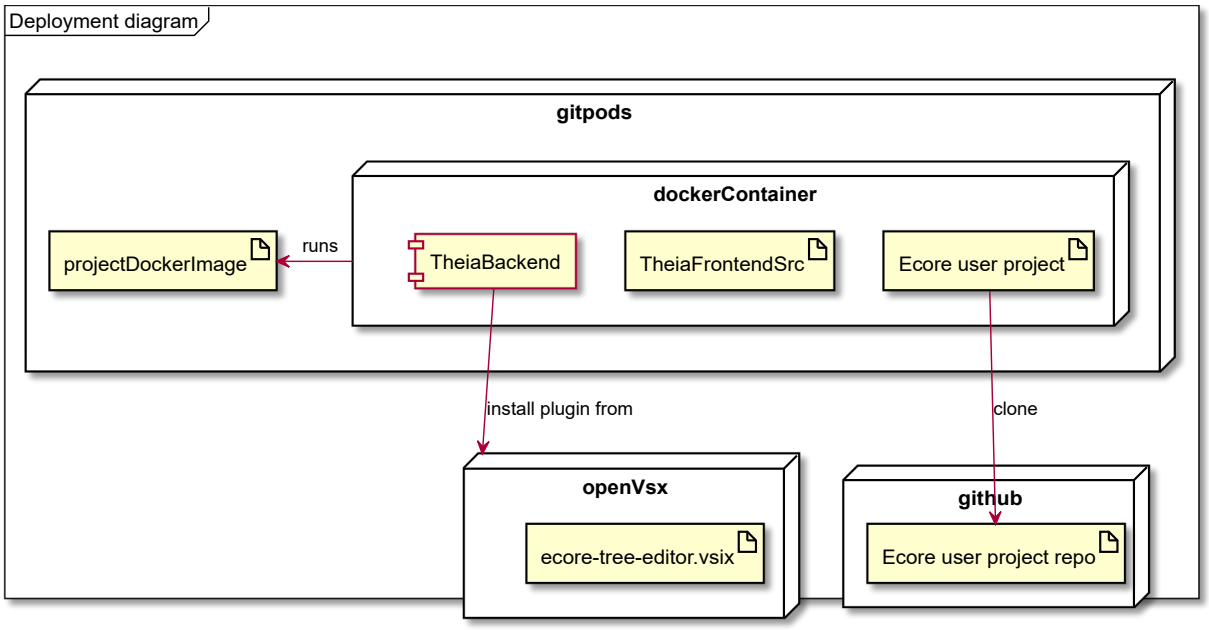
Design and trade-offs

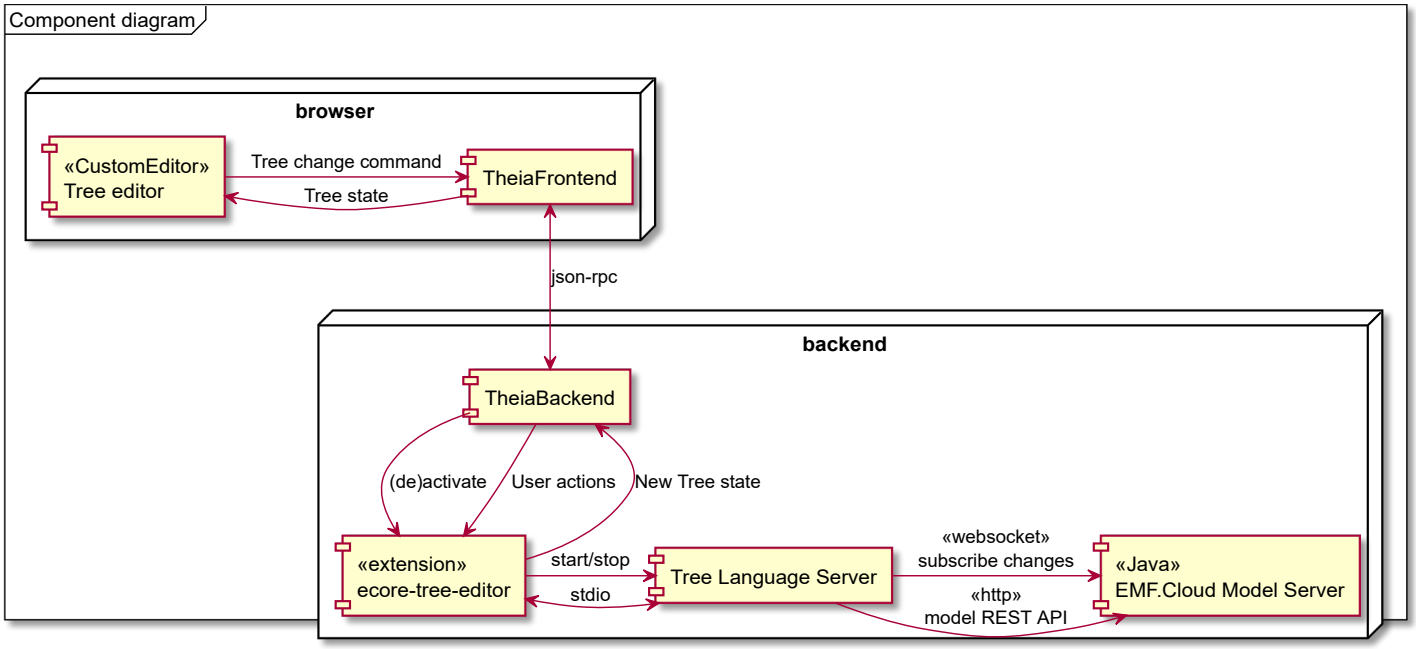
It will be a VSCode extension project in Typescript.
Using the VSCode extension API you can install the extension during runtime into a Theia instance. This is possibly needed for Gitpods (*assumption*).
A trade-off is that using Theia Extensions, you get full control. Here we are restricted to VSCode API and what subset of it Theia supports.

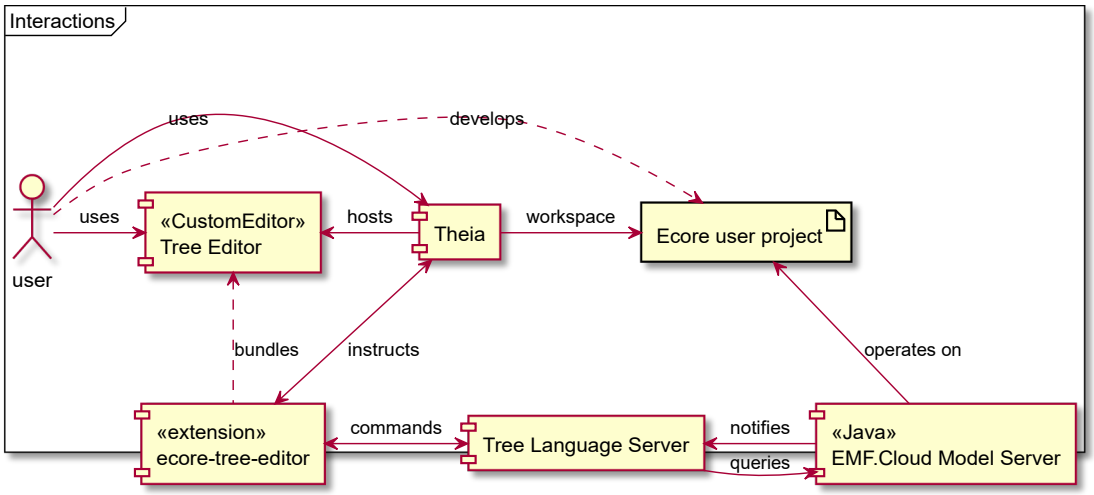
Using a VSCode extension, the text editor is limited to text.
Thus we have to rely on a Custom Editor API to create arbitrary layouts.
(An alternative is WebView , but I don't know which API Theia will support first).
The Custom Editor API is essentially WebView+Document APIs.
These WebViews have special requirements for serializing state,
and restrictions for where they can load resources.

`CustomTextEditorProvider` or `CustomEditorProvider` ? A `CustomEditorProvider` may be best.
A text based editor gets a document model for free, and related functionality like saving. If a tree can easily map to text editing, this could be beneficial. However, most changes to the document would not be performed by the editor, but the **model server**. The editor is mostly a model observer and change-command dispatcher; not an *editor* itself.

System-context-diagram







Degree of constraint

The project must work in Theia inside Gitpods.
Gitpods is the ultimate target.
However, **when** is an important question. Within 2 years from Jan/20201 is reasonable.
This timeframe allows for Theia to (maybe) support the CustomEditor or WebView VSCode APIs.

It should be an VSCode extension, but if Gitpods can load Theia Extensions, this constraint falls.

It may use EMF.Cloud's EMF Model Server if this simplifies model parsing or enhances tool/extension interoperability/cooperation among other extensions.

~~It should use Theia Tree View if this is possible to use in a VSCode extension.~~ (**Seems to not be possible.** It depends on Theia core and uses Theia Extension API).

- Alternatively, there is a VSCode TreeView for displaying trees in the Action Bar (e.g. the workspace file explorer). Its freedom (icons, right click actions etc) may be limited (*assumption*).
- https://github.com/mar10/fancytree
- https://www.jstree.com/

It should use JSON Forms for the detail view.
This supports React and Angular.

Alternatives considered

- VSCode Custom Editor API
  - This uses APIs not supported in Theia (*yet*)
    - `window.registerCustomEditorProvider`
- WebView
  - This uses APIs not supported in Theia (*yet*)
    - `window.createWebviewPanel`
  - Prefer custom editor over this
- EMF Forms. This seems to be a full editor on the web.
  If this could be set to target the Ecore metamodel itself, and then be embeded in VSCode WebView/CustomEditor,
  - Tutorial.
  - It uses Remote Application Platform (RAP) to render on web.
  - The Eclipse `Ecore Editor` from EMFForms is derivated from GenericEditor.
- EMF Forms Tree Master Detail used in EMF Forms.