Moving Model Driven Engineering from Eclipse to Web Technologies

Sören Domrös

Master's Thesis
November 15, 2018

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Eclipse is a popular IDE for model-driven-development. It is a current trend to move IDEs to the web. Web technologies enable to use flexible frameworks for UI development. Moreover, they enable container-based development. An IDE can be used in the web with nothing more than a browser with zero configuration and setup time. Web IDEs need separation between business logic and UI, which is not facilitated by Eclipse.

The academic KIELER project is an Eclipse-based IDE for model-driven engineering of SCCharts and other synchronous languages. KIELER can only run as a desktop application and still uses a SWT-based UI. A migration to web technologies seems promising and provides new opportunities for Human Computer Interaction (HCI). Web technologies enable to design applications, which run in the web or locally as an Electron app.

In this thesis the Theia framework is used to migrate KIELER from Eclipse to a client server architecture called KEITH using the Language Server Protocol (LSP). The already existing implementation of KIELER is reused to generate a language server backend using Xtext. The LSP and Xtext allow to support KIELER and KEITH development at the same time. The qualitative evaluation of this migration project shows that web technologies can be used to develop an IDE and that Theia is a viable IDE framework, which is highly extensible.

The resulting KEITH tool promises to be usable for teaching and conferences. The flexible UI allows to test and develop new UI concepts using HCI. The different setup methods of KEITH make it a highly flexible and configurable development tool.

Acknowledgements

I want to give special thanks to my advisor Alexander Schulz-Rosengarten and my professor Dr. Reinhard von Hanxleden for reading everything I gave them and giving constructive criticism. Moreover, I want to thank Steven Smyth for coming up the name KEITH (for Rich Charts (RiCharts)). Without him the resulting tool would be named something less awesome. I want to thank Darth Vader, my friends, Niklas Rentz, and everyone else who supported me during my thesis. Last but not least I want to thank Keith Richards for his music and the similarity of his name to the developed tool.

Contents

List of Figures

Listings

Chapter 1

Introduction

Model-driven engineering allows to create an abstract model of a system, which can be used to compile to different target systems and simulate the result. A text editor is not sufficient to develop and use such a model. An Integrated Development Environment (IDE) makes code more readable by providing syntax highlighting and indentation. An IDE provides content assist and refactoring support to ease development. Moreover, an IDE is able to navigate in the source code via symbol references, definitions and previous editing locations. Most modern IDEs, such as IntelliJ[1], Eclipse[2], and VSCode[3], provide these features and are highly extensible. IDEs for model-driven engineering are built to compile and simulate models.

One IDE for model-driven engineering based on Eclipse is the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) [HFS11; FH09b]. KIELER is used to develop Sequential Constructive Statecharts (SCCharts) [HDM+14]. SCCharts is a sequential constructive dialect of Harel's StateCharts [Har87]. The KIELER tool allows to compile and simulate SCCharts and other synchronous languages. Furthermore, KIELER provides a graphical and textual representation of the models.

Eclipse is programmed in Java[4]. Eclipse's UI is also programmed in Java using the Standard Widget Toolkit (SWT)[5]. SWT and Swing — another commonly used Java UI framework — are no longer actively maintained [Moh17]. Because of the missing maintenance of a Java UI a new UI technology shall be adopted to provide an up-to-date tool for academic research, which is what this thesis is about.

The new UI framework shall reuse the existing language features of KIELER as a service. Web technologies allow maximum UI customization and are therefore a promising UI technology. Web technologies are a recent trend in UI and application development and enable the developer to provide IDE support in the browser, which provides an easy to setup and reproducible development environment. Developers use web technologies not only to develop applications for the browser. Web technologies allow to run applications as desktop apps by bundling them as an Electron[6] app. The evaluation of these trends promises to be a new research topic for usability and Human Computer Interaction (HCI). Web or cloud IDEs can be used to ease development and collaboration of teams. They are a new trend in IDE development, as seen in the example of Eclipse Che[7].

1.1 Cloud IDE

Cloud IDEs are on the rise, as seen on Eclipse Che, CoRED [LNK+12], CEclipse [WLK+11], Adinda [DMC+10], mBed[8], and more. Many of them are IDEs for the web in the web, such as the online IDEs

[1] https://www.jetbrains.com/idea/
[2] https://www.eclipse.org/
[3] https://code.visualstudio.com/
[4] https://java.com
[5] https://www.eclipse.org/swt/
[6] https://electronjs.org/
[7] https://www.eclipse.org/che/
[8] https://www.mbed.com

JSFiddle[9] and the Online JavaScript Editor[10]. They are developed in the browser for the browser, which allows to test functionality in the future production environment. Furthermore, only a browser is needed to use them, which makes them usable even with tablets.

Many developers of mentioned IDEs want quick and easy to set up IDEs, which support up-to-date UI-technology. Developers want IDEs that are easy to extend and can support multiple languages. Since most of the mentioned cloud IDEs run in a browser, container-based development is possible. This makes development independent from the Operating System (OS), since the IDE runs on a shared server [HPH14]. Cloud or web IDEs have to solve the problems of workspace-management and security. These IDEs do not run locally, but on a designated server. An IDE running on a server is not allowed to have full access to the file system. Different users connecting to this IDE need different workspaces. Mutiara et al. regard security and license issues as future work in their project, since it is not always addressed in cloud IDEs [MRW14]. Therefore, some developers still want and need a locally running IDE. One option to deliver a desktop app and a browser version of an IDE is the Theia framework.

1.1.1 Theia

Theia[11] is an IDE framework developed by TypeFox[12]. Theia itself is implemented using web technologies. Moreover, Theia has its own diagram framework: sprotty[13]. The IDE framework allows to deliver an equivalent Electron app and browser version. Theia is a new IDE relative to Eclipse. Therefore, it needs a way to support additional languages, since it does not have access to such a rich extension environment as Eclipse. In Eclipse new extensions are added via the Eclipse Marketplace[14]. Theia achieves this via its general structure. Theia is an IDE framework consisting of different packages, which bundle functionality. The IDE is customized by adding extension packages, which are delivered via the package manager npm[15]. New languages are added differently. Theia is based on the Monaco Editor[16]. The integrated Monaco Editor allows to add language support through the Language Server Protocol (LSP) via new language servers. Therefore, a developer uses language servers to add language support to Theia.

1.2 The Language Server Protocol

The Language Server Protocol (LSP) is a framework to encapsulate part of the language specific implementation for an IDE into an own service [BGM17]. Auto-completion, find references, go to definition, and other features are part of commonly used IDEs. These features are language specific rather than IDE specific. The don't repeat yourself principle suggests that such implementation shall be reused, since two versions of the same code are harder to maintain [WAB+14].

Language features are added by IDE extensions to an IDE (e.g. via the Eclipse Marketplace). IDE extensions are written in the same language as the IDE itself and different IDEs use different APIs for their extension system. Therefore, extensions cannot be reused for different IDEs. Language features are commonly implemented for all target IDEs to provide support for one language, resulting in the m IDEs n languages problem [KPE16]. The problem states that to provide language support for

[9] https://jsfiddle.net/
[10] https://js.do/
[11] https://github.com/theia-ide/theia
[12] https://typefox.io/
[13] https://github.com/theia-ide/sprotty and https://github.com/theia-ide/theia-sprotty
[14] https://marketplace.eclipse.org/
[15] https://www.npmjs.com
[16] https://microsoft.github.io/monaco-editor/

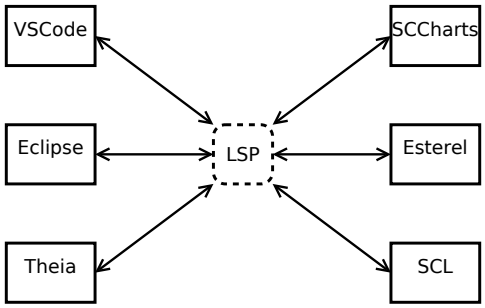


Figure 1.1. m IDEs n languages solution

m IDEs and n languages, $m \cdot n$ IDE extensions are needed. With the language server protocol the $m \cdot n$ language implementation problem is transformed to a $m + n$ problem. The LSP is used as a common communication standard for language features. Only one language server per language is needed and the IDEs only need to implement the protocol, as seen at the examples of language server implementations and language server support listed here [Tea18a]. The resulting $m + n$ problem is visualized in Figure 1.1.

Language support for multiple IDEs via the LSP was already done, as seen at the example of the data modeling language Yet Another Next Generation (YANG)[17]. YANG shows that it is possible to implement a language server for different IDEs with minimal effort by reusing most of the implementation [Köh17c]. This makes YANG development possible in four different IDEs: Yangster[18] as an Electron app, Yangster in the browser, Eclipse, and VSCode, as seen in Figure 1.2. All IDEs provide the same language features and an equivalent diagram view. Yangster uses the Theia framework and sprotty to generate a diagram representation of a YANG model. The example of YANG shows that language support can be added easily for IDEs, which support the LSP. Language servers reuse language support to enable model-driven development on different target platforms. This boosts language design and the use of Domain Specific Languages (DSLs), since IDE support can be provided easily by implementing or generating a language server. A language server makes the language itself more visible to other developers or teams, which might want to use the language, since it can easily be adopted to new IDEs.

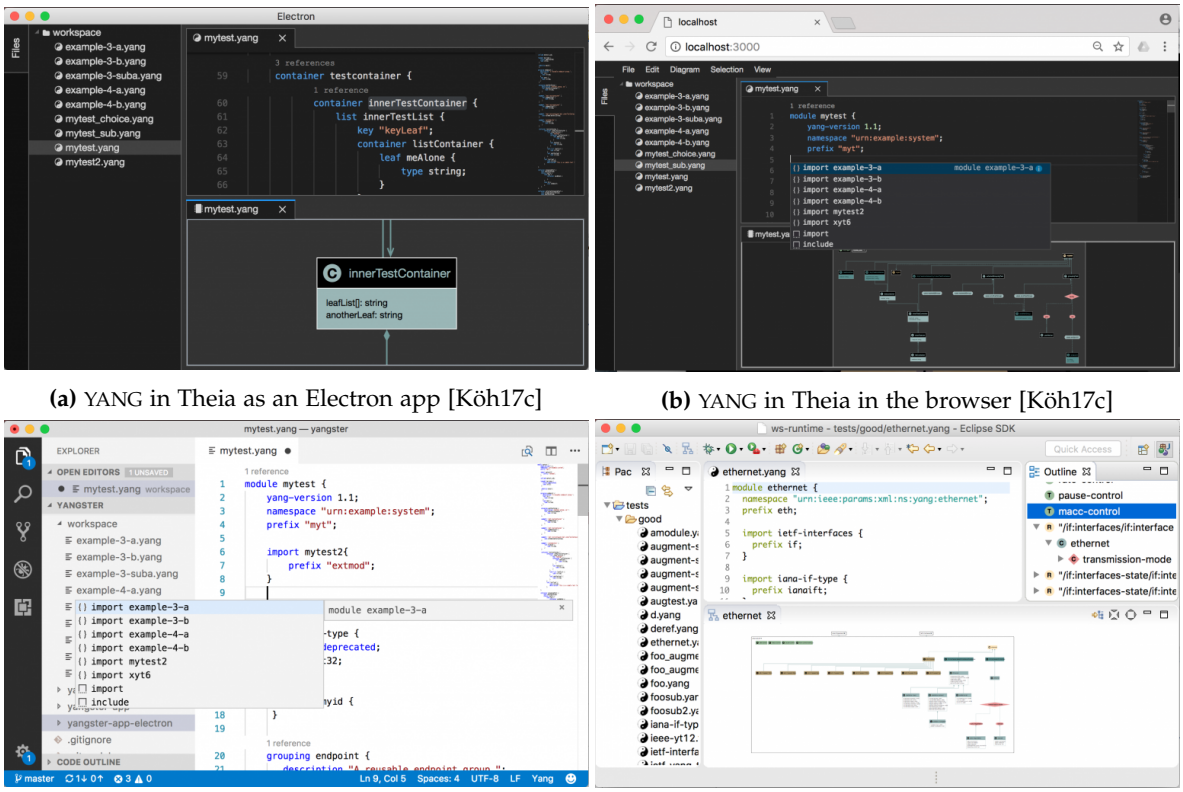
The recent trend of language servers as well as web technologies promises to be an interesting research topic. It does provide new environments and setups for an IDE. These setups can be used to analyze user behavior and the influence on HCI. Furthermore, the influence of a client server architecture on software development has to be analyzed.

1.3 Problem Statement

The goal of this work is to migrate the KIELER tool, a tool for model-driven development in synchronous languages based on Eclipse, to a web-based implementation in Theia called KIEL Environment Integrated in Theia (KEITH). A Theia extension achieves this. The Theia extension will consist of a language server for all languages supported by KIELER, such as SCCharts and its subgrammars. This language server is generated to support references, find definitions, do code completion, and more rich language features by reusing the plugins of KIELER. Furthermore, a Theia extension, which will support syntax highlighting, has to be developed. The Theia extension is able to call the KIELER Compiler (KiCo) by

[17] http://www.yang-central.org/twiki/pub/Main/YangDocuments/rfc6020.html

[18] https://github.com/theia-ide/yangster



(a) YANG in Theia as an Electron app [Köh17c]

(b) YANG in Theia in the browser [Köh17c]

(c) YANG in VSCode [Köh17c]

(d) YANG in Eclipse [Köh17c]

Figure 1.2. Yang for four IDEs

extending the LSP. Widgets, menus, commands, and keybindings for Theia are implemented to achieve this.

The stated project is used to evaluate the migration process from Eclipse to web technologies and Theia. The process will have special focus on what migration patterns are useful, what components have to be restructured, what tools are helpful in this process, and what can be learned from the migration. Large models are used to qualitatively evaluate performance of the Theia extension in terms of reactivity, usability, scalability, and maintainability by own experience reports. The non-functional properties of KEITH are compared to KIELER to present the differences, advantages, and disadvantages of the migration target platform. The resulting IDE will be used in further research about the new UI and HCI concept present in Theia.

1.4 Outline

In Chapter 2 the technologies and ideas that are used in the migration project are presented and explained. In Chapter 3 related work in form of similar technologies, IDEs and migration projects are presented. Chapter 4 covers the migration concepts. It explains what obstacles, problems and dependencies apply to a migration from Eclipse or a similar technology to a client server architecture. Furthermore, new UI concepts are explained and compared to the ones used in Eclipse. Chapter 5 describes how the actual migration is taking place and how the project is set up, what features are

supported, and what design decisions influenced them. Moreover, the tooling and setup are explained. It is elaborated how new features for the language server or the Theia extension can be added. In Chapter 6 the migration process and the KEITH product are evaluated qualitatively. Performance and features of KIELER and KEITH are compared with regard to usability, reactivity, and maintainability. Furthermore, the design decisions are reflected and evaluated. Different approaches are suggested if applicable. Chapter 7 summarizes the efforts and insights about this migration project and suggests future work on this topic and the resulting IDE.

Chapter 2

Preliminaries

This chapter introduces and explains the primary tools and concepts used or discussed in later chapters. This includes features commonly present in IDEs. Furthermore, Eclipse, the KIELER tool based on Eclipse, KIELER's primary language SCCharts, Xtext[1], the LSP[2], Theia, and TypeScript[3] are presented. In the following sections an IDE, which does not use the LSP or a similar client server architecture, is referred to as monolithic.

2.1 IDE Features

The first step in an IDE migration is to evaluate what features and functionality have to be migrated. Features of an IDE are part of this migration. The essential IDE features are elaborated in this section. An IDE must be able to edit text and do syntax highlighting for specific languages since they influence program comprehensibility [Ram86]. This alone is enough to edit code and even see some syntactic errors, since the highlighting seems to be wrong at some point. Syntax highlighting is not enough to enable the user to develop the full coding potential by concentrating only on the programming aspect. Not all examples mentioned here can be supported for every language, since not every grammar is designed for it or it is no language feature of that language.

A user makes mistakes. One of them are spelling mistakes. One way to avoid this is content assist. While typing an IDE suggests matching variable or class names, generates control blocks and loops on demand, supports indentation, and automatically manages imports. Formatting and with it indentation should also be supported to ensure readability and with that understandability and maintainability of a program [Lei80; MMN+83]. Since scopes and blocks are highlighted by indentation, bracket mismatches are more visible and the program itself is easier to understand. An IDE should support some kind of code formatting and allow the users to configure it and share the configuration with the team to support a common coding style. These are some requirements that help the user to structure and write code and lift an IDE over a common editor.

The next aspect is linting. Linting means static syntax checking including error or problem markers, style checking, finding dead code, finding unused variables, finding constructs which may lead to errors, and more [Joh78]. Linters are used to aid the programmer in addition to the compiler (if one exists) to write less error prone code, by pointing out possible errors. For linters it is important to filter these possible error messages, since the user may overlook the serious issues if there are too many problems detected [Joh78; Hol02]. An IDE should support linting and have an intuitive way of dealing with the noise of such linters by providing configurable filters.

Some errors cannot be found statically, such as logical errors or, from linters undetected, runtime errors. Therefore, IDEs need some kind of testing environment and logging. A debugging tool should allow to check integration of software and logical correctness of modules, classes, and functions (i.e.

[1] https://www.eclipse.org/Xtext/

[2] https://github.com/Microsoft/language-server-protocol

[3] https://www.typescriptlang.org/

unit tests). A debugger to configure breakpoints and step through a program is essential for languages, which run in real time. Debugging is most times the only option despite logging to see variable values and an option to understand the control flow of a program. Without debugging, software development cannot take place, since fixing and finding bugs is a tedious task with no tool support.

Apart from editing, the most common reason to use an IDE is compilation or execution of a program. IDEs provide support in library management, have integrated compilers and are able to deploy applications (e.g. in a `jar`). Integrated compilers and application deployment allow runtime-tests of the application and allow the user to see and develop UIs. Often external build tools such as Gradle[4] or Maven[5] are integrated to manage different modules or classes and build products. This way, the same tool can provide services for several phases in the development cycle, which suggests less problems with configurations of different IDEs.

In the lifetime of an IDE it is a use case to add additional language or tool support to the development tool by extensions. These extensions require the IDE to be customizable. A customizable IDE allows to not support a language in an IDE at its initial launch, since support can be added later via extensions. For example, Eclipse is customizable via a browsable market place (e.g. to add git support) and VSCode with an extension view, which can do the same. Extensions are the key to support new programming languages, since they can be added to an already compiled product at runtime. It is common for an IDE to have an extension interface to allow other developers to contribute and customize the IDE. An extension interface allows the IDE community to add new features. The community can support the original developer by adding tool support, which they need for their projects.

For most contexts an editor needs more than editing and syntax highlighting. Drag and drop of files from the file system should be supported. An editor needs window management to edit or compare code side-by-side. A representation of a file system, menus for configuration of program execution, as well as some kind of console output are fundamental features of an IDE and should be intuitively designed with HCI in mind.

Despite all the desired features, the user should not have the feeling that development and the editing experience is slowed down by configuration dialogs, syntax highlighting, content assist, or external tools. If content assist is integrated in the editor, it is often blocking. Blocking features slow down or stall the user in the editing process, since big files need longer to parse. Asynchronous, non blocking requests solve this issue. A editor-independent language feature implementation solves this by using the LSP.

2.2 Eclipse

Eclipse is a very popular IDE. The Eclipse Foundation[6] manages the development of various frameworks to add new tool support to their already rich infrastructure. Eclipse became famous for their Java IDE, but supports several other languages. Eclipse is highly extensible via its Eclipse Marketplace. The marketplace allows to add various tools to support development in nearly every language. Each Eclipse application has a `.ini` file, which can be used to configure the application. It is used to specify the Java version, maximum memory usage, whether Eclipse should open a UI, and more. The Eclipse Plug-in Development Environment (PDE) enables developers to add their own plugins to Eclipse. An Eclipse application uses extension points, which allows to dynamically load Eclipse plugins in the application using the Open Service Gateway Initiative (OSGi). Eclipse applications such as KIELER need these extension points to work. Eclipse in general is an ideal example on what an IDE should

[4] `https://gradle.org/`

[5] `https://maven.apache.org/`

[6] `https://www.eclipse.org/org/foundation/`

support. However, Eclipse also has its downsides. The trend in UI design and HCI concepts moved away from Eclipse. Eclipse uses the no longer actively maintained Java UI technology SWT. Moreover, Eclipse has extensive dialogs to configure its preferences and relies on various button toolbars to make functionality accessible to the user. This makes Eclipse's UI hard to comprehend [RT05; SDM+03].

The Eclipse Foundation also develops in other directions. Because of recent trends the Eclipse Foundation has its own cloud IDEs: Eclipse Che and Eclipse Orion. They are able to run in the browser or as desktop application. These cloud IDEs run in the web for the web. Eclipse Che and Eclipse Orion are fit to develop, deploy, and run applications in the cloud. The cloud IDEs are Eclipse's answer to development in the cloud and are able to adopt new UI and HCI concepts. Eclipse Che and Eclipse Orion also serve as IDE platforms together with Eclipse.

IDE platforms enable to build an own IDE based on an already existing one to reuse its infrastructure. One IDE based on Eclipse is KIELER.

2.3 KIELER

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER) is used to develop SCCharts as well as to compile and simulate them [HFS11; FH09b; HDM+14]. KIELER is able to show SCCharts in its graphical representation synthesized from a textual grammar, which allows to use the advantages from a textual and a graphical representation of a program [FH09a; Han18]. It is easier to change and use a textual model for version management. The graphical representation allows to get a better overview of the whole system and makes it easier to understand. How modeling in KIELER looks like can be seen in Figure 2.1. Textual and graphical model representation can be seen side by side. The graphical representation can be configured via an option menu on the right of it. The bottom bar holds a view for the projects, the compiler view, and views for simulation of the models. A data view displays the current variables. The data pool holds all variables and allows to change them to influence the simulation. Changes to the variables are applied in the next executed step in the simulation. The compiler view allows to use the KiCo to transform models via selectable compilation systems. This allows to generate C Code using an SCCharts model. The workspace is displayed in a package explorer view and holds all projects of the workspace. It allows to compile models by step-by-step transformations into an executable, other languages, or representations. The subresults, so called snapshots, of these transformations can be shown separately in form of their graphical representation. KIELER allows to edit and add new compilation systems and transformations to develop new implementations and transformations for other use cases. Furthermore, models can be transformed to C or Java Code, which can be executed and simulated inside KIELER. Visual feedback in the graphical model, seen in red in Figure 2.1, and a data view allow to test the developed automatons. KIELER supports to develop models for various synchronous languages while synthesizing a graphical representation of the model from the textual description. The main language however is SCCharts.

2.3.1 SCCharts and other Grammars

SCCharts is a sequential constructive StateCharts dialect, which is used for modeling of automatons [HDM+14; Har87]. These models have a textual and graphical representation, which is synthesized by KIELER or more concrete KIELER Lightweight Diagrams (KLighD) [SSH13]. SCCharts is a hierarchical grammar, which consists of the subgrammars KExt, KExpressions, KEffects, and Annotation.

SCCharts uses a different notion of time than most languages. Time is divided into discrete ticks, consisting of several micro steps, which represent the steps taken in the modeled automaton. SCCharts allows parallel regions, hierarchy, and sequential execution to model behavior. An SCCharts model is

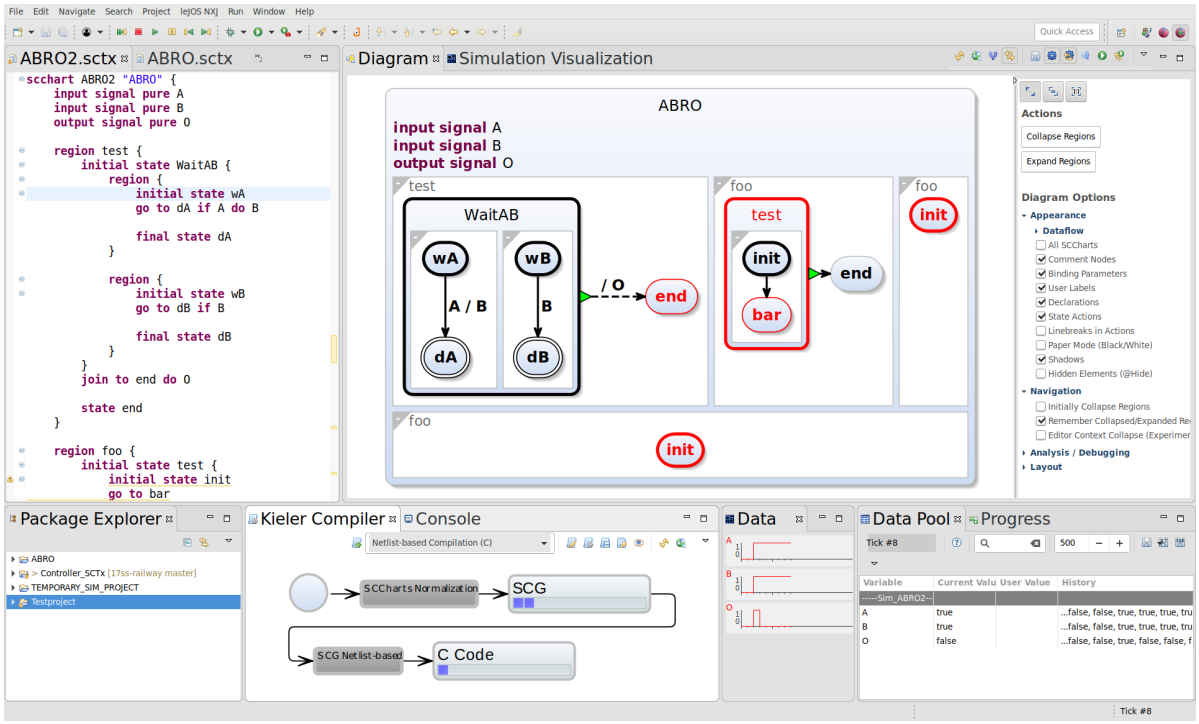


Figure 2.1. UI of KIELER

valid if the micro steps can be brought into a fixed order for every possible tick, while the constructiveness and the iur-dependencies are respected to achieve sequential constructiveness [HDM+14]. If this property is satisfied, there are no race conditions or unknown states in the modeled automaton. The SCCharts grammar is defined by Xtext, which automatically generates language support.

2.4 Xtext

Xtext[7] is a framework for DSLs. By defining a grammar, a full language infrastructure for Eclipse, a Language Server (since Xtext 2.11), or other tools can be generated and configured including syntax coloring, semantic coloring, error checking, auto-completion, formatting, hover information, marking occurrences, going to declaration, renaming refactoring, debugging, toggling of comments, outline view, structure view, quick fix proposals, finding references, showing call hierarchy and type hierarchy, and folding.

This is done by implementing an `XtextGrammar` and using an mwe2 workflow[8] to generate configurable fragments for these features. These fragments can be used to develop an Eclipse extension. This extension is used to provide previously mentioned language features for an editor or an IDE. Furthermore, the same generated code is used to generate a language server with the same amount of functionality, which allows to reuse code generated by Xtext grammars and mwe2 workflows. Xtext is able to manage hierarchical grammars, which is necessary for the development of some DSLs (e.g. SCCharts), which consist of several reused subgrammars. Together with the mwe2 workflow so-called

[7] `https://www.eclipse.org/Xtext/`

[8] `https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.xtext.doc%2Fcontents%2F118-mwe-in-depth.html`

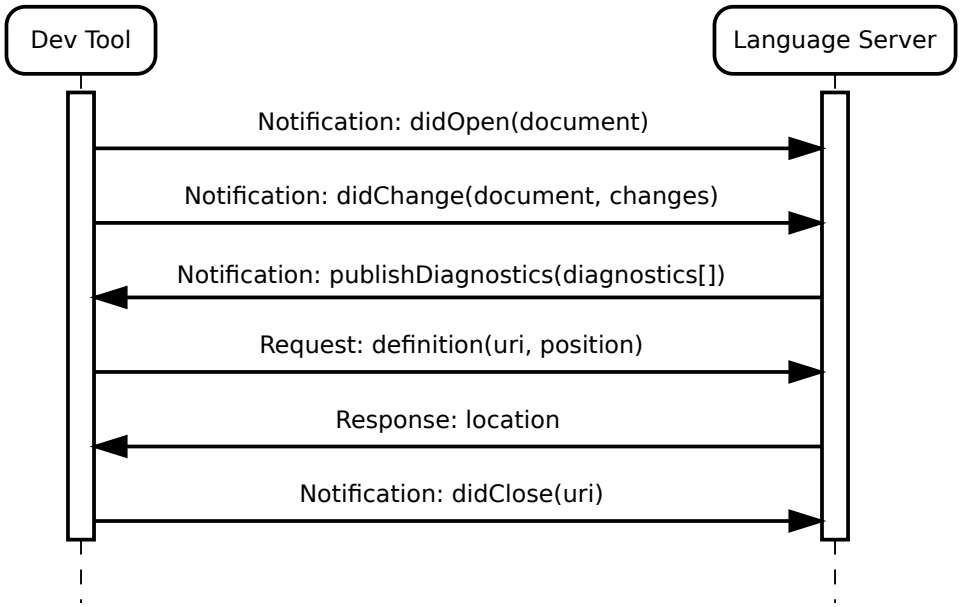


Figure 2.2. Language server communication during a session

fragments can be defined, which leverage the defined Xtext grammar to generate language features. These fragments can be manually extended or implemented if the default behavior does not match the desired one. Since not all languages can be taken into account, extensibility is crucial for such a framework. Otherwise, the scope of a DSL would be limited. In the KIELER tool Xtext generates language support for Eclipse, but equivalent language support can also be generated for a language server.

2.5 The Language Server Protocol

As seen in Section 1.2, the LSP tends to solve the m IDEs n languages problem by defining a common interface for editing tools and language servers. The LSP is an open-source communication protocol based on JSON Remote Procedure Call (JSON-RPC) 2.0[9]. The LSP was originally developed by Microsoft to support language servers for VSCode to provide a linter for warnings and diagnostics.[10] Later Microsoft continued developing this protocol together with CodeEnvy[11] and Red Hat[12] to be general enough to support multiple languages and added support for multiple IDEs and languages [Erc16; Tea18a].

As seen in Section 2.1, some of the desired features of an IDE are only language-dependent and can be provided without the rest of the IDE and support even more IDEs at once, as already mentioned in Section 1.2. This works as long as the protocol is general enough to support features for multiple languages. An example communication during a session can be seen in Figure 2.2 to explain how an editor uses such a language server.

The user begins the development process by opening a document, which results in a notification to the language server that a document with a specific URI was opened. The language server is notified if

[9] https://www.jsonrpc.org/specification
[10] https://microsoft.github.io/language-server-protocol/
[11] https://codenvy.com/
[12] https://www.redhat.com/

```
{
   "jsonrpc": "2.0",
   "id" :1,
   "method": "textDocument/definition",
   "params": {
      "textDocument": {
         "uri": "file:///p%3A/mseng/VSCode/
              Playgrounds/cpp/use.cpp"
      },
      "position": {
         "line": 3,
         "character": 12
      }
   }
}
```

Listing 2.1. 'textDocument/definition' request

```
{
   "jsonrpc": "2.0",
   "result": {
      "uri": "file:///p%3A/mseng/VSCode/
           Playgrounds/cpp/provide.cpp",
      "range": {
         "start": {
            "line": 0,
            "character": 4
         },
         "end": {
            "line": 0,
            "character": 11
         }
      }
   }
}
```

Listing 2.2. 'textDocument/definition' response

the corresponding file is changed by the user and answers with diagnostics for that file. If the user uses the go to definition feature, the development tool requests the definition of a position from the language server, as seen in Listing 2.1. A request consists of the protocol name JSON-RPC 2.0, an id, a name for the called method, and parameters. The id is used to cancel the request (e.g. a request for content assist is canceled, since a new line is focused). The method identifies the function, which is called in the language server. The name must not be the name of the method; it is rather used as an identifier. In the language server lsp4j[13] is used to annotate functions to respond to this method identifier. Parameters can be all serializable attributes. In this example, the file URI and position in the file identify the symbol of which the definition is requested. The language server responds with the position of the corresponding definition, as seen Listing 2.2, which is used by the editing tool to jump to the corresponding lines. The parameters of these message are the file URI of the requested definition and the range of its location in the file. Messages like this and messages for different language features are sent several times in a session. These request happen asynchronous and are non blocking, which promises not to influence performance and reactivity of the editing tool itself. After the user closed the file, a close notification is sent by the development tool to inform the language server that the file was closed. Of course, the communication is not limited to a single file; the URI of each document is used to identify and reference it.

As seen in Figure 2.3, it is possible and common to communicate with multiple language servers, which provide support for different languages. This feature enables an IDE to be configurable for specific development tasks, since every language with a language server can be supported with rich editing features. VSCode already uses this technology to add support for new languages on the fly via its extension system. Other IDEs can use this technology too to add language support via language servers.

Communication with a language server is possible via stdio, sockets, named pipes, or node-ipc. Since this migration project uses Xtext to generate a language server and uses Yangster as a reference, only communication via stdio/stdout and socket is supported. Yangster is based on the Theia framework, which naturally supports the LSP.

[13] https://projects.eclipse.org/projects/technology.lsp4e

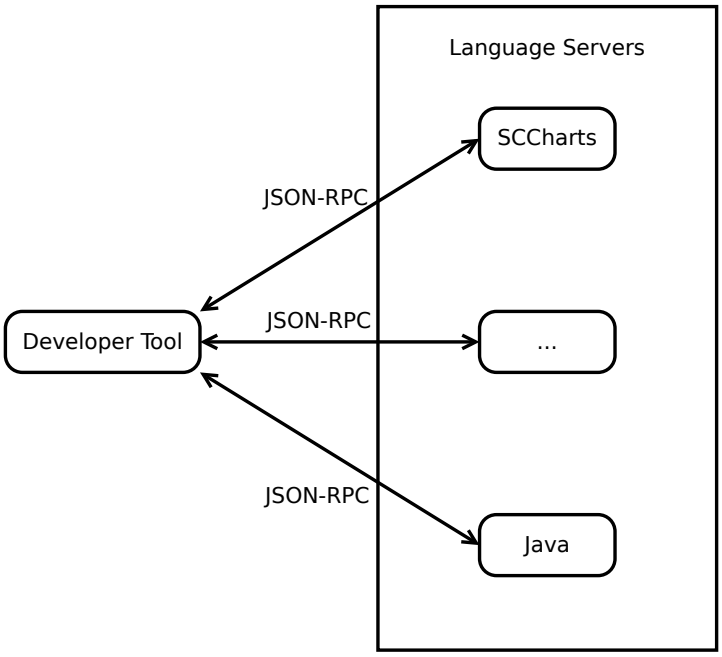


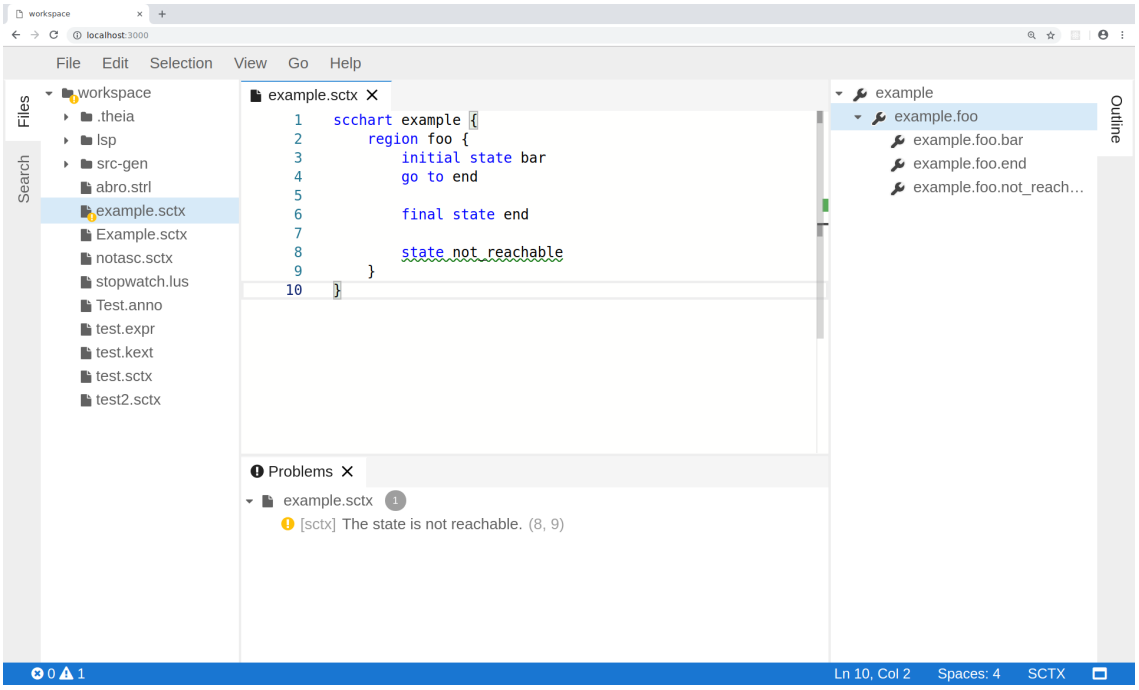
Figure 2.3. Communication with multiple language servers

2.6 Theia

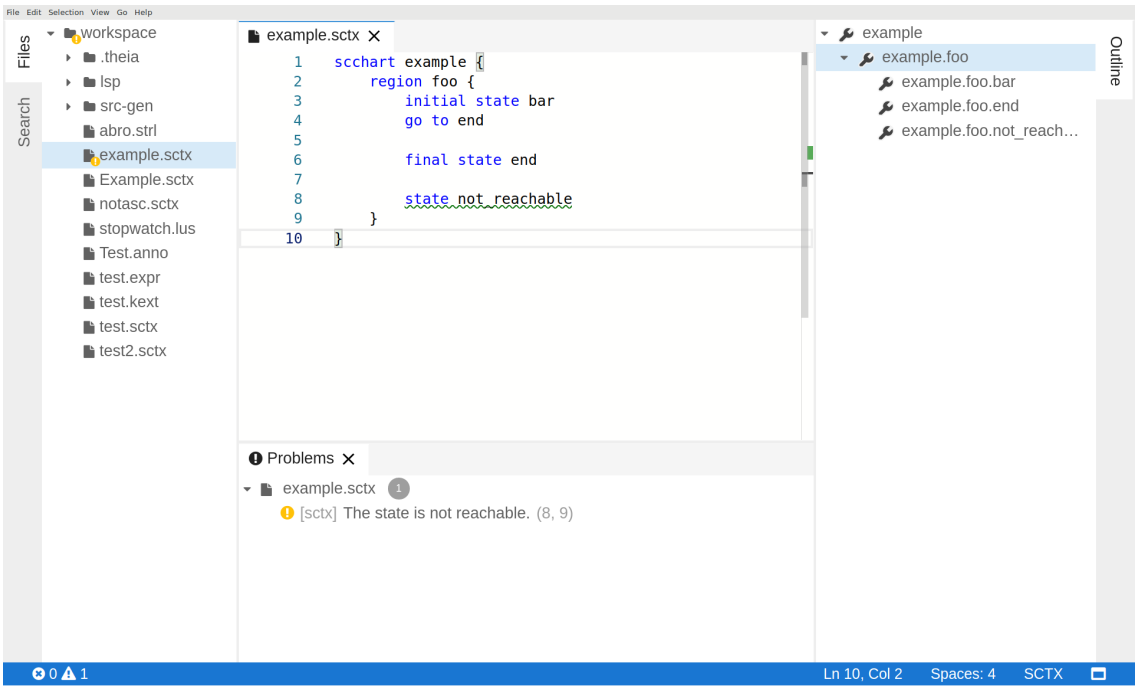
As already mentioned in Section 1.1.1, Theia is an IDE framework developed by TypeFox, which can be started in a browser or as an Electron app, as seen Figure 2.4 and Figure 2.5. Figure 2.4 shows that the browser version and the Electron version of Theia look exactly the same. Figure 2.5 presents how Theia's underlying components communicate. The three different components frontend, backend, and language server can run on separate machines. The backend runs on the system, which holds the file system of the workspace. Theia consists of a frontend, which runs in a browser, and a backend, which can run on any server and communicates with the frontend. The backend can be used to communicate with different language servers to provide rich language features. Figure 2.5a presents the browser version of Theia. The frontend runs in a common browser (e.g. Mozilla Firefox or Chrome). The backend is deployed locally or on a server and starts the application on a specific port, which is accessed by the browser and allows to use the IDE. If the backend runs locally it starts for example on `localhost:3000` to which the frontend connects in the browser. The backend communicates with one of the communication methods mentioned in Section 2.5 with the language server via JSON-RPC. The language server can either run locally or on a different server. Figure 2.5b shows Theia's architecture as an Electron app. The frontend runs in Chromium[14], which is a browser. In this configuration the backend runs locally and starts the application in Chromium. The language server is accessed via stdin/stdout. These three components are bundled in a single application via the Electron framework[15]. The communication with language servers and the whole UI design and concept is not new; Theia takes inspiration from Microsoft's VSCode[16] and adopts many of its features [Eff17].

Theia uses VSCode's features, as seen at the example of the Monaco Editor, the LSP, and the quick

[14] `https://www.chromium.org/Home`
[15] `https://electronjs.org/`
[16] `https://code.visualstudio.com/`

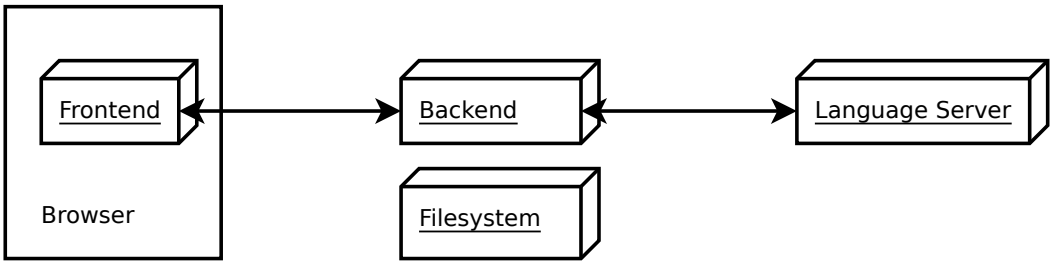


(a) Theia in the browser

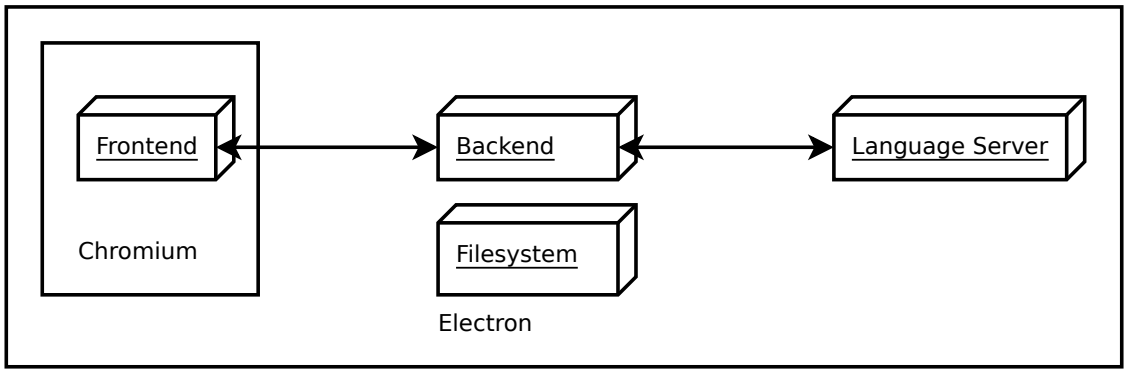


(b) Theia as an Electron app

Figure 2.4. Theia browser and Electron version



(a) Theia in the browser



(b) Theia as Electron app

Figure 2.5. Communication for different Theia products

open widget. Its backend and frontend are rather communicative. Therefore, VSCode itself is not optimized to run in a browser, although it is written in TypeScript (see Section 2.7 for explanation), which is commonly used to program web applications [Eff17].

Theia is easily extensible, since Theia itself consist of several extensions bundled together, as seen in Figure 2.6. Each package holds a separate feature of the IDE framework. Extensions are first class citizens in Theia and allow maximum customization [Eff17]. Because of this, Theia can be seen as a framework for an IDE and not only as an IDE itself. The user itself can decide whether git support, an outline view, or the ability to add new extensions at runtime via the extension system of Theia is needed. This feature makes Theia lightweight and feature rich at the same time. Together with Theia's ability to run in the browser, as seen Figure 2.5a, and as an app, as seen Figure 2.5b, it can run everything everywhere with the right support. A developer can deploy the application in a Docker[17] container, as seen in Figure 2.7. The frontend is accessible via a browser over the internet. The backend including the Theia backend and the language server runs inside a docker container and is started on a server. Docker container can be started for separate users. Docker is a container technology, which

[17] https://www.docker.com/

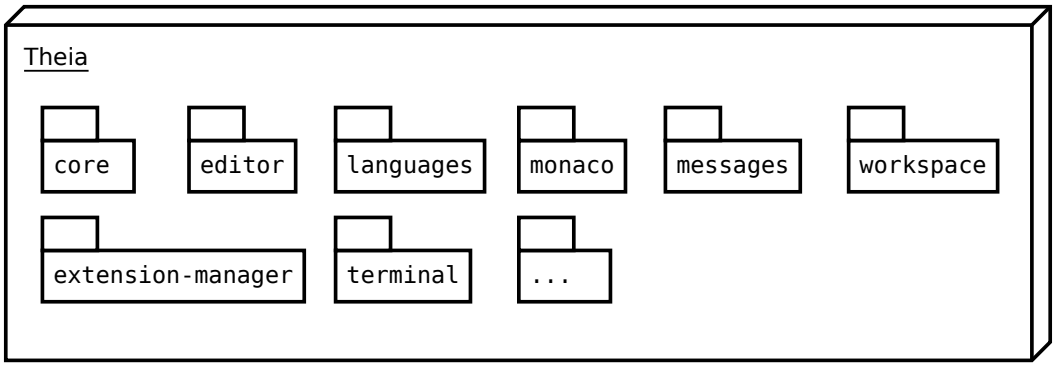


Figure 2.6. Theia package composition

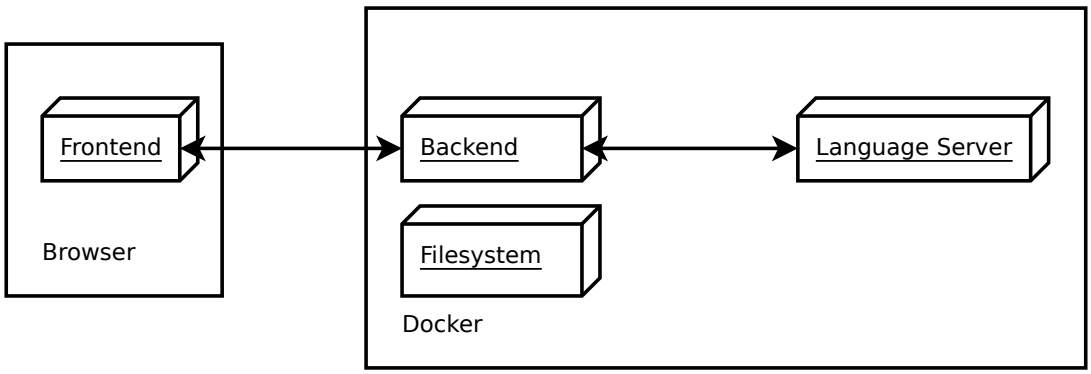


Figure 2.7. Theia started using docker

allows to deploy software in a standardized unit. The application itself can only access the Docker file system, which prevents access to the server itself and encapsulates the application. Containers can be saved as images, which allow to start containers in a specific, predefined configuration and share them with others. Container technology can be used to make Theia available to several users via the browser and provides different workspaces for every user. This technology is able to provide an IDE without any downloads and configuration accessible via a browser, independent of the users OS. The container setup has the potential to be used in teaching for programming courses and for conferences, since no setup time or configuration is needed.

The language server technology is utilized to provide language features for various languages which may implement the LSP. Therefore, Theia can provide rich language features for a wide range of languages. Theia provides many of the desired IDE features presented in Section 2.1, such as content assist, syntax highlighting, linting, and window and workspace management [Tea18a]. On the other hand, Theia is still in development and is not as long in use as other IDEs, which suggests that errors occur more often than in renowned IDEs such as Eclipse. Eclipse has a big user base and is every popular. Many corporations develop for Eclipse.

Theia is still incomplete, but everyone can contribute since Theia is an open-source project. Open-source projects tend to be the more successful, the more modular and innovative the framework or tool itself is [BC06]. Theia is highly modular, since each extension is completely separate from each other.

```
1 var test = "testing"
2 test = 25 // Type '25' is not assignable to type 'string'.
```

Listing 2.3. TypeScript is typed, even if the types are not explicit

It is highly extensible, while delivering a framework to develop IDEs and implementing the current trend in IDE development: the LSP. Theia itself is built using web technologies. Its main language is TypeScript.

2.7 TypeScript

TypeScript is a superset of JavaScript with optional typing. Optional typing means, values can be typed explicitly if needed. Otherwise their type is inferred. This means every JavaScript program is also a TypeScript program, but it may not be a valid program, since static analysis may find possible runtime errors beforehand, as seen in Listing 2.3. The comment "`Type '25' is not assignable to type 'string'`" indicates what error message TypeScript shows statically in a rich TypeScript editor. When the string `"testing"` is assigned to the variable `test`, Typescript infers that `test` must have the type `string`. If the developer later tries to assign a number to test, TypeScript throws a type error. JavaScript, however, behaves differently.

In JavaScript the program in Listing 2.3 produces a runtime error. TypeScript is able to find errors statically or on compile time, which result in a runtime error in JavaScript. This helps to comprehend the error message, since runtime errors are most times not comprehensible, since they are not able to reference the source of the error in the source code and provide no sufficient stacktrace.

This leads to the first major design goal of TypeScript: Statically identify constructs that are likely to be errors [Mic18]. The lack of typing and the error tolerance of JavaScript makes application development and debugging in JavaScript challenging and applications often error prone as evaluated by Mikkonen et al. [MT07]. Mikkonen et al. suggest a more incremental software development approach for JavaScript, which needs rapid prototyping to find mistakes early on. JavaScript is a dynamic language, which allows to add functions and variables on the fly and supports this incremental development approach.

Because of TypeScript's typing and its interface and module system adopted from the scripting language ECMAScript 6[18], also known as JavaScript 6, it is easier to develop large projects with it than with JavaScript. In comparison to JavaScript, TypeScript's optional typing makes it less error prone as the study of Ray et al. suggests, since strong typing seems to avoid more bugs than weak typing [RPF+14]. Ray et al. present that TypeScript has fewer bugfixes in relation to the overall commits from 2011 to 2014 with 14,987 commits and 2,443 bug fixing commits. JavaScript seems to be more error prone and has more bugfixes from 2002 to 2014 with 118,318 commits and 39,250 bug fixing commits. JavaScript has on average more errors than TypeScript regarding application, code analyzer, database, framework, library, middleware, and overall, as seen in Figure 2.8. The figure suggests that TypeScript produces less bugs than other languages. If the average of all languages is used as a reference, TypeScript is less likely to introduce security issues or concurrency issues than JavaScript [RPF+14]. Ray et al. conclude that programming languages in general have no affinity to bugs. They rather have an affinity to a specific type of bug. It has also be taken into account that TypeScript is used as a weakly typed language in 50% of variable assignments, which diminishes the results of the study by Ray et al. [RPF+14].

[18] `http://es6-features.org`

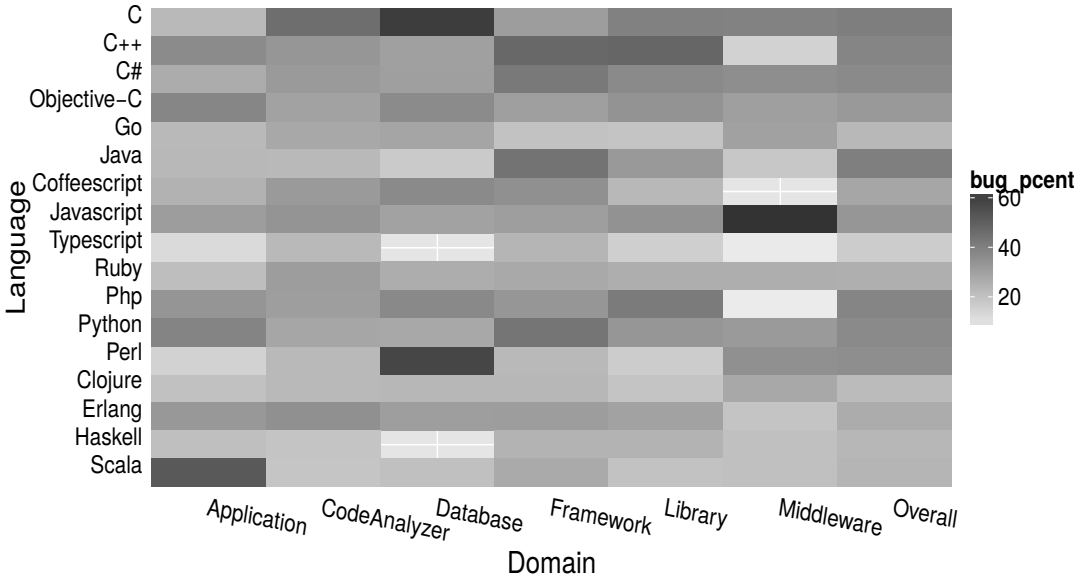


Figure 2.8. Variation of defect proneness of languages for a given domain [RPF+14]

```
class Pair {
   first: string
   second: string
}

function infer(): void {
   let pair: Pair = {first: "first", second: "second"}
}
```

Listing 2.4. TypeScript inference is able to infer a class from its attributes

Type correctness is not the same as program correctness, but often a type mismatch indicates undesired or unplanned program behavior or simply a non conventional way to solve a problem. Therefore, static type checking can be used to bugs early on and can help to achieve better programs. In TypeScript, every value is typed implicitly by default. If TypeScript cannot infer, which type should be used, `any` is used, which is the most general type. Ordinarily, TypeScript type inference proceeds "bottom-up". The different attributes of a structure are evaluated first. A structure with a specific set of attributes is automatically cast to the class resembling the structure, if all attributes are inferred. Listing 2.4 shows that the object consisting of the attributes `first` and `second` does not have to be cast to assign it to a `Pair`. The "bottom up" inference infers the attributes first. Since a string is assigned to the attributes `first` and `second`, it is evaluated that the object assigned to `pair` is indeed a `Pair`. This allows developers to have typing and still be flexible in their programming ability.

From TypeScript's goals and non goals the following conclusions can be drawn for TypeScript development [Mic18]:

The main goal is to find errors statically early on in development and not only on runtime. Static code analysis makes testing simpler and the developer does not have to use as much time to manually test the application for errors if they are found statically. Testing can be rather time consuming for

bigger applications, since the build of the product for runtime tests may take considerable time and slows down the development process.

A class system allows to reuse code more easily and makes it more maintainable. Moreover, the `this` operator can be used as in many other languages, where it refers to the object itself and not the current scope, by using bindings or the local fat arrow [Tea16]. The local fat arrow `=>` is used to define functions and changes the way `this` is evaluated, since the local fat arrow retains the scope of the caller, which allows to use `this` in the same way as in many other languages.

TypeScript always compiles to plain and clean JavaScript to be backwards compatible and does not break with the initial TypeScript implementation. Furthermore, a JavaScript program, which is valid at runtime, is a valid TypeScript program. However, expression level syntax should be avoided while programming in TypeScript.

Another goal is to use TypeScript as a cross-platform development tool, since it is based on JavaScript, which is browser and not OS-dependent (as long as specific libraries are not used). It is intended to use the intention of programmers and the already existing behavior of JavaScript to develop new features, regardless of features in different languages. This allows not to break with the community of JavaScript developers, while still introducing necessary features for simpler and less error prone development.

Moreover, excessive code optimization is avoided. Instead idiomatic and recognizable JavaScript is emitted. Simplicity and easy usage does also apply to the type system. The type system is no hindrance and does not slow down development, but rather helps the developer to identify problems.

Furthermore, the existing libraries of JavaScript are TypeScript compatible. It is not necessary to provide new ones for TypeScript development. In summary, the overall editing experience of JavaScript is not changed in TypeScript, but is made simpler and helps avoid common errors and mistakes, while keeping the syntactic noise of TypeScript as low as possible with the optional static typing.

2.7.1 Yarn and npm

TypeScript projects are configured using the so called `project.json`. This file is used to specify dependencies to other packages and define scripts to run and build the application. It can also be used to configure an Electron app and set author and license for that application. Dependencies can either be development dependencies (e.g. the linter for TypeScript tslint[19]) or dependencies of the product (e.g. `@theia/core`). It is possible to specify a version or a tag (e.g. `next` to use the newest running version) of a package. The `package.json` defines the setup of the project and is part of any organized TypeScript project.

Npm is a software registry for JavaScript packages[20]. It hosts these packages and provides packages listed in the `package.json` as dependencies. Since one of the main goals of TypeScript is not the break with JavaScript and be compatible with it, TypeScript is able to use these JavaScript libraries. Npm allows to share code for open-source packages, but can be used for private development as well. The `package.json` mentioned earlier is used to specify the used packages, but does not guarantee, which exact version is used. The package manager yarn can be utilized to achieve this. Together with the `package.json` yarn generates a yarn.lock file, which can be shared with other developers and checked into a version management system to ensure that the same versions of libraries are used.

[19] `https://palantir.github.io/tslint/`

[20] `https://docs.npmjs.com/`

Chapter 3

Related Work

Migration can take various forms and no migration project is the same, but they often share different aspects and approaches to the development problems. Migration projects can be used to learn from the taken efforts, but the reader has to bear in mind that not every aspect can be applied to every project.

Project migrations are presented to show similarities to other projects and to show the difference to the current one. Different Cloud IDEs are presented and compared to the used Theia framework including the reference project Yangster for the programming language YANG. In parallel to this project a diagram extension for KEITH is developed to add new functionality to the product. In this chapter the diagram extension project is delimited from this work. Language features can be added via the LSP and various language servers and LSP supporting IDEs do already exist. The Java language server is presented as an example for a language server. The LSP is not the only movement to develop language and IDE-independent protocols to provide language features. There are several other protocols which try to solve the *m* IDEs *n* languages problem, as seen at the example of Monto [KPE16], Microsoft's DebugProtocol[1], and the Kómpos protocol [MTA+17].

3.1 Project Migration

Several migration projects already exist and use different motivations and patterns to migrate, since they are applied to different contexts. Nevertheless, their ideas and motivation sometimes comply with other projects and can be used to achieve general knowledge about obstacles and goals of migration projects. To show this, some migration projects are presented here.

3.1.1 ARNO Project

The ARNO Project by Teppe is an experience report about the migration of the travel agency system Amadeus. Amadeus is used by the majority of travel agencies in Germany [Tep09]. What the Amadeus system can access is visualized in Figure 3.1. The Amadeus system allows to book travels with various hotel chains, airlines, and other transportation providers. Teppe focuses on the picking of a migration strategy and how cost and time influence this decision. The goal of this migration is to migrate from a mainframe platform to UNIX, since it promises new development tools.

Teppe identifies no new features at high cost as the main problem of migration projects. This makes the risk of a migration difficult to argue about. Moreover, Teppe identifies the lack of programmers in the language of the legacy system and the emerging of new promising development tools as the reason of many migration projects. Communication is key while migrating. Users and stakeholders should be included in the process. Communication with stakeholders is necessary to evaluate what features are needed and in which direction the product should be evolving. Three migration strategies are evaluated to plan the migration process:

[1] https://github.com/Microsoft/vscode-debugadapter-node

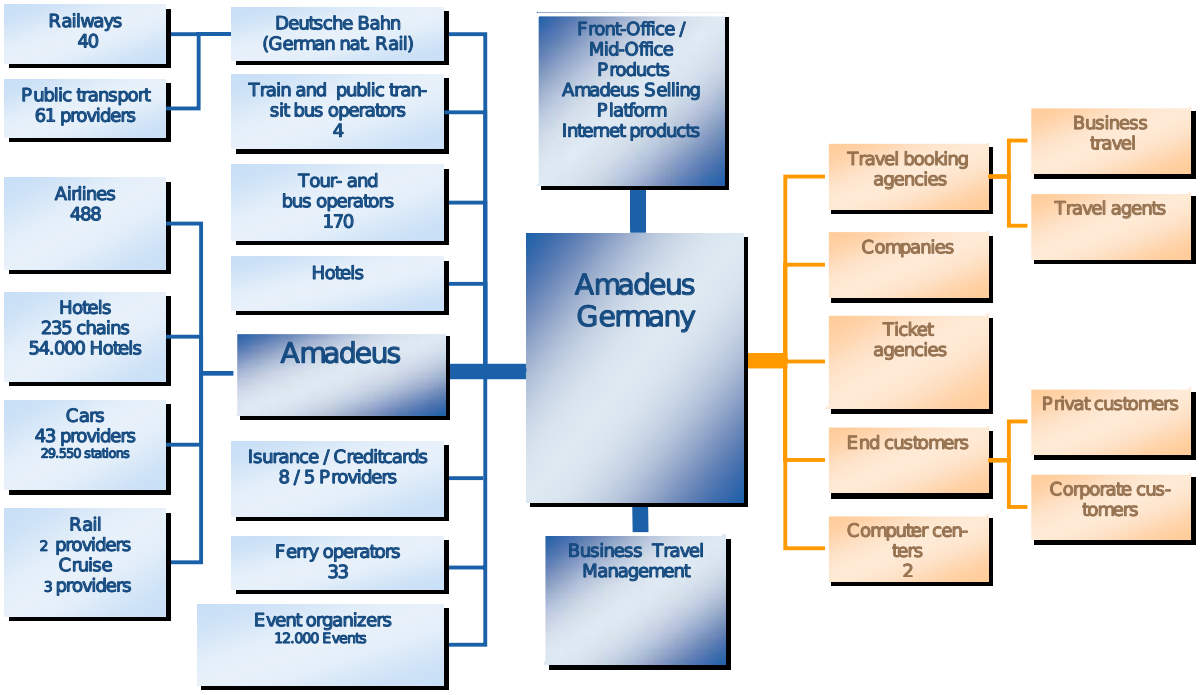


Figure 3.1. The Amadeus Germany System [Tep09]

Reimplementation Reimplementation requires to reimplement the whole application using the new framework. This is the most costly strategy and is likely to add the same errors as the legacy product in its earlier stages, which makes the strategy not relevant for the ARNO project.

Cross-compiler A cross-compiler compiles directly to the new UNIX environment. It allows to still program in the legacy language. It requires new programmers to learn a relatively old language.

Translator A translator to automatically migrate from one language to another is optimal for the ARNO project, since it allows step-by-step conversion of the code, while testing the correctness of the translation. This is superior to developing a cross-compiler and still develop in the legacy language, since that requires new programmers to learn the old language. Moreover, the translation strategy is cheaper than developing a cross-compiler. Furthermore, errors in the cross-compiler are difficult to find, since a step-by-step compilation and verification is not possible in this case.

Another important factor while migrating is time. The Amadeus system is a living system. Changes and corrections are made all the time and have to be adopted into the legacy system while migrating. The migration mechanism should allow to still build the legacy system and update step-by-step. The goal is to migrate as fast as possible to avoid changes that have to be applied to the new system. Teppe concludes that the staff has to be trained to migrate and to use the new frameworks and technology [Tep09]. Furthermore, code is automatically generated to avoid stagnation of the project while migrating and avoid previously made errors. Code can be generated from the living legacy system, in which these errors are already fixed.

Since a language is migrated in the ARNO project, not everything is applicable to the migration of KIELER, although many core ideas can be carried over. The reasons for migration are in both cases partly the same. New technology, which might make the tool more maintainable and separate different

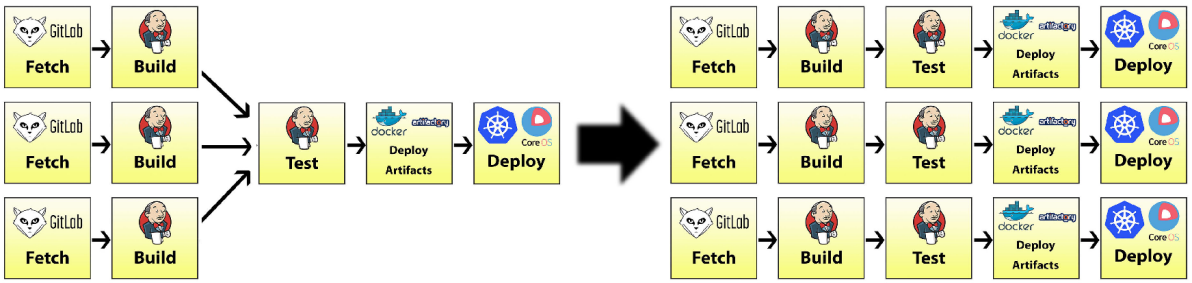


Figure 3.2. Continuous delivery using microservices [BHJ16]

concerns is desired for both projects. Step-by-step migration while testing the different features is necessary and guarantees that the legacy product is still runnable. Time is not this much of a factor in KEITH and constantly developed KIELER plugins are regularly merged, since these plugins are reused for the language server and its compilation capability.

3.1.2 Backtory

In the experience report by Balalaie et al. on the migration of Backtory to a cloud-native microservice architecture the main focus lies on the migration reasons and the migration goal [BHJ16]. Furthermore, it is discussed how continuous integration, continuous delivery and migration from a monolithic product to microservices can be achieved. Used migration patterns are presented. As migration reasons the need for reusability, decentralized data governance, automated deployment, and built-in scalability is mentioned together with the fear of technology lock-in. The migration is done step-by-step for the different services, while testing everything via a continuous integration pipeline using Jenkins[2], Artifactory[3], and Docker Registry[4]. The goal is to achieve continuous delivery with independent source code, configuration, and environment configuration, which allows to develop and deploy each of the components independently, as seen in Figure 3.2. The figure shows that each component has its own development circle and is deployed separately. Previously the components were built separately but tested and deployed together. Now each component is developed independent of each other. Using this, the developer is able to change the configuration without changing or recompiling the source code. Continuous monitoring is used to find bottlenecks in the microservice architecture and find performance anomalies with statistical models trained via normal monitoring data.

To continue effective development, the team structures have to be changed to enable DevOps. DevOps is a combination of development and operation. DevOps teams automate and monitor software releases during its whole life circle. As seen in Figure 3.3, teams shall be able to develop, deploy and maintain a single component all by themselves. Before the migration the teams had specific tasks and were only responsible for their development phase. In the microservice world a team is responsible for their whole service. DevOps includes development, quality assurance, and operation. Therefore, every team should have a member who is proficient in it. This results in mixing the old homogeneous teams into heterogeneous teams, which have experts for all different development cycles. Balalaie et al. add the following general remarks to microservice migration: The deployment of the development environment is difficult, since dependent services still exist. Interfaces or service contracts are likely to change if many services are changed. Domain experts and experts in microservice development are

[2] https://jenkins.io/
[3] https://jfrog.com/artifactory/
[4] https://docs.docker.com/registry/

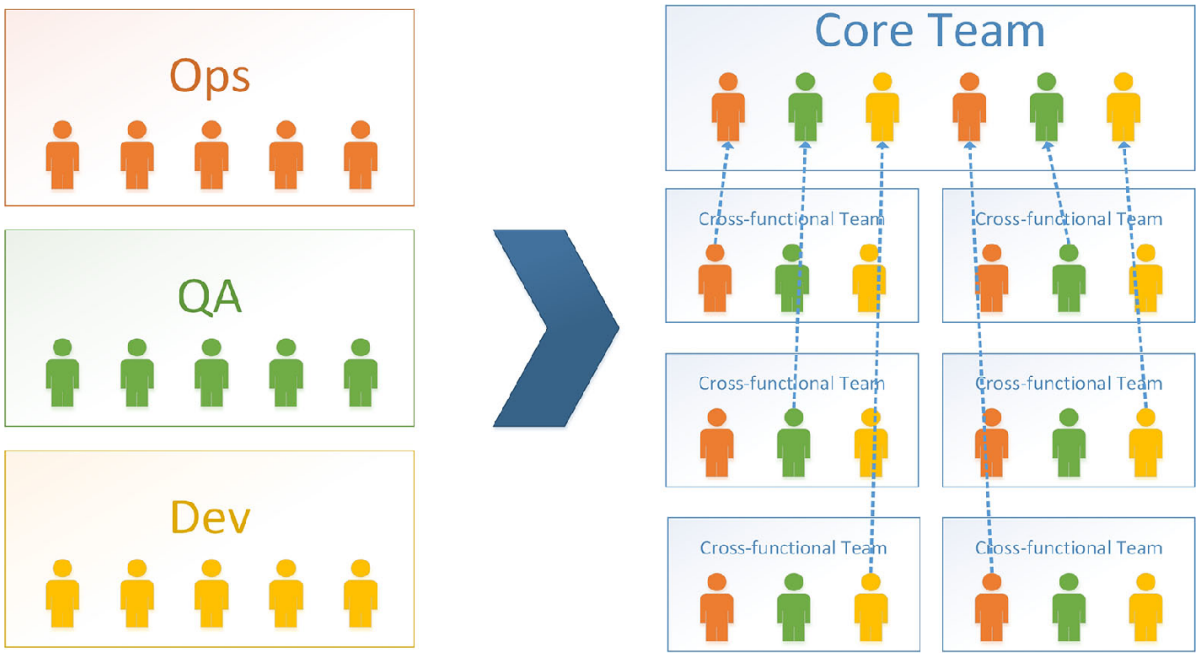


Figure 3.3. Team restructuring for DevOps [BHJ16]

needed for a smooth migration. Development templates for polyglot persistence are helpful and ease the development process. Microservices do not make the system less complex, since the flexibility and concurrency adds further difficulties.

Even though the migration of KIELER intends to separate different features of an IDE, it is not always comparable to a migrating to microservices. The team size and project size is not comparable, therefore the team composition aspect, the continuous delivery aspects, and the load balancing are not applicable to this scenario. Some ideas are present in both migration projects. In both cases the migration happened, because new technology should be adopted. Also, step-by-step migration is desired to be able to find errors in the new implementation. Furthermore, automation of migration steps is desired whenever possible and the lessons learned about decomposing a monolith are applicable.

3.1.3 Migration using Model-Driven Engineering

Fleurey et al. introduce model-driven engineering to migrate software projects [FBB+07]. The goal of the project is to migrate a large scale banking system from the web3[5] platform Mainframe[6] to Java 2 Platform, Enterprise Edition (J2EE)[7]. One of the main reasons for the migration are the quickly changing development techniques, paradigms, platforms, and the desire to adopt them. Moreover, pressure of users or unification of software systems when merging companies are driving causes. The migration is done via generating a code model and reverse engineering the pivot language meta-model from it, which is used to generate a platform specific model, as seen in Figure 3.4.

[5] https://web3js.readthedocs.io/en/1.0/
[6] https://mainframe.com/
[7] https://www.oracle.com/technetwork/java/javaee/appmodel-135059.html

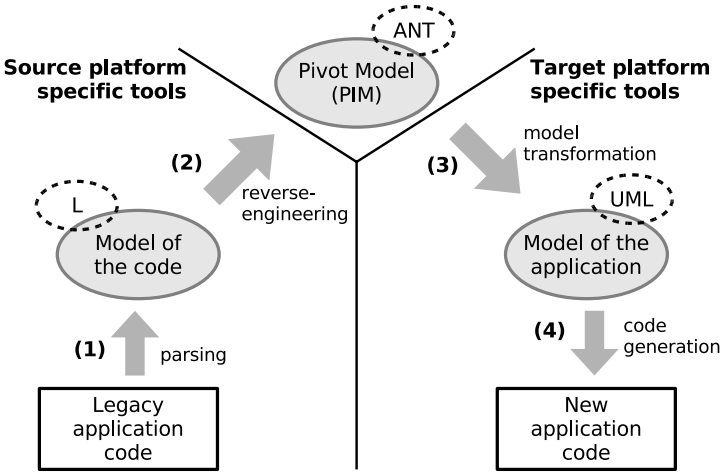


Figure 3.4. Model-driven migration principle [FBB+07]

The platform specific model is utilized to generate the new application. To be cost efficient the process is automated if applicable. Not everything can be automated, since one has to be sure that the new implementation is maintainable and does not repeat the mistakes and disadvantages of the legacy application. The goal is to make it more reliable, efficient, maintainable, and extensible. The migration process may take its time at the beginning, but after most of the application is migrated automatically only a few manual migration tasks remain. The approach of using model-driven engineering to migrate is cost and time efficient compared to the reimplementation approach. The advantages of this model-driven migration are the ability to reuse developed migration tools for similar projects and the cost efficiency. On the contrary it needs more time, until the customer can see any results, since the first step is to develop a model translation tool and to execute all other preliminary tasks. Feurey et al. suggest that the customers IT department should be integrated into the development process to combat this issue and to have experts, which may evaluate the progress. Furthermore, the cost of testing and ensuring that the new system works as good as the legacy system can be high, since it is done manually in many scenarios.

As seen on the other migration projects, not everything can be applied to the migration of an IDE. The half-automatic development seems to be the option of choice if specific tools already exist. Developing tools for these migration strategies in the case of KIELER might take longer than reimplementing the application. However, developing tools for the migration might seem helpful, since part of the architecture is changed. Such tools already exist. There are already tools to migrate from the old to the new Xtext generator framework in the mwe2 workflow. Therefore, model-driven engineering cannot be applied to the migration of KIELER.

3.1.4 Dublo Pattern

The Dublo architecture pattern is a migration strategy for monolithic business information systems to multi-tier architectures separated into interface, business logic, and data persistence by Hasselbring et al. [HRJ+04]. Hasselbring et al. focus on reuse of existing code and a smooth migration process, while the legacy product is still in use.

The migration pattern is driven by the fact that the legacy system cannot be disposed, since business has to go on during the migration. Documentation is often only present in the source code of the

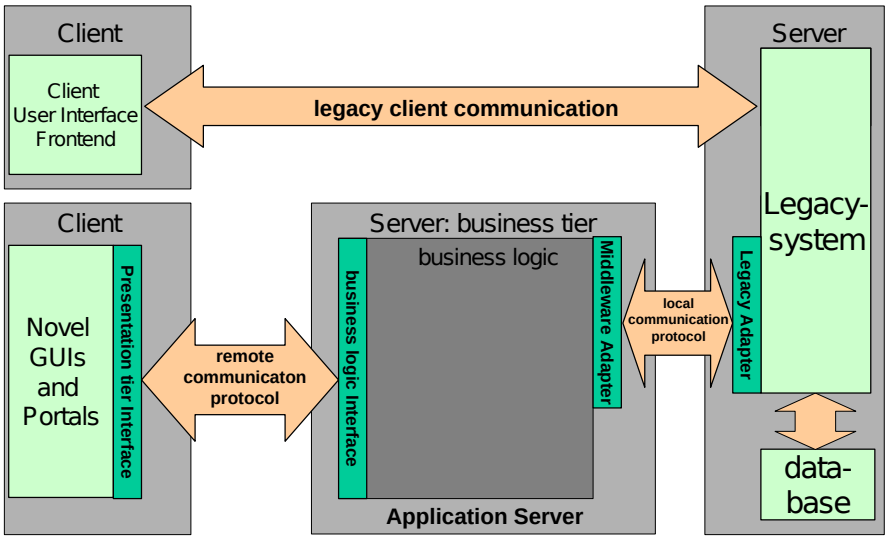


Figure 3.5. Structural view on the Doublo pattern [HRJ+04]

legacy system. Moreover, development of the legacy tool still has to go on while migrating. The goal is to achieve separation of concerns between business logic and UI which allows easier redevelopment of subsystems if needed. Separation of concerns is not always achieved in legacy systems, since often only one development language, which can support every aspect of the implementation, is used. Adapters enable to reuse part of the legacy system, while migrating and enable to run both systems in parallel, as seen in Figure 3.5. Two different clients run in parallel using different communication protocols. The adapters are used if a new component has to communicate with a legacy component, for example with the server. Furthermore, it is concluded that step-by-step development while migrating is more efficient than a complete redevelopment of the system, since testing can be done for single components and the rest of the system can be added as a mockup. Therefore, step-by-step development is the approach of choice and allows to reuse most of the business logic, while still being able to build the legacy system in parallel. Hasselbring et al. conclude that the degree of reuse of legacy code is limited by the context and is project specific. The Dublo pattern itself results in duplicate code, since the legacy system and the new system run in parallel with adapters to be able to interchange them in a living system. Furthermore, the performance is reduced, but flexibility and maintainability are increased.

The desire to still run and build the legacy system can be applied to the migration of an IDE, since at the beginning not all features are present in the new tool. Some users (i.e. the ones that use KIELER for their research) are not willing to switch to the new system. The problem of separation of concerns can be applied too, since the language server represents part of the business logic and the Theia frontend represents the UI. These two components communicate trough a common protocol: the LSP. This is similar to the interface between business information components.

3.1.5 General remarks

In conclusion, the different migration projects are not fully applicable to the presented migration scenario. However, their core ideas and motivations often match the ideas and motivation for migrating KIELER. They aim for separation of concerns, try to adopt new technology while reusing code, develop

step-by-step while testing the different components, and try to automate processes whenever possible. Even problems such as a smooth transition, development of the legacy system, advanced training in new technology (i.e. web technologies), and integration into the existing build process are discussed and their solutions can be partly applied.

3.2 Cloud IDEs

There are several implementations of cloud IDEs, which try to show the advantages of web technologies such as container-based development, lightweight client side IDEs, and separation of concerns between editor UI and language features. Moreover, automatic distribution and updating of an application are a trend, as seen at the example of Eclipse Che.

Many cloud IDEs suffer from the same problem: The IDE implements too few languages and is therefore used by too few developers, which leads to little support and few updates [LNK+12; WLK+11]. Often, time is spent to develop language support although language support already exists in another IDE.

3.2.1 CEclipse

CEclipse is an online IDE developed by Wu et al. used to solve the three major problems with cloud IDEs [WLK+11]: Function implementation, security, and advanced utilization. Wu et al. suggest that these problems are solved by service composition, program behavior analysis, and program behavior mining. As advantages of cloud IDEs the platform-independence and the easy to set up environment are mentioned, since the developer only needs a browser and an internet connection to work. Furthermore, online IDEs are suitable for collaborative development, as seen in Google Docs[8]. While CEclipse was developed, most cloud IDEs did not support rich editing features such as Eclipse. Moreover, these IDEs did not consider security and the mining of user data to help novice programmers by optimizing the overall editing experience. Wu et al. describe how one can migrate the Eclipse like functions to an online technology and how to compose the services for this task. Online IDEs also have new obstacles. The user is not allowed to have full access to the server on which the IDE is deployed. Static and dynamic program analysis are suggested to identify risky program behavior, which could lead to security issues. Therefore, file operations are banned, which might harm the system. Some APIs, which could harm the system (for example execution of files on the server), are banned and resource consumption is restricted to prevent an unavailable IDE because of attacks or mistakes.

These problems are not in the scope of this migration project, but are to consider for future work on KEITH. CEclipse is not ideal as a migration target. The LSP is not considered for CEclipse since it was developed after the development of CEclipse and was therefore not available. CEclipse uses a client server concept, but fails to deliver a protocol as general and extensible as the LSP. Therefore, CEclipse does most likely not support SCCharts.

3.2.2 Eclipse Che

Eclipse Che[9] is a partly open-source developer workspace server and cloud IDE, built for teams and organizations. It was released in 2016 and was one of the first IDEs to fully support containerized development workspaces and IDE features such as debugging, refactoring, and content assist [Mic17].

[8] https://www.google.com/docs/about/

[9] https://www.eclipse.org/che/

The IDE supports portable workspaces, which can be managed with container images. Eclipse Che can be integrated into Kubernetis[10] and OpenShift[11] to scale vertically and horizontally [Lor18]. This enables Eclispe Che to support multi-user management for workspaces and team development. The workspaces are collaborative and can be saved as snapshots, while having a workspace agent to provide additional features such as monitoring. User management enables to control user permissions in workspaces for team development. Teams should be able to organize, deploy, and communicate via Eclipse Che. As most IDEs it provides git support, debugging features, and a plugin framework. Moreover, it enables DevOps teams to monitor the status of deployed machines. Eclipse Che is able to mimic production environments to allow to test in a genuine working environment. Overall it is designed to minimize the effort to set up development environments, since the IDE itself and the workspaces can be saved as container images, shared, and restarted in seconds [Ben17]. Eclipse Che supports C++, Java, JavaScript, PHP, Python, Ruby, SQL, and has adopted the LSP, which allows to support more languages. Currently Eclipse Che is working on a Theia integration, which allows to use Theia as an alternative IDE in Eclipse Che [TT17].

Although Eclipse Che is a modern IDE, Theia is chosen as a target IDE for migration. The reference project Yangster presented in Section 3.2.4 does use Theia as well. Theia itself seems to be more flexible in its setup methods and in its API. Xtext, which is already used to generate language support for custom DSLs, and Theia are both maintained by TypeFox[12], which promises support for both technologies working together. Nevertheless, since a language server for SCCharts is developed, an integration into Eclipse Che could be part of future work to bring SCCharts development in more IDEs.

3.2.3 CoreD

CoreD by Lautamaki et al. is a cloud IDE for Java development, which provides not only syntax highlighting, content assist, and error and warning annotations, but also compilation [LNK+12]. Lautamaki et al. criticize that most cloud IDEs do only support syntax highlighting and indentation, but are missing all rich editing features desktop IDEs provide.

CoreD can provide automatic distribution, installation, updating, and independence of the development environment. It uses the Vaadin framework[13], the Ace web editor[14], and the Java Development Kit (JDK) to provide a rich editing environment for Java, while still being extensible for other languages. All development features are implemented in separate and replaceable components. The tool itself works similar to the LSP implementation in various IDEs since the client requests highlighting or similar services from the server. However, the server does not use an interface to deal with most known programming languages and is Java oriented. CoreD allows multiple users in one workspace and implements a protocol for collaborative editing as well as a feature to lock certain parts of the program to edit them alone. All users can make notes on code and set code markers to allow communication while editing. Every part of this implementation is interchangeable, which allows to adopt new technologies more easily and allows to exchange the code editor.

This project does not use the language server technology, but adopts several concepts from it such as separation of concerns, easier adaptability of new languages, and classifies editors, which do not provide rich language features, as problematic for software development. The LSP however enables Theia to support new languages with less effort and the whole extension infrastructure makes it even more configurable. Moreover, Theia can also be used as a desktop application.

[10] https://kubernetes.io/
[11] https://www.openshift.com/
[12] https://typefox.io/
[13] https://vaadin.com/
[14] https://ace.c9.io/

```
@JsonSegment('diagram')
public interface DiagramEndpoint extends Consumer {
   @JsonNotification
   void accept(ActionMessage actionMessage);
}
```

Listing 3.1. Diagram extension of LSP

3.2.4 Yangster

Yangster can be seen as an example project, since there has been a successful implementation of language support for different IDEs while reusing most parts of the language plugin in form of a language server [Köh17c]. Yangster can be seen as a guideline for building an SCCharts language server, since YANG also supports diagram generation by extending the existing LSP. On the long run this will also be necessary for KEITH, since future work will be to integrate the diagram view and creation.

YANG has shown that it is possible to deliver IDE support for multiple IDEs with minimal effort while reusing most of the code [Köh17c].[15] The creators of the YANG language server are confident that a YANG extension for any IDE can be created with minimal effort if the target IDE supports the LSP. Since this project is quite successful, a migration to a language server and a client side IDE sounds promising. The only difference to KEITH is that a full migration to Theia needs not only language features such as content-assist, hovers, jump to definition, finding references, and finding diagnostics, but also compilation, result navigation, and simulation. This can be managed by extending the LSP, as seen in Listing 3.1 at the example of a diagram server extension [Köh17b; Köh17a]. The `DiagramEndpoint` interface is used to add a JSON-RPC method to the language server, which adds additional services to the LSP. This allows the Theia application to communicate via actions with the diagram server. LSP extensions can be used for KEITH to compile models.

3.3 Diagram extension for KEITH

Parallel to this project, Rentz develops a diagram extension for KEITH by leveraging the sprotty framework[16] to generate diagrams for grammars supported by diagram syntheses in KIELER [Ren18].

The diagram extension project reuses code from the KIELER tool and tries to be compatible with the implementation of diagrams in KIELER, but uses sprotty's functionality for diagram generation on the Theia client. Furthermore, it defines a language server extension for the *show diagram* use case, similar to the diagram extension used in the Yangster project in Section 3.2.4. KEITH and the diagram extension use the same technology stack, but have different focuses. The project of Rentz tries to migrate to a new diagram frontend. Rentz reuses the existing Theia framework in its development setup, while the project discussed here focuses on build setup, language server integration, project structure, and compilation of SCCharts, as seen in Section 2.3.1. Furthermore, the KEITH tool is developed and delivered as an Electron app, while the diagram extension only works in the development setup. Together, these two projects are intended to migrate KIELER's key functionality to web technologies and to further develop KIELER by separating UI and functionality. Moreover, Java or — in the case of Eclipse — SWT is not a desired framework to build UIs; web technologies seem to be the right approach to achieve a maintainable project without hindering expressiveness and are therefore used in both projects.

[15] https://github.com/theia-ide/yang-lsp/blob/master/README.md#release-engineering

[16] https://github.com/theia-ide/sprotty

3.4 Alternative LSP projects

The LSP is adopted by several IDEs, such as Eclipse Che, Eclipse LSP4E[17], IntelliJ/JetBrains IDEs, Vim[18], VSCode, MS Monaco Editor[19], Atom, Emacs[20], Sublime[21], Theia, and more [Tea18a]. Many IDEs seem to have LSP support and are therefore able to provide language features for every language server that might be implemented. The developing LSP infrastructure enables developers to easily provide language support for new languages. Only a language server is needed to add a new language to on IDE which supports the LSP.

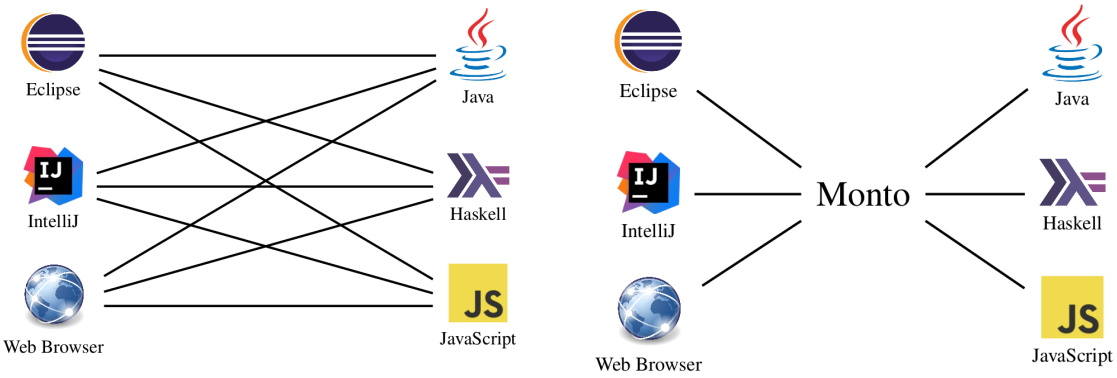
One example for a language server implementation is the Java language server[22]. It is written in Java and can provide language features for the same language. Such features are already implemented in several IDEs, such as Eclipse and many others. The language server depends on Eclipse LSP4E[23], Eclipse Java development tools (JDT)[24], and more plugins to provide support for the LSP, Java, Maven[25], and Gradle[26]. As the KEITH language server, the Java language server can connect via socket or stdin/stdout to enable debugging and a connection to a remote language server. There are already several client implementations for this server including Theia, Neovim[27], Emacs, Atom, and VSCode. Such a client implementation needs to implement Gradle and Maven support as well as compilation capabilities.

3.5 Monto

Monto is a framework by Keidel et al. for an intermediary representation and architecture to provide language and IDE-independent services for programming languages [KPE16]. It is used to find a solution with linear complexity to the m IDEs and n languages portability problem [KPE16]. How this can be achieved is presented in Figure 3.6. Without the protocol each IDE needs an own language implementation. This results in nine implementations for three IDEs and three languages. If Monto is used, only six implementation — three for the IDEs and three for the languages — are needed. Monto is used as a middleware server. One language needs only one server implementation and an IDE needs only to support the protocol of the Monto broker. Keidel et al. suggest that these features of Monto make it easier to add new language services, since fewer implementation have to be made. Monto works via a central broker, which manages the communication between IDEs and language plugins. Monto supports incremental update and is stateless to decrease the performance penalty of the middleware.

The disadvantage of Monto is that fewer IDEs support the protocol than the LSP. Moreover, the desired target IDE Theia has no Monto integration. Monto support cannot be generated by Xtext, but Xtext is able to generate a language server. The Xtext support makes the LSP far more easy to adopt than Monto. Therefore, Monto is not implemented here and the LSP is used instead to provide language features for KEITH.

[17] https://github.com/eclipse/lsp4j
[18] https://www.vim.org/
[19] https://microsoft.github.io/monaco-editor/
[20] https://www.gnu.org/software/emacs/
[21] https://www.sublimetext.com/
[22] https://github.com/eclipse/eclipse.jdt.ls
[23] https://projects.eclipse.org/projects/technology.lsp4e
[24] http://www.eclipse.org/jdt/
[25] https://maven.apache.org/
[26] https://gradle.org/
[27] https://neovim.io/



(a) *m* IDEs, *n* languages without Monto [KPE16]

(b) *m* IDEs, *n* languages with Monto [KPE16]

Figure 3.6. The *m* IDEs *n* languages portability problem by Keidel et al. [KPE16]

3.6 Debugging Protocols

Microsoft's open-source DebugProtocol[28] is fairly similar to the LSP. It does provide a separation of client and server such as the LSP and is therefore reusable for several IDEs. The DebugProtocol allows to write a debugging server in any desired language, which enables the developer to choose a language that is most proficient in its job and allows tool developers to adopt new languages more easily [M17]. The differences between the debugging protocol and the LSP are the following: The debugging protocol server stores all the state, while the LSP only stores the index of the files. The debugging protocol does not support cancelable request and is not JSON-RPC 2.0 compatible [M17]. The goal is to use the debugging protocol to utilize the full debugging capability in form of launching programs, a view for processes and threads, stack traces, a run control (to step, continue, and run), breakpoint support, variables, source code lookup, support of stdin/stdout, console support, and expressions. These features are sufficient to adopt the DebugProtocol for debugging of SCCharts if the protocol is integrated into Theia.

Another debugging protocol is the Kómpos protocol proposed by Marr et al. [MTA+17]. It is mainly used for concurrent debugging in multiple concurrency paradigms. Marr et al. state that it supports different breakpoints and stepping operations to adopt different concurrency models such as message-passing and shared memory, while still being able to handle the models the same way. The protocol allows to visualize and debug different kinds of concurrency with an agnostic debugger, even in real time. However, this protocol is not needed for debugging of SCCharts, since time is handled differently. Time is divided into discrete ticks, therefore no complicated notion of concurrency is needed. Hence, this approach cannot be applied without an overhead, which outweighs its usefulness. Therefore, Kómpos is not considered for KEITH.

[28] https://github.com/Microsoft/vscode-debugadapter-node

Chapter 4

Migration from Eclipse to Web Technologies

This section presents the main concept and design decisions for an IDE migration from Eclipse to web technologies. First, the reasons to migrate and different strategies are presented and evaluated. Moreover, common obstacles of a migration, the problem of OS-independence, and the need for migration of knowledge are discussed. The usage of the resulting product of the migration for future projects is presented, together with general problems while migrating. Since the UI is part of the migration, the UI concepts of Eclipse and modern IDEs, which use web technologies, are compared.

4.1 Migration Strategy Discussion

An IDE migration from Eclipse to web technologies using a client server architecture has to be carefully planed. A migration strategy should include a solution for common migration problems. Furthermore, OS support and knowledge migration have to be kept in mind while migrating. The resulting product should be more maintainable, scalable, and reactive than the previous implementation.

4.1.1 Migration Reasons

There are several reasons to migrate to a different platform. New technology has to be adopted in the quickly changing world of software development [Tep09; FBB+07; BHJ16]. Often new technologies are needed because of dependencies to other projects or just to avoid technological lock-in by migrating to more flexible frameworks [BHJ16]. Migration should lead to a more maintainable software. Therefore, separation of concerns[1] is another migration reason [HRJ+04; BHJ16]. If one chooses to migrate, a different framework or language is adopted in most cases. Software migration leads to migration of knowledge, since developers are not familiar with new technologies or languages [Tep09]. Migration of knowledge in new developers can also be a reason to migrate. Programming languages such as the common business-oriented language (COBOL) are nowadays only used to maintain legacy applications. New software developers are most times not proficient in COBOL. A migration to a new language promises less costs for training new employees, since it is difficult to use a language on the long run if no expert developers know it.

4.1.2 Migration Strategy

Whatever reasons motivate the migration, time and money are important factors in a migration project. How much money and time have to be invested in a migration project is influenced by the migration strategy. Three migration strategies are evaluated in this context:

Reimplementation A full reimplementation involves the development of a completely new product without reusing any sources. This migration strategy is also evaluated in its usage for the project migrations in Section 3.1.1 and Section 3.1.4.

[1] https://deviq.com/separation-of-concerns/

Translator The translator strategy involves the development of a translation tool for the source code of the legacy tool. This allows a step-by-step migration and verification of the translated components.

Reuse backend The backend implementation to provide language features already exists in the Eclipse implementation. To avoid duplicate code and minimize the reimplementation effort, the backend is reused and used as a service via a server component. The frontend is reimplemented using a new IDE framework and web technologies.

As also mentioned by Teppe and Hasselbring et al., the full reimplementation approach is the most costly and time consuming one [Tep09; HRJ+04]. Moreover, a full reimplementation is not necessary in an IDE migration. If the previous implementation separated UI and business logic, the functionality including all grammars and language features can be reused.

A translator is not applicable in this scenario. The backend can be reused and does not have to be translated to move to web technologies. A translation to a different language or framework is not needed here. Translation is also not applicable to the UI. Part of the migration involves the implementation of new UI concepts. New concepts cannot be generated by a simple translation tool. Moreover, the development of such translation tool is too costly and has no added value to the project in the future, which can justify its development.

Reusing the backend implementation seems to be the optimal solution. It allows to support the Eclipse IDE and the new IDE in parallel.

4.1.3 Reusing the Backend

The goal of this migration strategy is to reuse most of the already existing UI-independent code of the Eclipse based IDE, which allows to support both the Eclipse IDE and the new tool in parallel if needed.

Prerequisites

A migration from Eclipse to web technologies must have the following prerequisites to use this migration strategy.

The business logic and editor functions of the Eclipse application have to be independent of the Eclipse UI, since they are bundled into an independent server component in this migration. If this is not the case, the Eclipse application has to be restructured. The goal of this restructuring is to separate the existing plugins of the Eclipse IDE in UI-plugins and non UI-plugins. Every non UI-plugin has no dependencies to a UI-plugin. Furthermore, IDE-independent functionality has no dependencies to UI-plugins. The IDE-independent features have to be provided to the Eclipse frontend via an interface. This interface allows to interchange implementations easier. To be able to use all advantages of this strategy, the code basis for the development of the Eclipse IDE has to be in some kind of version management system. This version management system can hold the Eclipse IDE and the backend server of the new tool on different branches. Providing them in the same repository is not mandatory, but eases development in later stages of the migration.

Advantages

The *Reusing backend strategy* has the following advantages over the other introduced strategies:

Reusing the editor functions and language features is quicker than the full reimplementation. Moreover, reusing the backend introduces fewer bugs than reimplementing everything. Reusing the backend prevents the new tool from diverting in functionality. The backend is already implemented in the Eclipse IDE and in use for a while. Therefore, some bugs and other mistakes were already

found and fixed. A full reimplementation on the other hand is likely to introduce the same bugs again [Tep09]. The same programmers tend to do the same mistakes and the same development strategies and patterns do this too. Reusing the backend is also quicker than developing a translator, since the project is relatively small. A translator is only justified if it can be used for more than one project or has other future uses.

Since the backend is reused, the new tool has the same behavior as the Eclipse product. Therefore, the Eclipse product can be used to cross-validate the functionality of the new IDE. Functions of the Eclipse IDE and the new tool have to deliver the same output. Cross-validation during development reduces the time and money spent to test the new IDE after the migration. If restructuring of the Eclipse IDE is needed, the previous version of the IDE can also be used to validate the changes. An existing test base can be used to verify the backend of the new tool.

Moreover, the Eclipse IDE and the new IDE based on web technologies can be supported in parallel if needed. The backend is reused, but the frontend changes. The backend can be reused from the same version management repository. This allows to support the Eclipse IDE and the backend of the new tool on different branches in the same version management repository. Therefore, changes to the Eclipse IDE backend can be easily merged into the new tool.

Migrating to a client server architecture facilitates separation of concerns. Separating business logic and UI makes a project easier to maintain, as also experienced by Hasselbring et al. [HRJ+04]. If the server is used as a service and functionality is not reimplemented on the client side, duplicate code is avoided. Moreover, the separation into client and server allows to use different languages in each of them. Therefore, problems can be solved in the component that is best suited for it.

Migration Steps

The migration aims to build a working prototype as early as possible in the migration. The prototype shall consist of a client, which has a working editor, and a server component, which can provide part of the backend functionality of the Eclipse tool, for example rich language features for the mentioned editor.

The first step in the migration process is to fulfill the prerequisites. An Eclipse IDE is not always structured into a backend and a frontend. Not every feature of the Eclipse IDE has to be made UI-independent at the start of the project. Features that are not necessary for the prototype can be restructured later when they are needed.

The next migration steps can be seen in Figure 4.1. The Eclipse IDE is separated into backend and frontend. The frontend consists of an editor UI and a model UI. The backend consists of the editor functions and the business logic, which provide all IDE-independent functionality including language features. This backend is accessible by the frontend packages via an interface. The new IDE consists of two components: a client and a server component. The client is implemented using web technologies and provides a new editor UI and a new model UI. The server component consist of an editor function package and a business logic package. These packages are reused from the Eclipse IDE. Therefore, the server is written in the same language as the Eclipse backend. An interface, for example the LSP, allows to use the backend implementation in the server as a service for the frontend implementation.

A real-world IDE might have several more backend or frontend packages for more features (e.g. simulation support in KIELER). The frontend of the Eclipse application is based on the Java UI framework SWT. The frontend of the new tool does not redundantly copy the existing Eclipse UI. Web technologies have established design concepts, which can be used as a guide. However, they are flexible enough to mimic the Eclipse UI if needed. The LSP defines a standard IDE message interface, which is independent of the used programming language. If this interface is not sufficient, it can be extended by new

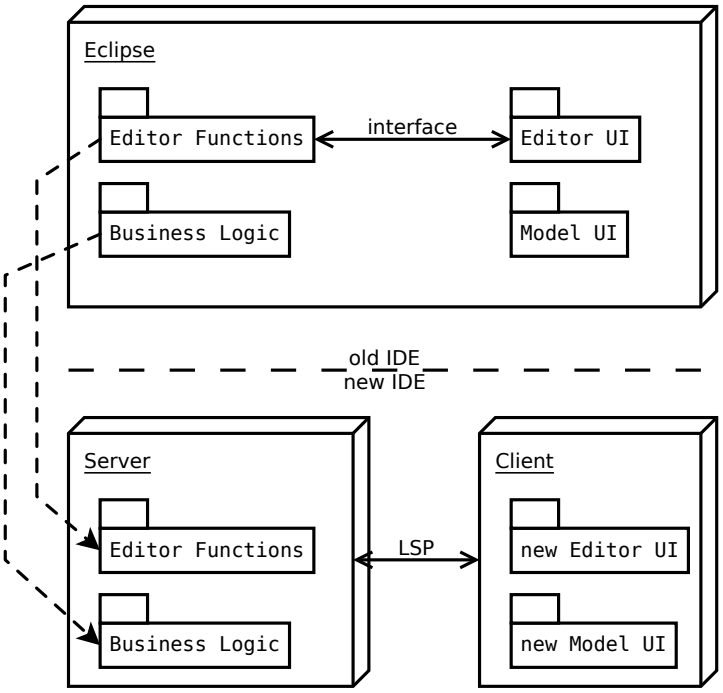


Figure 4.1. Concept of reusing backend strategy

messages for custom needs. Using the LSP is not mandatory. Every message interface such as Monto presented in Section 3.5 can be used instead, as long as the interface is able to express the necessary messages the IDE and its supported languages need. However, it has to be evaluated if the framework can support all languages and features required for new IDE. This is done by evaluation the use cases of the Eclipse IDE.

If the LSP is used, the server component does not have to be implemented manually. A generator framework can be used instead. Some generation frameworks are also able to generate a server, which provides language features, as seen in the example of Xtext in Section 2.4, which generates a language server, as seen in Section 2.5. If no generator framework is currently used in the Eclipse implementation, the migration might be the right time to adopt one. Adopting a new grammar framework will slow down the migration process and might cause errors or missing functionality, since the current developers are inexperienced with the new framework. However, a generator framework for a supported DSL helps to maintain the language features and makes it easier to add new functionality in case of a grammar change. There are several grammar frameworks available. Part of the design decisions while migrating is to find a suitable one that can support all necessary languages and features, which are required for the IDE and grammar.

The resulting prototype of the new IDE can be extended step-by-step with new functionality. New features of the Eclipse IDE are restructured to be IDE-independent and are provided by the reused backend. If necessary, the interface between the client and server component is extended to support the new function. The client is also extended to use the new feature. These steps can be done several times and allow to cross-validate functionality using the existing Eclipse IDE. The product is iteratively developed, which allows to present the progress to stakeholders and allows to discuss new feature implementation with them. Experts for the Eclipse IDE can be involved in this process as well. These experts know the use cases and how the backend functionality of the Eclipse IDE is used.

4.1.4 Migration Obstacles

During the migration from Eclipse to web technologies several obstacles might occur. Some of them occur since the Eclipse IDE and the new IDE are supported in parallel for the duration of the migration process or beyond that. Many of them can be avoided with the strategy presented in Section 4.1.3.

While developing the new tool, the Eclipse IDE may be further developed. To avoid duplicate bug fixes, the changes of the previous tool have to to be adopted into the new system while it is still in development, as suggested by Hasselbring et al. [HRJ+04]. If this is not done, both products may divert from each other and all bug fixes and changes have to be done twice. The presented strategy avoids these problems. Since the Eclipse backend resides in a version management system, changes to the backend implementation of Eclipse can be merged into the development branch for the new IDE. It is advised to use version management tool support to avoid mistakes while merging bug fixes. Manually copy and pasting the changes is often error prone, since humans tend to do mistakes while copying redundant code as also experienced by Teppe and Altadmri et al. [Tep09; AB15]. The project setup of the Eclipse IDE and the server component of the web-based tool have to be in the same repository for this to work.

The restructuring of a product is advised to be done incrementally to keep track of the changes and to find errors when they appear, as suggested by Teppe and Hasselbring et al. [Tep09; HRJ+04]. This is possible using the presented strategy. The prototype allows to be incrementally extended. After each step the changes can be evaluated and the updated product can be cross-validated with the Eclipse IDE.

The grammar of the languages supported by the Eclipse IDE may change in the future. If a generator framework such as Xtext is used, changes to the grammar can be applied automatically to the language feature implementation in the backend. Moreover, reusing the backend implementation allows to share grammar changes between the Eclipse IDE and the server of the new IDE. This enables to maintain both at the same time.

Syntax highlighting is needed to provide language support for an IDE. Syntax highlighting is a client side feature and often editor and therefore IDE-dependent. The highlighting is grammar-dependent. Such a feature can be provided by a server component or as a stylesheet, which is common for web editors. If Xtext is used in the project, it can be leveraged to automatically generate a highlighting stylesheet for any web editor. This makes the syntax highlighting resistant to grammar changes.

Using the *reuse backend strategy* does not only solve problems, but also introduces new obstacles and problems.

Migrating to a different IDE does not solve architectural problems. A restructuring of the Eclipse IDE into backend and frontend is only possible if the project structure allows it. Developing a prototype, which is extended later, is only possible if the target IDE framework allows it. A target platform for the new IDE has to be chosen carefully.

The Eclipse IDE backend and the server of the web based tool may have different dependencies. Dependencies to the same library might have different version requirements. To develop the Eclipse IDE and the backend server in parallel, merging between them in a version management tool is advised. Conflicting dependencies of the two components prevent that. If both IDEs should be supported in parallel, the dependencies have to be compatible.

It is not always better to compute everything on the server side. The presented migration strategy allows to reuse the legacy backend implementation without much effort. However, sometimes it is more performant to develop a client side implementation, which produces duplicate code. For every new feature it has to be evaluated, whether a client side implementation can be beneficial. This adds complexity to the design decisions and slows down the development process, since the decisions are more complicated and have a higher impact on the project.

4.1.5 OS-Independence

It is important to make an IDE usable on several OSs such as Linux, Windows, and MacOS, since developers may use different setups. If one OS is not supported, a considerable amount of developers may not use the tool. For IDEs the user base is crucial, since many of them are open-source. Therefore, the user base also influences the developer base.

Eclipse is able to deliver an OS-independent product, which can be built cross-platform. However, an Eclipse IDE has native dependencies to each OS regarding its file system and UI. Building an Eclipse product is done via Java and Maven or a similar software management tool. Java itself runs on the Java Virtual Machine (JVM) and is therefore platform-independent. Maven is able to provide libraries for different OSs if they are needed, which facilitates cross-platform build. The Eclipse framework includes prebuilt binaries for the different OSs. The technology stack used to build an Eclipse IDE enables to develop a platform-independent tool, which can be built cross-platform.

Migrating to web technologies influences OS-independence, since the application runs in the browser, either in an actual browser, such as Chrome[2] and Firefox[3], or in a Chromium inside an Electron app. OS-independence is not always trivial if web technologies are used. The backend server itself remains OS-independent as the Eclipse application and can even be provided as an Eclipse product.

The client implementation in web technologies possibly influences OS-independence. JavaScript or a JavaScript dialect such as TypeScript, explained in Section 2.7, can have native dependencies. Native dependencies have to be built for the specific OS, to add e.g. UI elements for the different OSs. JavaScript packages do not always provide prebuilt binaries for the different OSs. If native libraries are needed for the client, they cannot be built cross-platform. Therefore, an Electron product has to be built directly on all supported OSs. The problem of native dependencies can be solved by using tools such as Appveyor[4], Travis[5], or Microsoft Azure[6] to build and test the product inside a container. These technologies allow to build an Electron application if no server with a supported OS can be provided. The tools Appveyor and Travis do not need a server with a specific OS and are therefore less dependent on the existing deployment infrastructure. Not every product qualifies to use these tools. Some tools need a specific version management system. Some need a purchase of a license to access all features. Moreover, the developer itself may not want to give access to the source code to such a tool, because of license issues. An Electron build or bundling tool might be needed (e.g. electron-builder[7] or electron-packager[8]) if the product should be delivered as a desktop Electron app. A bundling tool allows to customize the build, adds special cases for the different OSs without touching the code base, and can provide installers for different OSs.

Providing the resulting IDE as an online IDE accessible over the internet, without an Electron app, eliminates dependencies to different OSs, but introduces another dependency. Since the resulting product runs in a browser, the UI is browser-dependent. Different browsers support different fonts, have different UI components, or different bugs. This does not only apply to different browsers, but also different versions of the same browser or the same browser in different OSs. In case of an Electron app the used browser is always a Chromium, which limits the browser versions that have to be supported.

[2] https://www.chromium.org/
[3] https://www.mozilla.org/firefox/
[4] https://www.appveyor.com/
[5] https://travis-ci.org/
[6] https://azure.microsoft.com
[7] https://github.com/electron-userland/electron-builder
[8] https://github.com/electron-userland/electron-packager

4.1.6 Migration of Knowledge

When migrating from Eclipse to web technologies, the developer knowledge has to be kept in mind, as also experienced in the ARNO project in Section 3.1.1. UI technologies, concepts, and languages change and require developers to learn them. To keep up with this change, the staff has to be trained or new developers, which are already proficient in web technologies, are recruited. It is important to give the developers time to understand the new technology, while undergoing a migration. Therefore, too drastic changes to the used technologies should be avoided. Too drastic changes can be compared with a full reimplementation. This is also the case for a migration from Eclipse to web technologies, since the UI language changes from Java, which leverages the SWT framework, to a JavaScript dialect, HTML, and CSS. The separation into frontend and backend enables to have two separate teams to develop each component using an interface between the server and the client.

4.1.7 The Migrated Product

The migration from Eclipse to web technologies results in two different components: the server, which provides languages features, and the client.

The server can be delivered as a separate product. Therefore, it can be reused in different projects, which makes the client implementation variable and interchangeable. Separation of concerns and an easier to maintain code basis is achieved by interchangeable components [HRJ+04; BHJ16]. With a general enough implementation, this server can be used to implement new extension for different development tools, by using it as a service. The developer can decide which tool to use.

The client itself is implemented using web technologies. Therefore, it has the ability to run in a browser. The resulting product has the potential to run as a desktop IDE or as an online IDE if the used IDE framework supports this. The developer has to include this fact in the decision for a suitable framework.

Even when migrating, innovation and technological advancement does not stop. A migration has to be quick to avoid cost and to not be outdated [HRJ+04]. Therefore, a migration should be carefully planed and the target should be easier to maintain and to expand to ease new migrations or implementations in the future. A distributed system is sometimes easier to maintain, since every component can be developed on its own and is interchangeable [BHJ16]. Hence, decomposing a monolith into a client server application seems to be a step into the right direction.

4.1.8 Generalization of IDE Migration Problems

Although the strategies and ideas are applied to the context of an IDE, some of the deliberations can be applied to a more general context.

The first step of a migration should always be the evaluation of migration strategies regarding time, resource consumption, and the migration target. A migration to an undesirable architecture or framework is not only waste of time and money, but leads also to no technological advancement. Separating a monolithic program in smaller services helps to achieve better maintainability and makes the different components interchangeable by providing an interface for communication [BHJ16]. Interchangeable components ease future migrations. One has to keep in mind that distributed services always have an overhead for communication, and separating a program into services is a non trivial task [BHJ16].

Furthermore, while migrating testability should be ensured. A system which is not testable is not useful for any developer, since bugs cannot be reliably found [Bin94]. In a distributed system not

only the different components, but also their communication has to be debugged or monitored as suggested by Joyce et al. [JLS+87].

The overall separation into a business logic and UI can be applied to a different context and are also applied to the migration of business information systems [HRJ+04].

Time is an important factor in the migration process as well as money and have to be considered when deciding for a migration strategy [Tep09]. The longer a migration needs the more changes will be made on the legacy system and the longer both projects have to be supported in parallel [HRJ+04]. Time can be reduced with automation and the reuse of software without redundantly copying it [FBB+07]. Redundant software only costs time and carries over all problems to the new environment [Tep09].

Another problem might be the documentation. Often the legacy system is not well documented, which may lead to erroneous plans to migrate and might slow down the process. The legacy system itself can serve as documentation [HRJ+04]. Nevertheless, domain specific experts, communication with the user, and stakeholders or other evaluation methods are needed to migrate [Tep09; ZKC+07].

4.2 UI Design in Web-based IDEs

In the migration from Eclipse to web technologies not only the functionality, but only also the UI have to be migrated. Web technologies introduce new UI concepts. Modern IDEs often use web technologies for their UI and adopt these concepts. Theia, the IDE framework which will be used to as a target IDE for the migration of KIELER to KEITH, is inspired by the UI design and concept of VSCode. VSCode is a newly introduced and established IDE, which is implemented using web technologies. VSCode is an example of a modern IDE.

4.2.1 Eclipse

Eclipse is a desktop IDE, which can do almost anything with its rich extension environment. Its functionality is mostly accessed by menus (e.g. the context menu) and configuration wizards. The UI focuses on buttons with icons and the context menu to add new functionality to the Eclipse native features. An example how the Eclipse UI looks like can be seen in Figure 2.1. The KIELER tool displayed in the figure focuses on buttons to provide functionality. The UI concept tries to make every feature discoverable by the user via button icons. The concept associates buttons also via their placement. Every view has its own toolbar, which provides functionality in form of buttons, selectboxes, and more, to access functionality regarding the view itself. Icons, hovers, or short description of functions in menus guide the user and display all possible options. Expert users know where to find functions or know how to utilize existing search functions to find them. Novice users tend to have some problems using Eclipse. They do not find all the functionality Eclipse provides, since the UI is rather complex [RT05; SDM+03].

4.2.2 Comparison to Modern web IDEs

The newly introduced IDE VSCode is used as an example for a modern and web-based IDE. These web technologies facilitate separating between display options configured in the CSS styles and function calls via the UI. This enables to separate different concerns and to reuse UI styles without redundantly copying them. The IDE focuses on what it is designed for: to write code using model-driven engineering. Therefore, VSCode tries to provide the UI of a simple editor. VSCode hides it functionality not in extensive wizards or menus, but rather in the command palette, a command search bar at the top,

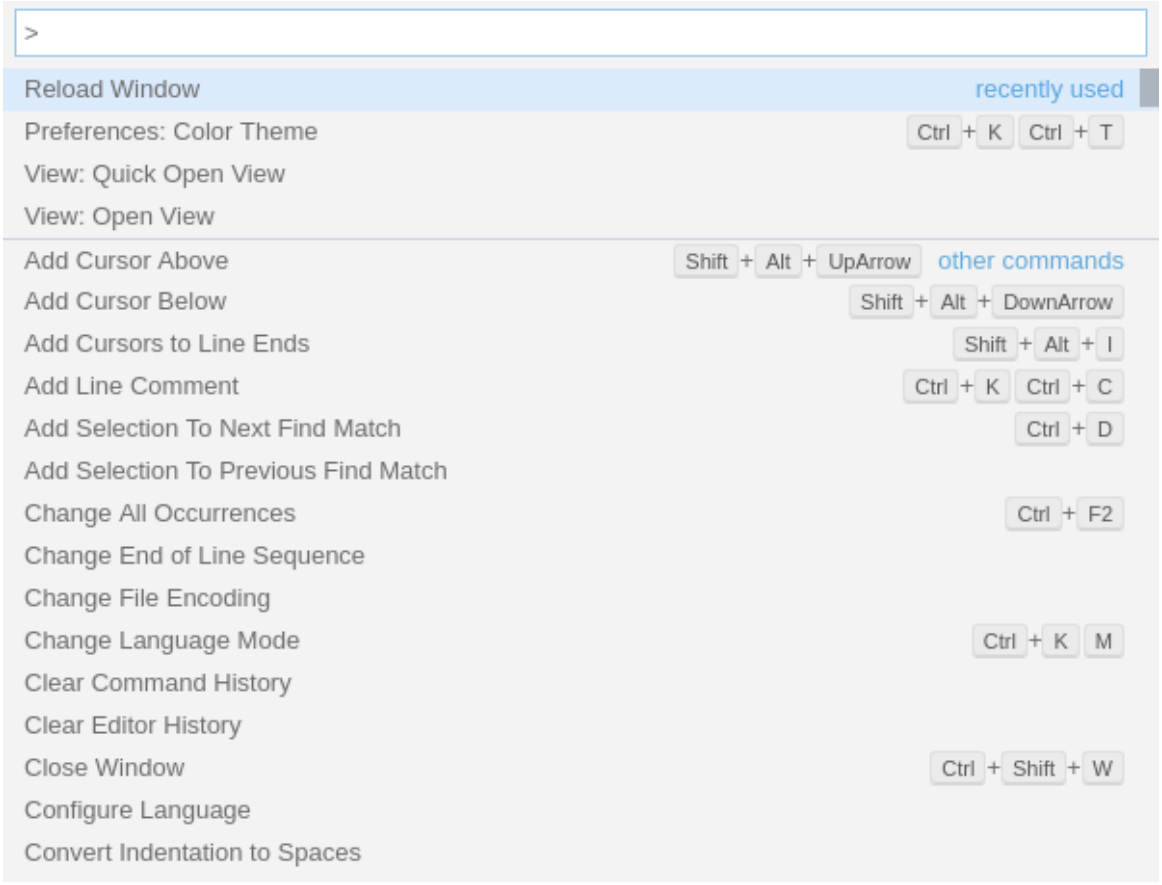


Figure 4.2. Command palette in VSCode [Tea18b]

as seen in Figure 4.2. The command palette is not visible in the IDE. It only becomes accessible if it is opened via a corresponding shortcut or menu entry. The command palette consists of a text field, which is used to specify a command. If the command palette is opened a ">" is already in the text field, which is the prefix to execute a command. Below the text field executable commands are listed. They are divided into recently used commands at the top and other commands, which are sorted alphabetically. Together with a command a shortcut is specified, which can be used to execute the command. If a string is written in the text field the available commands are sorted by relevance.

The command palette holds every registered command and is searchable by regular expressions if ">" is typed. For example, the command for changing the color theme of the IDE is called "Preferences: Color Theme". It consists of three of the major keywords associated with the color theme and its preferences. Often, a command prefix is used to associate a command to a context. In the previous example the context are the IDE preferences. Furthermore, entering "?" results in a listing of all possible commands. No prefix allows to search and browse the files of the workspace. In Eclipse these functions require the user to know shortcuts or where in the menus they might be situated if the corresponding shortcut is unknown. The main idea of VSCode is to hide most of its functionality from the user and make commands searchable and browsable. The developer needs to know an associated shortcut for a command or how it is called to use this command. Otherwise, all possible commands have to be browsed. VSCode tries to reduce the noise of the UI with this concept to help a developer to focus.

VSCode displays important messages via pop-ups. While migrating the UI to a VSCode like UI, the usage of these pop-up messages has to be evaluated, since they divert the attention of the user. VSCode also uses buttons, but these are designed to blend with the whole UI and are most times only visible if the corresponding widget is focused of hovered over. Web technologies can be leveraged to apply Eclipse's button focused UI also to VSCode if desired. To summarize, VSCode tries to look like a text editor while still delivering rich language features for model-driven development on demand to not divert the users attention.

Chapter 5

Transforming KIELER into KEITH

As already mentioned in Section 1.3, the goal is to migrate from the Eclipse IDE KIELER to KEITH, an IDE built with web technologies using the Theia framework, and a language server. This chapter presents how this migration took place and presents and explains the design decisions. Furthermore, general remarks about the use of the Theia framework are made.

The resulting prototype of the KEITH Electron application can be downloaded at the following URL: `https://rtsys.informatik.uni-kiel.de/~kieler/files/nightly/sccharts-integration/`

5.1 Migration Strategy

The migration strategy elaborated in Section 4.1.3 requires to restructure KIELER to separate UI and non-UI functions.

KIELER uses Xtext to specify its grammars and generate language features. The newest Xtext version already provides separation of UI-plugins and non UI-plugins, as seen in Figure 5.1. The new Xtext version is required, since it implements necessary language server features. Before the upgrade of Xtext each grammar had a corresponding plugin, which implements language features, and a UI-plugin, which implements all other functionality required for that language. After the upgrade of Xtext an IDE-plugin for each grammar namely `*.sccharts.ide` is introduced. This plugin holds all functionality that has formerly been in the `*.sccharts.ui`-plugin and is not UI-dependent.

Xtext is also able to generate a language server using the existing infrastructure. The next step is to build a Theia application, which uses the generated language server. The generated language server is used to register the supported languages. The language server needs two different starting methods. The language server connects via stdin/stdout if it is bundled as an Electron app or if the path to the language server is on the same system as the Theia application. If the language server runs in its development setup or server deployment setup, it connects via socket. The KEITH language server

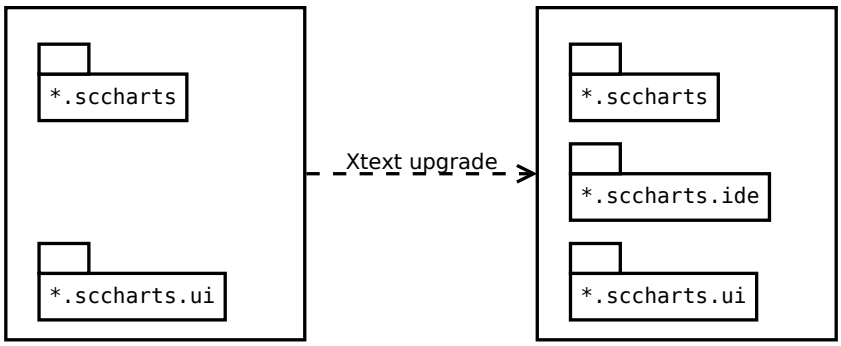


Figure 5.1. Change in plugins of KIELER as a result of the Xtext upgrade

needs support for all languages included in KIELER, compilation and simulation capabilities, and the ability to synthesize diagrams. The language server is bundled as an Eclipse plugin, since the existing Eclipse plugins need extension points and OSGi to access required plugins from the KIELER project, as already mentioned in Section 2.2.

The Theia application KEITH is implemented via a Theia extension. A Theia extension is a package bundled with other Theia packages to build a Theia application, as seen in Figure 2.6. The Theia extension implements the new UI in web technologies and is the client side implementation mentioned in Section 4.1.2. Theia provides classes and interfaces to connect to a language server. The migration has shown that it helps to use the `hello-world-extension` for Theia as a prototype and extend it step-by-step through language support via a language server and new widgets, which are called views in Eclipse.

5.2 Features

The desired features for KEITH are determined by the tool KIELER. On the long run every available feature should be ported to KEITH. The primary use cases can be seen in Figure 5.2.

KIELER has two different actors: a developer and a user. The developer extends the user and can use any use case the user has. The development of KEITH focuses primarily on the user use cases, since most of the developers desire to continue their work with the KIELER tool and not every aspect of KIELER that is needed for development is already available in KEITH. All use cases that are currently not implemented in KEITH are shown smaller in italic. KEITH supports modeling of programs. This process includes writing in an editor, syntax coloring for the model language, content-assist, and other language features such as diagnostics. Moreover, diagram synthesis is part of the modeling process. The diagram synthesis is only implemented as a mockup view. A user can compile and simulate models. As part of the compilation the user can selected the compilation system and preferences. The user can view compilation snapshots. The developer can additionally add new compilation systems and has access to more preferences for advanced features.

The main use case is to develop a model for SCCharts or other synchronous languages, which are included in KIELER. This requires the user to be able to edit a file in an editor. The editing experience should be supported by the IDE by adding syntax highlighting, delivering content assist if required, and displaying warnings or other diagnostics to help the user find problems or errors in the model. Furthermore, the user needs a diagram view of the model, which should be synthesized automatically. This is part of the project of Rentz [Ren18] and therefore not in the scope of this work. Diagrams are only displayed by a mock view and are generated using already existing technology, which is automatically part of the language server. The communication needed for an interaction with a diagram server is modeled to be compatible with future implementations. The diagram server will be part of the language server and is realized via an extension to the LSP. The developed models can be compiled using different compilation systems, which require to set compilation preferences (e.g. inplace compilation). The compilation results in a step-by-step transformation from one model to another by separate defined transformations. Inplace compilation means that a source and target model of a transformation are the same [MFH09]. The resulting snapshots are selectable and displayable as their diagram representation and browsable by the user to understand the transformations or use the snapshots for debugging purposes. The compiler needs a widget to manage the compilation settings, select the compilation system, and browse the compilation results. The simulation of a model also needs widgets to display and set the current variables and signals and a way to select which file to simulate. As seen in KIELER, it is desired to mark the simulation progress in the diagram representation of the model. Since this is not in the scope of this work and requires to merge this implementation

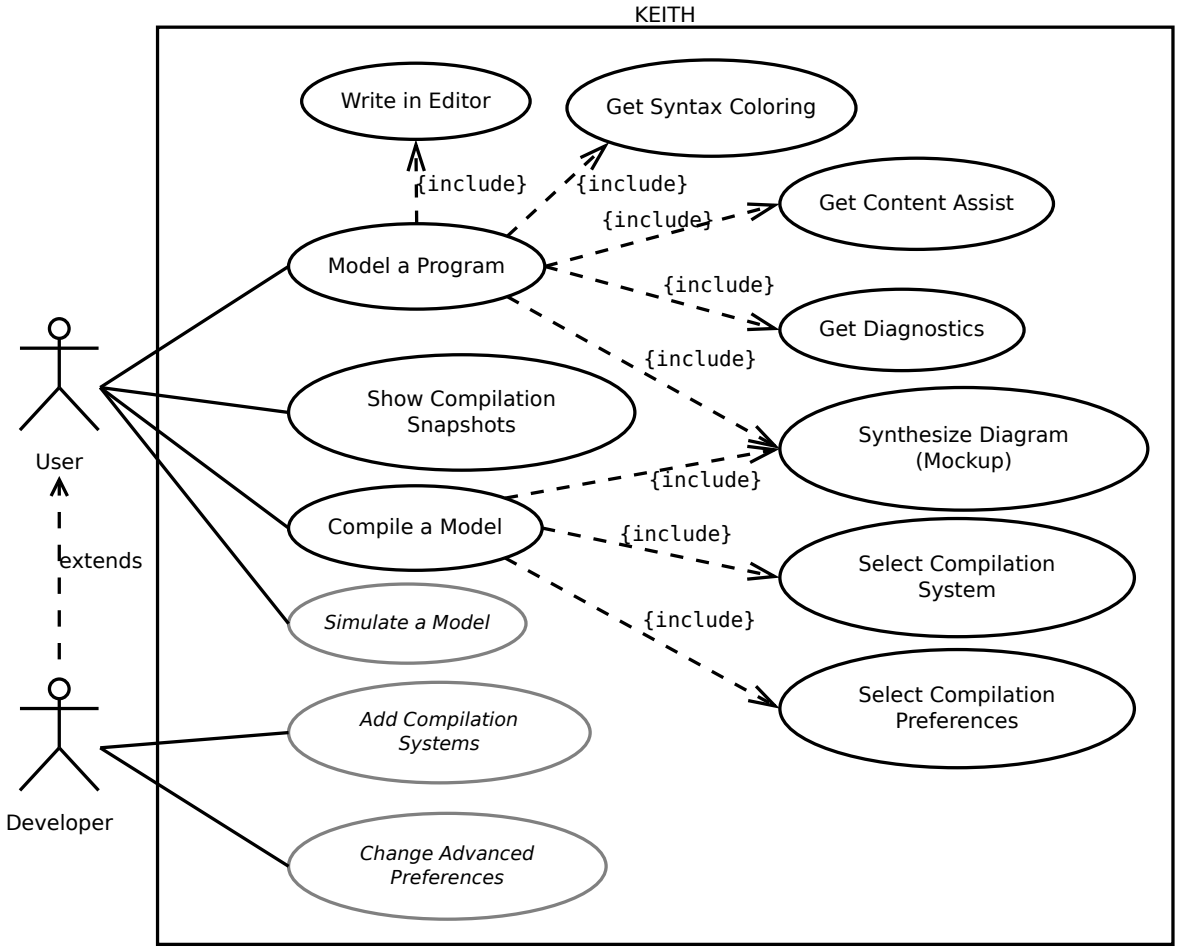


Figure 5.2. Use cases for common users and developers desired to be in KEITH

with the work of Rentz [Ren18], it is considered as future work on this project. All remaining use cases are only relevant for the developer user and are therefore not in the scope of this migration project and regarded as future work.

A widget for compilation is needed. How such a widget may look like is seen in Figure 5.3. The widget consists of some buttons to change preferences, a selectbox to select a compilation system, and a list of snapshots. The widget itself normally resides in the bottom bar. The snapshots are named. If a transformation has more than one snapshot only the first one is named. The other snapshots for this transformation are visualized smaller to show their affiliation to the previous one. Furthermore, colors are used to indicate, whether diagnostics (i.e. infos (turquoise), warnings (yellow) and errors (red)) are attached to the snapshots. The available compilation system seen in the select box are not implemented statically, but are requested from the language server on update of the widget. Furthermore, the list of available snapshots is the response of a compilation and is saved on the client side for each file of a model. This allows to use the Theia application with new language servers to test functionality without building a new Theia application. The frontend without the language server can be delivered separately and allow to connect to different language servers via a socket connection.

Since Rentz [Ren18] has not yet finished with his implementation of a diagram widget, a mockup

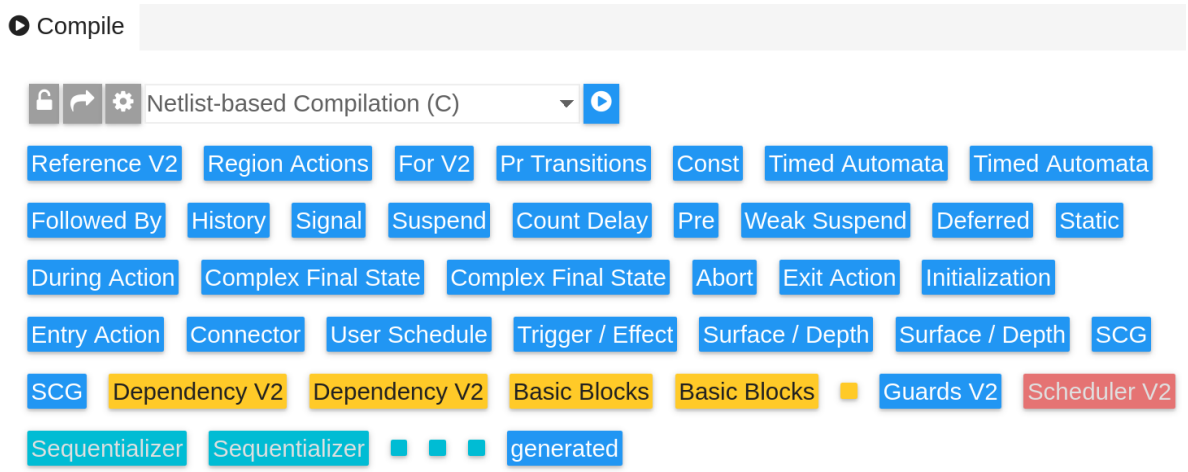


Figure 5.3. Screenshot of the `CompilerWidget`

to simulate the communication and find possible obstacles is needed to visualize the compilation snapshots. Mockup diagram generation is used to generate an SVG. This generation can be requested from the LSP in the compilation view via the snapshot buttons and is displayed in a widget as HTML, which is interpreted by the browser as an SVG and rendered accordingly. The communication to compile a model and show a corresponding snapshot can be seen in Figure 5.4. The language server is called via the compile command, which needs the model URI, the compilation system, and a boolean. The URI identifies the model in the file system. The `compilationSystem` string identifies the compilation system on the language server. The boolean defines whether inplace compilation shall be turned on or off. These information is used to call KiCo, the compiler of the KIELER tool, which compiles the model and generates the corresponding snapshots [SSH18]. These snapshots are saved for each URI and allow the user to access them later. As a result the description of the snapshots are sent back to the Theia application. Such a description consists of the processor name and the index of the snapshot with its name. A processor can have several snapshots as output. The index is used to order them. The descriptions are used to update the compiler widget and show the snapshots, as seen in Figure 5.3. The snapshot descriptions are clickable and trigger a request to the language server, which uses its mocked diagram functionality to produce an SVG, which is returned to the Theia application. The SVG is displayed it in the diagram widget. Theia's extensible framework allows to specify keybindings to navigate through these compilation snapshots.

A screenshot of the KEITH tool can be seen in Figure 5.5. The left bar holds the file system. The file system indicates with icons whether a file has diagnostics. The divisible main bar consists of the code editor and the diagram view side-by-side. The editor view shows diagnostics at its scrollbar. Several editor files can be opened at the same time and are displayed next to the opened editor. The diagram widget shows an SVG. The mockup diagram widget is not zoomable, but scrollable and is therefore not able to display the whole diagram at once. The right bar is not opened but can hold an outline view or a view for diagram preferences. It is seen in the top right of the screenshot and holds a closed outline view. The bottom bar holds the `CompilerWidget`, which allows to compile the active model in the code editor, and the problem widget. The problem widget holds all diagnostics of all files. The `CompilerWidget` can also be seen in Figure 5.3 and was explained earlier. The status bar is at the bottom. It shows the number of problems and errors, the current line and column in the editor, how tabs are displayed, and the editor language SCTX. A blue status bar indicates that a connection to

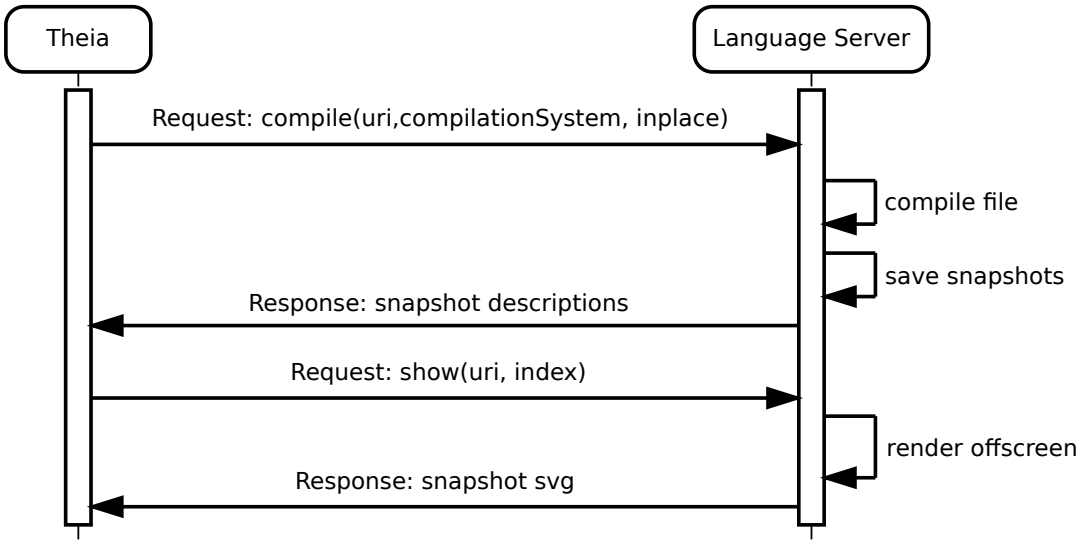


Figure 5.4. Communication for *compile* and *show snapshot* workflow

the language server can be achieved. The status bar is shown in orange if no connection is possible or the connection is lost. All widgets can be moved to other bars if desired. Furthermore, the main and bottom bar are splittable, which allows to open the diagram widget next to the editor in the main bar. The left and right widget are theoretically also splittable, but this functionality is deactivated per default.

5.3 Build Setup

Since the legacy IDE KIELER is built via Maven and tycho[1] in the continues integration Atlassian tool Bamboo[2], it is desired in this project to integrate the build of the KEITH app in Bamboo as well.

The language server itself is bundled as an Eclipse application to be able to use all plugins of the required projects. This Eclipse application can reuse the extension points of the KIELER tool. These extension points allow the compiler of KIELER to have access to the compilation systems, which are defined in different plugins. Since the language server is an Eclipse application, it is automatically built for every OS and has separate `.ini` files. The language server is built the same way as KIELER and shares KIELER's plugins. The only difference is its main class to start the application. Eclipse is able to be built cross-platform, since the precompiled binaries are accessible by Eclipse and result in a language server for every supported OS.

The Theia extension itself has to copy that language server, which is done automatically, and package it to an Electron app containing both the language server and the Theia extension. In this process the `.ini` file of the language server is changed to start an Eclipse application without the UI with the `-nosplash` option. Other used options can be seen in the project repository of the language server.

[1] `https://www.eclipse.org/tycho/`

[2] `https://de.atlassian.com/software/bamboo`

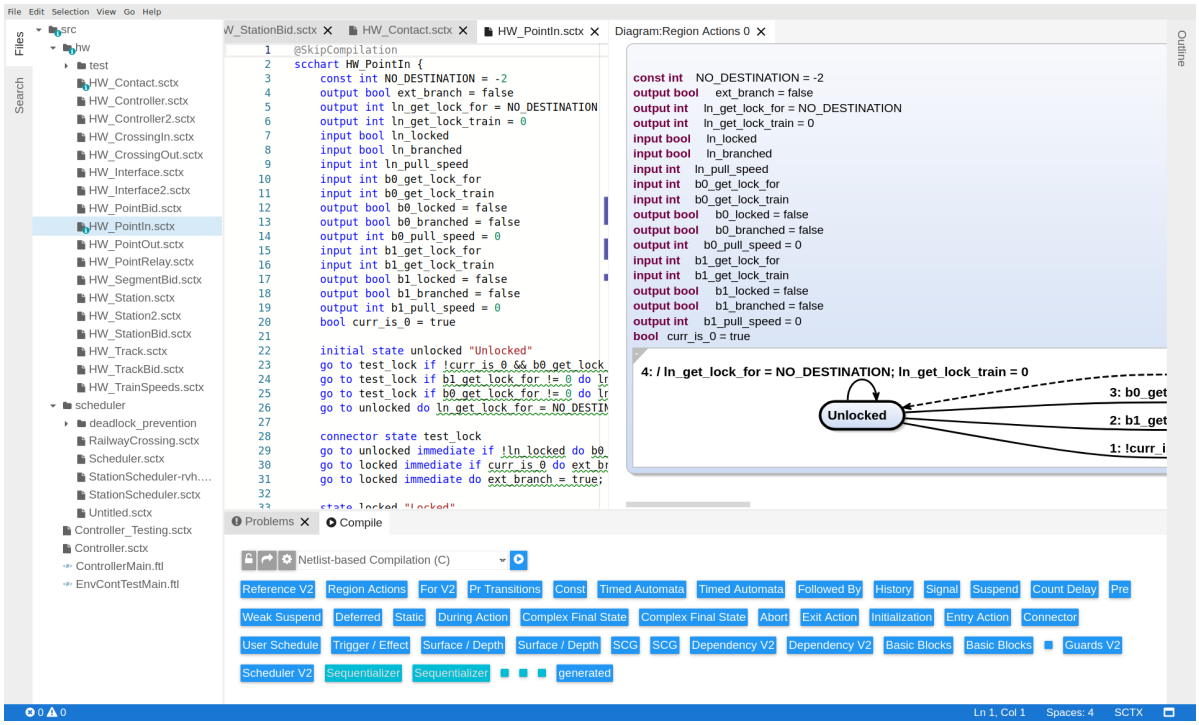


Figure 5.5. Screenshot of KEITH

5.3.1 Bamboo Build

The build jobs and their tasks of the KEITH build in Bamboo, which can be seen in Figure 5.6, are similar to the build of the KIELER tool. Many of the different jobs are coped and adopted to KEITH because of this similarities.

The KEITH build consists of five stages: `Build Update Site`, `Deploy Update Site`, `Build Product`, `Build Theia App`, and `Deploy Product`. All accept the `Build Theia App` stage are part of the KIELER build. The `Build Update Site` stage has the build job `Compile and Package Update Site`. This job has two tasks: one to check out the KIELER repository and one to build the plugins via Maven. The next job is `Deploy Semantics p2 Repository`. It has one task to make the plugins available on an update site. The `Compile and Package Product` job is used to build an Eclipse application. In the case of KIELER, the KIELER tool is built. In the case of KEITH this job builds the language server and bundles it as an archive for Windows, Linux, and MacOS. The tasks checkout the repository and use Maven to build the product. The fourth job is OS specific and is missing in the build of KIELER. Bamboo build agents build the KEITH product natively on each OS. For every job the following tasks are executed. The working directory is cleaned. After that the source code is checked out from the development repository. The third tasks uses yarn to get all required packages by executing `yarn` in the corresponding folder in the repository. Next `yarn build` is executed to build the application. The last task executes `yarn package`, which bundles the application as an Electron app or installer, as specified in the `electron-builder.yml` using the electron-builder framework. The last job `Deploy Semantic Product` makes KEITH available for download. In the build of KIELER, the KIELER tool is copied to its update site. Bamboo allows to add dependencies in form of artifacts to each job, which allow to share the language server or KEITH application between the different tasks.

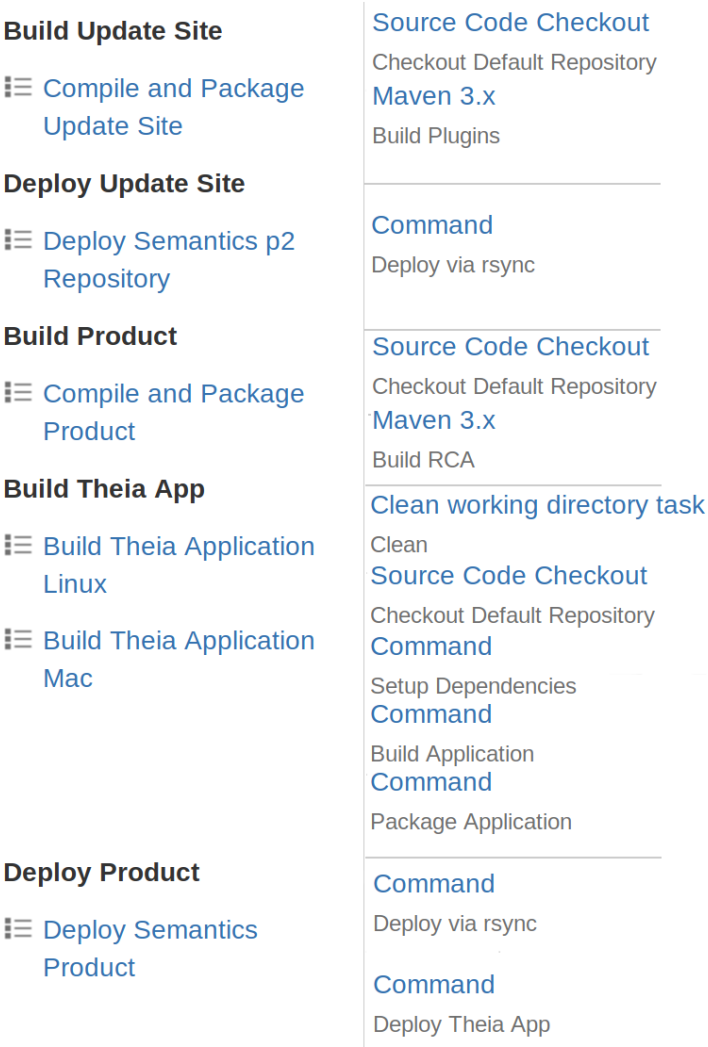


Figure 5.6. Overview of the Bamboo jobs and tasks necessary to build KEITH

5.3.2 Prerequisites

To build the KEITH product, the following tools are needed: All prerequisites to build the KIELER project, since many of its plugins are reused for the language server. So among others Java, Xtext, Xtend, and Maven are needed. To build the Theia extension Python 2.x[3], Java 8.x[4], node 8.x[5], the package manager yarn[6], gcc[7], make[8], and g++[9] are needed to build dependency packages and to build the product as an Electron app. Since the Electron app has to be built natively, the prerequisites for Theia extension have

[3] https://www.python.org/
[4] https://java.com/de/download/
[5] https://nodejs.org/en/
[6] https://yarnpkg.com
[7] https://gcc.gnu.org/
[8] https://www.gnu.org/software/make/manual/make.html
[9] https://linux.die.net/man/1/g++

to be installed on every machine that builds a version of it (e.g. on the Mac that builds the MacOS version via a Bamboo build agent). Moreover, the application uses Maven and npm to get the build dependencies. An internet connection is required to build KEITH. If no internet connection can be provided, a locally running npm repository and a local Maven repository have to be sufficient to build the KEITH product. If this cannot be provided, KEITH cannot be built or used in its development setup. To run the development setup only Python, Java, Maven, yarn, and node are required, since not all dependencies have to be compiled using gcc, make, or g++.

5.3.3 Building a Product for different OSs

As mentioned in Section 5.3.1, KEITH is built for the different OSs via electron-builder. Electron-builder is an npm package that allows to bundle an application as an Electron app or installer and to configure this application. Electron-builder[10] can be used to build an application for Linux, Windows, and MacOS with the same source code. This is done by configuring the OS specific build via a `.yml` file.

Since the product has native dependencies it is necessary to build natively on each OS. The reference project YANG presented in Section 3.2.4 uses Appveyor and Travis to build and test their Yangster Electron app. Since Travis only works for GitHub[11] projects, but the KIELER tool is hosted via BitBucket[12], this is not possible in the case of KEITH. Therefore, such tools are not considered here and the product is built natively via Windows and MacOS build agents in Bamboo, which are executed on machines with the corresponding OS. This allows to deliver the product the same way as the KIELER tool, but requires machines with the corresponding OSs to build the product.

5.4 Migration Process and Development

The general strategy for this project are already explained in Section 5.1; only the concrete migration plan and order are missing. The elaborated dependencies between the different components result in the following plan:

The first step of the migration is to generate a running language server from the existing Xtext grammars and have it connect to a Theia application to be able test the backend of the Theia application while developing it. The language server is developed first, while the Theia extension is only developed to the point that it can connect to a language server and provide language features for one language in an editor. This allows to test the server generation via Xtext and shows the limits of this solution early in development.

5.4.1 Upgrading Xtext

The upgrade of Xtext has the following results: An IDE-plugin, which holds all IDE-independent code of a corresponding UI-plugin, is created, as already explained Section 5.1. Furthermore, the new Xtext generator framework is implemented for the mwe2 workflow to generate the support for SCCharts and other languages included in KIELER. The Xtext version is upgraded from 2.10 to 2.14 to be able to generate a language server with all necessary features. TypeFox, one of the companies behind Xtext, has developed a guide and a generator to do this in Eclipse.[13] Classes can be generated in Java or Xtend. Xtend is the preferred language of the developers of KIELER and is also used for KEITH.

[10] https://github.com/electron-userland/electron-builder
[11] https://github.com/
[12] https://bitbucket.org/
[13] https://typefox.io/xtexts-new-generator-migration

Conversion of Java files to Xtend can be done automatically via the context-menu entry `Convert to Xtend` in Eclipse. A guide is used to convert to the new Xtext generator architecture.[14] This guide describes how an mwe2 workflow is created using the new infrastructure. Moreover, a template with all potential fragments is part of the guide, which is needed to add custom behavior. The new workflow is cross-validated with the KIELER tool regarding its functionality to avoid future problems, as suggested in Section 4.1.2.

5.4.2 Syntax Highlighting for Theia

The LSP cannot be used to provide syntax highlighting for KEITH. Moreover, syntax highlighting cannot be automatically generated by Xtext for the Monaco Editor, which is the editor used in Theia, since no suitable generator fragment exists.

While upgrading the Xtext generator, a new fragment is introduced to generate a syntax highlighting file for the Monaco Editor. A change in the grammar can result in wrong syntax highlighting, but since the highlighting file is automatically generated from the grammar, this is avoided. A manual change of the highlighting file is not necessary and the process can be automated. The `MonacoHighlightingFragment` automatically generates a file that recognizes all keywords of a language and highlights strings, comments, and more according to the standard in SCCharts and its sublanguages. An example SCCharts model can be seen in Listing 5.1. Strings are enclosed by quotation marks and shown in blue, as seen in line four. Keywords are shown in purple. Comments are enclosed by `/*` and `*/` if they are block comments, as seen in line one to three. One-line comments begin with `//` and are colored in a light grey-green, as seen in line eleven and 29. This fragment can be reused for other languages. By extending the class, methods for comment and string recognition can be overwritten if the language marks them differently. The different generation methods can be reimplemented to fulfill the different needs by being able to configure the highlighting and recognition of comments, strings, numbers, whitespace characters, escapes, and more.

5.4.3 Prototype for KEITH

The next step is to build a prototype for KEITH to ensure that the generated language server works. After the migration of Xtext and the development of the `MonacoHighlightingFragment` are finished, the language server can be implemented and tested against a generic Theia extension: the `hello-world-extension`[15]. The `hello-world-extension` needs a backend extension to connect to a language server. Moreover, the languages have to be registered in the frontend extension of the Theia extension for KEITH. This results in a prototype that can access the language features provided by the generated language server. Such a prototype gives insight on how processes can be automated and shows obstacles of the used technology. Automation helps to avoid problems on grammar change and helps to add new languages.

5.4.4 Extending the KEITH Prototype

As shown in Section 5.2, KEITH needs more than language features. Compilation and diagram generation are needed and corresponding widgets for them. These are developed incrementally. If a new widget is added, it most likely requires communication with the language server, which needs a new extension to the LSP. The LSP extension has to be tested, before new features are added. This

[14] `https://www.eclipse.org/Xtext/documentation/302_configuration.html`

[15] `https://www.theia-ide.org/doc/authoring_extensions`

```
scchart ABRO2 "ABRO" {
   input signal pure A
   input signal pure B
   output signal pure O

   region test {
   // inital state of the region; awaits A and B
   initial state WaitAB {
      region {
         initial state wA
         if A do B go to dA

         final state dA
      }
      region {
         initial state wB
         if B go to dB

         final state dB
      }
   }
   join to done do O

   // final state
   state done
   }
}
```

Listing 5.1. Example SCCharts model

may influence the KIELER plugins and may require to change them while developing, since the IDE-plugins require functionality that was not already restructured to be IDE-independent. Moving the IDE-independent part from the UI-plugins to IDE-plugins is one of the most time consuming tasks. It has to be decided which features shall be restructured as IDE-independent and used by KIELER and KEITH. Moreover, it has to be evaluated for each feature, whether it should be implemented on the client side instead, which results in duplicate code in the Theia frontend and KIELER. Since this has to be done for every feature added to KEITH, it heavily influences the new architecture and is on the critical path in the development of KEITH.

Regular merges with the KIELER production branch are advised to keep the reused plugins up to date. Incremental updates allow to present the current implementation regularly to stakeholders and experts to ensure that it is developed correctly and required features are kept in mind.

5.5 Language Server

As mentioned in Section 5.1, the language server is built as an Eclipse application. This option is chosen over building a `jar`, since extension points for Eclipse are already provided in the Eclipse application. OSGi allows to register plugins via extension points. The compiler plugin needs the different compilation system. These are loaded via the extension points and therefore essential for the KEITH language server and KIELER. Furthermore, this setup allows to use the same build system as

the KIELER tool, since the only difference between the build of the language server and KIELER is the starting class and the fact that the language server needs no dependencies to UI-plugins. The language server needs an extension to the LSP, since the LSP does not support compilation of files and generation of diagrams, which should be added to KEITH. How such an extension can be added is presented in Section 5.7. Client side compilation is not an option, since it requires to reimplement the compiler in TypeScript on the client side. Compilation is slow in the browser and results in duplicate code, since the compiler does already exist in the KIELER plugins, which are part of the language server. Duplicate code is not desired and produces obstacles regarding maintainability, as mentioned in Section 4.1. Models are compiled in the language server using an LSP extension as it is common for language server implementations.

The language server can be started in different ways, as seen in Section 2.5. Therefore, two different starting methods are needed. One to connect via socket and one via stdin/stout. The difference between these two is how the logging and the input and output are configured. Parameters for the Eclipse application are used to configure the start method. Most of the implementation is reused with this approach and duplicate code is avoided.

5.6 Theia extension

As already mentioned in Section 2.6, Theia consists of several extensions. Yangster[16], theia-rust-extension[17], or all packages in Theia itself[18] can be used as an inspiration on how to implemented a new Theia extension. A Theia application can add packages for an outline view, an editor, an output view, a problem view, a preferences view, and more. An extension for a new language usually consists of backend and frontend. The backend is responsible for the connection to the language servers and enables to use the different connection types that are supported by the LSP, as described in Section 2.5.[19]

5.6.1 Theia Backend

A Theia extension can connect to several language servers and provide language support for all languages that provide a language server.

Although KEITH supports many languages, KEITH needs only one language server, since the KEITH language server supports all languages present in the KIELER project. Other language servers only provide language features for one language. This is inferior for KEITH, since the languages share many plugins in KIELER. Building a language server for each language results in a far bigger product and lots of redundant code, since all necessary plugins have to be added for every language server. The plugins used to compile and simulate are not language specific and have to be present in every language server that provides support for a language included in KIELER. They can only be efficiently reused if only one language server is built and used instead.

Listing 5.2 shows how a backend extension is structured. The backend has a `ContainerModule` that binds the `KiCoLanguageServerContribution`, as seen in line two. The `KiCoLanguageServerContribution` extends the `LanguageServerContribution`. It specifies the main language it supports, which is `"SCCharts"` with the id `"sctx"`. A contribution defines a `start` method that connects to the language server. The

[16] `https://github.com/theia-ide/yangster`

[17] `https://github.com/theia-ide/theia-rust-extension`

[18] `https://github.com/theia-ide/theia/tree/master/packages`

[19] All in this section described classes can be found at `https://github.com/theia-ide/theia` or are part of the developed Theia extension for KEITH

```
export default new ContainerModule(bind => {
  bind<LanguageServerContribution>(LanguageServerContribution).to(KiCoLanguageServerContribution)
  // more language servers
  ...
});

@injectable()
class KiCoLanguageServerContribution extends BaseLanguageServerContribution {
  readonly id = "sctx"
  readonly name = "SCCharts"
    start(clientConnection: IConnection): void {
      // call language server and connect to it
  }
}
```

Listing 5.2. Backend extension of Theia extension

```
protected get globPatterns() {
   return [
      "**/*.sctx",
      "**/*.scl",
      ...
   ]
}

protected get documentSelector(): string[] {
   return [
      "sctx",
      "scl",
      ...
   ]
}
```

Listing 5.3. Register language for syntax highlighting

`start` method enables the different connection types of a language server. Only one language can be registered directly at the language server contribution and not all supported languages. The other languages are registered in the `LanguageClientContribution` and the pattern for the supported languages, as seen in Listing 5.3. The `globPattern()` method specifies a pattern for the file extensions for SCCharts and the Sequentially Constructive Language (SCL). The `documentSelector()` methods specifies the corresponding language identifiers. These two methods are used to register all languages the KEITH language server can provide. The `LanguageClientContribution` allows to specify a list of patterns for the supported language extensions and allows to specify the supported language identifiers. The mapping between language identifier, which is unique and identifies the language, and language name, which is shown as editor language in the IDE, is done via the monaco highlighting registration in the frontend-extension, as seen in Listing 5.4. The language itself with its id, alias, file extensions, and mimetypes is registered. Moreover, the language id is associated with a corresponding highlighting stylesheet for the Monaco Editor. Since the `LanguageClientContribution` only registers file extensions and the language identifier, the registration in the frontend is needed to associate the file extension with the name of the language.

```
// register my language
monaco.languages.register({
   id: "sctx",
   aliases: ["SCCharts", "sctx"],
   extensions: ['.' + "sctx"],
   mimetypes: ['text/' + "sctx"]
})
monaco.languages.onLanguage("sctx", () => {
   monaco.languages.setLanguageConfiguration("sctx", sctxConfiguration)
   monaco.languages.setMonarchTokensProvider("sctx", sctxMonarchLanguage)
});
```

Listing 5.4. Bind language for syntax highlighting for the Monaco Editor

```
commands.registerCommand(SHOW_PREVIOUS, {
   execute: () => {
      // execute the command
   }
});
```

Listing 5.5. Register a command in the `CommandContribution`

Most of the IDE's capability to call the language server and write to a stream is hidden from the user via abstraction. Only the parts that should be configurable can be configured. This makes the Theia framework easier to maintain and to understand. The frontend of Theia implements the functionality of the extension by implementing different widgets.

5.6.2 Theia Frontend

Despite the connection to a language server the Theia extension for KEITH consists only of new commands to compile or show compilation results, new menus or menu entries, new keybindings, and widgets to show and compile the models.

Theia allows to register new commands by registering them in a `CommandContribution`, as seen in Listing 5.5. The command with the name `SHOW_PREVIOUS` is registered without any parameters. In KEITH, commands to navigate between the compilation snapshots called `SHOW_PREVIOUS` and `SHOW_NEXT` and a command to open the `CompilerWidget` are specified. Commands are registered via a `CommandRegistry` and can be defined to execute anything. This can be used to add new custom functionality needed by the individual developed extension. Commands are accessible via the command palette the same way as in VSCode, as mentioned in Section 4.2.2.

Menus are used to access IDE features such as commands or widgets that are also present in the command palette. A menu entry specifies its location in the UI, the corresponding command, and an alias for the command. This concept facilitates separation of concerns between business logic and UI. Menu entries do only bind functionality and do not influence the behavior themselves. How this can be done in TypeScript can be seen in Listing 5.6. The `SHOW_PREVIOUS` command is added to the edit/find section in the context menu. The label for this command is `"Show previous snapshot"`. Theia's menu framework allows to add menus in all menu bars and add new submenus for every command. Moreover, commands can be bound by keybindings to add a new way to access functionality, as seen in Listing 5.7. The `SHOW_PREVIOUS` command shall be executed if the `KeybindingContext` with the id `"keith.keybinding.context"` evaluates to `true` and the `Alt` and `j` keys are pressed. Keybindings always

```
@injectable()
export class KeithMenuContribution implements MenuContribution {

   registerMenus(menus: MenuModelRegistry): void {
      menus.registerMenuAction(CommonMenus.EDIT_FIND,
         {
            commandId: "SHOW_PREVIOS",
            label: 'Show previous snapshot'
         }
      );
   }
}
```

Listing 5.6. Example of a `MenuContribution`

```
registerKeybindings(keybindings: KeybindingRegistry): void {
   [
      // keybiding for SHOW_PREVIOUS command
      {
         command: "SHOW_PREVIOUS",
         context: "keith.keybinding.context",
         keybinding: "Alt+j"
      },
   ]
   // register each keybinding
   .forEach(binding =>
      {
         keybindings.registerKeybinding(binding);
      }
   );
}
```

Listing 5.7. Keybinding registration

have a context that specify when they are usable to filter for unwanted behavior and unnecessary exceptions. A keybinding context, shown in Listing 5.8, consists of a context identifier and the function `isEnabled`, which specifies if a keybinding can be executed. The `"keith.keybinding.context"` is enabled if its condition, here represented with `<condition>`, is `true`.

Widgets can be defined to add new views to Theia. Theia currently supports a widget for git integration, output, outline, preferences, extension management, editor, problems, and more. All of them are developed and published in separate projects and can be added on runtime to the IDE via the extension-manager package. When developing a new widget the existing widgets and their usage should be examined to adopt their design decisions if applicable.

The user can be notified about current activities by using the `MessagerService` for pop-up messages or the `OutputWidget`, which is predefined by Theia to log activities. The pop-up messages are used carefully in KEITH to not divert the attention of the user and to only inform about important events. Too many messages are not comprehensible by the user. Theia itself uses pop-up messages to display warnings and errors that occurred while executing Theia.

The following commands are added to implement some of the functionality of the KIELER tool: A compile command is added, which cannot be accessed via the command palette. The user can only

```
@injectable()
export class KeithKeybindingContext implements KeybindingContext {
   constructor() {

   }

   readonly id = 'keith.keybinding.context';

   /**
    * Function which indicates whether keybindings are executed
    * @param arg Keybinding which should be executed
    */
   isEnabled(arg: Keybinding): boolean {
      // compute <condition>
      return <condition>
   }
}
```

Listing 5.8. Example of a `KeybindingContext`

compile using the `CompilerWidget`, since it filters the available compilation systems and guarantees an active editor with a language that is supported by the language server. The available compilation systems are requestable from the language server for the different file extensions. The compilation preferences (e.g. auto compile) are changeable. A command to show the compilation snapshots and commands to navigate through them is provided.

5.6.3 Creating a widget

The Theia extension for KEITH implements two different widgets, one for the compiler and one to display the generated diagram as a mockup, since the real diagram view is implemented by Rentz [Ren18] parallel to this project.

The `CompilerWidget` needs to request information about known compilation systems from the language server and has to be able to be updated on request. The `CompilerWidget` can be seen in Figure 5.3. It extends the `ReactWidget`, which allows to implement the required feature. Since compilation preferences are stored on the client side, it is also possible to implement the `StatefulWidget` interface to load and store this preferences on close. This `StatefulWidget` interface can be compared to the `IMemento` interface in Eclipse. A `StatefulWidget` is currently not implemented for the `CompilerWiget`, but can be done as part of the future work on this project.

The second implemented widget is the so called `TextWidget`, which receives the text to display as a parameter. The `TextWidget` can be seen in Figure 5.5 in the main window on the right side. The `TextWidget` is used as a prototype for the diagram view. The diagram is displayed as an SVG, which is generated by KLighD [SSH13] via the offscreen renderer. Since the diagram is generated as an SVG the result can be displayed in a browser and is rendered if given the `TextWidget` as parameter. The SVG is stored in the HTML DOM and can be rendered by all common browsers. Therefore, the `TextWidget` is used as a mockup widget for diagrams. This is only a prototype solution. It is used to show the functionality the diagram widget can provide in the future. The communication between the language server and Theia can be mimicked to find possible problems that affect the final implementation.

For all features, despite editing, a new widget might be needed. Theia allows to add custom widgets to an extension by registering them in the `WidgetManager` or by attaching them to the

```
@injectable()
export class HelloWorldWidget extends BaseWidget {

   constructor(
   ) {
      super();
      this.title.label = "Hello World!"
      this.id = "hello-world"
      this.addClass('hello-world-widget')
      this.node.innerHTML = "Hello World!"
   }
}
```

Listing 5.9. Example how `HelloWorldWidget` can be implemented

`FrontendApplication`. The latter requires the user to open, close, and find the widget manually. The `WidgetManager` can be invoked to do this more comfortable. Therefore, it is advised to always use the `WidgetManager` to create widgets. All widgets extend the `BaseWidget` implementation, which allows to override default event methods e.g. on-close, on-update, and more. A widget has a title, a CSS class for its style, and some kind of HTML content. Furthermore, a scrollbar, drag and drop, an opening area (e.g. bottom), and closing behavior or possibilities can be configured in the implementation. A simple example of a widget can be seen in Listing 5.9. This widget has the title `Hello World!` and the id `hello-world`. The id is used to identify the widget using the `WidgetManager`. The class is the corresponding CSS class. A widget can have several CSS classes. The content of a simple widget is defined by its HTML content. The widget in Listing 5.9 has the title `Hello World!` and the text content with the specified style `Hello World!`. More complex widgets can be used to provide more functionality.

Theia itself has already implemented different widgets for specific use cases. Some widgets have a state that should be stored if closed and restored if the widget is reopened. These widgets can implement a `StatefulWidget`, which allows to implement relevant methods to store and restore the state on reopening of said widget. This means the state can be individually defined by the user. However, this can lead to errors on changing the state definition, since both methods have to be changed to reopen a stateful widget correctly.

Subclasses of the `BaseWidget` are `ReactWidget`, `TreeWidget`, `EditorWidget`, `TerminalWidget`, and more. Each of them is used for a specific purpose. The `TerminalWidget` is used to add a terminal window to Theia. It allows to open a terminal in Theia and execute shell commands. The `EditorWidget` allows to use an editor by adding the UI implementation, matching to a file in the file system, and adding open/close mechanisms. An `EditorWidget` can be seen in Figure 5.5. The program is modeled in it and it can be seen in the main windows on the left side. These two widgets are very specialized and are in most cases not extended. The `ReactWidget` on the other hand is suitable to be reused to implement own behavior. The `ReactWidget` simplifies UI generation and updating of the widget, since the React framework[20] can be leveraged. A `render` method can be implemented. The `render` method is called automatically on creation and can be invoked if needed. This allows to dynamically update the view on specific messages or events (e.g. some data that was requested earlier). The `CompilerWidget` seen in Figure 5.3 implements the `ReactWidget`. The `TreeWidget` extends the `ReactWidget` to use its render and upate capability. It allows to specify a lazy tree structure, which is displayed on the widget. The concept of laziness is used to load subtrees only if its needed, which allows to manage big trees with fewer performance issues. An example for a `TreeWidget` can be found in the `OutlineWidget` of Theia,

[20] `https://reactjs.org/`

```
new SCTXIdeSetup {
   override createInjector() {
      Guice.createInjector(Modules2.mixin(new SCTXRuntimeModule, new SCTXIdeModule, new
          KeithServerModule))
   }
}.createInjectorAndDoEMFRegistration()
```

Listing 5.10. Register an example language in the injector

```
def Class<? extends ILanguageServerExtension> bindILanguageServerExtension() {
   KeithLanguageServerExtension
}
```

Listing 5.11. LSP extension registration

which is used to display variables and the program structure of a model in KEITH. Another example of a `TreeWidget` can be seen in Figure 5.5 in the form of the file system on the left side.

In general, the already implemented Theia widgets can serve as documentation on when, how, and why a widget is implemented, structured, and called the way it is. Before adding an own widget, it has to be evaluated what the widget should display and what features it needs to decide which widget can be extended.

5.7 Extending the LSP

The LSP is often not enough to implement language specific behavior for a language. To add new functionality the language server has to be extended and the extension have to be used on the client side in Theia.

5.7.1 Server side LSP extension

KEITH needs a language server that can compile, synthesize diagrams, and, in the future, simulate. These features are not part of the LSP and are added via extensions. The supported languages have to be registered via the injector to add the language server functionality, since Xtext uses injection. The KIELER application does provide the necessary infrastructure to do this, as seen in Listing 5.10. Xtext generates a `SCTXIdeSetup`, a `SCTXRuntimeModule`, and a `SCTXIdeModule`. These are registered in the injector, which is used by the language server to access their language features. The Eclipse Modeling Framework (EMF) is registered in the injector via the `RuntimeModule` and the `IdeModule`. For each grammar the `RuntimeModule` and the `IdeModule` have to be registered using their corresponding `IdeSetup`. Additionally, it is possible to bind several new classes in an own module. KEITH uses a `KeithServerModule` that binds a `LanguageServerExtension`, as seen in Listing 5.11, which allows to add custom messages to the LSP. The method is used to bind the `KeithLanguageServerExtension` that implements the `ILanguageServerExtension`.

This server module is used to bind several custom implementation, but is only used for the `LanguageServerExtension` in the KEITH language server. The `KeithLanguageServerExtension` does implement all JSON-RPC methods registered in its interface `CommandExtension`, as seen in Listing 5.12. The `CommandExtension` serves as an interface for a language server extension. The `JsonSegment` defines the

```
1 @JsonSegment('keith')
2 interface CommandExtension {
3
4    /**
5     * Compiles file given by uri with compilationsystem given by command
6     */
7    @JsonRequest('compile')
8    def CompletableFuture<Object> compile(String uri, String command);
9
10   ...
11 }
```

Listing 5.12. Example of the `CommmandExtension`

prefix `"keith"` for the new messages defined in the `CommandInterface`. A `compile` method is specified in line eight, which is annotated as a `JsonRequest` with the name `"compile"`.

There are three different kinds of JSON-RPC methods: requests, notifications and responses. In the case of KEITH, requests are needed, since the Theia extension always asks for information or for the execution of a command. Furthermore, it does always await a result message.

The `LanguageServerExtension` implements the methods defined by the `CommmandExtension` interface and returns the result in form of a `CompletableFuture` to allow asynchronous messages. The LSP is easy to extend with custom commands for compilation or potential diagram generation. The language server makes use of the KIELER plugins, which can be easily added to the language server Eclipse application. This allows to add new functionality by adding new methods to the language server extension interface `CommandExtension` and implement these in the custom `KeithLanguageServerExtension`. The concrete function call is done by lsp4j[21], Xtext, and JSON-RPC. It enables to send messages to the language server and accept the result.

5.7.2 Client Side LSP Extension

The LSP can be formally extended on the language server side, as seen in Section 5.7.1. On the client side it is extended by executing new requests that catch the return value. An asynchronous request can be sent via the language client, as seen in Listing 5.13. The `executeCompile` method needs the compilation system as the parameter with the name `command`. It is an asynchronous method, which returns a promise. This promise is undefined until the method returns. The `await` keyword allows to wait until the method returns a value and fulfills the promise. Via the language client, accessible by a asynchronous method, a request can be sent to the language server. The method is `"keith/compile"`, which requests the execution of a method on the language server that is annotated, as seen in Listing 5.12. The executed method is the `compile` method specified in the `KeithLanguageServerExtension`. The parameters are `uri`, `command`, and `inplace`. The parameter `uri` refers to the URI of the model in the file system. The second parameter specifies the compilation system and the last one is a boolean that specifies whether inplace compilation is turned on or off. The return value is a `CompletableFuture` to allow asynchronous request. This allows to continue execution while waiting for the return value.

To implement a client side extension to the LSP a different request has to be sent to the language server. A command name and parameters allow to send such a request. The language client allows to send such a request asynchronous to the language server.

[21] `https://github.com/eclipse/lsp4j`

```
async executeCompile(command: string): Promise<void> {
   // initiale compilation
   const lclient = await this.client.languageClient
   const snapshotsDescriptions: CodeContainer = await lclient.sendRequest(
      "keith/compile",
      [uri, command, this.compilerWidget.compileInplace]) as CodeContainer
   // save result
}
```

Listing 5.13. Request to the language server in Theia

Remarks on Requests and Notifications

Requests are asynchronous and non blocking for the editor. Even if a compilation of a big model, which consumes many resources, is invoked, the request has no impact on the reactivity of the editor, since the request is handled by the language server. The language client also offers to send notifications to the language server that do not have a return value. Arbitrary messages can be sent via JSON-RPC between the Theia extension and the language server, as long as they are serializable into a finite string, as already mentioned in Section 2.5.

The LSP allows to send responses and notifications to the Theia extension to get data from the language server. Responses are part of a request/response pair. Therefore, these responses are used if an asynchronous function calling the language server should wait for a return value. A notification can be sent any time and does not require synchronization. The language server may always send notifications to inform the Theia extension that some action has to be or has been performed. The `DiagramExtension` for Yangster mentioned in Section 3.2.4 implements this concept for its diagram widget. One example of a request can be seen in Listing 2.1 and Listing 2.2, which show how a get definition request and response look like. Since the editor file with the corresponding definition has to be opened, it has to be waited until the response returns the location of the definition to jump to it. Request itself are cancelable. The request for content assist is cancelable via clicking in the editor or focusing another widget. A request that cannot be canceled has the potential to freeze the editor if the language server is busy or unavailable. Moreover, a go to definition implementation via notifications is not possible. The server could send the answer notification any time. Therefore, the jump to the definition can happen way after it was first requested if the language server cannot answer due to a high load. To avoid this scenario, a request is used in that case. A notification makes more sense, if the task may take a long time and execution can be safely resumed. The implementation of the `DiagramWidget` in sprotty uses such notifications. The Theia client notifies the language server (or diagram server) that a new diagram should be generated. This notification results in the creation of a new diagram that is sent via notification to the `DiagramWidget` in Theia and updates the diagram accordingly. These asynchronous notification define an interface to request an update of the diagram. The interface can be used by different widgets to request a redraw, for example to display the compilation snapshots via the `CompilerWidget` if requested.

5.7.3 Combining two LSP extensions in Theia

The steps to add a new LSP extension are shown on the example of the LSP diagram extension on which Rentz [Ren18] is currently working on and can be used as a guide. Two LSP extensions for Theia are combined to one language server and one Theia product as presented in the following sections.

Combining LSP extensions and Theia extensions will be part of future work on the KEITH project and are therefore important for KEITH.

Combine two Language Server Extensions

First, the new languages must be added. Therefore, the Xtext language has to be registered as shown in Listing 5.10. If the language is already recognized by the language server one has to add a `ServerModule` with a corresponding binding, as seen in Listing 5.11. If there is already a binding, the defined JavaScript Object Notation (JSON)-commands are added to the `CommandExtension` interface, as seen in Listing 5.12, and the implementation is moved to the `LanguageServerExtension` that implements these commands. One has to be aware that JSON messages with the same name are not compatible and have to be renamed for the language server to work with them. Therefore, a `JsonSegment` is used to generate a prefix for every group of commands. This means these prefixes have to be unique to avoid multiple command bindings.

To separate concerns in the implementation it is advised to only register the `ServerModule` of the second implementation separatly from the first `ServerModule`. The LSP extensions can be added or removed by changing only one function. This makes the project easier to understand and to maintain, since functions with a different context are separated into different classes. Moreover, functionality is easier removed as well as readded to the project if the different functions are capsuled in different `LanguageServerExtensions`.

Combining the two extensions is not always trivial if the two implementation have different dependencies. A common target platform has to be defined for Eclipse and conflicting dependencies have to be resolved. It is advised to set up a common target platform with expert developers of the KIELER tool and cross-validate the product build.

Combine two Theia Extensions

Combining two Theia extensions can be done without much effort if they are set up correctly and do not have circular dependencies or different version requirements.

One option is to publish the extensions as separate npm[22] packages. Publishing via npm is fairly simple and can be one done for every directory that has a `package.json`.[23] Separate publishing allows to have different development circles for the extensions and allows to compose a KEITH product with different versions for the underlying extensions. A product only has to specify the packages as dependencies in the `package.json` to deliver the content via the extensions.

Another option is to have the different extensions in different directories in the project repository. The option is chosen for the prototype KEITH implementation, but promises problems regarding Electron according to a discussion with leading engineers at TypeFox. Therefore, this implementation will be reworked into publishing via npm packages in future work on this project.

5.8 Development Setup

The KEITH tool and the language server are further developed and have to be debugged. Debugging requires a developer setup for the KEITH tool to debug the different components and their communication.

[22] `https://www.npmjs.com/`

[23] See `https://docs.npmjs.com/getting-started/publishing-npm-packages` for documentation

```
"scripts": {
   ...
   "start:backend:socket": "node ./src-gen/backend/main.js --root-dir=../workspace --LSP_PORT=5009 --
       port=3001 --loglevel=debug",
   ...
}
```

Listing 5.14. Command to connect the Theia application via a socket

The language server is started directly from Eclipse and connects via socket to the Theia extension. This socket can be monitored via the TCP/IP view from the Eclipse web-developer tools. Therefore, the Theia extension is not started on the port the language server uses, but on the one the TCP/IP view uses. This TCP/IP view forwards the data to the language server and allows to see the different requests. The same can be achieved by using VSCode's LSP-Inspector[24], which allows to monitor all messages on a specific port. In this case the Theia extension has to connect directly to the language server, since no port forwarding is needed. Both enable the developer to see the JSON-RPC messages send to and by the language server.

Debug Theia Extension

The Theia extension itself is started by running `yarn run start:backend:socket` in the `app` folder after compiling it. This command is specified in the corresponding `package.json`. As seen in Listing 5.14, it can be used to specify the initial workspace, the port of the language server, and port of the frontend application. A VSCode start command comes in handy to debug the node backend directly in VSCode and is used instead of the command defined in Listing 5.14 if the node backend is debugged. VSCode's command are specified in `.vscode` folders in the workspace. An example how such a start command looks like can be seen in Listing 5.15. The VSCode command specifies the main class of the program, the initial workspace, port of the language server, loglevel, port of the frontend, and the location of output files. Furthermore, several other preferences can be set.[25] It starts the Theia application and waits for the language server to connect, after the extension is accessed via the browser (e.g. connect to `localhost:3001`). The frontend can be debugged in the browser via the developer tools of the used browser, which enable to debug functionality as well as the CSS of the UI. The ports, on which the language server is started and the application is running, are specified in the `package.json` inside the `app` directory. The ports can be changed, as long as the new port is free and the `LSP_PORT` corresponds to the port on which the language server is started.

Debug Language Server

The language server can be debugged by running the `LanguageServer` class as an Eclipse application in debug mode and adding `"socket"`, the port number, and an optional host address to the arguments. This allows to specify where and on which port the language server is started. The port is the port of the TCP/IP monitor or the `LSP_PORT`, which allows to connect to the Theia extension if the port is specified, as elaborated in Section 5.8. The debugging itself is standard Java debugging and leverages all of Eclipse's debugging features [Sin17].

[24] `https://marketplace.visualstudio.com/items?itemName=octref.lsp-inspector-webview`

[25] `https://code.visualstudio.com/docs/editor/debugging`

```
{
   "type": "node",
   "request": "launch",
   "name": "Start Browser Backend Socket",
   "program": "${workspaceRoot}/app/src-gen/backend/main.js",
   "args": [
      "--root-dir=./workspace", // initial workspace
      "--LSP_PORT=5009", // port of language server
      "--loglevel=debug",
      "--port=3001", // port to connect via browser
      "--no-cluster"
   ],
   "env": {
      "NODE_ENV": "development"
   },
   "sourceMaps": true,
   "outFiles": [
      "${workspaceRoot}/node_modules/@theia/*/lib/**/*.js",
      "${workspaceRoot}/app/lib/**/*.js",
      "${workspaceRoot}/app/src-gen/**/*.js",
      "${workspaceRoot}/app/src-gen/**/**/*.js",
      "${workspaceRoot}/extension/lib/**/*.js",
   ],
   "smartStep": true,
   "internalConsoleOptions": "openOnSessionStart",
   "outputCapture": "std"
}
```

Listing 5.15. Command to start Theia application for socket connection inside VSCode

Debugging in the Browser

The last step is to open a browser on the port and host as specified in Section 5.8. As mentioned before, the browser itself can be used to debug the frontend and allow to configure CSS styles and properties for UI development. The web developer tools of the browser allow to inspect elements in the frontend and see their corresponding HTML and CSS properties. This allows to find errors and problems, which may occur. Furthermore, all frontend files can be debugged with standard debugging functionality such as breakpoints, step over, step into, and more. Nearly every browser has its own debugging tools, which deliver more or less the same functionality[26][27]. Debugging of a Theia extension takes place in the browser, since the backend is used to connect to the language server, and most new functionality is added in the frontend. This makes the debugging of the frontend in the browser the most frequently used debugging option.

[26] https://developer.mozilla.org/docs/Tools/Debugger
[27] https://developers.google.com/web/tools/chrome-devtools/

Chapter 6

Evaluation and Experience Report

At the beginning of this thesis the different steps were planed, but not every aspect or effect of the migration can be seen upfront and problems and risk cannot be planed in their full extent. Therefore, the general migration process with its different steps defined in Section 5.1 is qualitatively evaluated here, together with the overall performance of the developed tool. All mentioned remarks about the migration project and the development of the new KEITH tool try to be objective, but are influenced by personal observations and are subjective user experiences.

6.1 Migration Process

As seen in Section 5.1, the migration consists of an upgrade of the Xtext version in the KIELER plugins, the implementation of a prototype language server and Theia extension, and the iterative development of all necessary features for KEITH. While evaluating the development of the Theia extension and the language server, the reader has to consider that the developer of the language server and the Theia extension for KEITH is already proficient in Java development and is only a novice in web technologies. Therefore, the developer is not an expert in TypeScript development, which affect the remarks of developing with TypeScript.

The paradigm to reuse the existing code basis to avoid inconsistencies between the KIELER tool and the language server is applied in this migration project. Bug fixes and new features can be shared between KIELER and KEITH. Moreover, this allows to restructure the KIELER plugins itself to separate between the different concerns: UI and business logic. The reimplementation of the prototype compiler widget in KEITH also proved successful, since it allows to add the paradigms of Theia via web technologies. A direct translation of the widget UI breaks with Theia's UI and its HCI concept, since it reintroduces Eclipse's UI concept into Theia, which causes the same noisy UI and is not desired, as mentioned in Section 4.2.

6.1.1 Language Server

The language server consists of the KIELER plugins and wrapper code generated by Xtext to provide its functionality. Running it as an Eclipse application allows to reuse the existing extension points to provide its functionality as a service, as elaborated in Section 5.5. First, the separation of UI-plugins into UI-plugins and IDE-plugins was planed, which needs an upgrade of the Xtext version.

Xtext Integration

Since the language server generation is done by Xtext, the development process for a first language server worked out smoothly. The existing Xtext language server framework allows to register needed languages to make them accessible for the language server. Therefore, Xtext is a good choice for any project that uses Xtext grammars or plans to use them and tries to develop a language server.

Regardless of that, if a promising generator framework emerges, it can be used instead. The Xtext framework did not cause any trouble while migrating.

Upgrading Xtext

For upgrading the Xtext generator environment, the guide provided by TypeFox was used, as elaborated in Section 5.1 [Köh16]. The general steps consist of the creation and preparation of an IDE-plugin, the conversion to Xtend, and the change of the mwe2 workflow. Conversion to Xtend can be done by Eclipse, but the resulting code may be difficult to understand, due to the new dialect and missing comments in the translated code. Manual effort may be needed to do this. Furthermore, problems with SCCharts, which is a hierarchical grammar, occurred in KIELER, since the use case is not documented in the migration guide. The behavior of the grammars (e.g. for SCCharts) are checked after the migration to be sure that all required fragments are added and that they deliver the same functionality. This proved to be necessary, since the new Xtext generator infrastructure changed some fragments, abolished some fragments, and changed the name of others. The upgrade itself is well documented by TypeFox, one of the maintainers of Xtext. IDE support in Eclipse is provided to ease the upgrade.

Dependency Analysis

Since the language server itself shall not have dependencies to Eclipse UI-plugins, the plugin dependencies have to be checked and evaluated. The previous step already provided this in KIELER, which allows to skip this step. It was initially planned to do a full dependency analysis and reverse engineering of the plugins, but this is not needed, since the Eclipse application can be started without a UI. Omitting the dependency analysis saves time in the migration, since only the features that are currently necessary are migrated. For other migration projects, it might be helpful to evaluate the known dependencies beforehand and to solve obvious problems such as non UI-plugins having dependencies to UI-plugins, as mentioned in Section 4.1.3.

IDE-plugin Development

While iteratively developing the Theia extension and the language server, certain parts of the UI-plugins were deemed not IDE-specific and have to be moved to an IDE-plugin. For example, the list of compilation systems for a file were originally only available in the `kicool.ui`-plugin and were moved to the `kicool.ide`-plugin to use them for KEITH. The list of compilation systems is part of the compiler functionality that is not needed for the prototype of KEITH. The restructuring is a blocker that does not allow to merge the restructuring in the master development branch of KIELER, since the original developer does not comply with the resulting restructured plugin. To avoid possible duplicate code in the UI of KEITH and problems that might occur from this strategy, presented in Section 4.1.3, the whole workflow and user stories of the KIELER tool were planed to be evaluated by an expert. This was, however, not done, since a step-by-step restructuring seemed to work without it. Therefore, not every aspect of the needed functionality is restructured and parts of the migration are done if they are needed. Every restructuring step has to be discussed and evaluated before doing it to avoid the problems that were encountered in the migration of KIELER. Iterative development of KEITH and the language server is used to get a prototype early on in development and enables the developer to restructure UI-plugins later. Evaluating everything beforehand hinders development, since a minimal running example is achieved later in the development process. This minimal prototype for KEITH is used to identify critical problems and the critical path of the project early on. It helps to assign priorities to the different tasks.

Extensibility

The LSP is designed to be extensible and the Xtext implementation enables the developer to add own extensions to the language server easily and unhindered. New extensions can be added, as seen in Section 5.7.1, with minimal effort and do not cause any problems on the language server side. They are only limited in their expressiveness by the fact that the return value of messages has to be serializable. Languages are registered in the injector, which makes it easy to add new languages. The supported languages can be configured for different language servers and different products. The language and syntax highlighting registration in language server and Theia extension have the potential to be automated. This guarantees that all desired languages are added into the product the right way even on grammar changes. Furthermore, the language server has the potential to propagate the supported languages to connecting applications to allow to add them dynamically into an IDE. These features are currently not implemented, but are considered as future work on this project.

The language server allows to reuse the existing plugins of the KIELER implementation. This solves the issue of legacy product divergence that was identified as one of the main problems in migration projects, as mentioned in Section 4.1. Nevertheless, this requires a distinction between UI-dependent and UI-independent implementations in the KIELER plugins. Since this is a non-trivial task, it has the potential of errors or wrong design decisions. Some part of the functionality performs better on the client side, for example filtering of compilation systems. Some is better done on the server side, since an implementation already exists in the plugins used for the language server. Hence, such functionality can be easily integrated into the language server, but potentially causes too much communication between client and server, which might hurt the performance of the product. A client side implementation on the other hand results in duplicate code. The functionality that is already implemented in the language server has to be developed in TypeScript in the Theia extension for KEITH as well. This is more difficult to maintain, since both implementations have to be changed in case of an update. These necessary design decisions do influence the overall extensibility, since the place of the implementation is relevant for the overall performance. The design decisions have to be discussed with experts and stakeholders to prevent wrong decisions, which have to be corrected later. This slows down the development process.

Size Problems

The bundled version of the product, including the language server, has a size of 361 MiB.How this size is spread across the components can be seen in Figure 6.1. KEITH and the language server have a size of 361 MiB, the language server has a size of 258 MiB, the plugin folder has a size of 254 MiB, and all UI-plugins and the plugins included to deliver an Eclipse application have a size of 114 MiB. The UI plugins and the Eclipse application plugins are only included, since extension points are used and the language server must be delivered as an Eclipse application. A language server only has to be built as an Eclipse application if extension points are strictly needed. If KIELER and therefore KEITH is built without extension points, the size of KEITH can potentially be reduced by 114 MiB. The language server can be delivered as a fat `jar` instead.

KEITH is bigger than standard IDEs and exceeds the size of lightweight editors, which Theia claims to be, by far. The language server is to blame, which is bundled as an Eclipse application. Most language servers do not provide compilation and diagram generation together with support for multiple languages. More than six programming languages are supported by the language server and KEITH. Although the size of the language server is a problem and its reduction is part of future work on this project, the size is justified through the supported languages and features.

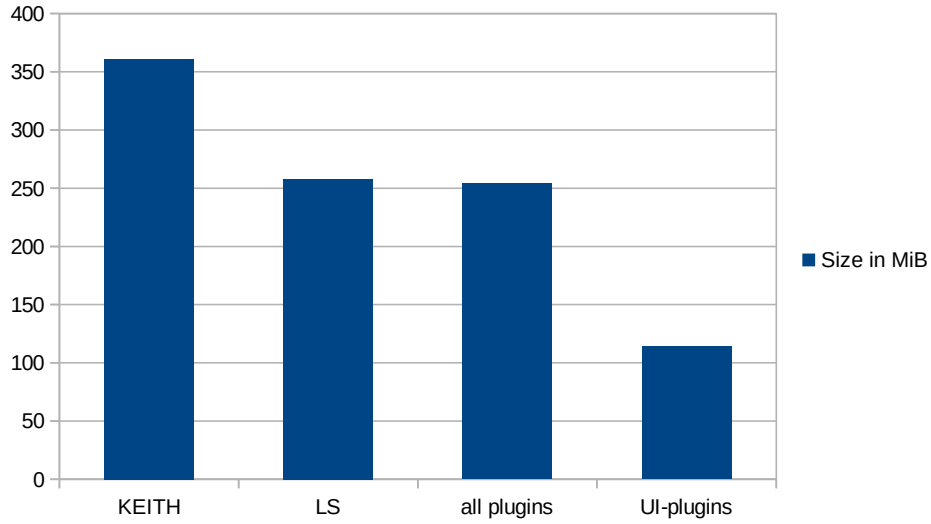


Figure 6.1. Size distribution in KEITH and the language server

Execution Modes

The language server supports two different execution modes to enable the developer to use it in various contexts, as explained in Section 2.6: the connection via socket and the connection via stdin/stdout. The socket connection is vital for the development process, since it is used to debug the application, as seen in Section 5.8. However, only the connection via socket is part of the development setup and can be debugged. The connection via stdin/stdout is only available if the language server is already bundled as an Eclipse application. This Eclipse application cannot be debugged with standard debugging tools. Bugs that only occur on connection via stdin/stdout are therefore not efficiently debugable.

Conclusion

In general, the migration to a language server worked out smoothly. After the restructuring of the plugins was finished, a starter class was enough to run the first language server. However, implementing new extension and therefore the migration of former UI-plugin functionality to IDE-plugins may cause problems in the future. The decision whether code should be duplicated in a client side implementation or reused from the legacy plugins has to be evaluated for each occurring problem and is a difficult software engineering task. Moreover, the language server as an Eclipse application is quite big and its size can be reduced. The different execution modes are not equivalent and only one of them can be debugged with all tool support.

6.1.2 Theia Extension

Theia's documentation mainly consists of code examples. Since there are already several Theia extensions, they can be used as an example on how to use the technology and where and how concepts and features are used.

Developing a Theia extension for KEITH is not only coding in TypeScript and developing new LSP extensions to add a new features. A non-negligible amount of time is used to build and test

the application. Theia allows to use a watch mode while compiling, which listens on changes and updates the running application on reload in the browser. This feature may be good, but has to be used together with the `-mode development` Theia build-option to be quick enough to build the product without a significant delay. Building an application takes time and not every change can be applied immediately to the running Theia application. While developing, the watch mode was not used, since it was confusing. It was not always clear which version of the tool is currently running, since delays while building were too big. Not only running of the application may be an issue, but also debugging has to be kept in mind.

Debugging

To debug a Theia extension, the frontend, the backend, and the language server have to be observed, as elaborated in Section 5.8. This leads to three different locations and tools that may be used for debugging, not including tools to monitor the sent messages. Switching between all these tools was confusing in the development of KEITH and hinders debugging of the application, since the current status of KEITH is more difficult to monitor.

Development tools such as VSCode allow to set breakpoints in classes that are ignored such as breakpoints in the frontend classes. Since the frontend of Theia runs in the browser, it is debugged in the browser, while the node backend is debugged in VSCode. While developing KEITH, it was a confusing feature of VSCode and led to some time spent trying to fix this by configuring the debug setup in VSCode.

Monitoring LSP messages can be done via the Eclipse TCP/IP view, which is provided by the Eclipse web developer tools[1]. This option has the disadvantage of slowing down the communication and also slowing down the responsible Eclipse application.

The development in the browser allows to debug CSS and frontend functionality via the developer tools present in all major browsers (e.g. Firefox and Chrome). Many different tools are used to debug a Theia extension, but only one or two of them are used at the same time. Most times, only one part of the application is debugged, which simplifies the debugging process of KEITH and makes fewer tools relevant. Using fewer tools helps understandability and therefore maintainability of KEITH. Nevertheless, different tools and components make certain bugs more tedious to track if the whole tool chain is required. This resulted in a confusing debug environment while developing KEITH, which hurts maintainability and understandability and makes the project setup and development tiresome. Developers have a steep learning curve, because of the different tools that are needed for development.

While using these debugging tools the following observations are made regarding their usability. During the development of KEITH debugging in VSCode failed at the beginning, because the source mapping in VSCode was not turned on. Without source mapping breakpoints in TypeScript are not mapped to the corresponding JavaScript files.

The TCP/IP view causes some problems. If too many, too big messages are sent, the TCP/IP view is able to crash Eclipse, since all sent messages are saved by Eclipse. If it is used to debug the messages sent to the KEITH language server, the language server should be restarted every time the Theia application is restarted. This was discovered while sending SVG files to the Theia application. On the bright side, the TCP/IP monitor is only necessary if the communication is debugged and can be omitted if this is not the case or the VSCode LSP Inspector can be used instead. The only thing that changes if the TCP/IP view is omitted is that the Eclipse application gets the `LSP_PORT` of the Theia application as an argument, as elaborated in Section 5.8.

[1] `https://marketplace.eclipse.org/content/eclipse-web-developer-tools-0`

Using TypeScript

Theia extensions are developed in TypeScript. Since TypeScript, despite all its features and abstractions, still compiles to JavaScript, it inherits some of its disadvantages. Specifically, runtime errors and their lack of expressiveness are still present in TypeScript. TypeScript can avoid most of these problems, but not all of them. Wrong bindings, problems while registering Theia contributions, or other setup problems were difficult to find and to fix while developing the Theia extension for KEITH. If a class is bound wrong, the exact reason cannot be found by the compiler, as seen at the error message `Error: Ambiguous match found for serviceIdentifier: e planner.js:74`. This error trace occurred because of a wrong binding and shows an exception that occurred in a generated file. This is a problem, especially for novice programmers. Most of these problems have a one line solution that may be hard to find since injections are used to enable a high level of abstraction and to solve problems with the scoping that may occur in JavaScript. Moreover, TypeScript's package system leaves the implementation open to bugs. The packages have various dependencies, which may change. The `yarn.lock` file used to save the package versions may not be enough to solve this issue. It allows the user to update specific libraries to a specific release, but does not have the ability to specify a compatible version range, as it can be done in other language e.g. for Java in Eclipse. Package managing in general does not have the same tool support in web technology IDEs such as VSCode, as it is implemented for Java in Eclipse. This results in manual and therefore often error prone configuration and problem solving in the quickly changing environment of JavaScript packages. This can be a tedious task and needs experienced developers. The open-source channels of Theia and other development tools can be used to get help and to find bugs.

Bugs and Documentation

Theia is still in development and sometimes has bugs. Since Theia is an open-source project and the extensions are published via npm, Theia has many developers who are willing to fix bugs quickly. Bugs can be reported using the open-source channels GitHub[2] or Gitter[3]. This does allow to have direct contact to developers and enables the developers of KEITH to ask questions and to get help in case of serious problems. Tickets with a reasonable amount of effort are fixed quickly and the growing Theia community promises earlier found bugs and more active developers for this tool. Furthermore, Theia's extension updates are continuously delivered, since npm is used. TypeScript itself allows to specify not only major versions, but can be used to reference the last working version via the `next` keyword.

Developing a Theia extension can be learned in a short amount of time. As mentioned in Section 5.4, there is already a `hello-world-extension` for Theia and several existing frontend and backend extensions. These extensions serve as documentation for the use of widgets and existing Theia core classes. However, documentation in form of similar projects might not always be the best idea. The existing implementations are not bug free, which may cause problems in the own project if they are redundantly copied. This relates to errors caused by copy and pasting legacy code into the own implementation, mentioned in Section 4.1. If an error is fixed in this copied code, the fix is not distributed to the own implementation. Furthermore, it is often not documented, which example implementation is used to inspire the own implementation. This leaves no way to know where to look to solve occurring problems. The existing open-source channels can be used to get help from the community or the original developers, as mentioned earlier.

[2] `https://github.com/theia-ide/theia/issues`
[3] `https://gitter.im/theia-ide/theia`

Extensibility

Theia promises to be highly extensible. The Theia native packages enable to configure Theia in different ways from the most basic editor functions to git extensions, keymap extension for rebinding of keybindings, an extension manager to add new extensions on the fly, and more. This allows to add desired functions. It is recommended to only use native Theia packages with the same version number to avoid problems caused by inconsistent versions that occurred in KEITH at the beginning of the development. To get the advantage of the rich extension environment, the packages of KEITH should be updated regularly to be able to adopt to API changes and be able to use Theia's new features. Of course, this does not mean that API changes cannot be controlled. The `yarn.lock` can be used to control these updates and allow to always build a working version, regardless of API changes, as mentioned in Section 6.1.2. If the `yarn.lock` is not changed, a working version can always be built as long as libraries are still provided.

As described in Section 5.7.2, the client side extension of the LSP requires minimal effort and allows to call the language server in all contexts that allow asynchronous calls. New functionality can be added easily. The same can be applied to commands, menus, keybindings, and widgets, as presented in Section 5.6.2. Several widgets can be added, which can be independent from each other to allow modular programming. Theia's package structure encourages to publish a widget and its functionality in an own component, as seen in the native Theia modules[4].

The package system of Theia, presented in Figure 2.6, shows that Theia is as extensible as claimed. Every aspect of its implementation is designed to add new functionality independent of the existing packages. Furthermore, the existing packages allow to configure Theia on runtime or bundle specific extensions into a product, making it not only extensible, but also configurable.

Conclusion

The migration to a Theia extension for KEITH is limited by the underlying language TypeScript. Stack traces of errors are often not usable and project management has little tool support in VSCode. Moreover, bundling as an Electron app is not trivial and Theia's different deployment methods do not behave the same in all cases. Despite that, Theia is highly extensible and configurable and can be adopted to work in different scenarios and deployed in different ways. Debugging of Theia is also possible, but limited by the need to use different tools to debug communication and different components.

6.1.3 OS-Independence

The need to build a running Theia product for different OSs is destined to cause further problems. Building the language server causes no problems, since Eclipse uses prebuilt Eclipse binaries to build the application for different OSs. If the application is not delivered as an Eclipse application the JVM enables the application to be platform-independent. Bundling the Electron app for several OSs is a non-trivial task. Electron-builder is used to build a KEITH product for each OS. The documentation, by Electron Userland[5], for the tool was not always easy to find while developing KEITH. Windows and MacOS have the option of signing the KEITH Electron app. Windows allows to execute unsigned apps, while MacOS does not if the signing is just left out in the build tool. This requires manual configuration to disable it or sign it with only the existing open-source channels as documentation. This can be a tedious task, since the try and error principle is not applicable here because the rebuild of

[4] https://github.com/theia-ide/theia/tree/master/packages

[5] https://electronjs.org/userland

the product takes too much time to do this efficiently. The Electron product needs roughly 30 minutes to build. Furthermore, the different versions have to be built natively on each OS. The use of tools such as Travis and Appveyor can solve this problem, but these are not considered for this project, as mentioned in Section 5.3.3. This requires to have these OSs available to execute the build jobs via Bamboo build agents, which have to be maintained. Hence, KEITH could profit from using tools such as Travis or Appveyor to build cross-platform. Furthermore, the configuration files of these tools can be checked in into the version management system to document changes in the configuration and workflow.

One of the goals of the migration is to have a platform-independent framework to develop an IDE using web technologies. Theia promises that. Despite this promise full platform-independence is not achieved. The UI is not platform-independent. The UI looks different in different browsers and it looks different on different OSs in the bundled Electron app. This does not only apply to a different style for selectboxes, frames, or similar UI elements, but rather to the alignment of icons. This is more difficult to debug the UI for all OSs, browsers, and different starting methods of Theia. Non platform-independent code is not limited to the UI. The same development setup seems to cause problem when developing with a Windows OS. The connection to a language server cannot be sustained on Windows, since the path in the filesystem is erroneous resolved and causes an exception in the language server. The bundled Electron application also seems to have different behavior on different OSs. One exception that is ignore on Linux causes the language server to disconnect on MacOS.

6.2 Development Tools

Not only the migration process and the resulting application KEITH, but also the used development tools and their setup time have to be evaluated. Development tools heavily influence usability, maintainability, and therefore how easy the migration and further development takes place. In this section development in VSCode and Eclipse are evaluated.

6.2.1 Development in VSCode

Since the Theia extension is developed in TypeScript, an IDE that supports web development has to be used. VSCode itself is implemented in TypeScript and one of the main influences in the design of Theia. The developers of Theia are using VSCode for their development of Theia, which suggests that it is the ideal tool for this task. Using the same IDE promises reproducible problems and therefore more reliable help via the open-source channels. It is also possible to use a different development tool, but since VSCode is written in TypeScript it has native support for the language. However, since the language server is developed in Eclipse, the Theia extension can also be effectively developed in Eclipse to reduce the used technologies.

VSCode itself does not require any project setup to edit. It allows to add folders from the file system into the workspace without any configuration. To start the application a build via the `yarn` command is needed. A start command runs the application. VSCode can suggest the newest version of package dependencies in the `package.json`, but does neither add dependencies automatically, nor finds conflicts between versions. VSCode recognizes the git repositories in which the opened folders reside and supports version management for git. Moreover, VSCode can debug the backend implementation of Theia by defining executions in form of JSON files. These have the advantage that they are versionable and can therefore be shared via a git repository. Versionable executions and no workspace configuration allow to recreate development setups more easily and ease development in teams. However, VSCode cannot debug frontend files, but allows to add breakpoints in them that are

ignored and marked as inactive if the program is run. These features and only minor inconveniences in the package management make VSCode ideal for developing a Theia extension in TypeScript.

6.2.2 Development in Eclipse

Eclipse seems heavy weight in comparison to VSCode. Eclipse is currently used in the development of the KIELER tool. It requires extensive setup, but is ideal to develop a language server in, since Xtext is integrated. The setup is configured by an Oomph setup[6]. Since KIELER is developed in Eclipse, Eclipse experts are present in the development team. Therefore, the knowledge about the extensive setup can be shared with new developers.

6.3 Performance Testing

Migrating to a new framework often happens with the intention to improve performance of the legacy application. Since KEITH itself does use the plugins of KIELER as a service, the overall performance, e.g. the compile times, will not improve. However, since the product is separated into two different components it promises better reactivity, usability, scalability, and maintainability. Therefore, performance is evaluated qualitatively in the sense of these non-functional properties. For the evaluation, several small models and a big model of a railway controller for a model railway are used. My participation in the railway project[7] provides the insights about developing in KIELER and modeling with SCCharts. The railway project requires to build a controller for a model railway network via SCCharts, which involves to route and synchronize up to ten different trains on a complicated model track. Therefore, some of the evaluations of KIELER apply to the version used in summer semester 2017.

6.3.1 Reactivity

Reactivity is a non-functional property that describes the time that passes between user input and reaction of the tool. The reactivity of the KIELER tool and KEITH is only subjectively evaluated and was not measured in this evaluation.

The KEITH Theia application does not need time to set up the workspace on startup similar to VSCode. On the other hand KEITH needs time to start and connect to the language server. In browser mode via socket, the connection time is rather small, since the language server itself is already running before the Theia application connects. If KEITH is started as an Electron app and connects via stdin/stdout, the initial start and connection of the language server may take some time. The language server itself is only started if a corresponding model is opened (e.g. a `.sctx` file to develop SCCharts). Nevertheless, the editor itself is ready on start up and code can be written directly after the application is started. KIELER is rather slow on startup. The application takes considerably longer to open, but is working immediately with all rich language features when it opens. If the start of the language server is added to the startup time both applications are nearly equal. Nevertheless, Theia feels subjectively more reactive, since the editor itself opens up faster and allows to edit models immediately.

The asynchronous request to the language server allows to cancel each request and does not limit the reactivity of the editor. This is achieved since compilation, mock diagram generation, and rich language features are implemented on the language server. Even if a large model such as the railway controller is compiled, the editor and the whole IDE are still functional. However, the language server

[6] `https://projects.eclipse.org/projects/tools.oomph`

[7] `https://rtsys.informatik.uni-kiel.de/confluence/x/VABgAQ`

is unavailable for other services. Models can still be edited and requests to the language server in the form of content assist can be canceled, as mentioned in Section 5.7.2, even though they do not get an answer. KIELER on the other hand handles requests differently, since it is an Eclipse application. Nearly all requests are blocking, which results in the following problems: Opening large models or importing big projects into the workspace slows down KIELER heavily and blocks the application. While a new big model is opened the developer cannot do anything else than wait. All other requests are queued and are suspended. Since it is not easy to cancel *open file* or *build workspace* requests in an Eclipse application, this slows down the development process and frustrates the user as experienced in the railway project. Opening the content assist menu for a large model sometimes freezes the whole application in prior versions of KIELER, which blocks all functionality for a short duration, as experienced in the railway project. Since Eclipse itself is a rather memory hungry application this can slow down the whole system and not only the development tool. On the other hand compilation itself does not block other language features in KIELER. Moreover, the browser used for KEITH is also memory hungry and needs many of the systems resources. The overall better performance also helps to achieve a better reactivity in KIELER.

Usability

Reactivity relates to usability. Usability is the non-functional property that describes, how well the user of a tool is able to access functionality and describes the overall user experience. The evaluated usability in this section is purely subjective.

A non-reactive tool does hinder the user and diminishes the user experience of a tool. A tool that is not reactive is less usable and therefore not desired. The convenience of functions and how they are accessible influences usability. KEITH does currently not support all features provided by KIELER. Usability is evaluated using only the already implemented features.

The Theia framework hides most of its functionality in the command palette. For me, this reduces the noise of the IDE. Moreover, commands are searchable via the command palette. This eases the struggle in extensive menus, as experienced in the command palette for VSCode while developing KEITH. Editing of a model is not influenced by compilation or other calls to the language server, because of the already mentioned asynchronous requests. KEITH feels like a lightweight editor in terms of usability because of the previously mentioned features.

KEITH compiles slower than the KIELER tool. Nevertheless, the compilation time and communication overhead is rather small for small or medium sized models and does not hinder the user. Compilation of large models tends to block the language server for several minutes and to hurt usability. The compilation systems are automatically requested on a model change. A change of the current editor model to a large model (i.e. the railway controller) produces noticeable delays while updating the compiler view. This can be optimized in future work, since only the file extension is relevant and the size of a file stands in no relation to the available compilation systems. KIELER, however, needs this feature, since not every model is represented in the file system.

6.3.2 Scalability

Scalability is the non-functional property that describes how well a distributed system can support more users by adding more of a specific component. Scalability cannot be evaluated for KIELER, since it is not a distributed system.

Theia allows various different setups and connection types, as presented in Section 2.6. However, this does not mean that the application can scale. The connection via stdin/stdout does not allow to reuse the language server between different Theia instances if it is bundled in an Electron app.

Several Theia applications can connect to the same language server using the connection via socket. However, compiling large models via one KEITH instance fully blocks the language server for the other ones. Therefore, it is not advised to reuse a language server for more than one IDE instance. One Theia application can use more than one language server for the same language if some kind of middleware is used for load balancing and to route the messages. Such a middleware is currently not provided by the Theia framework.

Scalability can be achieved via docker or similar container platforms. Figure 2.7 shows the separation into a Theia client and a docker container. In this scenario, the Theia application is accessible via a browser over the internet. A server hosts container images and starts new ones on demand if a new user connects to the server via a browser. A container itself contains the KEITH application together with the language server. It is also possible to host the language server on a different server and use the socket connection feature to connect KEITH to it. However, this produces more communication overhead.

6.3.3 Maintainability

Maintainability is the non-functional property that describes how well a software can be maintained and how well new features can be added. Maintainability is influenced by extensibility. The remarks regarding maintainability are purely subjective. A tool that allows to add new features easily is easier to maintain, since new functionality can be added without much effort. Since KEITH does currently not implement all features of the legacy IDE KIELER, it is necessary to add them later on. Therefore maintainability is important for KEITH.

As described in Section 6.1.1 and Section 6.1.3, different execution modes and OS-independence influence maintainability and will be an obstacle in the future development of KEITH.

The language server itself uses the same code basis as the plugins of the KIELER tool. This allows to reuse the code basis of KIELER and to maintain the language server together with the KIELER implementation. However, the bug fixes and improvements of the KIELER plugins have to be regularly merged into the KEITH development branch and vice versa to keep changes synchronizes, since it is currently not possible to build KEITH on the master branch. The restructuring of the UI-plugins was not approved, as mentioned in Section 6.1.1. This may result in a version conflict between the language server and KEITH in the future, since one product may divert to use different libraries if they are not merged for a long time. New development dependencies between unrelated products are created that have to be maintained and may result in a breaking conflict in the future if one product is not able to adopt certain libraries. Furthermore, new features for KIELER or KEITH have to be designed with separation of concerns in mind. Most features have an IDE-independent core implementation that is used as a service for the corresponding products. The UI for KEITH or KIELER has to be designed to use the implementation from IDE-plugins to avoid duplicate code, as mentioned in Section 6.1.1. IDE-independent feature implementations enforce separation of concerns on the whole project landscape, which may lead to better and easier to maintain code. However, different tools, languages, and development teams decrease maintainability. Moreover, separation of concerns comes with the price of new dependencies and more products to maintain, as well as more difficult design decisions when adding new features. For every new feature it has to be evaluated if it should be implemented on the client side of Theia, which may result in duplicate code, since this feature is also provided by the KIELER product. However, a client side implementation performs often better, since no communication with the language server is needed.

The developer base for the used languages have to be taken into account when evaluation maintainability. The KIELER product is developed in Java or Xtend. KEITH, however, uses TypeScript as main

development language and HTML and CSS to design UIs. Moreover, the Theia framework is new and yet only known to few people. This leads to fewer developers for the new product. New developers have to be trained and the already employed developers have to be taught how to these web developer languages and the Theia framework to develop KEITH in the future. Fewer capable developers influence maintainability and limit the work on KEITH.

The maintenance of the product is not only influenced by the own developers, but rather by the maintainers of the Theia framework, namely TypeFox. TypeFox controls what is added to the released packages and therefore controls how quick bug fixes are applied. The Theia framework is an open-source project, therefore it allows to apply bugfixes ourselves and contribute to the project via pull requests. However, this is rather time consuming and limits the work on KEITH itself. More company support for Theia can solve this. More users lead to more active developers of the framework and therefore more developers who solve open issues.

6.4 Use of KEITH in Teaching

The Theia framework was adopted to develop a new academic tool for research and teaching. Future work on this project involves to make KEITH ready for that use case.

KEITH feels like a lightweight tool. However, the application itself together with the language server is rather big (roughly 400 MB). This can hurt usability on small devices with limited storage.Students at the Kiel University have accounts with limited storage on the servers at the department of computer science. A solution with docker or similar containers, to make the IDE accessible online, is currently not implemented. Furthermore, students work on the same servers and share their ports, which hinders the use of the socket mode in KEITH. All these obstacles hinder the use of KEITH for students at this university and have to be kept in mind, when planning to use KEITH for teaching purposes. These problems can be avoided by providing a viable container solution or by using the Electron app and disregarding its size.

The Theia framework does currently not contain any features for the container setup. Starting container platforms on demand has to be implemented around Theia. Hosting a KEITH application without a container platform is not an option. KEITH allows to access the file system and to open a terminal on the hosting server. This can lead to abuse of these features by accident or malicious intent. A viable docker infrastructure can enable the usage of KEITH in teaching.

6.5 Comparison

KEITH does not improve the overall performance of the product and does not implement all of KIELER's features, but surpasses the legacy implementation in other disciplines: The strong separation between UI and business logic allows to write IDE-independent features. These features are accessible as a service via a language server. This enforces the separation of concerns pattern, since duplicate code is avoided in the migration. KEITH reduces the noise of the UI by hiding the functionality in the command palette. KEITH is more reactive, since requests to the language server are asynchronous. However, KIELER has a higher performance, since no communication overhead is needed. Moreover, KIELER is not as big as KEITH because of taken design decisions, as elaborated in Section 6.1.1. KIELER has no option to scale by using more than one UI, IDE, or business logic component. KEITH allows to start the language server and the Theia extension separately, which allows the application to scale. However, scaling is not always applicable. It is not advised to use the same language server with several Theia instances. KEITH is able to reuse the plugins of KIELER. This makes the migration easier and the two products able to be

maintained in parallel. KEITH adds dependencies to several open-source projects. Furthermore, KEITH is developed in TypeScript. Since KIELER is not developed in TypeScript, no experienced developers are available for KEITH development. Therefore, the development of KEITH is destined to be slower and more error prone than development for KIELER. The UI of KEITH behaves differently in different browsers and OSs, which makes maintenance more difficult. The different setup methods of KEITH are not always helpful, since they produce different bugs. These bugs are harder to reproduce, since the Electron app tends to run into different problems than the unbundled version or browser version. KIELER on the other hand does not have this problems to this extent. KIELER is browser-independent, since it does not run in a browser. Furthermore, KIELER has no different setup methods, which therefore cannot cause problems. Using web technologies requires to use several frameworks to develop, which are difficult to keep track of. Many other obstacles of the development with the new Theia framework in TypeScript are yet to find. However, Theia itself seems promising as a tool, because of its high extensibility and its versatile deployment methods.

Chapter 7

Conclusion

This thesis focuses on the migration from Eclipse to web technologies. The KIELER project is migrated to a Theia extension for KEITH that reuses KIELER as a language server. In this chapter the insights about the migration, the resulting product, and implementation process are summarized. Future work is presented to show the potential and new projects to further develop KEITH.

7.1 Summary

This migration project from Eclipse to web technologies results in KEITH. A Theia application that uses the LSP to reuse the backend of the KIELER tool as a language server to provide rich editing features and compilation in the browser or in an Electron app is developed.

7.1.1 Migration

The main strategy of this migration is reusing code to develop two IDEs in parallel with the same backend. Furthermore, continuous development, extensibility, and automation are kept in mind to make the implementation dynamic and less error prone, since manual configuration or manual copying of files is avoided. The language server technology together with an Xtext grammar enables exactly that by using the same Eclipse plugins used for the language features of the KIELER tool to generate a language server that can be easily extended. This makes it possible to reuse and merge the backend implementation of KEITH with the language features of KIELER. Therefore, one of the main problems in migration projects can be avoided, since bug fixes and changes to KIELER are automatically available to be merged into the KEITH language server. Only the UI dialogs and widgets have to be reimplemented.

Migrating to web technologies results in some problems. The technology stack changes and current developers are not familiar with the new technology. Training of old and new developers in the usage of web technologies is needed in this migration. Furthermore, web technologies have to be added to the existing build system. Cross-platform build is not as easy as it is with an Eclipse application, since many packages have native dependencies and are not prebuilt for every OS. Furthermore, resulting applications are partially browser and OS-dependent. The native dependencies require to provide a server with a corresponding OS to build a product for all OSs.

7.1.2 Implementation

The migration of the KIELER tool results in the Theia application KEITH that can run in the browser or as an Electron app. This new product does use the KIELER plugins as a language server, which allows to compile SCCharts models and other synchronous languages. Moreover, it provides rich editing features for the languages present in the KIELER project. To build a deliverable product, the application is bundled as an Electron app for all major OSs. Moreover, the application can run in a browser and has the potential to run in a container, which can be used for teaching, as elaborated in Section 6.4.

Parallel to KEITH, KIELER can be further developed without interfering with the development of KEITH, since the backend is reused.

KEITH has advantages by being able to send asynchronous request to the language server and by being configurable via different production setups. The Theia UI introduces promising new concepts that reduce the noise of the UI, but change the interaction with the IDE. Separation of concerns is enforced by KEITH trough the use of the LSP. The maintainability might be boosted by the concept. Future migration are easier, since the backend functionality can be provided as a language server, which can be adopted by any IDE that supports the LSP. Only the frontend implementation has to be replaced.

7.2 Future Work

Since this migration project took place in the scope of a master's thesis, not everything could be done, implemented, and therefore evaluated. Future work focuses on adding new features to KEITH, optimizing the build infrastructure, evaluating problems, starting new projects using the underlying technology, analyzing KEITH in the scope of UI concepts and HCI, and starting new collaborations with TypeFox to further develop Theia.

7.2.1 Restructuring of KEITH and Further Development

Currently the developed Theia extension does reside in a directory in the KIELER repository and is integrated into KEITH by a static reference to the directory in the `package.json`. Other Theia extensions are published via npm and can be added through the `package.json` into the project and can also be added at runtime through the extension manager. Publishing via npm is one of the steps to achieve more automation in the product build. It has to be evaluated whether a locally running npm registry is required to bundle the extension only in KEITH or whether the extension should be released as a standalone package.

Currently the diagram widget and diagram synthesis implemented by Rentz [Ren18] is not fully integrated in the project and diagrams are only shown in a mock view using an SVG synthesized by the offscreen renderer of KLighD. To use the diagram view, Rentz has to develop an interface to use it. This interface has to be called in the show snapshot use case instead of the mock diagram view. Integrating the diagram extension does not only involve its functionality, but also the project setup and the bundling into different packages and publishing via npm. Implementing that, allows to install this extensions in every Theia application via the extension manager of Theia. Furthermore, adding the diagram extension does influence the language server extensions, which have to be changed accordingly.

Since not all features of KIELER were migrated to KEITH, the missing ones have to be added. Simulation of SCCharts currently does not work and has to be added in form of a simulation widget and a view for the simulation data. Furthermore, many developer features such as tracing or the developer use cases seen in Figure 5.2 have to be added. To achieve this, more functionality of the KIELER tool has to be made UI-independent and restructured to be used by the KEITH language server. This is a non-trivial task, as mentioned in Section 6.1.1. It may lead to problems, since the UI and overall UI-concept of Eclipse and Theia are different and sometimes use opposing concepts, as mentioned in Section 4.2. It has to be evaluated how and in what scale this affects the parallel development of this two tools.

The `CompilerWidget` does currently not store its state. However, this may be relevant to store

compiler preferences. Therefore, this widget can implement the `StatefulWidget` interface to achieve this.

Currently the UI of KEITH is not browser- and OS-independent, as mentioned in Section 6.1.3. Selectboxes and menubars look slightly different. Button labels and icons are not properly aligned. A UI solution that prevents browser dependency or masks these problems has to be developed. Moreover, the application itself behaves differently on different OSs. This promises to be a problem in the future. It has to be evaluated how it effects the maintainability of the project. The overall OS- and browser-independence have to be tested and evaluated to find problems and report them to the Theia developers or fix them ourselves.

The performance evaluation in Section 6.3 has shown that the filtering of the compilation systems takes quite some time when working with big models. The reused implementation of the KIELER project parses the whole file to recognize the supported compilation systems. An option to solve this particular problem is to do all filtering on the client side and only request the compilation systems on connection to the language server. Therefore, it has to be evaluated, in what cases it is better to duplicate code to achieve reactivity for KEITH or whether there are options to prevent this problem in Theia.

7.2.2 Build Setup and Automation

The integration in the Bamboo continuous integration system have to be completed and all workflows for e.g. highlighting generation have to be automated if not already done. This does include the error messages and recognition by Bamboo, which does not seem to work in all cases for TypeScript errors and yarn. Having continuous integration in mind, the TypeScript implementation needs test cases to find problems with the UI or functionality automatically.

The language registration in language server and Theia extension is done manually by moving the generated highlighting configuration, mentioned in Section 5.4.2, in the Theia extension project and registering the supported languages in the language server and the Theia extension. As part of the overall automation process, Xtext can be leveraged to automatically generate language registration and syntax highlighting for both components. Automation promises to avoid errors when changing the supported languages of an IDE and makes the project easier to maintain. This is one of the first steps in future work on this project for more automation.

7.2.3 Future Tools for Theia and KEITH

Microsofts DebugProtocol is used by VSCode to implement a debugging server. This technology is also interesting for Theia and KEITH to implement a debugging server for SCCharts. In future work on KEITH, the use of a debugging server via a debug adapter can be evaluated. A debug adapter is also interesting for the whole Theia project and can be reused for different Theia implementations.

Since the language server already exists, it can be used to develop different IDE extensions, as seen in the example of YANG [Köh17c]. These can be used to develop in SCCharts or other languages in different IDEs and to raise the overall awareness of this language and its features. Diagram generation requires an extensive frontend implementation. Whether the diagram generation should be integrated into extensions for different IDEs has to be evaluated.

7.2.4 Usage for Teaching

In the future, the browser mode of KEITH should be usable for conferences, teaching, or demos for big audiences, as mentioned in Section 6.4. It can be used to provide an IDE without setup time

or configuration. To achieve this, the problem of workspace management has to be solved. Docker can be used to start various language servers or Theia servers with a predefined workspace inside an encapsulated environment. These containers have to be somehow mapped to users. Since Theia itself does currently not support a solution for this, some kind of user workspace mapping has to be implemented. Gitpod[1] can be used instead to achieve an easy to set up online IDE. Gitpod is a tool developed by TypeFox to automatically set up workspaces in an online IDE for specific branches and commits in GitHub projects. The tool itself is currently in beta, but in a meeting with TypeFox it was suggested to use it in an educational context to test Gitpod with several users at the same time. Using Gitpod requires to evaluate the resulting costs of this tool.

During the migration project collaboration with TypeFox, the developers of Theia, took place. Theia is still in development. Further collaboration with TypeFox can point out problems and missing functionality of the open-source project. Moreover, the Gitpod tool can be tested together with TypeFox and its use for teaching or conferences can be evaluated.

7.2.5 Build Language Server as fat jar

One of the ideas behind Theia is to have a lightweight easy to set up editor that is highly configurable. Currently, the whole Eclipse UI is delivered in the language server, since it is bundled as an Eclipse application. Since the language server for KEITH can support diagram generation, compilation, and is a full grown Eclipse application, it is quite big. It is not necessary to build the application as an Eclipse product, as elaborated in Section 6.1.1. The language server can also be delivered as fat `jar` if the extension points are abolished.

7.2.6 Publishing of Xtext Fragment

The `MonacoHighlightingFragment` is interesting for developing Theia extensions for DSLs written in Xtext and can be used to generate syntax highlighting automatically, as elaborated in Section 5.4.2. After this fragment is cleaned up and is generalized, it can be made open-source and be committed to the Eclipse project to be used and maintained for different projects.

7.2.7 Research regarding Usability of KEITH

KEITH was not formally evaluated in this thesis. The different setup methods of KEITH use new development and UI concepts. These concepts influence how the user develops SCCharts. HCI can be used to evaluate further, how KEITH influences development and how users adopt these new UI concepts. The different setup methods can be evaluated at the same time. An academic lecture that previously used the KIELER tool can be used to evaluate these influences.

[1] `https://www.gitpod.io/`

Bibliography

[AB15] Amjad Altadmri and Neil C.C. Brown. "37 million compilations: investigating novice programming mistakes in large-scale student data". In: *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. SIGCSE '15. Kansas City, Missouri, USA: ACM, 2015, pp. 522–527. ISBN: 978-1-4503-2966-8. DOI: 10.1145/2676723.2677258. URL: http://doi.acm.org/10.1145/2676723.2677258.

[BC06] Carliss Y. Baldwin and Kim B. Clark. "The architecture of participation: does code architecture mitigate free riding in the open source development model?" In: *Management Science* 52.7 (2006), pp. 1116–1127. DOI: 10.1287/mnsc.1060.0546. eprint: https://doi.org/10.1287/mnsc.1060.0546. URL: https://doi.org/10.1287/mnsc.1060.0546.

[Ben17] Florent Benoit. *Checkpoint / restore eclipse che in seconds*. 2017. URL: https://che.eclipse.org/restore-eclipse-che-in-seconds-1523434217ab (visited on 08/31/2018).

[BGM17] Dirk Bäumer, Erich Gamma, and Sean McBreen. *Language server protocol*. 2017. URL: http://www.eclipse.org/community/eclipse_newsletter/2017/may/article1.php (visited on 05/24/2018).

[BHJ16] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables devops: migration to a cloud-native architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[Bin94] Robert V. Binder. "Design for testability in object-oriented systems". In: *Commun. ACM* 37.9 (Sept. 1994), pp. 87–101. ISSN: 0001-0782. DOI: 10.1145/182987.184077. URL: http://doi.acm.org/10.1145/182987.184077.

[DMC+10] Arie van Deursen, Ali Mesbah, Bas Cornelissen, Andy Zaidman, Martin Pinzger, and Anja Guzzi. "Adinda: a knowledgeable, browser-based ide". In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ICSE '10. Cape Town, South Africa: ACM, 2010, pp. 203–206. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810330. URL: http://doi.acm.org/10.1145/1810295.1810330.

[Eff17] Sven Efftinge. *Theia – vscode in the cloud*. 2017. URL: https://typefox.io/theia-vs-code-in-the-cloud (visited on 05/24/2018).

[Erc16] G. Ercan. *A common interface for building developer tools*. 2016. URL: https://developers.redhat.com/blog/2016/06/27/a-common-interface-for-building-developer-tools/ (visited on 07/16/2018).

[FBB+07] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jézéquel. "Model-driven engineering for software migration in a large industrial context". In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2007, pp. 482–497.

[FH09a] Hauke Fuhrmann and Reinhard von Hanxleden. *On the pragmatics of model-based design*. Technical Report 0913. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2009.

[FH09b] Hauke Fuhrmann and Reinhard von Hanxleden. *The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage*. http://www.informatik.uni-kiel.de/rtsys/kieler/. 2009.

[Han18] Reinhard von Hanxleden. "Automated graph drawing". University Lecture. 2018.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: https://doi.org/10.1016/0167-6423(87)90035-9. URL: http://www.sciencedirect.com/science/article/pii/0167642387900359.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. "SCCharts: Sequentially Constructive Statecharts for safety-critical applications". In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*. Long version: Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, December 2013, ISSN 2192-6274. Edinburgh, UK: ACM, 2014.

[HFS11] Reinhard von Hanxleden, Hauke Fuhrmann, and Miro Spönemann. "KIELER—The KIEL Integrated Environment for Layout Eclipse Rich Client". In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '11)*. Grenoble, France, 2011.

[Hol02] Gerard J Holzmann. "Static source code checking for user-defined properties". In: *Proc. IDPT*. Vol. 2. 2002.

[HPH14] J. Hausladen, B. Pohn, and M. Horauer. "A cloud-based integrated development environment for embedded systems". In: *2014 IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA)*. Sept. 2014, pp. 1–5. DOI: 10.1109/MESA.2014.6935577.

[HRJ+04] Wilhelm Hasselbring, Ralf Reussner, Holger Jaekel, Jürgen Schlegelmilch, Thorsten Teschke, and Stefan Krieghoff. "The dublo architecture pattern for smooth migration of business information systems: an experience report". In: *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*. IEEE. 2004, pp. 117–126.

[JLS+87] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. "Monitoring distributed systems". In: *ACM Trans. Comput. Syst.* 5.2 (Mar. 1987), pp. 121–150. ISSN: 0734-2071. DOI: 10.1145/13677.22723. URL: http://doi.acm.org/10.1145/13677.22723.

[Joh78] S. C. Johnson. "Lint, a c program checker". In: *COMP. SCI. TECH. REP*. 1978, pp. 78–1273.

[Köh16] Jan Köhnlein. *Xtext's new generator: migration*. 2016. URL: https://typefox.io/xtexts-new-generator-migration (visited on 09/24/2018).

[Köh17a] Jan Köhnlein. *Extending a language server with sprotty diagrams*. 2017. URL: https://typefox.io/extending-a-language-server-with-sprotty-diagrams (visited on 08/23/2018).

[Köh17b] Jan Köhnlein. *Sprotty – a web-based diagramming framework*. 2017. URL: https://typefox.io/sprotty-a-web-based-diagramming-framework (visited on 08/23/2018).

[Köh17c] Jan Köhnlein. *Yang-tools: one language server for four ides*. 2017. URL: https://typefox.io/yang-tools-one-language-server-for-four-ides (visited on 05/24/2018).

[KPE16] Sven Keidel, Wulf Pfeiffer, and Sebastian Erdweg. "The ide portability problem and its solution in monto". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2016. Amsterdam, Netherlands: ACM, 2016, pp. 152–162. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997368. URL: http://doi.acm.org/10.1145/2997364.2997368.

[Lei80] Dennis W. Leinbaugh. "Indenting for the compiler". In: *SIGPLAN Not.* 15.5 (May 1980), pp. 41–48. ISSN: 0362-1340. DOI: 10.1145/947639.947644. URL: http://doi.acm.org/10.1145/947639.947644.

[LNK+12] Janne Lautamäki, Antti Nieminen, Johannes Koskinen, Timo Aho, Tommi Mikkonen, and Marc Englund. "Cored: browser-based collaborative real-time editor for java web applications". In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. CSCW '12. Seattle, Washington, USA: ACM, 2012, pp. 1307–1316. ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145399. URL: http://doi.acm.org/10.1145/2145204.2145399.

[Lor18] Mario Loriedo. *The new superpowers of che workspaces*. 2018. URL: https://che.eclipse.org/the-new-superpowers-of-che-workspaces-243967a2010 (visited on 08/31/2018).

[M17] Tracy M. *Debug protocol vs language server protocol*. 2017. URL: https://kichwacoders.com/2017/11/08/debug-protocol-vs-language-server-protocol/ (visited on 07/12/2018).

[MFH09] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. *Semantics and execution of domain specific models*. Technical Report 0923. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, 2009.

[Mic17] Brad Micklea. *The evolving cloud development market*. 2017. URL: https://che.eclipse.org/the-evolving-cloud-development-market-2657aaf83e6c (visited on 08/31/2018).

[Mic18] Microsoft. *Typescript design goals*. 2018. URL: https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals (visited on 08/24/2018).

[MMN+83] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. "Program indentation and comprehensibility". In: *Commun. ACM* 26.11 (Nov. 1983), pp. 861–867. ISSN: 0001-0782. DOI: 10.1145/182.358437. URL: http://doi.acm.org/10.1145/182.358437.

[Moh17] Dominik Mohilo. *What theia is all about — a classic ide built with modern technology*. 2017. URL: https://jaxenter.com/theia-ide-efftinge-interview-134467.html (visited on 05/24/2018).

[MRW14] A. B. Mutiara, R. Refianti, and B. A. Witono. "Developing a saas-cloud integrated development environment (IDE) for c, c++, and java". In: *CoRR* abs/1411.5161 (2014). arXiv: 1411.5161. URL: http://arxiv.org/abs/1411.5161.

[MT07] Tommi Mikkonen and Antero Taivalsaari. *Using javascript as a real programming language*. Tech. rep. Mountain View, CA, USA, 2007.

[MTA+17] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. "A concurrency-agnostic protocol for multi-paradigm concurrent debugging tools". In: *SIGPLAN Not.* 52.11 (Oct. 2017), pp. 3–14. ISSN: 0362-1340. DOI: 10.1145/3170472.3133842. URL: http://doi.acm.org/10.1145/3170472.3133842.

[Ram86] Gerard K. Rambally. "The influence of color on program readability and comprehensibility". In: *SIGCSE Bull.* 18.1 (Feb. 1986), pp. 173–181. ISSN: 0097-8418. DOI: 10.1145/953055.5702. URL: http://doi.acm.org/10.1145/953055.5702.

[Ren18] Niklas Rentz. "Moving transient views from eclipse to web technologies". unpublished master thesis. 2018.

[RPF+14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. "A large scale study of programming languages and code quality in github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 155–165.

[RT05] Peter C. Rigby and Suzanne Thompson. "Study of novice programmers using eclipse and gild". In: *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. eclipse '05. San Diego, California: ACM, 2005, pp. 105–109. ISBN: 1-59593-342-5. DOI: 10.1145/1117696.1117718. URL: http://doi.acm.org/10.1145/1117696.1117718.

[SDM+03] Margaret-Anne Storey, Daniela Damian, Jeff Michaud, Del Myers, Marcellus Mindel, Daniel German, Mary Sanseverino, and Elizabeth Hargreaves. "Improving the usability of eclipse for novice programmers". In: *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*. eclipse '03. Anaheim, California: ACM, 2003, pp. 35–39. DOI: 10.1145/965660.965668. URL: http://doi.acm.org/10.1145/965660.965668.

[Sin17] Sarika Sinha. *Debugging the eclipse ide for java developers*. 2017. URL: https://www.eclipse.org/community/eclipse_newsletter/2017/june/article1.php (visited on 10/04/2018).

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. "Just model! – putting automatic synthesis of node-link-diagrams into practice". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '13)*. San Jose, CA, USA, 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSH18] Steven Smyth, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Towards interactive compilation models". In: *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*. accepted. 2018.

[Tea16] Microsoft Team. *'this' in typescript*. 2016. URL: https://github.com/Microsoft/TypeScript/wiki/%27this%27-in-TypeScript (visited on 09/18/2018).

[Tea18a] Sourcegraph Team. *A community-driven source of knowledge for language server protocol implementations*. 2018. URL: https://langserver.org/#implementations-server (visited on 05/24/2018).

[Tea18b] Visual Studio Code Team. *User interface*. 2018. URL: https://code.visualstudio.com/docs/getstarted/userinterface (visited on 09/04/2018).

[Tep09] Werner Teppe. "The arno project: challenges and experiences in a large-scale industrial software migration project". In: *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE. 2009, pp. 149–158.

[TT17] Sven Efftinge (TypeFox) and Anton Kosyakov (TypeFox). *Running theia in eclipse che*. 2017. URL: https://www.eclipsecon.org/europe2017/session/running-theia-eclipse-che (visited on 08/31/2018).

[WAB+14] Greg Wilson et al. "Best practices for scientific computing". In: *PLOS Biology* 12.1 (Jan. 2014), pp. 1–7. DOI: 10.1371/journal.pbio.1001745. URL: https://doi.org/10.1371/journal.pbio.1001745.

[WLK+11] L. Wu, G. Liang, S. Kui, and Q. Wang. "Ceclipse: an online ide for programing in the cloud". In: *2011 IEEE World Congress on Services*. July 2011, pp. 45–52. DOI: 10.1109/SERVICES.2011.74.

[ZKC+07] Hong Zhou, Jian Kang, Feng Chen, and Hongji Yang. "Optima: an ontology-based platform-specific software migration approach". In: *Quality Software, 2007. QSIC'07. Seventh International Conference on*. IEEE. 2007, pp. 143–152.

Abbreviations

API Application Programming Interface

ARNO Application Relocation to New Operating System

COBOL common business-oriented language

CSS Cascading Style Sheets

DOM Document Object Model

DSL Domain Specific Language

EMF Eclipse Modeling Framework

HCI Human Computer Interaction

HTML Hypertext Markup Language

IDE Integrated Development Environment

IP Internet Protocol

IT Information Technology

iur initialize-update-read

J2EE Java 2 Platform, Enterprise Edition

JDK Java Development Kit

JDT Eclipse Java development tools

JSON JavaScript Object Notation

JSON-RPC JSON Remote Procedure Call

JVM Java Virtual Machine

KEITH KIEL Environment Integrated in Theia

KiCo KIELER Compiler

KIEL Kiel Integrated Environment for Layout

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client

KLighD KIELER Lightweight Diagrams

LSP Language Server Protocol

OS Operating System

7. Abbreviations

OSGi Open Service Gateway Initiative

PDE Plug-in Development Environment

SCCharts Sequential Constructive Statecharts

SCL Sequentially Constructive Language

SVG Scalable Vector Graphic

SWT Standard Widget Toolkit

TCP Transmission Control Protocol

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

YANG Yet Another Next Generation