Standalone Web Diagrams and Lightweight Plugins for Web-IDEs such as Visual Studio Code and Theia

Christoph Fricke

Bachelor's thesis
September 28, 2021

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

Advised by
M.Sc. Niklas Rentz

Selbstständigkeitserklärung

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

Kiel, 28.09.2021

C. Fricke

Abstract

Web technologies have become more capable and are more widely used than ever before. The continuous improvements to available web APIs and the addition of new APIs have enabled more complex, web-based applications. One of the most popular code editors, Visual Studio Code, is built with web technologies, and every day new applications are developed with a web-first approach.

Previous work by Domrös and Rentz has created the KEITH project, which integrates tooling for the KIELER project in the web-based IDE platform Theia. This work ports the created diagram view for KLighD-synthesized models to Visual Studio Code and a standalone browser view to enable new use cases.

A solution for such implementation can be approached from multiple angles. Different approaches are discussed and ultimately one approach is picked and implemented for this thesis, based on requirements that should be fulfilled by this approach. The new diagram view avoids code duplication between Visual Studio Code and the standalone view and is extensible for future work.

To achieve this, abstractions have been identified that not only simplify the development of features but also make the new diagram view accessible beyond the scope of the KIELER project. The newly released applications that are the result of this work make the diagram view for KLighD models accessible to many more potential users.

Acknowledgements

I would like to thank Prof. Dr. Reinhard von Hanxleden for the opportunity to write my bachelor's thesis in his working group. Many thanks go to Niklas Rentz who advised my thesis and has provided continuous support, ideas, and feedback. Furthermore, I would like to thank Alexander Schulz-Rosengarten for his feedback regarding the approach for a Visual Studio Code extension.

I would also like to thank Lena Fricke and Claus-Dieter Piontke for proofreading and reviewing this thesis. Lastly, I am very grateful for my family who continuously supported and motivated me while writing my thesis.

Contents

List of Figures

Listings

Chapter 1

Introduction

Today's software solutions often rapidly grow in size and complexity. This decreases the ability to fully understand the system. At the same time changing requirements have to be met, bugs have to be fixed, and the whole system has to be maintained and updated with technological advances and new features.

To tame the complexity, system architects and developers rely on visualizations to obtain an overview of the system and to communicate the system architecture to new team members. Further, visualizations are used to communicate new requirements and to detect potential problems during the design phase.

These visualizations have to be maintained manually more often than not [SSH13]. Without a strong commitment to keeping the documentation and visualizations up to date, both become outdated over time and no longer apply to the current software system. Generating visualizations directly from the source helps to avoid outdated visualizations, but they might not be as meaningful as handcrafted visualizations, since general purpose languages are not able to semantically express the intent behind the code. Domain specific languages not only enable source code that is focused on the problem domain, but enable the generation of more meaningful visualizations as well.

The Real-Time and Embedded Systems Group of the Department of Computer Science at Kiel University develops and maintains the research project Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER). The project aims to integrate different modeling languages, such as their synchronous language Sequentially Constructive Charts (SCCharts) [HDM+14], into the Eclipse platform. Supported languages are enhanced with interactive diagrams, whose view models are synthesized using the KIELER Lightweight Diagrams (KLighD) framework [SSH13].

Over the years, web technologies have advanced and have become more capable than ever. The development of Node.js, a JavaScript runtime, and frameworks, such as Electron[1], has allowed web technologies to spread into many new application areas. Most notably, newer and more lightweight Integrated Development Environments (IDEs), such as Atom or Visual Studio Code (VS Code), are built with these technologies and have quickly gained popularity. The 2021 Stack Overflow survey asked 82 thousand developers about their IDE of choice. Around 71% answered that they use VS Code, making it the most popular IDE according to the survey [Ove21].

Together with the release of Microsoft's VS Code, the company popularized the idea

[1] A framework for cross-platform desktop applications built with web technologies.

of separating programming language-specific functionality from IDEs. A language server provides functionality, such as hover information and suggestions, for a language behind a clearly defined communication protocol, called the Language Server Protocol (LSP)[2]. IDEs can integrate such language servers, which allows them to support more languages with less effort.

The working group has seen the development of the LSP and the advancement of web technologies as an opportunity to experiment with new technologies and discover new research topics that arise with web technologies [Ren18]. Further, it opens their project to more tools and thus makes it accessible for more developers. Past work by Domrös and Rentz has created a language server and integrated parts of their tooling into the open-source IDE platform Theia [Dom18]. Based on the new custom IDE, they created the Kiel Environment Integrated in Theia (KEITH) project, which contains a web-based diagram visualization using a library called Sprotty [Ren18]. The diagram synthesis with KLighD is implemented behind the LSP in the language server, while Sprotty is used to transform the resulting model to Scalable Vector Graphics (SVG) and display it in Theia.

In version 1.4 Theia deprecated its support for the LSP to reduce maintenance cost [Köh20]. Instead, developers should wrap their language server in a VS Code extension, which are supported by Theia since its 1.0 release [Eff20b].

This change in Theia's development forces the working group to port their tooling developed for Theia to a VS Code extension to support newer versions of Theia. Obviously, developing a VS Code extension will add support for VS Code to their project. The development of a VS Code extension is further motivated by requests of industry partners to support the tools they commonly use, such as VS Code.

However, the need for visualization motivated in the beginning is not fully covered by tooling that is integrated into IDEs. Even though VS Code and Theia start fairly quick compared to Eclipse, if a developer wants to quickly open an interactive visualization, a standalone diagram view is often preferable. Furthermore, interactive visualizations can improve the communication inside an organization. The ability to share a link to an interactive diagram view that runs in a browser allows non-technical and technical people to benefit from the visualization without installing specific applications, such as an IDE. This can be coupled with the possibility to enhance other digital media with interactive visualizations. Many web-based wiki solutions, such as Confluence, are able to embed external websites. Embedding a diagram view in such wiki solutions allows a reader to interactively explore a visualization, which can help to further comprehend documentation [RDH20].

1.1 Problem Statement

The main focus of this thesis is to explore and implement a solution for a diagram view in a standalone website, Theia, and VS Code. The diagram view must support diagram models that are synthesized with KLighD, so it is usable by the KIELER project. Instead of implementing the

[2] https://microsoft.github.io/language-server-protocol

diagram view from scratch, the current solution in KEITH should be refactored to be reusable as a foundation. In the end, all diagram-related features that are available in KEITH should be available in VS Code and a standalone website as well.

Throughout this thesis, the term *platforms* will be used to refer to VS Code, Theia, and a browser as a whole. Common occurrences include *target platforms* or *new platforms* when Theia is not considered since it is an already supported platform. Even though the term *platform* is often used to refer to operating systems, it is also used to refer to technologies upon which other applications are developed[3]. In this case, VS Code, Theia, and a browser are applications that serve as runtime platforms, which will be used to provide diagram visualizations to users of the KIELER project.

Further requirements can be imposed to refine the problem statement and to determine the quality of a solution. It should focus on avoiding code duplication between the different target platforms. If possible, a core implementation that contains most features should be reused for each platform. KIELER is not the only project that contains languages which can be visualized with the use of KLighD. Further, the new platforms are not exhaustive and do not cover all possible use cases. Therefore, a solution should be extensible with new features and open for easy adoption by other projects.

1.2 Outline

Chapter 2 introduces JavaScript libraries that have been used extensively in the implementation for this thesis, as well as foundational software, such as KLighD, VS Code, and Node.js. It is followed by related work that inspired the solution for this thesis in Chapter 3. Chapter 4 discusses and outlines concepts and approaches for the different platforms, which are composed to an overall architecture and approach. Some core components have been created that enable and simplify the implementation of diagram-related features, which are introduced in Chapter 5, as well as an outline of the implementation for each platform. The new implementation is compared with KEITH in Chapter 6. Furthermore, it reasons about the diagram view performance and outlines the effort required to adopt it for other projects. Chapter 7 summarizes this thesis and motivates future work.

[3] https://www.techopedia.com/definition/3411/platform-computing

Chapter 2

Used Technologies

Developing a diagram visualization for three different platforms requires knowledge about many libraries and applications. Each platform provides its own guidelines and Application Programming Interfaces (APIs) that have to be understood to successfully target the platform. Furthermore, different libraries are used to support the development of a diagram visualization. Additional libraries are used to increase the maintainability of the codebase, and to create a build setup to bundle and distribute the code for each platform.

This chapter introduces various libraries and applications that are used throughout the thesis. The list is not exhaustive but includes all relevant technologies that are used extensively or fulfill an important role.

2.1 KIELER Lightweight Diagrams

KIELER Lightweight Diagrams (KLighD) is a framework introduced by Schneider et al. [SSH13] that can be used to create diagram visualizations with automatic graph layout and interactivity. The framework is independent of any rendering library and thus can be integrated by multiple applications. KLighD synthesizes a diagram model which describes how the visualization is supposed to look and is mapped into a final visualization by a rendering library. Furthermore, the framework uses an input language called *KRendering* [SSH13]. KLighD allows adding support for any model by registering a model transformation, which describes how an input maps to *KRendering*. A transformation is referred to as a *synthesis*. The resulting model contains layout information, which however is not final. Instead, it is further used by KLighD to calculate the final position for each element in the model.

KLighD supports interactive features for the diagram models, such as semantic zooming, which includes expanding and collapsing regions in a model to show or hide more elements [SSH13]. Furthermore, KLighD is able to evaluate options that are provided by the synthesis and change the resulting diagram model [Ren18]. When a user triggers an interactive feature or changes an option, the framework has to rerun the layout or even re-execute the synthesis.

KIELER uses KLighD as its main framework for automatic diagram views, which has been added to KEITH as well [Ren18]. In KEITH, the diagram generation is separated between a client and server, with the model synthesis being implemented on the server. This thesis reuses the existing server and integrates the web-based rendering on the client into the new target platforms.

2.2 Node.js, npm and yarn

Traditionally, JavaScript has been developed for usage in a browser. It was intended to be a scripting language to add interactivity to otherwise static web pages. Today, JavaScript is one of the most used and demanded languages with one of the largest developer ecosystems, partly due to a project by Ryan Dahl.

In 2009, Ryan Dahl released the JavaScript runtime Node.js[1], which enables the execution of JavaScript outside the browser. Node.js is built on the open-source JavaScript engine V8 developed by Google, which is used to compile and execute JavaScript in Google Chrome and other Chromium-based browsers. The asynchronous event-driven runtime makes Node.js a good candidate for Input/Output (I/O)-heavy applications, such as web servers [Fou21].

Nowadays, developing software for applications that run in either Node.js or a browser heavily relies on open-source libraries. Most of the time, the npm[2] registry is used to distribute libraries as so-called *packages*, and currently hosts more than 1.7 million packages. To manage dependencies to other packages, Node.js-based software projects use npm's provided Command Line Interface (CLI) or the package manager yarn[3], which is a popular alternative that also utilizes the npm registry. The tooling required to build and maintain web-based software is often developed with JavaScript and distributed with npm, thus using the same technologies as the rest of the codebase. This eases the development and usage of new tools for the JavaScript community, because more developers are able to contribute.

Together with Node.js, npm forms the foundation of any modern, web-based software by providing dependency management, tooling setup, and script execution capabilities. This is the case for this thesis as well, which uses Node.js and yarn. To bundle the code for distribution and to ensure compliance with JavaScript coding standards, an extensive tooling setup has been established that relies on multiple npm packages, such as Webpack, ESLint, and pkg.

2.3 TypeScript

Over time web-based applications have grown in size and complexity as the web ecosystem continuous to grow with more capabilities. Developing these big applications often involves multiple software developers. Even if an application is developed by a single person, it gets harder over time to remember every part of the codebase. Using JavaScript, a dynamically typed language, does not help to maintain the codebase either. Common questions arise during development that are hard to answer without reading the implementation. What was the shape of the object that the function expects? What was the signature of the required callback? These questions are hard to answer with a dynamically typed language, which

[1] https://nodejs.org

[2] https://npmjs.com

[3] https://classic.yarnpkg.com

often causes runtime errors that could have been fixed during development with a statically typed language [Mic21e].

To improve development with JavaScript, Microsoft has developed TypeScript[4] which is a superset of JavaScript. This means that any JavaScript code is also valid TypeScript code, but can be enhanced with type definitions [Mic21e]. The addition of interfaces, generics, and access modifiers makes TypeScript suitable for object-oriented programming. To be able to properly type existing and often flexible JavaScript code, TypeScript requires a powerful type system. TypeScript's type system is based on structural subtyping, where two types are compatible if their structure matches. This contrasts with nominal typing used by most object-oriented languages [Mic21e]. Additionally, TypeScript has features, such as type inference, conditional types, union types, type narrowing, and discriminated unions. They allow TypeScript to fully type libraries that have a very flexible API.

This work has used TypeScript extensively. In fact, all code developed for this thesis is written in TypeScript. All libraries that are used include TypeScript definitions as well, which makes it easier to work with them. To execute TypeScript code, a build system has been established to transpile TypeScript back to JavaScript.

2.4 Visual Studio Code

Visual Studio Code (VS Code) is a modern code editor developed by Microsoft that is based on web technologies. While the product is published for free under the Microsoft product license[5], it is developed together with the community as an open-source editor under the MIT license[6]. VS Code is written in TypeScript and is powered by the Monaco editor[7], which is developed by Microsoft as well. The Monaco editor was first developed as part of VS Code to provide the core of its rich editing experience. Since 2016, Monaco has been extracted to empower anyone to develop rich code editing experiences for the browser [Mic16a]. Furthermore, VS Code utilizes the Electron[8] framework to distribute the web-based product as a cross-platform desktop application. The editor provides a powerful, well documented extension API[9] to integrate additional features, theming, and support for many languages and developer tools.

Since its version 1.0 release in April 2016, VS Code has grown a large community of developers that use the editor daily and actively extend its capabilities with new extensions. At the time, the editor already achieved over two million installs since the public preview release a year before [Mic16b]. In the year 2021, the editor has been used by around 71% of the 82 thousand developers that were asked about their regularly used editors in the yearly Stack Overflow survey [Ove21]. This makes it one of the most used editors today. Together

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

[5] https://code.visualstudio.com/License

[6] https://github.com/Microsoft/vscode

[7] https://microsoft.github.io/monaco-editor

[8] https://www.electronjs.org

[9] https://code.visualstudio.com/api/references/vscode-api

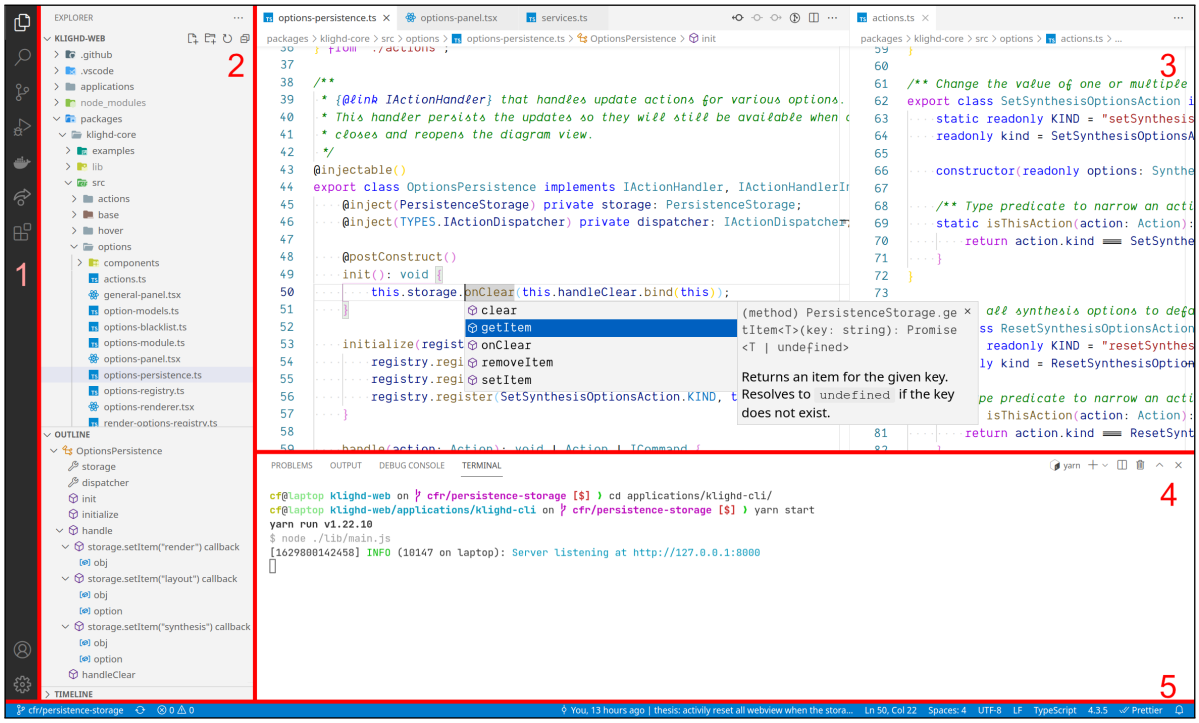


Figure 2.1. Overview of major VS Code UI components, which have been highlighted with red lines. The components are 1: an activity bar, 2: a sidebar, 3: a main area, 4: a panel, 5: a status bar.

with such popularity, the amount of available extensions has also grown rapidly to support a wide range of usage, which is why many consider VS Code an IDE.

Compared to other IDEs, such as Eclipse or Jetbrain's IDEs, VS Code takes a different approach to its User Interface (UI). Instead of many panels and a toolbar that is cramped with various buttons and context menus, VS Code has a more minimal UI. Most interactions that do not involve code editing are defined as commands that can be invoked from a command palette. Further, commands can also be invoked by buttons that are placed in small groups throughout the UI. VS Code tries to keep the number of buttons to a minimum, and locates them near the context they will actually affect [Mic21b]. For example, commands for running and debugging a program are located in a *Run and Debug* view.

Figure 2.1 provides an overview of the major components that define the UI of VS Code. These components form the shell and the different areas of the editor. Other items are placed inside these components. These items include toolbars and views for different content, such as a file explorer, an integrated terminal, and an outline for the opened file [Mic21b]. Following the numeration in Figure 2.1, the container components are:

1. The *activity bar* which is probably one of the most characteristic features of VS Code. Multiple views in VS Code are grouped into view containers. Each container is represented by an icon in the activity bar. For example, the *Extensions* view container uses four boxes as an

icon in the activity bar and contains views that show the installed, recommended, enabled, and disabled extensions.

2. The *sidebar* which is a resizable container that can be collapsed or expanded. It displays the currently active view container that has been selected in the activity bar.

3. The *main area* which contains the editor where files are opened in editor tabs. The area can be split horizontally and vertically, similar to the behavior in many other IDEs. More advanced content can be opened in a custom editor or a webview, which are displayed in the editor area as well.

4. The resizable *panel* which contains the integrated terminal, as well as a debug console, output, and problems view.

5. The *status bar* which contains common information about the opened file, such as the current line number, indentation scheme, and language. Furthermore, it contains information about the current git repository if the opened folder is part of a git repository.

2.5 Language Server Protocol

Together with the release of VS Code, Microsoft popularized the Language Server Protocol (LSP)[10]. The idea is to separate programming language-specific functionality from IDEs. Instead of implementing features, such as auto complete, go to definitions, and hover information for each language, over and over again, an IDE should only have to implement these features once. With the LSP, any IDE only has to implement support for the protocol and integrate language servers for the languages it wants to support. The functionality for each language is implemented by the corresponding language server and communicated between the IDE and language server with the LSP. Therefore, a language server can be used by many IDEs and simplifies their support for multiple languages. Especially languages with a smaller community are able to support multiple IDEs by building one language server and multiple small plugins for each IDE to integrate their language server.

The communication defined by the LSP uses JSON-RPC[11] messages to be independent of any programming language [Mic21d]. The LSP does not make assumptions about how the messages are transported. While any transport is possible in theory, often Inter Process Communication (IPC) between the IDE and language server is used.

To separate the visualization of a diagram between a client and a server, KEITH extends the LSP to communicate diagram-related messages between the client and server [Ren18]. The server in KEITH is a language server with support for KIELER model languages, which has been extended with KLighD synthesis support. Figure 2.2 visualizes the LSP communication that is required to display a diagram model, which is synthesized with KLighD on the language server.

[10] https://microsoft.github.io/language-server-protocol/overviews/lsp/overview

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

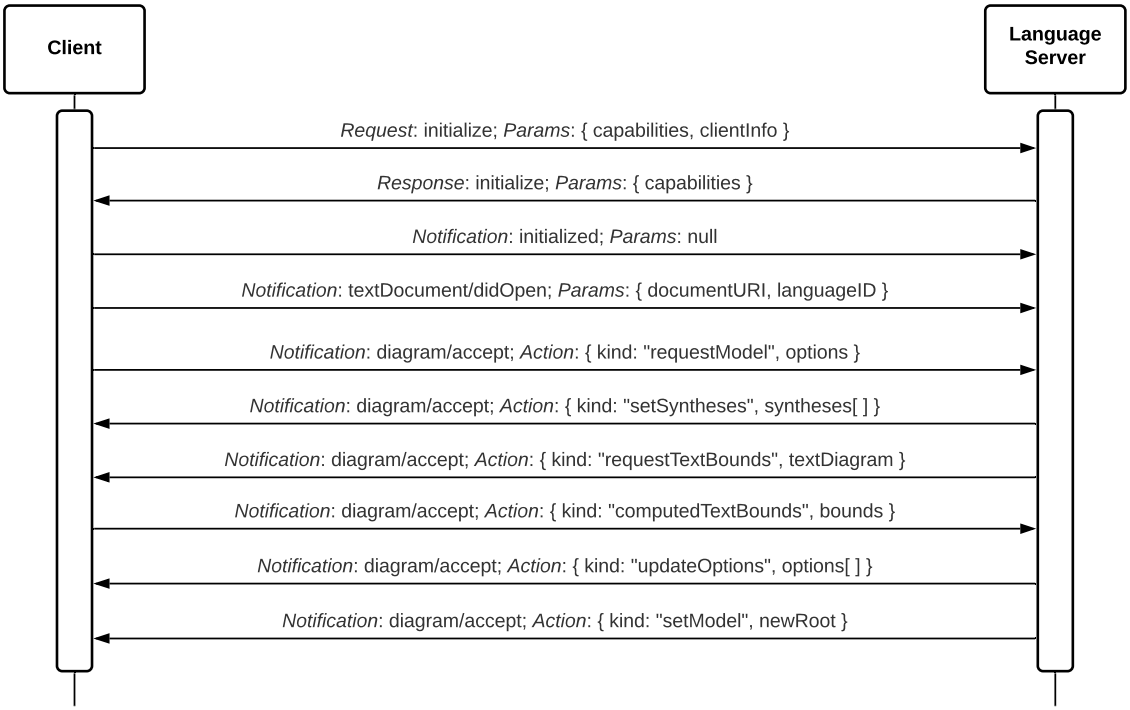


Figure 2.2. LSP communication sequence required to display a diagram.

Before a diagram model can be requested, the connection between the client and server has to be initialized, and the server has to be notified about the document that will be visualized. During the initialization, the client and server exchange their capabilities, which describe the features they are able to support [Mic21d]. Afterwards, the client informs the server about an opened document by sending a *textDocument/didOpen* notification with the document's Unique Resource Identifier (URI) as a parameter. The server is able to access the content of the document through the URI. Alternatively, the content can be sent as a parameter as well.

After this standard LSP communication, the customized diagram communication is able to start. Each message is send as a *diagram/accept* notification with a diagram action as a parameter. Every action has a unique kind and potentially more data. The client starts the message exchange with a *requestModel* action to request a diagram model for a previously opened document. The server responds with a list of KLighD syntheses that are available for the requested model, and requests a text bound calculation from the client [Ren18]. After the client has responded with calculated text bounds, the server sends the synthesized diagram model to the client, as well as a list of available synthesis and layout options that can be changed by a user.

The software developed for this thesis uses the same message sequence between the client and language server for the new platforms. The standalone diagram view also has

to implement the whole initialize and open document communication, which in contrast is already implemented in VS Code by the platform.

2.6 Theia

Theia[12] is an open-source IDE developed by TypeFox. The UI of Theia is very similar to VS Code. In fact, Theia makes use of the Monaco editor as well [Eff17].

Just as VS Code, Theia can be used as a desktop application that is powered by Electron as well. What makes Theia unique to VS Code is the fact that it can also be used in a browser. In this scenario, the IDE is separated between a client and a server, which does not have to be executed on the same computer as the browser that runs the client. Therefore, it is possible to run the IDE in a cloud application scenario [Eff17], which has already been done by TypeFox with Gitpod[13]. Gitpod is an IDE as a service that can be used to access a development environment from anywhere with any device as long, as it is able to run a browser. Theia has been used as the underlying IDE in Gitpod, but they recently added support for VS Code as well [Eff20a].

Generally, this trend towards VS Code can be observed in Theia as well. Not only does Theia look very similar, but it also supports the usage of VS Code extensions [Eff20b]. This integration has matured enough, so Theia deprecated support for features, such as the LSP, in their own extension API to reduce maintenance cost [Köh20].

Theia has been used as the IDE for KEITH. However, with the deprecation of features that are relevant for KEITH, Theia can no longer be updated to the latest version. Therefore, KEITH is stuck with an older version until all features are moved to a VS Code extension, which in return can be used with newer versions of Theia. This thesis has started the transition to a VS Code extension by porting the diagram support that has been previously implemented for Theia.

2.7 Sprotty

Sprotty is a web-based framework for rendering interactive diagrams that is developed by TypeFox [Köh17]. The goal of the framework is to be highly customizable and extensible. To achieve this goal, Sprotty centers its design around the Inversion of Control (IoC) container library InversifyJS[14]. Every part of the framework can be customized and configured by registering custom implementations in the IoC container or by using the provided default implementations.

This approach allows Sprotty to be extensible for different usage scenarios and environments. Sprotty provides integrations for VS Code extensions[15], Theia extensions, and

[12] https://theia-ide.org

[13] https://gitpod.io

[14] https://inversify.io

[15] https://github.com/eclipse/sprotty-vscode

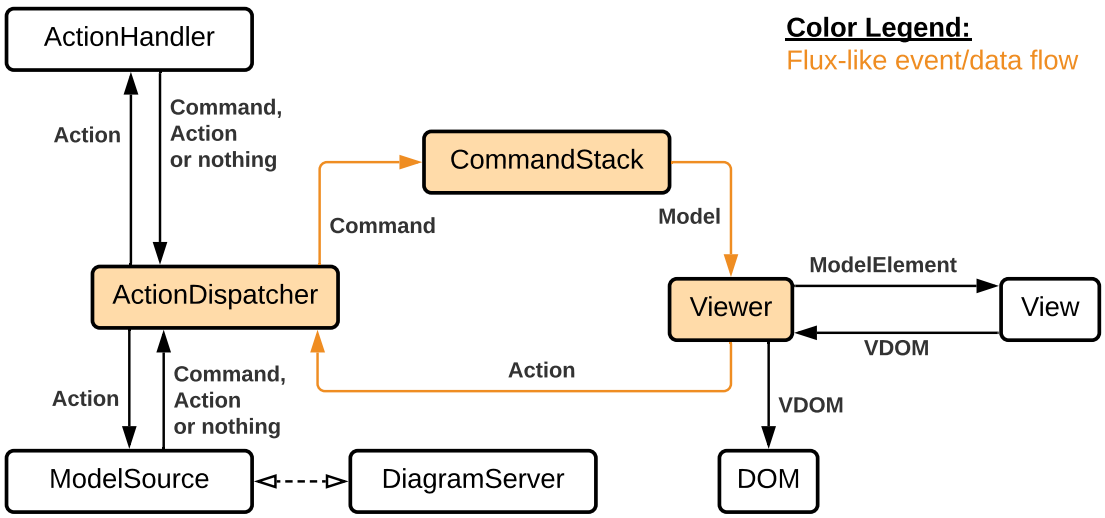


Figure 2.3. Overview of Sprotty's Flux-like architecture inspired by its documentation [Spö19].

standalone views in a browser. Furthermore, the framework can also be separated between a client and a server. In this setup, the client only stores the current diagram model. The server is responsible for generating and changing the model for the client. Model changes and other interactions with the diagram visualization are represented as action objects, which are communicated as JavaScript Object Notation (JSON) between the client and server [Köh17].

The architecture of Sprotty is inspired by Flux and follows a unidirectional event and data flow [Spö19]. The unidirectional flow of a Flux architecture can be summarized in three steps. The view dispatches actions which describe an intent. Actions are handled by a data store that updates the stored state corresponding to the given action. The view is notified about the update and re-renders based on the new state. The implementation of this idea in Sprotty is visualized in Figure 2.3. The unidirectional Flux-like flow is highlighted in orange.

A diagram visualization in Sprotty is started with a model request action. The action is handled by a *ModelSource*. The *ModelSource* returns an action that contains the diagram model and is required in every Sprotty IoC container. As explained above, the *ModelSource* can either be implemented locally or as a *DiagramServer* that synchronizes its handled actions with a server. Actions in Sprotty are always dispatched through the *ActionDispatcher*, which forwards actions to registered *ActionHandlers*. They convert an action into a command, further actions, or handle them without returning anything.

Commands describe how the locally stored diagram model should be modified, and describe how the modification can be undone and redone. The local diagram model is stored and managed by the *CommandStack*, which executes and potentially buffers incoming commands.

The diagram model is provided to the *Viewer*, which uses registered *Views* to map the

model to a virtual Document Object Model (DOM) structure, which is described in the next section. It is used to patch the actual DOM with the SVG representation of the model, so the diagram is displayed to a user. When a user interacts with the visualization, new actions are dispatched and the Flux-like cycle is executed again.

Sprotty is used in KEITH to render diagram models that are received from the language server [Ren18]. The implementation for this thesis continues to use Sprotty in the same way as KEITH, and extends the idea of a unidirectional, action-based flow with support for additional application data and views that are not concerned with the diagram model. The extension is part of the Sprotty IoC container and reuses the *ActionDispatcher* system.

2.8 Snabbdom

Sprotty uses Snabbdom to define its diagram model views declaratively. Snabbdom[16] is a virtual DOM library that focuses on simplicity and performance. In a web page, the DOM represents the structure of the page so that JavaScript is able to interact with it and change the structure, style, and content. A virtual DOM is a data structure used by many UI libraries, such as React and Vue.js, that describes how the actual DOM should look like. Based on the virtual DOM, libraries are able to identify changed nodes during a render update and apply granular changes to the DOM, instead of replacing the whole structure [Rea19].

In Snabbdom the application UI is expressed as a function of its state. The function receives arbitrary data, referred to as *props*, and returns a virtual node. Multiple functions are composed together to create the virtual DOM for the entire application. Functions that are composed to a virtual DOM structure are commonly referred to as *components*.

JSX[17] is an approach to compose UI components in JavaScript with a familiar syntax. It reads similar to HTML but can be written in JavaScript files. Since JSX is just JavaScript at runtime, it requires a build step that transpiles JSX expressions to pure JavaScript functions, which can be called by a UI library [Rea20]. Snabbdom provides an additional library[18] to use their function, used to describe virtual nodes, as a JSX compilation target.

Listing 2.1 highlights the usage of JSX and Snabbdom in the implementation for this thesis. The `SynthesisPicker` component describes a select UI that a user can interact with to change the synthesis for the visualized diagram model. The available options and currently selected option change based on the given `props`. Curly braces can be used to embed JavaScript expressions inside JSX. For example, inside Listing 2.1 a JavaScript expression maps an array of syntheses information to option nodes. Another assigns the synthesis identifier as a value attribute to each option node.

[16] `https://www.npmjs.com/package/snabbdom`

[17] `https://facebook.github.io/jsx`

[18] `https://www.npmjs.com/package/snabbdom-jsx`

```tsx
interface SynthesisPickerProps {
   currentId: string;
   syntheses: { displayName: string; id: string }[];
   onChange: (newValue: string) => void;
}

function SynthesisPicker(props: SynthesisPickerProps): VNode {
   return (
      <div classNames="options__column">
         <label htmlFor="synthesisSelect">Current synthesis:</label>
         <select
            id="synthesisSelect"
            classNames="options__selection"
            on-change={(e: any) => props.onChange(e.target.value)}
         >
            {props.syntheses.map((synthesis) => (
               <option value={synthesis.id} selected={synthesis.id === props.currentId}>
                  {synthesis.displayName}
               </option>
            ))}
         </select>
      </div>
  );
}
```

Listing 2.1. `SynthesisPicker` expressed as a UI component using Snabbdom flavored JSX.

2.9 Fastify

Since Node.js is an asynchronous event-driven runtime, it is a good candidate for web-server applications. Due to the event loop, Node.js is able to handle multiple connections concurrently [Fou21]. Although Node.js is designed without threads since JavaScript is a single-threaded language, the built-in *cluster* module can be used to enable load balancing between multiple processor cores. While Node.js has built-in support for creating HTTP servers, multiple libraries have been developed to simplify the implementation of a Node.js based web server.

One of the most popular Node.js libraries for creating a web server is Express[19]. Although the library currently has 16 million weekly downloads, it has not received many updates in recent years. At the time of writing, the latest stable version has been released two years ago and the latest alpha version over a year ago. This causes the library to lack support for newer JavaScript features, such as *Promises*[20], which are a data structure that represents the eventual completion of asynchronous operations.

[19] `https://expressjs.com`

[20] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`

A newer alternative for a web-server library is Fastify[21]. The library is actively maintained and receives regular updates. Furthermore, in synthetic benchmarks conducted by Fastify to evaluate the overhead of a library, Fastify is able to handle significantly more requests per second than Express[22].

Fastify has been built to be extremely modular [Fas21]. Everything is a plugin, which adds encapsulation and modularity to the library. A plugin can contain anything and might include route handlers, middleware, and request hooks. Furthermore, Fastify provides an official plugin to handle WebSocket connections.

The official support for WebSocket connections is important for this thesis and the main reason for using Fastify instead of Express. The modular approach further increases the reusability of the Fastify server developed for this thesis. A Fastify server has been configured that can serve standalone diagram views. Other projects or future work are able to reuse the server and extend its functionality by registering custom plugins.

2.10 Commander

Commander[23] is an npm package that is used to develop CLI applications. It is one of the most downloaded packages with 67 million weekly downloads. Commander takes a declarative approach to writing a Node.js CLI. The CLI script is able to focus on the actual logic and does not have to handle parsing and validation of options and arguments. Instead, a script declares available commands and specifies their expected options and arguments, as well as a callback function that will be called with the parsed arguments and options when a user executes its command. Furthermore, Commander provides helpful information to the user with an auto-generated help text. A script is able to modify and change parts of the text if required.

The software developed for this thesis uses Commander to implement a CLI for the standalone view that is easy to maintain. The help text provided to the user also serves as the user documentation.

[21] https://fastify.io

[22] https://www.fastify.io/benchmarks

[23] https://www.npmjs.com/package/commander

Chapter 3

Related Work

Using web technologies to visualize diagrams is not a new concept. Many frameworks and libraries exist to ease the implementation of diagram visualizations with web technologies. An overview of different, web-based diagram libraries has been created by Eugene [Eug20]. However, while many applications are able to display diagrams in a browser, only a few can also be used in VS Code, with support often added by third-party developers. The next section introduces a project that uses web technologies for diagram visualizations. It is followed by Sprotty examples that provide inspiration for a possible solution and the KEITH project, which serves as the foundation for this thesis. The last section introduces an architecture pattern that inspires the overall architecture for this thesis.

3.1 Mermaid

Mermaid[1] is a project that aims to help documentation catch up with development by providing easy-to-edit diagram visualizations for documentation. It specifies a Markdown-inspired syntax to define diagrams as text, so it integrates into Markdown more fluently, which is often used as a markup language in documentation tools. Mermaid has support for different diagram types, such as flowcharts, sequence diagrams, and class diagrams. The Mermaid documentation provides an overview about all supported diagram types and their syntax.

Next to integrations for various documentation platforms and generation tools, Mermaid provides a standalone live editor as well that can be easily accessed in a browser[2]. Figure 3.1 shows the editor visualizing a sequence diagram. The left side provides a basic text editor to edit the diagram definition, which also provides an idea about the syntax used by Mermaid for sequence diagrams. The live editor includes a synchronized preview of the editable diagram definition, which can be exported as an image or SVG. Furthermore, Matt Bierner developed a VS Code extension[3] that adds Mermaid support to the Markdown preview integrated in VS Code.

While Mermaid provides tools for platforms that are also focused on by this thesis, the diagram visualization and generation are part of the client. In contrast, this thesis continuous to separate the diagram visualization and generation between a client and a language server.

[1] https://mermaid-js.github.io

[2] https://mermaid-js.github.io/mermaid-live-editor

[3] https://marketplace.visualstudio.com/items?itemName=bierner.markdown-mermaid

Figure 3.1. Mermaid live editor that runs in a browser.

Furthermore, Mermaid's live editor combines a simple text editor and diagram view, whereas this thesis focuses on the diagram view without the ability to edit the source.

3.2 Sprotty Examples

TypeFox has developed numerous examples and smaller projects that highlight different use cases of Sprotty. Generally, these examples serve as inspiration for this thesis and highlight different aspects that must be taken into consideration. Such aspects include the communication between a client and server for the different platforms and more broadly the integration of Sprotty into these platforms.

The *sprotty-server* repository[4] contains an example that uses Sprotty in a standalone website. The example synthesizes the diagram model on a server, which is communicated to the website through a WebSocket connection. This highlights the use of Sprotty in a standalone view with a client-server separation. However, the communication does not use the LSP, which is a requirement for this thesis since the diagram synthesis with KLighD is part of a language server. The communication between a website and a language server is implemented by a CodeMirror LSP adapter project[5], which inspires the implementation for the standalone view in this thesis.

Another inspiring example is an npm package dependency inspector created by Type-

[4] https://github.com/eclipse/sprotty-server
[5] https://codemirror-lsp.wylie.su

Fox [Spö18]. The website of the inspector[6] mainly consists of a dependency graph visualized with Sprotty. However, it also contains an additional UI next to the diagram to search and select packages for which dependencies should be visualized. The implementation of the UI is independent of the Sprotty diagram and is added as an additional feature to the website. The standalone view for this thesis also requires additional UI on top of the diagram to display diagram options that allow a user to change aspects of the diagram synthesis. Therefore, the package dependency inspector showcases a possible implementation of additional UI in a standalone view.

Lastly, TypeFox also provides an example of a Sprotty VS Code integration [Köh19]. The example contains a language server for a simple state machine language. Further, the example contains a language extension that integrates the language server in VS Code. Sprotty is used to visualize a state machine model in a VS Code webview. To integrate Sprotty with VS Code, the *sprotty-vscode* glue code is used. This integration example inspires a possible VS Code integration of KLighD diagrams for this thesis. It fulfills many requirements that have to be met by a solution and is used as an integration guide for *sprotty-vscode*, since the documentation around the glue code is sparse.

3.3 Kiel Environment Integrated in Theia

This thesis is of course closely related to the master's theses of Rentz [Ren18] and Domrös [Dom18], who created the initial implementation of the KEITH project. The KEITH project has implemented a language server for model languages supported in KIELER. The language server has been integrated into the web-based Theia IDE. Furthermore, the language server is used to synthesize diagram models with KLighD. Since Sprotty has the ability to separate diagram generation and visualization between a client and a server, it is used to visualize the synthesized diagram models in the Theia client. Their theses have focused on Theia as a target platform for their implementation. In contrast, this work has developed a solution for multiple target platforms. The solution is based on their implementation in KEITH and reuses as many parts as possible. The language server can be completely reused without changes. The existing Sprotty code on the client is reused with small modifications. Additional features related to the diagram visualization were implemented with Theia dependent code, which have to be rebuilt to be useable in multiple target platforms.

3.4 Software Architecture

While the examples developed by TypeFox inspire implementations for the new target platforms, they only focus on one platform at a time. To achieve the goal of a high reusability and easy adoption by other projects, an architecture has to be found that contains as much logic as possible in a central core package to reuse the logic.

[6] `http://npm-dependencies.com`

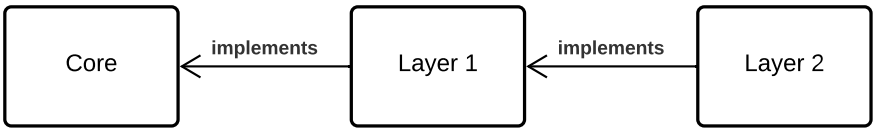


Figure 3.2. Concept behind a layered architecture similar to the clean architecture.

In software development, every project is concerned with reusable code and abstractions. Creating a well-structured codebase with clear abstractions and reusable parts is important for software that should be maintainable and extensible with new features. Over the years many architecture patterns have been created and generalized, which try to provide a foundation for a well-structured codebase. Since no codebase is the same and every project has different prerequisites and requirements, no architecture pattern exists that fits them all. Instead, understanding the idea behind each pattern can help to find an architecture that best fits the own project. The solution might even be a combination of two or more patterns, which only contains parts of each pattern and applies them to different areas of the codebase.

One architecture pattern that inspired the overall architecture of the implementation for this thesis is the clean architecture introduced by Martin [Mar12]. While the exact definition of the clean architecture does not fit well with the prerequisites for this thesis, which is the existing code in KEITH, the general idea of the architecture can be applied to achieve greater code reusability.

It divides the software into layers that form around a software core. Code towards the core contains policies and business logic and is rather abstract from any concrete implementation [Mar12]. This changes with each layer built on top of the core. Contracts for features, such as data persistence and transports, are replaced with actual implementations. The idea is roughly visualized in Figure 3.2. The key point and main inspiration for this thesis is the flow of source code dependencies. Dependencies in the clean architecture are only allowed to point inwards [Mar12]. This can be achieved with the dependency inversion principle. If the core logic requires access to a database, it should define an interface for the required data persistence functionality and only rely on the interface. The interface is implemented in an outer layer that has access to a database and the implementation is provided to the core by that layer. This avoids dependencies that point outwards, since the inner layer only relies on its defined interfaces.

This idea has influenced the software developed for this thesis, which implements an architecture that consists of many reusable parts and can be easily adopted by other projects. The existing diagram rendering core in KEITH is reused and extended with most features that are related to diagram visualizations, and has no implemented dependencies to platform-specific functionality.

Chapter 4

Concepts

This chapter gradually introduces and outlines the new architecture proposed for this work. The current implementation in KEITH serves as a foundation for the new architecture, which reuses as much code as possible. Therefore, the structure of KEITH and existing diagram-related features are introduced first to understand what can be reused.

Often, multiple approaches exist to support a target platform. The following sections introduce the approach taken for each target platform and discuss possible alternatives. Afterwards, the approaches are combined into the new overall architecture that has been implemented for this thesis. As part of the new architecture, decisions about the implementation of previously existing features and their location in the new architecture are outlined. This work has put a focus on a coherent UI for additional diagram features, which is discussed in the last section.

4.1 Understanding the Foundation—KEITH

KEITH is using Theia as its IDE platform, which enables it to run in a browser and as a desktop application. Since it behaves the same as any text editor, the user is able to open and edit files along other common IDE features. This work will focus on the diagram-related features that are special for KEITH and are available for languages supported by the KIELER project. More information about KEITH can be found in its documentation[1].

A user is able to visualize the model in a supported file by opening the context menu and selecting *Open in Diagram*. This opens a new editor tab that shows the diagram visualization.

4.1.1 Available Diagram Features

Based on the screenshot of the KEITH editor in Figure 4.1 and actual usage, the following features can be identified that are related to the diagram visualization. These features where first implemented and are further described by Rentz [Ren18]. They should be present in the new platforms as well. Therefore, it is important to not only identify the features but also determine how they are implemented in KEITH to identify parts that can be reused in the new architecture. The following features are also marked in Figure 4.1 for better clarity.

[1] https://rtsys.informatik.uni-kiel.de/confluence/x/TYAkAw

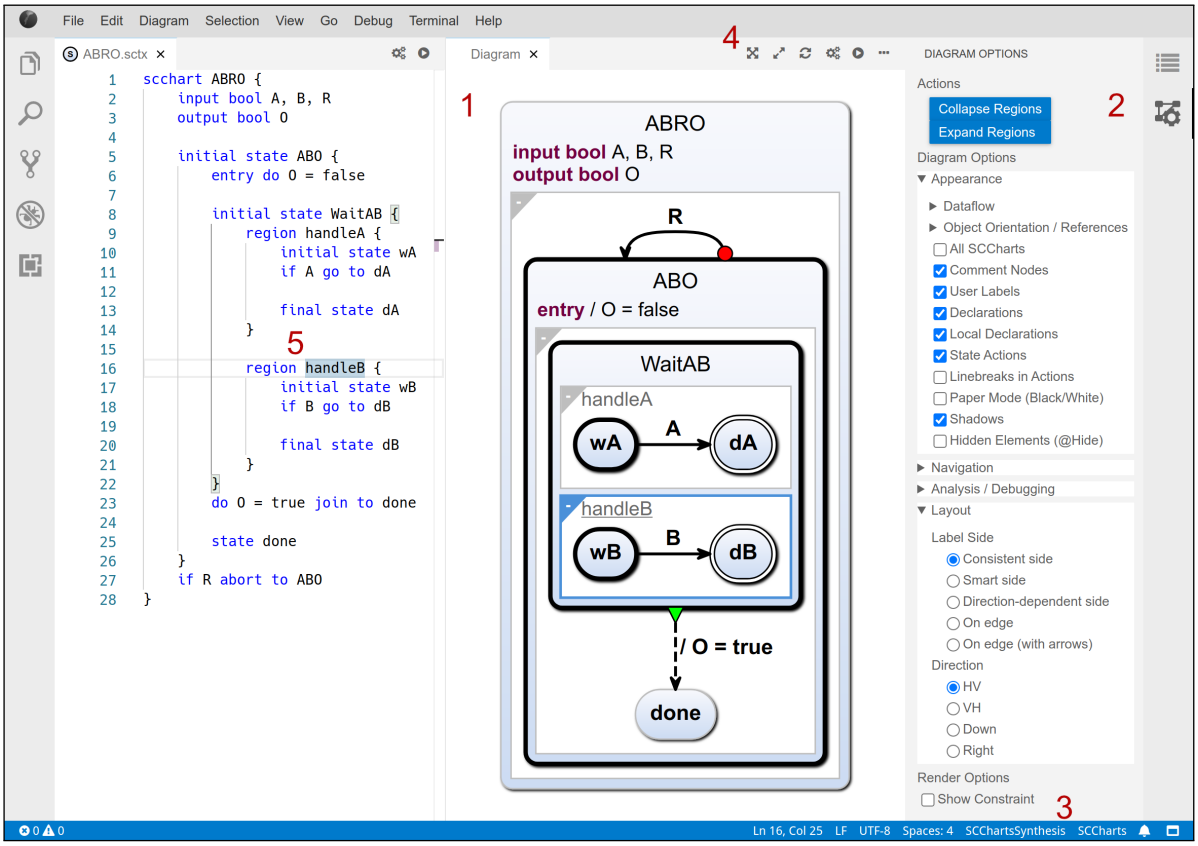


Figure 4.1. Screenshot of the KEITH editor with the current model visualized as a diagram. Features are 1: a diagram view, 2: an options view, 3: a synthesis picker, 4: actions, 5: selection synchronization.

1. A *diagram view* uses a Sprotty IoC container to visualize the opened model and provides a user the ability to interact with the diagram. The interactions include zooming, panning, selecting elements, and showing hover information.

2. Next to the diagram view, an *options view* is located, which is implemented as a Theia widget. The options view contains synthesis and layout options, which affect how the diagram model is synthesized by KLighD. Changes to the options are persisted and are used to initialize the options in a future session.

3. A status bar contains information about the *current synthesis* that has been used by KLighD. In Figure 4.1, the synthesis has been set to the *SCChartsSynthesis*. Clicking the label reveals a dropdown selection in Theia where the user is able to select another synthesis from a list of available syntheses provided by KLighD.

4. A toolbar on top of the diagram view contains *actions* that affect the current visualization. For this thesis, the *center*, *fit*, and *refresh* actions are relevant. *Center* resets the zoom of the diagram view and centers the diagram. *Fit* zooms the diagram view and centers the

diagram to fit into the diagram view while consuming all available space. *Refresh* requests a refresh of the diagram model from KLighD.

5. The user is able to *synchronize the text selection* in the editor with the element selection in the diagram view. For example, selecting `handleB` in the text editor selects the corresponding region in the diagram view, which is indicated by a light blue border around the region. This feature can be enabled for both directions or a single direction, e.g. only selections in the diagram view highlight the corresponding text, or turned off completely in Theia's editor settings.

To support all of these features in the new target platforms, many have to be completely re-implemented in the client to make them usable in the new target platforms. As indicated in Section 3.3, the original implementation of these features is used as a reference for the new implementation.

4.1.2 Project Structure

The KIELER language server in KEITH runs on the Java virtual machine and is not part of the web-focused KEITH repository. Since the LSP is developed to support multiple platforms with one language server by relying on a platform-independent protocol [Mic21d], each new target platform can reuse the language server as long as it is able to communicate with the server. VS Code, being the first IDE that supported the LSP, is able to reuse the language server. With some considerations about the connection, a browser is also able to communicate with the language server. Therefore, the language server implementation can be reused and does not require changes as long as the exchanged messages presented in Section 2.5 remain the same. It does not prevent the implementation of previous features for each target platforms.

The main reason for a required re-implementation is a dependency to Theia that exists for most features on the client. Obviously, the Theia API is not available in VS Code and a standalone browser view. Therefore, they have to be implemented without a dependency to Theia. Currently, KEITH consists of multiple Theia extensions. All extensions are Node.js packages, which are managed together in a repository. Figure 4.2 visualizes the relation between KEITH's packages that implement diagram-related features.

The *keith-sprotty* package contains a Sprotty IoC container that is able to render models that are synthesized by KLighD. The container registers Sprotty views that describe how the diagram model is translated to SVG, so it can be displayed by the diagram view [Ren18]. *Keith-sprotty* imports and registers an IoC container module that is provided by *keith-interactive*. The module adds interactive layout capabilities to the Sprotty IoC container, which were introduced by Petzold [Pet19] and Schönberner [Sch19]. Both packages do not rely on Theia and can be considered the core of the diagram visualization in KEITH.

The container is integrated into Theia by the *keith-diagram* package, which uses the glue code provided by the *sprotty-theia* library. A `ModelSource`, which is required by a Sprotty container, is not implemented in *keith-sprotty*. Since the model is provided by the language server, a `DiagramServer` that functions as a `ModelSource` is implemented in *keith-diagram* and

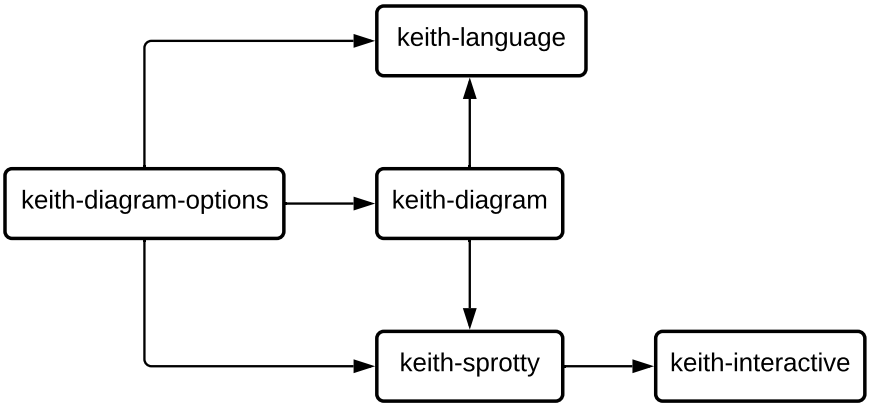


Figure 4.2. Dependencies between KEITH's packages that implement diagram related features.

registered in *keith-sprotty*'s IoC container. The `DiagramServer` uses a language client provided by the *keith-language* package to communicate with the KIELER language server.

Most of the available features listed above are implemented in the *keith-diagram* package. An exception is the diagram options view, which is implemented in the *keith-diagram-options* package. The view is based on a Theia Widget to integrate it into Theia's UI. Changes to the options are communicated to the language server with the language client provided by *keith-language*. Further, the package depends on *keith-sprotty* to import TypeScript interfaces for the different options. Information about the current diagram is provided to the options view by *keith-diagram*.

The new solution is only able to reuse the *keith-sprotty* and *keith-interactive* packages, because they are the only packages that do not depend on Theia. Therefore, the Sprotty IoC container that has been created in KEITH can be reused.

In a diagram visualization that is based on the Sprotty framework, the Sprotty IoC container forms the core of the visualization. Therefore, the remainder of this thesis will refer to the *keith-sprotty* and *keith-interactive* packages as the core or diagram core. The Sprotty IoC container in these packages will form the core of the new architecture.

4.2 Visual Studio Code Support

VS Code extensions are plugins that extend VS Code with new capabilities, such as support for new languages, color themes, new views, and more. Support for new languages is usually achieved with a language server that adheres to the LSP and is integrated into VS Code by an extension.

VS Code provides an npm package called *vscode-languageclient*[2] that should be used to start

[2] `https://www.npmjs.com/package/vscode-languageclient`

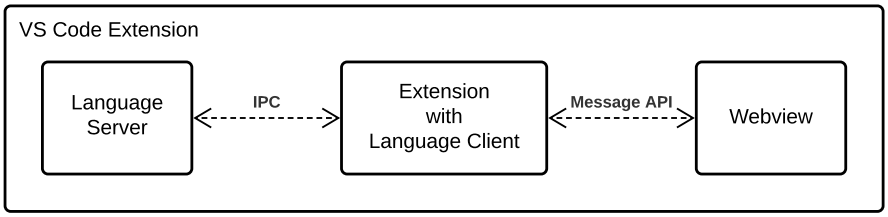


Figure 4.3. Communication flow in a VS Code extension that uses a language server and webview.

a language server in a VS Code extension [Mic21c]. This npm package creates and configures a language client, which is used to manage and to connect to a language server. During runtime, an instance object of such a language client can be used to communicate with the language server that is started and managed by the instance. The communication can use different transports, such as sockets and IPC.

To continue to start quickly with many extensions installed, VS Code activates extensions only when they are required. All extensions define activation events, which allows VS Code to call their activation function when a defined event occurs. Such events may include opening a file for a language that is supported by the extension, or executing a command that should be handled by the extension.

This work has implemented and published a VS Code extension to support VS Code as a new target platform. The remainder of this section introduces two VS Code extension concepts that are important for the implementation for this thesis. Refer back to Section 2.4 for general information about VS Code and the general structure of VS Code.

4.2.1 Webviews

Webviews are special views in VS Code that are able to display fully customizable content. A webview in VS Code works similar to an Inline Frame (iframe) element in a website [Mic21f]. VS Code displays a webview as a distinct editor in the editor area, where it can be used in split view together with other open editors. Since webviews are able to run arbitrary code, they are isolated from the rest of the extension. Similar to an iframe, the content of a webview is defined by a custom HTML structure, which is able to load custom JavaScript and CSS styles.

Due to the isolation of webviews, a webview is not able to call code from its extension directly. Instead, a message API has to be used to send data as messages to and from the webview [Mic21f]. The communication between a webview and its extension is visualized in Figure 4.3. If a webview wants to send a notification to a language server, it has to send the notification as a message to the extension. The extension has to handle the received message event and has to proxy the included notification to the language server through a language client instance.

If a diagram that is rendered with Sprotty should be displayed in VS Code, a webview

has to be used to display the custom content [Köh19]. The webview contains the Sprotty IoC container and has to be managed by the extension. Managing a webview includes creating the webview if a visualization is requested for a file. Further, the file URI has to be communicated to the webiew, otherwise Sprotty cannot start to request a diagram model from a language server. The actions that are communicated between Sprotty and a language server have to be proxied by the extension as shown in Figure 4.3. When a user switches to a different file that is supported as well, the existing webview should be notified to start updating the visualized diagram for the new file. In case the webview is closed by the user, existing references and resources have to be cleaned up.

All this boilerplate to wire-up and manage a webview is almost the same for every diagram visualization in VS Code that uses Sprotty. Therefore, Sprotty created a glue code library, called *sprotty-vscode*, that hides this boilerplate [Köh19]. Instead of implementing its own glue code, the implementation for this thesis uses *sprotty-vscode* to rely on an existing boilerplate, so it does not create a potentially worse copy that increases maintenance cost for future maintainers of the project. If possible, required changes should be contributed back to the community and not to its own copy.

4.2.2 Inter-extension Communication

Custom commands can be defined by an extension to create interactions for the user [Mic21a]. These commands are defined in an extension configuration file. The implementation of commands is later registered when the extension is activated by VS Code. Inside the configuration, contribution points can be defined as well to place toolbar buttons or context-menu items in the UI, which trigger their corresponding commands.

However, commands are not only useful to create user interactions. They can be executed by other extensions as well [Mic21a]. This is one of two possibilities to enable inter-extension communication, which can be useful to provide common functionality to many extensions through commands in a shared extension that they can depend on.

Listing 4.1 contains a minimal example of a communication between two extensions that uses command calls. *Extension A* registers a basic echo command that returns any argument it receives. *Extension B* declares *Extension A* as a dependency in its configuration file to ensure that *Extension A* will be installed when *Extension B* is installed by a user. In the example, *Extension B* calls the echo command with an object and stores the result. Comparing the object with the echoed object by referential equality reveals that they have the same reference. This shows that objects are passed by reference between commands in VS Code and no copy is created.

Therefore, a language client instance, which is provided to another extension through a command call, would be the same instance and would contain a working connection to the language server that is represented by the client. This pattern can be used in the implementation for this thesis to further minimize the boilerplate for projects that want to visualize KLighD-synthesized diagram models in VS Code.

The VS Code extension that is developed for this thesis is therefore split into two extensions.

```
// Extension A
vscode.commands.registerCommand("extA.echo", (obj: unknown) => {
   return obj;
});

// Extension B
const obj = { key1: "Hello", key2: "World" };
const res = await vscode.commands.executeCommand("extA.echo", obj);

console.log(obj === res); // => true
```

Listing 4.1. Example of a command call between two VS Code extensions.

The first extension is a diagram extension that uses *sprotty-vscode* to visualize Sprotty diagrams. It uses a language client to communicate to a language server with KLighD synthesis support. Similar to the dependency inversion principle discussed in Section 3.4, an instance of the language client is provided by other extensions through a command call. The second extension integrates the KIELER language server in VS Code and depends on the first extension for visualization support.

4.3 Theia Support

Section 2.6 already mentioned the future of the LSP support in Theia. Supporting many programming languages and frameworks is a difficult task for any IDE [Eff20b]. While the LSP removes most of the work required, a language server still has to be integrated into an IDE. Further, an IDE has to actively keep up to date with the LSP and implement support for changes and additions to the protocol. IDEs can solve this either by growing a massive community of extension developers, which is hard to achieve, or by hiring many developers, which is very costly [Eff20b]. As Efftinge puts it, Theia solves this problem by leveraging the largest and most active IDE extension community [Eff20b]. Instead of growing a massive community themself or hiring dozens of developers, Theia taps into the VS Code community by supporting the VS Code extension API.

With the support of VS Code extensions, Theia deprecated their own support for the LSP in version 1.4 [Köh20]. Developers should implement their language extensions as VS Code extensions and use them in Theia directly.

Therefore, the solution for this thesis will not focus further on the Theia-based IDE developed in KEITH. Instead, it will ensure that the VS Code extension developed during this thesis can also be used in Theia and therefore be used by KEITH.

To publish their VS Code extensions, developers rely on the Visual Studio Marketplace[3] from Microsoft. VS Code integrates with the marketplace and allows users to download

[3] https://marketplace.visualstudio.com

extensions from within their IDE. However, the marketplace's Terms of Use[4] prevent non-Visual Studio products from accessing the marketplace. As an open-source alternative, the Open VSX registry has been developed to host and download VS Code extensions from a vendor-neutral marketplace [Eff20b].

Theia uses the Open VSX registry as its default extension registry [Eff20b]. Furthermore, a custom Theia IDE is able to provide built-in extensions by defining a URL to a VS Code extension, such as one hosted on the Open VSX registry, in its build configuration [Fug20]. Therefore, Theia powered IDEs, such as Gitpod or KEITH, can either install or pre-bundle VS Code extensions from the Open VSX registry.

To ensure that the VS Code extensions developed for this thesis are accessible in Theia, the extensions are not only published to the Visual Studio Marketplace, but also to the Open VSX registry.

4.4 Standalone Diagram View

A standalone diagram view that is displayed in a browser can be approached from two general directions. Each has unique advantages and disadvantages, which are discussed in this section.

In both cases, the diagram view has to establish a full-duplex communication with the language server, which means that the server has to be able to send messages to the client and the other way around. To achieve this in a browser, a WebSocket connection has to be established.

While the KIELER language server supports socket connections for debugging purposes, it does not support WebSocket connections. Further, a language-server process is supposed to handle a single connection and the KIELER language server does not accept multiple concurrent socket connections. If WebSocket support is implemented in the language server directly, it should be able to handle multiple concurrent connections as well to support use cases, which are discussed in the next subsection.

A cleaner approach implements a web server as an intermediate proxy layer, which is required anyway to serve the website. The web server can be extended with WebSocket support, which is outlined in Figure 4.4. If a client tries to establish a WebSocket connection, the web server spawns a language server for that client and proxies all messages via IPC. With this approach, the language server does not have to be modified. Further, it only needs to handle a single connection, because the web server spawns multiple language servers for multiple connections.

4.4.1 Hosted Service

The first approach to support a standalone diagram view uses a hosted diagram service. The service consists of a web server, which serves the user interface and creates a language-server

[4] https://aka.ms/vsmarketplace-ToU

Figure 4.4. Web server that proxies multiple WebSocket connections to language server processes.

process for incoming connections, as discussed above. A user navigates with a browser to a registered domain to use the service. To view a diagram visualization, the user has to select a file that contains a supported model from his/her computer. The diagram service does not necessarily require the user to upload a file to make it accessible for the language server, because the content of a file can also be transmitted during the LSP *textDocument/didOpen* notification sent by the client [Mic21d]. However, to enable more advanced use cases other than viewing a diagram visualization, the service would have to upload and store the selected file.

This approach enables embeddable diagram views and shareable links for interactive diagram visualizations that were motivated in Chapter 1. If a file is uploaded to the server, it is possible to generate a URL for the contained model, which makes the model accessible to other users when the URL is shared. A similar URL can be used to link to a minimal interactive view of a diagram that can be embedded into other websites with an iframe element.

However, some concerns arise when a file is stored on the server. Policies in organizations often prevent sharing of closed source code. This might include uploading a model file to such service as well. To work around this, organizations would have to host the diagram service themself. Furthermore, the Real-Time and Embedded Systems Group noted that they are currently not planning to host a diagram service, since storing the files securely requires additional care.

In conclusion, this approach enables advanced use cases that are only possible with a hosted diagram service, but is currently not feasible due to concerns about hosting such service.

4.4.2 Locally Running Service

The second approach to support a standalone diagram view uses a locally running diagram service. The web server together with the language server runs locally on the user's computer. To access the diagram view, a user has to start the web server and navigate with a browser to the opened port on the special *localhost* URL. Since the language server runs on the local computer, it has access to the user's file system and is therefore able to access the content of a local file URI that is provided with the LSP *textDocument/didOpen* notification.

However, how does the diagram service know which file should be visualized? If a user selects a file on the website similar to the hosted approach, the website would not have access to the actual file path on the local computer. Browsers are sandboxes and prevent a website from reading the local file path. Therefore, the language server would not receive a URI it can access. To solve this problem, a user could specify a file URI within the website's URL when the diagram view is opened. This skips the whole file selection process on the website since the URI of the file that should be visualized is already known and can be made accessible to the language server. The following template shows a URL that follows this approach:

```
http://localhost:8000?source=file:///<path-to-diagram-file>
```

The URL search parameter `source` contains the file URI that will be provided to the language server. The placeholder `<path-to-diagram-file>` refers to an absolute path in the user's file system.

The process that is currently proposed for visualizing a model with this approach is rather tedious. A user has to manually start the web server, identify the absolute path of the file that contains the model, and manually construct and navigate to the website URL to open the diagram view. These steps can be automated and hidden with a small CLI application. The user provides a file path to the application, which starts the web server and opens the diagram view with the constructed URL in a browser. The file path provided by the user does not need to be absolute and can be converted to an absolute path, further simplifying the process for the user.

A locally running web server that is started by a CLI application does not support the more advanced use cases, which could be supported by the previous approach. Since the URL of the diagram view is only accessible on the local computer, it cannot be shared within an organization or embedded into online documentation websites. However, it does not require a hosted service and is therefore feasible for the Real-Time and Embedded Systems Group. Furthermore, if the documentation is also hosted locally with the diagram source files, the CLI can be used to start the web server to view embedded diagram visualizations in the local documentation website.

Going forward, the CLI application approach is implemented for this thesis. All features of the web server for this approach are also required for a hosted service approach. In case the latter should be implemented in the future, the existing web server can be extended to support file uploads.

4.5 New Architecture

Now that the different approaches for each target platform are outlined, they can be combined into an architecture for the solution implemented for this thesis. The architecture focuses on reusability and extensibility. This is achieved by sharing as much code between each platform as possible, and by providing multiple possibilities for future work to extend this solution.

Figure 4.5 visualizes the new architecture at an abstract level. In the diagram, components

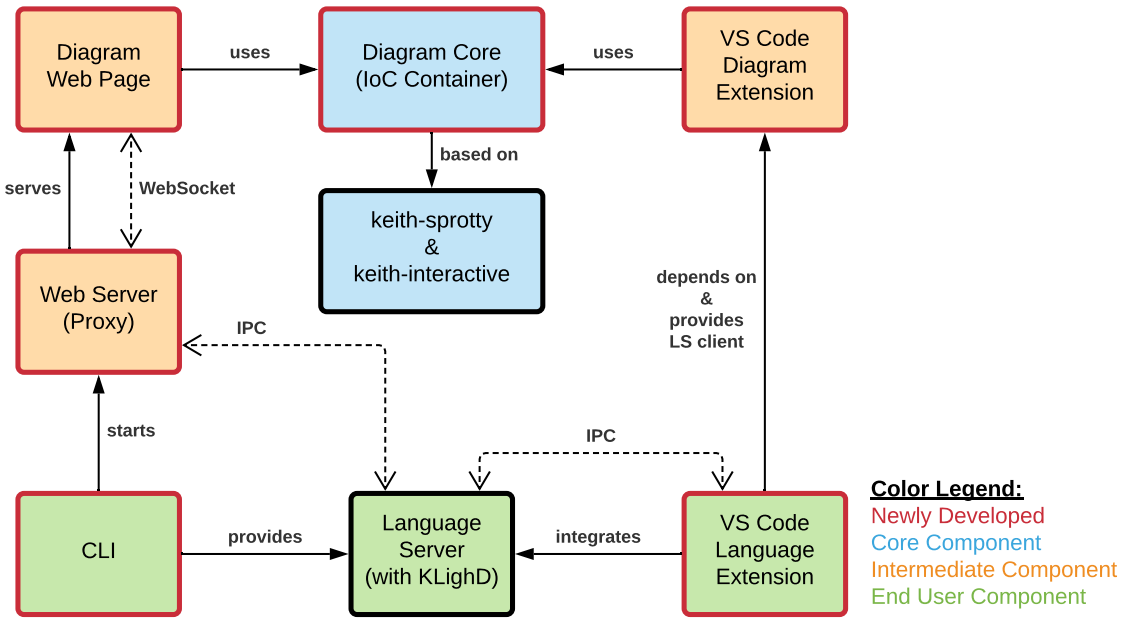


Figure 4.5. Connection and communication between the different components in the new architecture.

that are newly developed for this thesis are outlined in red. Furthermore, each component is color-coded by the role it fulfills. Core components are colored blue indicating that they contain core logic and are platform-independent. They are used by intermediate applications, which are colored orange and provide support for different platforms but are not supposed to be used directly by end users. Instead, end users have control over the green colored components, which are end user applications. A special case is the *Language Server* component. While it is not directly an application the end user interacts with, it can be replaced by the user, and a different *Language Sever* can be used to visualize a model for a different model language.

The architecture is built around the *Diagram Core*. It is based on the Sprotty IoC container that had already been created for KEITH by Rentz [Ren18] and is extracted from *keith-sprotty* and *keith-interactive*, as it is explained in Section 4.1.2.

Following the architecture for the standalone diagram view on the left, the *Diagram Core* is integrated into the *Diagram Web Page* to create a functional web page that displays a diagram visualization. The implementation follows the approach discussed in Section 4.4.2. The *Diagram Web Page* is built into static assets that consist of HTML, JavaScript, and CSS files, which are served by the *Web Server*. When the *Diagram Web Page* is opened, it creates a WebSocket connection to the *Web Server*, which spawns a *Language Server* process and forwards incoming messages to and from the *Language Server* via IPC. However, the *Web Server* does not have a hard dependency to the *Language Server*. Instead, the user provides a path to

a *Language Server* through the *CLI*. The *CLI* is the user-facing application that starts the *Web Server* and provides the given path to a *Language Server*. Further, the *CLI* opens the *Diagram Web Page* in the browser that is set as the system's default browser by the user. Therefore, the standalone diagram view can be used via the *CLI* with any *Language Server*, as long as the *Language Server* supports the extended LSP defined by Rentz [Ren18].

The support for VS Code and consecutively Theia is visualized on the right side of the architecture overview. It uses the approach outlined in Section 4.2 and Section 4.3. The *Diagram Core* is integrated into a webview by the *VS Code Diagram Extension*, which is able to display Sprotty diagrams in VS Code. The integration uses the *sprotty-vscode* glue code created by TypeFox to manage the webview and to route Sprotty actions as messages from the webview to the language client [Köh19]. In VS Code, a language client provides a configured connection to a *Language Server*. The language client is not part of the *VS Code Diagram Extension* in the new architecture. Instead, it is provided to the extension by an additional *VS Code Language Extension* through a command execution, which also activates the *VS Code Diagram Extension*. The *VS Code Language Extension* is the actual extension that integrates and bundles a *Language Server*. It depends on the *VS Code Diagram Extension* to provide diagram visualizations for the languages supported by its *Language Server*. Since the *VS Code Diagram Extension* does not contain a language client, it is not ready-to-use and should not be installed by end users directly. Instead, it enables other projects to add diagram visualizations to their VS Code extension with minimal effort. As part of this work a *VS Code Language Extension* is created to integrate the KIELER language server, which had already been created for KEITH by Domrös [Dom18] and Rentz [Ren18].

In conclusion, if other projects have a *Language Server* with KLighD synthesis support, they have three options to extend and use the diagram visualization. Ordered from most to least complexity, they can integrate the *Diagram Core* for maximum flexibility as every component can be replaced in the IoC container, extend the *Web Server* with additional functionality and display the *Diagram Web Page* in an iframe, or create a *VS Code Language Extension* that depends on the *VS Code Diagram Extension*. Alternatively, the CLI can be downloaded[5] and used directly with their *Language Server* if a basic browser-based diagram view fulfills their use case.

4.5.1 Location of Diagram Features

While Figure 4.5 provides an overview of the new architecture, it does not clarify where the diagram-related features that were identified in Section 4.1.1 are located.

One approach would be to keep the *Diagram Core* a simple Sprotty IoC container that is able to render a diagram model to SVG. All other features, such as the synthesis picker and diagram options widget, would be implemented in the *Diagram Web Page* and *VS Code Diagram Extension*. Furthermore, the model source that is required by the Sprotty container would have to be implemented by each platform. Depending on the platform, it requires a different connection type to the *Language Server*. In the *Diagram Web Page*, a WebSocket connection has

[5] https://github.com/kieler/klighd-vscode/releases

to be used to communicate with the *Language Server*. In the *VS Code Diagram Extension*, the webview message API is used to exchange Sprotty actions between the Sprotty container in the webview and the extension, where they are routed to the *Language Server* by the language client.

However, with this approach, all features would have to be implemented twice. Another project that wants to build upon the *Diagram Core* would also have to implement these features if they should be available as well. This goes against the idea of achieving a high reusability. Ideally, all features are only implemented once. Therefore, every feature should be implemented in the *Diagram Core*, because it is shared between all platforms. To achieve this, the IoC container in the *Diagram Core* should not be considered as a container for the Sprotty framework that happens to be a generic IoC container, but rather as a generic IoC container that happens to support diagrams with the Sprotty framework. With this mindset, it is easier to perceive that the IoC container is able to hold generic application data, such as diagram options or the names of available syntheses, as well as additional UI components, which are required for the diagram options view and synthesis picker.

To move the model source into the *Diagram Core*, an implementation that does not depend on a specific connection has to be used. The chosen approach relies on the dependency inversion principle and the idea of architecture layers introduced in Section 3.4. Inside the *Diagram Core*, the model source implementation injects and uses a `Connection` interface from the IoC container. However, the `Connection` is not implemented in the *Diagram Core*. Instead, it is implemented and registered in the IoC container by each target platform that integrates the *Diagram Core*. This approach hides the model source specific logic, such as handling custom Sprotty actions, from the applications, and at the same time keeps the *Diagram Core* free from platform-specific logic related to the transport of actions.

4.6 Reusable User Interface for Diagram Options

This section takes a step back and focuses on concepts for a UI to display diagram options and a synthesis picker. When it comes to a UI, trade-offs have to be made between implementing a UI that feels as native to each platform as possible, and a UI that is reusable between all platforms.

A UI for diagram options that is native to VS Code would have to be implemented as a view in VS Code. An additional view container could be placed in the activity bar to include the diagram options view in the sidebar. The synthesis picker could be placed in the status bar to show the current synthesis and a VS Code selection dropdown when it is clicked, similar to the previous approach in KEITH.

However, such UI implemented for VS Code is not usable in the standalone view and requires an additional implementation just for the standalone view. Furthermore, expanding the solution to more platforms requires an additional implementation of the UI as well. This goes against the goal of achieving a high reusability and the idea to implement all features in the diagram core, which has been outlined in the previous section. It forces each platform to

re-implement diagram options and a synthesis picker just to have a UI that feels as native as possible.

An architecture that implements the diagram options and synthesis picker in the diagram core requires a solution for rendering an additional UI, which is not related to the diagram visualization, inside the diagram core. While such a solution does not use a native UI approach, it is able to provide one UI implementation for all platforms. Depending on the visual design of the UI, it might be able fit in quite well with small compromises in the user experience. An advantage of such solution is that the UI is always implemented near the diagram visualization, keeping them contextually close together.

One solution is an additional layer on top of the IoC container in the diagram core. It might be implemented with a UI library, such as React[6], to create a component tree for the diagram view and additional UI, which can be rendered into the DOM by any platform that integrates the diagram core. However, this approach abstracts the IoC container it is layered on top of, which makes the approach less flexible for modifications by integrating platforms, since they are no longer able to replace every part of the diagram core. Furthermore, the approach prevents the usage of the *sprotty-vscode* glue code, because *sprotty-vscode* requires access to the Sprotty IoC container. The glue code should be used to integrate the diagram core in VS Code to avoid re-implementing the webview management code, which is provided by the library.

Instead, an undocumented feature in Sprotty can be used to implement additional UI as part of the IoC container. Sprotty has a concept called *UI extensions*, which are only documented in the git commit[7] that introduced them. A UI extension is rendered in a separate DOM element on top of Sprotty diagrams. It is registered in the IoC container and thus is part of the general Sprotty architecture (Section 2.7), which enables it to dispatch actions for user interactions.

For the implementation of this thesis the *UI extensions* approach is used to add a UI and thus all features in the diagram core, while preserving the ability to use *sprotty-vscode* and the ability to replace any part of the diagram core. Therefore, the goal of a high reusability can be achieved and is not prevented by a UI solution.

4.6.1 Behavior of the Additional User Interface

The diagram options UIs in both KEITH and KIELER have been implemented next to the visualized diagram, as it can be seen in Figure 4.1. The available screen space for the diagram view is shared between the diagram options UI and the visualized diagram. While the options can be collapsed in both implementations and moved to different widget regions in KEITH, they are often kept open and consume screen space.

For future use cases, the standalone diagram view might be embedded into documentation where the available space can become quite tight. Furthermore, the available space for the diagram view in an IDE can also be quite tight if a file is opened next to diagram view or the IDE does not consume all available screen space. Therefore, it must be possible to hide

[6] https://reactjs.org

[7] https://github.com/eclipse/sprotty/commit/f67631f3503e7402ecacf440fdb5639f7ca94370

the diagram options, as it has been before, to use all available space for the diagram view. This is even more important in documentation where the focus is on the diagram and not the diagram options.

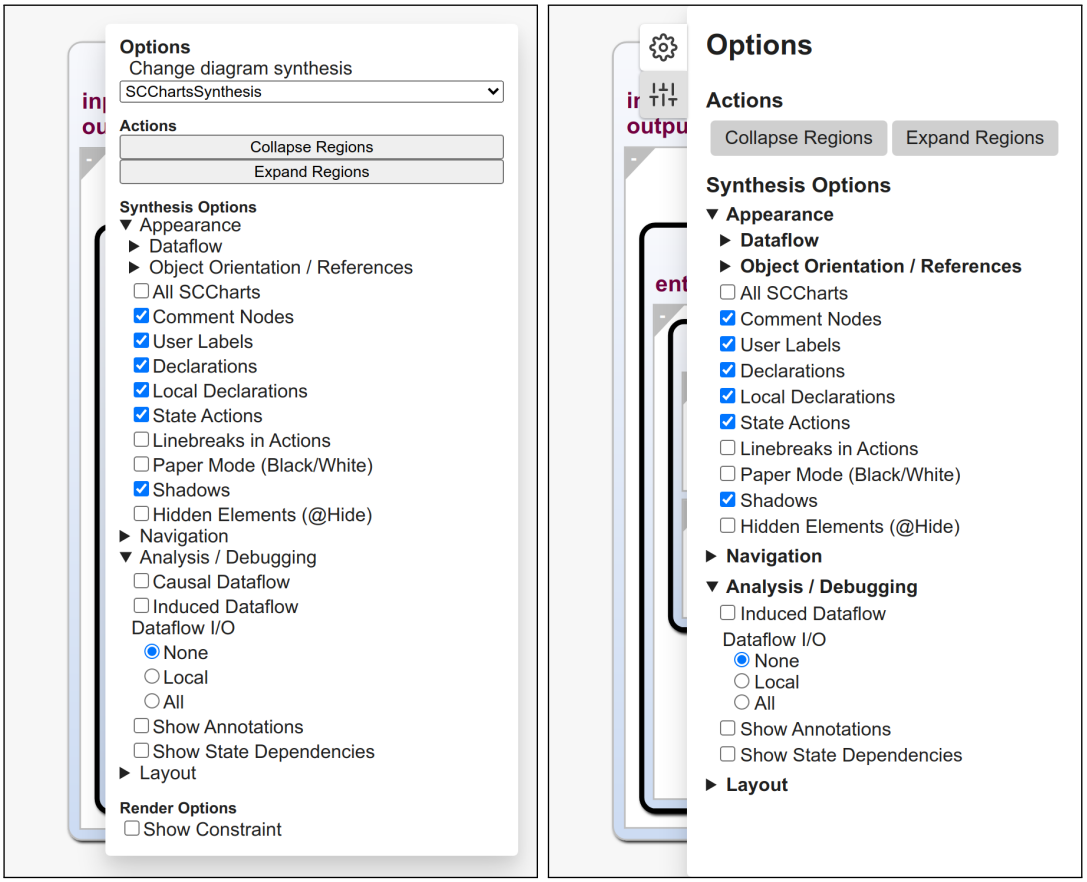
Under the assumption that a user focuses on the diagram view most of the time and only interacts with the diagram options for short periods, this thesis diverges from the previous UI approach in KEITH and KIELER. The diagram options UI is hidden by default and can be shown by clicking a small trigger. When the UI is shown, it should be placed on top of the diagram visualization instead of next to it. Most of the time, the diagram is still visible enough to see the effects of an option change. While the diagram might be mostly covered in smaller views, there is not enough space to fit the options UI next to the diagram comfortably. This is solved by rendering them on top of the diagram. The diagram options UI gets closed as soon as the user closes it explicitly or starts to interact with the diagram again, shifting the focus away from the options. In short, the new UI shares the available screen space between the diagram and diagram options, and gets out of the way, as it is only visible when it should be.

This work has experimented with two approaches for a floating diagram options UI. The result of each approach can be seen in Figure 4.6.

The first approach in Figure 4.6a uses a popover, which floats near the right edge of the diagram view. The effect is visually achieved through the use of a soft shadow and a margin to the edges of the view. The trigger to show the popover is a small icon button that is not visible in Figure 4.6a. It is underneath the popover in the top right. A mouse press outside the popover hides it again.

A drawback of this approach is the lack of support for new features. During the implementation for this thesis, other theses, which were implemented at the same time, have already shown interests in supporting additional UI as well. They would have to implement another popover and trigger to follow a coherent design. The additional triggers may be placed in the top right with a margin to the edges as well, which creates a cluster of icon buttons. This cluster might grow rather quickly, thus consuming too much space in the diagram view, caused by the fact that they are not placed completely on the edge. Furthermore, switching between the different UIs would require closing the current one to open another one. This additional step might get annoying over time and worsens the user experience.

The second approach in Figure 4.6b tries to fix the drawbacks of the first approach. It is a sidebar placed on the right edge of the diagram view that is inspired by a drawer. The trigger icon buttons are grouped underneath one another on the right edge. When a trigger is clicked by the user, the sidebar opens and slides in on from the right, while floating on top of the diagram view. The sliding effect is animated and just like before, the floating effect is achieved with a soft shadow. When a trigger is clicked while the sidebar is already open, the sidebar content is replaced with the content that should be shown for the clicked trigger. Therefore, the user is able to switch between the UIs without having to close the sidebar first. Clicking the already active trigger, highlighted in light gray, or interacting with the diagram view closes the sidebar, which causes the sidebar to slide out. The trigger icon buttons remain visible on the right edge, ready to be clicked again.



(a) UI floating as a popover.

(b) UI floating as a slide-in panel.

Figure 4.6. Two approaches to show a floating UI for diagram options above the diagram.

With this approach new features are able to contribute additional UIs to the diagram view, which are collected and displayed in a coherent approach. All triggers are placed to the far right of the diagram view, thus not consuming as much space as the first approach. The UI that can be contributed to the sidebar has been named a *sidebar panel*. A sidebar panel contains an icon that should be used for its trigger and the content that should be shown when its trigger is active.

For this thesis, the first approach has been abandoned, and the second approach has been implemented. This is the reason why the diagram options in Figure 4.6a look different compared to Figure 4.6b and has nothing to do with the actual approaches. The popover never left the proof of concept phase, since the sidebar has solved its drawbacks. The implementation for this thesis splits the synthesis picker and diagram options into two sidebar panels. The first panel contains the synthesis picker, as well as user preferences and quick actions. The second panel contains the diagram options that are provided by the diagram synthesis and can be seen in Figure 4.6b.

Chapter 5

Implementation

Implementing the architecture that has been developed in the previous chapter requires smaller and bigger software changes and additions. Many of them follow the Sprotty examples that have been introduced in Section 3.2 and are nothing special for this thesis compared to the examples. For the most part, these include the integration of the Sprotty IoC container in *sprotty-vscode* and the web page. The code for the entire implementation can be found in a software repository on GitHub[1], except for the KIELER language server VS Code extension, with can be found on BitBucket[2].

This chapter instead focuses on aspects of the implementation that are not straightforward to implement with the Sprotty examples and KEITH as a template. The main focus is on the implementation of concepts that enable the integration of other diagram-related features in the Sprotty IoC container. This chapter discusses how the diagram core is able to rely on infrastructure that is different between platforms, and how arbitrary data can be stored in the IoC container. Furthermore, it introduces the sidebar API that has been designed to add a coherent UI to the diagram view. Lastly, it outlines the implementation of the CLI and VS Code diagram extension, as well as the motivation for moving the source code to GitHub and the new build process that has been enabled by this move.

5.1 Services

Services are a new concept that has been added to the diagram core. If a feature in the diagram core requires functionality that might be different between each platform, it should use a service. Common functionality that is different for each platform is infrastructure, such as the connection to a language server and storage capabilities. A service abstracts this infrastructure behind an interface, which is registered in the IoC container, so it can be used by other parts of the diagram core. This approach follows the ideas of the clean architecture that is introduced in Section 3.4. The diagram core is able to rely on these services without them being implemented in the core. Each platform that integrates the core has to create a concrete implementation for each service interface, and registers them in the IoC container. Otherwise, the container cannot be initialized, because it would not be able to inject all services into its modules.

The implementation for this thesis requires the following services, which are introduced

[1] http://github.com/kieler/klighd-vscode

[2] https://git.rtsys.informatik.uni-kiel.de/projects/KIELER/repos/keith/browse?at=refs%2Fheads%2Fcfr%2Fmaster

and motivated in the next subsections. While they cover all basic infrastructure that is currently required for a diagram visualization, they can be further extended and more services can be added if new requirements arise in the future.

5.1.1 Connection

Each Sprotty IoC container requires an implementation for a `ModelSource`. The `ModelSource` can also be implemented as a `DiagramServer` class that forwards requests for a diagram model to a server, such as a language server. To connect a `DiagramServer` with a server, Sprotty uses class inheritance to implement different connection types. For a standalone view, a `WebSocketDiagramServer` class can be used or further extended to connect to the server via a WebSocket. A VS Code extension that uses *sprotty-vscode* can integrate or extend the class `VscodeDiagramServer`, which forwards Sprotty actions from the Sprotty IoC container, which runs in a webview, to the extension.

If the diagram core uses the suggested approach of class inheritance to add a `DiagramServer` implementation to the IoC container, the implementation would have to be provided by the integrating platforms. The existing `DiagramServer` implementation in KEITH registers custom Sprotty actions, which should be handled by an instance of the class and forwarded to the actual server. This logic would have to be duplicated between each platform. Even though the only difference between these implementations is how they send and receive Sprotty actions to and from the server.

Instead of relying on inheritance, object composition can be used to implement one `DiagramServer` in the diagram core, which relies on a `Connection` service to communicate Sprotty actions with a language server. Therefore, the logic for the `DiagramServer` only has to be implemented once. The only part that has to be implemented by each platform is the `Connection` service that is genuinely different between them.

Apart from sending and listening for received Sprotty actions, the `Connection` service also supports sending LSP notification messages for other features in KEITH, which do not use Sprotty actions to communicate their data to the language server. Such features include diagram options and user preferences, which enable or disable the selection synchronization between the diagram view and text editor. Furthermore, the `Connection` service allows the diagram core to wait until a connection is established before it starts to send messages. This is achieved with an `onReady` method that returns a `Promise`, which is resolved by an implementation once the connection is established, or resolved immediately if the connection is already established.

5.1.2 Session Storage

KLighD supports images in a diagram model. The content of the images that are included in a model is sent separately as *base64* encoded data. In KEITH, received images are cached by the client for the duration of a user session. When the user closes the application, the cache is

emptied. Additional client-server communication allows the server to request a list of cached images to avoid sending all images for every subsequent diagram model update.

This functionality has been ported to the new implementation for this thesis as well. The logic is implemented as part of the `DiagramServer` in the diagram core and requires a `SessionStorage` service to cache received images. The interface of the service is identical to the standard `Storage` API[3], and is just a type alias in case custom modifications have to be made to the service interface in the future. The `Storage` API represents a key-value store with the ability to save a given value under a given key. A value for a key can be updated by setting a new value, and values can be retrieved or deleted for a given key. An implementation of the `SessionStorage` service should ensure that the storage is cleared after a user ends the session. Often, the web-native `sessionStorage` object should be sufficient and can be used as an implementation if available.

5.1.3 Persistence Storage

KEITH is able to persist changed diagram options between user sessions and informs the language server about the persisted options during the LSP initialization. For example, if a user changes the layout direction of a diagram from down to right, a new diagram will also use a right layout direction when the diagram view is opened again. To persist the changed options and notify the server about them, they are stored in a map collection, with the option identifier as the key and the changed option value as the value. Different option types, such as synthesis options and render options, are persisted as separate groups in a key-value store.

To long-term persist the different option types in a key-value store, a `PersistenceStorage` service has been defined for this implementation. The interface for the service could also follow the `Storage` API. However, to long-term persist data in VS Code, the diagram core has to send the data from the webview to the extension, where VS Code's `Memento` API[4] can be used to properly persist the data, which stores data based on keys as well. Furthermore, the data has to be persisted in the VS Code extension, because it has to be able to inform the language server about the persisted data during the LSP initialization, which happens before a webview with the diagram core is opened.

Since receiving all persisted data in the webview via message events is an asynchronous operation, data stored with the `PersistenceStorage` service can only be safely accessed asynchronously. Therefore, the synchronous `Storage` interface has to be slightly modified to allow asynchronous reads for implementations of the `PersistenceStorage` service.

The modification of the `getItem` method returns a `Promise` instead of the value, which should resolve to the value once an implementation of the `PersistenceStorage` service is able to provide the value.

Furthermore, writing a value to a key has been optimized in the service interface for asynchronous access. Often, the value for a key should not be replaced but rather updated based on the old value. This would require receiving the value first before it can be written.

[3] `https://developer.mozilla.org/en-US/docs/Web/API/Storage`

[4] `https://code.visualstudio.com/api/references/vscode-api#Memento`

To avoid the necessary read for an update, an implementation of the service is required to accept a callback function that returns the new value, instead of expecting the new value directly. When the `setItem` method is called with the key to change and an update callback, the callback function is called with the current value once this data can be accessed, and the return value of the callback is used as the new value for the key. This avoids explicitly reading the value for an update and makes code that calls the service a little more compact.

5.2 Managing State

There are multiple features that require a way to manage general data, which is not the diagram model, in the IoC container of the diagram core, so it can be accessed and modified during runtime. Such data includes diagram options provided by a KLighD synthesis, as well as user preferences and the current synthesis, and is often referred to as *state* in frontend development. Other parts of the IoC container, such as sidebar panels or diagram model views, have to be able to subscribe to the state that is managed by a registered class, so they can react to changes. Otherwise, the UI would not update when the state changes and would display an outdated state.

To achieve this, it is not required to use an extensive observer pattern that is typically found in object-oriented programming with multiple interfaces, which have to be implemented by both providers and observers. In JavaScript, functions are first-class citizens and can be passed as arguments. A simple but effective observer pattern in JavaScript uses callback functions that are registered in a state management class. Anytime the state changes, all subscribers are notified by calling their provided callback, which prompts them to react to the changes.

One approach to updating state that is stored in the IoC container is to call modifying methods on the storing class directly. While this object-oriented programming approach works fine, state management can be integrated into the IoC container with the Flux-like unidirectional event/data flow that is already established by Sprotty. Figure 5.1 visualizes how Sprotty's architecture has been extended to manage state in the IoC container with multiple `ActionHandlers`. The concept for such `ActionHandlers` has been abstracted into `Registries`, which are highlighted green in Figure 5.1. An abstract base class ensures that every inheriting class implements the `IActionHandler` interface. Furthermore, it abstracts the pattern used for subscriptions that is described above. This allows a `Registry` that extends the abstract class to not be concerned with managing subscription callbacks. It only has to implement access to the managed state and a `handle` method to handle received Sprotty actions.

To modify the state in a `Registry`, a custom Sprotty action should be dispatched in the container. If a `Registry` is registered to handle the action, the `ActionDispatcher` will call the `Registry` with the action and its state can be updated. After an update, a `Registry` should call its `notifyListeners` method, which is provided by the abstract base class and informs all subscribers about the state change.

The biggest advantage of extending Sprotty's architecture is that actions can be handled by multiple `ActionHandlers`. A state update does not map directly to a method call. Instead, the

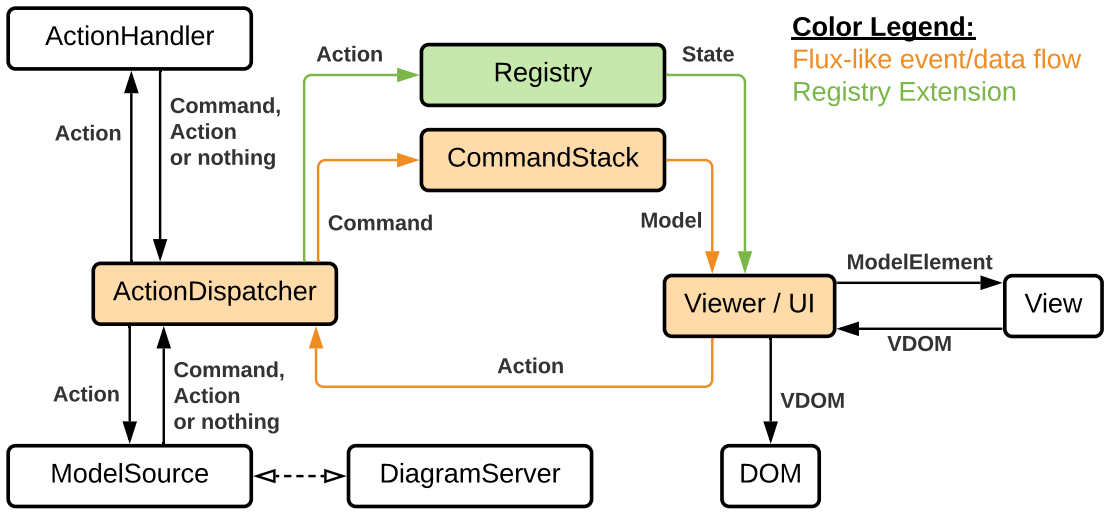


Figure 5.1. Extension for the Sprotty architecture to use its Flux-like flow for state management.

update that is supposed to happen is described by a custom Sprotty action. For example, an update to synthesis options is handled by the `OptionsRegistry` and by an `OptionsPersistence` action handler, which calls the `PersistenceStorage` service to persist changes to all kinds of diagram options. Furthermore, actions can be dispatched from anywhere through the `ActionDispatcher` provided by Sprotty, even from outside the diagram core by an integrating platform. An application is able to update user preferences, which are stored by the `PreferencesRegistry` in the diagram core, by dispatching a `SetPreferencesAction` without knowing anything about the `PreferencesRegistry`.

5.3 Sidebar API

Services add functionality to the diagram core that depends on the capabilities of integrating platforms. Registries add the ability to store and manage state in the diagram core IoC container. These new concepts that have been added to the core can be used to develop features in the diagram core. However, a user would currently not be able to interact with a feature, since the ability to create an additional UI is still missing. Section 4.6 discusses the concepts for a UI in the diagram core and introduces the sidebar as a concept to create a coherent UI. This section will introduce the sidebar in more detail and outlines how developers are able to create a sidebar panel for their features.

For the sidebar, an API has been designed that allows the addition of sidebar panels by registering them in the diagram core IoC container, so they can be injected and displayed

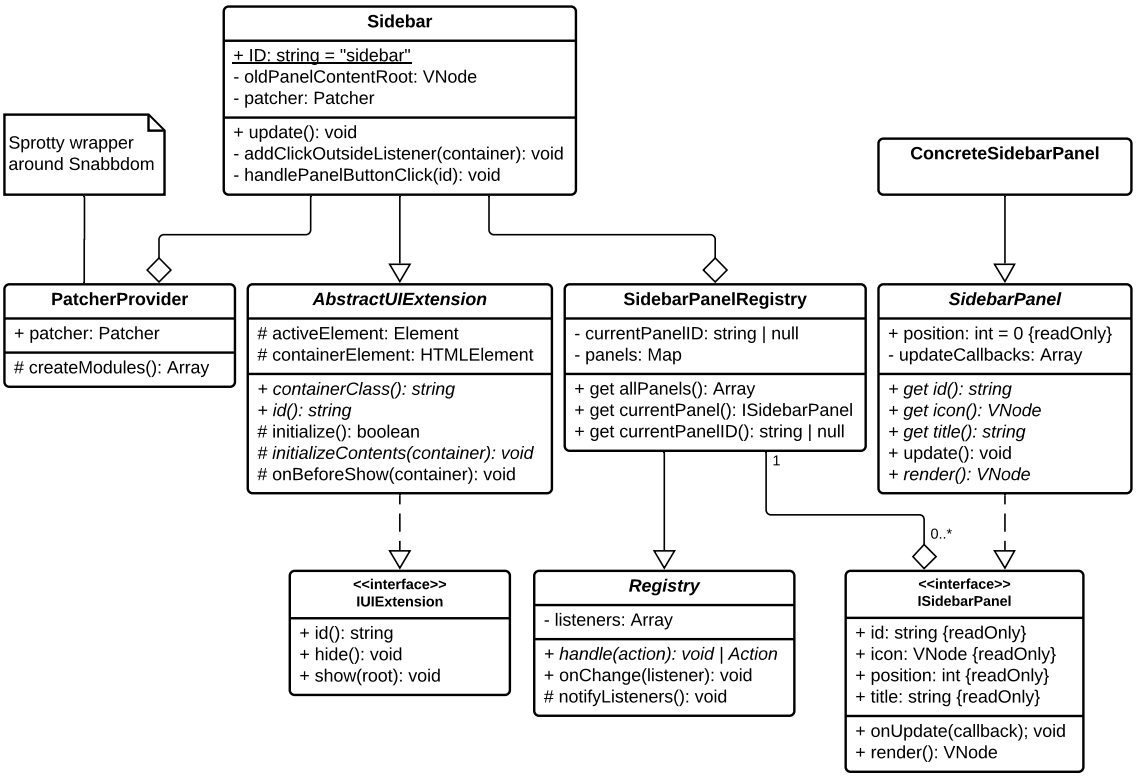


Figure 5.2. Hierarchy of the classes and interfaces that form the sidebar API.

by the sidebar. Furthermore, a sidebar panel has to be able to re-render when the state that influences the displayed content changes.

The classes that form the sidebar API and their relations are visualized in Figure 5.2. As mentioned in Section 4.6, the sidebar is built on Sprotty's undocumented *UI extensions* feature. A UI extension has to implement the `IUIExtension` interface and has to be bound to the UI extension symbol in the IoC container. The symbol is used to access all `IUIExtension` implementations, so their visibility state can be managed by Sprotty. This multi-dependency injection mechanism has also been used for the new sidebar API to access all registered sidebar panels in the IoC container. To create an `IUIExtension`, an implementation should inherit from the `AbstractUIExtension` class. It implements most boilerplate code for mounting the `IUIExtension` in the DOM, and shifts the focus of inheriting classes to providing the UI content. The sidebar API contains a `Sidebar` class that extends the `AbstractUIExtension` class and thus implements the `IUIExtension` interface. It is responsible for rendering and updating the sidebar that consists of registered sidebar panels.

Existing features in Sprotty that extend the `AbstractUIExtension` class create the UI extension content imperatively with the native DOM API. However, based on experience, imperatively creating a DOM structure that updates when dependent state changes, quickly

becomes tedious and unpleasant to work with. Therefore, most JavaScript UI libraries rely on a declarative approach to create the UI based on state with JSX or templates. Furthermore, Sprotty depends on Snabbdom to create views for the diagram model declaratively with JSX. Sidebar panels should also be able to define their content declaratively to create a more pleasant API. To achieve this, the `Sidebar` class depends on a `PatcherProvider` class provided by Sprotty, to create a virtual DOM root with Snabbdom's patch function, which is rendered in the DOM. Upon render updates, the created virtual DOM root is patched with virtual nodes that are created by sidebar panels.

A panel that should be rendered by the `Sidebar` has to implement the `ISidebarPanel` interface. The interface requires a panel to define general content, such as the panel title and trigger icon, as well as a `render` method that returns the panel content as a virtual node, which is created with Snabbdom. Furthermore, a subscription mechanism, which is similar to Section 5.2, has to be implemented by a panel to inform the `Sidebar` when it should re-render. Instead of calling an update method on the `Sidebar` directly from within a panel, the `Sidebar` subscribes to every panel with a callback function, which can be called to request a render update. This inversion of control allows the `Sidebar` to ignore a render-update request in case the `Sidebar` is closed, or the requestor is not the currently shown panel. The implementation of the subscription mechanism is hidden from sidebar panels by the abstract `SidebarPanel` class. Instead of implementing the `ISidebarPanel` interface directly, panels should inherit from this abstract class. It exposes an `update` method, which must be called to request a render update from the subscribed `Sidebar`.

Instances of `ISidebarPanel` that are registered in the IoC container are managed in the `SidebarPanelRegistry` class. It extends the abstract `Registry` class presented in Section 5.2 to inherit a subscription mechanism and the `IActionHandler` interface. Opening and closing the sidebar or switching between panels is realized with actions that are dispatched by the UI and handled by the `SidebarPanelRegistry`. It updates its reference to the current sidebar panel accordingly and informs all subscribers about the state change. Besides subscribing to each panel provided by `SidebarPanelRegistry`, the `Sidebar` subscribes to the `SidebarPanelRegistry` as well, to keep the sidebar UI synchronized with the opened/closed state and currently open panel.

5.3.1 Creating a Sidebar Panel

The sidebar API hides the complexity of showing, hiding, and switching different sidebar panels from features that want to contribute an additional UI to the diagram view. Furthermore, all boilerplate that is required to update the UI after state changes is hidden by the `Registry` and `SidebarPanel` class. The code that is still required is reduced to a minimum. A feature that adds a sidebar panel only has to provide its content, which is done declaratively with JSX, as well as an identifier, a title, and an icon for the sidebar panel. Apart from the identifier, all of these directly contribute to the UI shown to the user.

A small example of a sidebar panel implementation is shown in Listing 5.1. The panel displays a counter, which can be incremented and decremented between zero and ten. After

```tsx
export class CounterPanel extends SidebarPanel {
   private count = 0;

   get id(): string { return "counter-panel"; }
   get title(): string { return "Counter"; }
   get icon(): VNode { return <i attrs={{ "data-feather": "percent" }}></i>; }

   render(): VNode {
      return (
         <div>
            <p>Current count: {this.count}</p>
            <button on-click={() => this.changeCount(1)} disabled={this.count >= 10}>
               Increment
            </button>
            <button on-click={() => this.changeCount(-1)} disabled={this.count <= 0}>
               Decrement
            </button>
         </div>
      );
   }

   private changeCount(by: number): void {
      this.count += by;
      // Requests a re-render from the Sidebar
      this.update();
   }
}
```

Listing 5.1. Example of a `SidebarPanel` that displays an interactive counter.

the current count is changed, the panel requests an update from the `Sidebar`, to display the new count in the UI. The code for more complex panels scales with the UI that should be shown to a user. In this basic example, the state is managed inside the panel. Features with a more complex state should use a `Registry` to manage the state. If a `Registry` is used, the `SidebarPanel` should subscribe to the `Registry` with a callback that calls `update` on the `SidebarPanel`, to reflect `Registry` changes in the UI. An example[5] of this approach can be found in the source code for this thesis. The created sidebar panel provides a UI for state that is stored in multiple registries and subscribes to all of them, after an instance of the panel is created by the IoC container.

Sidebar panels, registries, and services are the foundation upon which the other diagram features that are identified in Section 4.1.1 have been built in the diagram core. They abstract most non-feature-specific logic, so the code for each feature can be written in a straightforward way. For the most part, requirements for a feature can be reduced to storing state in registries,

[5] https://github.com/kieler/klighd-vscode/blob/main/packages/klighd-core/src/options/general-panel.tsx#L59

communicating actions to and from a language server, and displaying an interactive UI, such as diagram options, in the sidebar.

5.4 Implementing the CLI Application

The standalone diagram view that is opened by a CLI has to wrap the diagram core in a web page. This requires implementations for all services. An instance of the `SessionStorage` service is directly provided with the web-native `sessionStorage` object, which implements the `Storage` interface. The implementation for the `PersistenceStorage` service is a lightweight wrapper around the web-native `localStorage` object to support the small deviations from the `Storage` interface. To implement the `Connection` service, a WebSocket connection is established with the web server, which forwards messages to the language server. The connection is wrapped by the *vscode-ws-jsonrpc* library to abstract and support the communication of LSP-conform messages, which despite its name functions in a browser as well.

With an implementation for all services, the web page is able to create the diagram core IoC container and registers instances for all services in the container. After the LSP-initialization messages are exchanged and the language server is notified about the file that will be visualized, the diagram core is able to dispatch a model request action to start the visualization. In case an error occurs on the connection or during the initialization process, a small popup UI has been added to the standalone view, which is used to inform the user about problems.

The web server that forwards LSP messages has been created with Fastify, which has been introduced in Section 2.9. For an incoming WebSocket connection, the server spawns a language server process and establishes message forwarding between the connection and process. Furthermore, the web server is used to serve the standalone diagram view as a static website, which is created by bundling the sources into HTML, JavaScript, and CSS. In the future, the server instance can be extended with more functionality by adding additional Fastify plugins to the server instance.

The CLI application that is used to wrap the web server for users has been created with Commander, which has been introduced in Section 2.10. It allows a user to open the standalone view for a file from the command line and provides usage information in form of a help text. By using Commander, argument parsing and help text generation is handled by the library, and did not have to be implemented explicitly. To simplify the distribution and installation of the CLI for users, it is packaged into a self-contained executable. For each major operating system, an executable is packaged that contains the CLI script, web server, web assets, and a Node.js runtime. Therefore, a user can download the single executable file and does not have to install a specific Node.js version just to use the CLI application.

Both the CLI and web server have been implemented with Node.js libraries. The alternative would have been Java applications to use the same runtime as the language server. However, in that situation they would have been part of the KLighD repository and not the web-based repository. Since the web page for the standalone diagram view has to be able to import

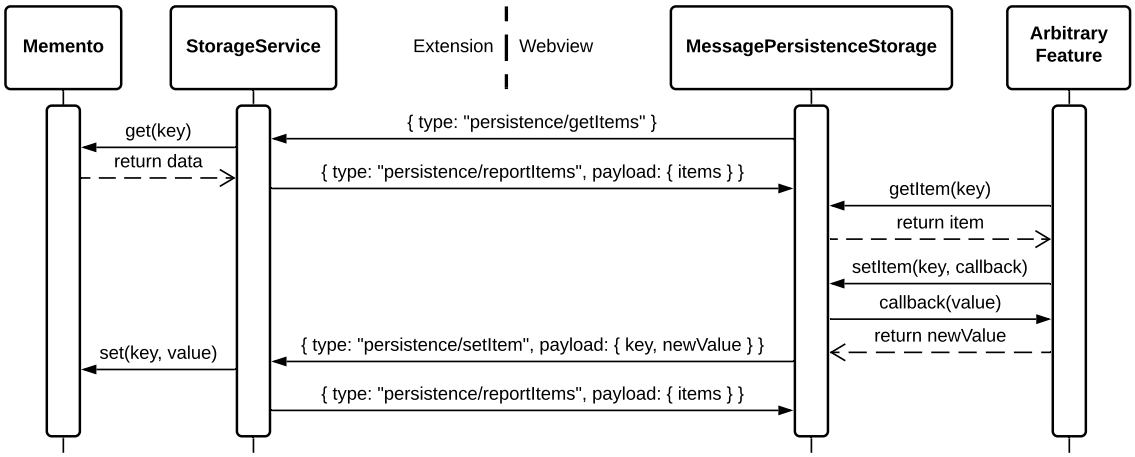


Figure 5.3. Sequence diagram visualizing the communication for persisting data from the webview in VS Code.

the diagram core and is bundled with a Node.js tool-chain, it is part of the web-based repository. Creating the web server and CLI in Java, would create a cyclic dependency between the repositories, where the web-based repository depends on the language server for development, and the KLighD repository depends on the web assets to build the web server. To avoid this cyclic dependency between the repositories, the CLI and web server have been developed with Node.js, so only the web-based repository has a dependency on the language server for development.

5.5 Implementing the VS Code Diagram Extension

As it has been discussed in Section 4.2.1, the VS Code diagram extension uses *sprotty-vscode* to integrate the diagram core in VS Code. Since the diagram core is executed in a webview, the required services have to be implemented as part of the webview as well. Similar to the standalone diagram view, an instance of the `SessionStorage` service is provided by the web-native `sessionStorage` object. Therefore, images that are stored by the diagram core are cached until the webview is closed. To properly implement the other services, they have to access APIs that are running in the extension code.

As outlined in Section 5.1.3, the `Memento` API has to be used to long-term persist data in VS Code. Therefore, the `PersistenceStorage` service is implemented in two parts. An example communication between the two parts is visualized in Figure 5.3. The first part, which is the `MessagePersistenceStorage` class, runs in the webview where it implements the `PersistenceStorage` interface and is usable by the diagram core. The implemented methods forward changes via messages to the second part in the extension. The second part, which is the `StorageService` class, receives the messages and converts them to operations that are

applied to the data stored with the `Memento` API. The changed data is reported back as a message to the `MessagePersistenceStorage`, which updates its data cache accordingly.

The implementation of the `Connection` service uses messages as well. It sends notifications for the language server as messages to the extension, where *sprotty-vscode* forwards them to the language server. In return, *sprotty-vscode* sends notifications that are received from the language server as messages to the webview, where they are handled by the `Connection` service implementation and forwarded to subscribers.

Originally, the webview abstraction and message forwarding in *sprotty-vscode* only handle messages that are Sprotty action messages, which are sent to the language server as *diagram/accept* LSP notifications. However, the `Connection` service sends other LSP notification types as well, which were defined for KEITH to communicate changes of the diagram options. Furthermore, the webview abstraction has to be able to send arbitrary messages to the webview as well to support the custom messages for the `PersistenceStorage` service implementation.

Therefore, the implementation in *sprotty-vscode* has been partly overwritten to enable the communication of custom messages and forwarding of other LSP notification types. Since *sprotty-vscode* uses a class-based design for its implementation, the modifications have been implemented with custom subclasses that overwrite the behavior of their parent classes.

Such modifications to *sprotty-vscode* have also been applied for an editor syncing feature that is available in KEITH. When a user changes the active editor, the diagram view changes as well to show the diagram for the content of the currently active editor. The user can disable this feature, which causes the diagram view to not update when the active editor changes. This behavior is only implemented in the VS Code diagram extension and not in the diagram core, because it requires access to the extension and makes no sense in the standalone diagram view, where the user cannot change between multiple source files. This feature is not supported by *sprotty-vscode*, so it has been implemented by overwriting a `reloadContent` method on its webview abstraction, which is called whenever the active editor changes.

5.6 Publishing the Applications

Many projects host their open-source software on the development platform GitHub[6]. This is also the case for KLighD, which is part of the KIELER organization on GitHub. During the development for this thesis, the diagram view on the client has been decoupled from the KIELER language server. It is no longer dependent on this specific language server and can be used by other projects as well. While working on the implementation for this thesis, we[7] decided to move the software, which is able to visualize KLighD-synthesized models, from BitBucket to GitHub next to the KLighD repository. GitHub is widely accepted in the open-source community for hosting source code, which further increases the visibility and discoverability of the client code.

[6] `https://github.com`

[7] "We" refers to the author and other members of the Real-Time and Embedded Systems Group.

Apart from the increased discoverability, this also enables an automated publishing process for the applications with GitHub Actions[8]. Actions are GitHub's implementation of a continuous integration service that is deeply integrated in its platform. They can be configured to trigger on various events that occur in a GitHub repository, such as creating an issue, opening a pull request, pushing code to a branch, publishing a release, and many more. Actions are configured in *workflow* files that define triggers and jobs that are executed for the workflow.

The new GitHub repository[9], which contains most of the software developed for this thesis, has been fitted with GitHub Actions automations as well. The automations that are currently applied consist of three workflows:

- When a pull request is opened against the main branch or code is pushed to the main branch, a general continuous integration workflow is executed that confirms that the code can be build and analyzes the code for problems with ESLint[10].
- When code is pushed to the main branch, a packaging workflow, which is comparable to a nightly build, builds the applications in the codebase and attaches them as artifacts, which can be downloaded, to the workflow run.
- When a new version is published in the repository as a GitHub release, a publishing workflow builds the applications and publishes the VS Code diagram extension to the Visual Studio Marketplace and the Open VSX registry, and the extension bundle is attached to the release notes in case someone wants to install the extension from a local source. The self-contained CLI application is attached to the release notes for download as well.

These automations ensure that the code, which is contributed via pull requests in the future, does not break the application builds before it is merged on the main branch, and are common for many projects that use GitHub Actions for their continuous integration. Distributing a new version to users has become as simple as publishing a new release[11] with informative release notes. Ensuring that the applications are actually distributed to all relevant places is taken care of by an automation.

During the migration to GitHub, the resulting Node.js packages have been renamed to closer represent their connection to KLighD. The diagram core and interactive module are now called *klighd-core* and *klighd-interactive*. The *klighd-cli* package contains the CLI application, standalone view, and web server. The VS Code diagram extension is now called *klighd-vscode*.

The *klighd-cli* and *klighd-vscode* packages have already been published with the automated approach. The VS Code language extension that integrates the KIELER language server is currently not published and can only be installed from a local bundle. Previously existing features that are not directly part of the diagram visualization, such as simulation and compilation, should be added first and are out of scope for this thesis.

[8] https://docs.github.com/en/actions

[9] https://github.com/kieler/klighd-vscode

[10] https://eslint.org

[11] https://github.com/kieler/klighd-vscode/releases

Chapter 6

Evaluation

The main goal of this thesis has been to find a solution and implementation that ports the existing diagram view in KEITH to VS Code and a standalone view. While the performance of the diagram view is generally important for a pleasant user experience, it has not been a pressing criterion for this work. Instead, this work has been focused on scalability to other projects and platforms.

This chapter discusses the expected performance for the new implementation and compares available features of the new implementation with KEITH. Lastly, it outlines the implementation effort for new platforms and projects.

6.1 A Word About Performance

The performance of a diagram view that is separated between a client and server is influenced by multiple factors. For the most part, the previous performance of the diagram view in KEITH has remained largely the same for the new target platforms. However, while the performance regarding the diagram plays a big role, the render speed and responsiveness of the new sidebar also affect the user experience.

6.1.1 Diagram Visualization Performance

A big factor is the time that is required to communicate the diagram model between the client and server, and the time required to render the diagram model on the screen. Since both client and server are located on the same computer in this implementation, the communication time is mainly affected by the size of exchanged messages. To render the received diagram model, Sprotty translates it to SVG, which is rendered on the screen by the browser engine. Therefore, the rendering performance is mainly affected by the SVG rendering speed of the browser engine.

This work has not made any changes to the messages that are communicated between the client and server. The message payload and order has remained the same. Furthermore, it does not change the translation of the diagram model to SVG. Therefore, the timings evaluated by Rentz [Ren18] still hold for this implementation and are still relevant.

The standalone diagram view is often used in a full-screen browser window, thus the browser has to render a bigger area compared to VS Code and KEITH, where the view is opened next to the model source. Therefore, it can feel noticeably slower compared to VS Code or the previous implementation in KEITH. Rentz introduced a new approach for rendering shadows

in the diagram view [Ren18]. While these shadows look smoother and more realistic, they are more intensive to render for the browser. Paired with the often bigger rendering area in the standalone view, they are the main cause for rendering-performance decreases. Disabling the shadows in the diagram options and resizing the browser window to a smaller area can help to increase the rendering performance in the standalone view, in case panning the diagram feels slow.

6.1.2 Sidebar Rendering Performance

Diagram options that require a long time to show their new state after a user integration may be confusing as they do not provide instant feedback to the user. The time required to show an update is influenced by the render speed of the UI, as well as the time that is required to communicate a change to the language server and back, which is the source of truth for the diagram options. To avoid waiting for the server communication, the implementation of the diagram-options sidebar panel uses optimistic updates. The stored state is changed locally, causing the sidebar panel to re-render and instantly reflect the change, while the change is communicated to the language server. Afterwards, the optimistic update is replaced with the newly received options and the sidebar panel is updated once again to show options from the source of truth. If nothing goes wrong, the optimistic update already reflects the new options. Paired with quick re-rendering times of the sidebar, optimistic updates are able to provide instant feedback to users.

Throughout the development and testing of the sidebar, the time that is required to re-render the sidebar has been continuously measured. While a single option update re-evaluates the whole sidebar content, Snabbdom only updates the single option in the DOM. Switching between panels or opening the sidebar requires Snabbdom to update the whole DOM structure for the sidebar content. Opening and switching between a sidebar panel require between 3 to 10ms to update. Updating a single option usually finishes within 2ms and rarely spikes up to 5ms. These numbers measure the time that is required to update the DOM, and do not include the browser paint times. If a target of 60 frames per second is considered, the browser has to finalize a new frame every 16ms. The measured timings for a sidebar update fit within this frame budget, thus the sidebar does not prevent the browser from reaching this target. While these timings can vary for different systems and system load, they indicate and quantify the general responsiveness of the new sidebar UI.

6.1.3 Startup Performance

The client and server have to establish a connection and exchange the LSP-initialization messages. In VS Code and KEITH, a user waits for the IDE to finish its startup, which also includes the initialization of the LSP communication. However, in the standalone view the user would see a blank page while the LSP communication is not initialized. During the development, the time that is required for the LSP initialization has been measured as well. The WebSocket connection between the client and web server is established after 40 to 80ms.

Afterwards, the client has to wait around 2 to 5s for the KIELER language server to answer the LSP *initialize* request. To show some feedback until the LSP communication is initialized, a loading spinner has been added to the standalone view that informs the user about the initialization step.

6.2 Feature Comparison with KEITH

The new implementation for VS Code and the standalone view contains all diagram-related features that exist in KEITH and have been identified in Section 4.1.1. Throughout this thesis, the importance and necessity of moving all diagram-related features into one diagram core to avoid duplication has been discussed. While these features still exist, they had to move to a different location in the UI. This section outlines the changes made to the features and their new location in the UI.

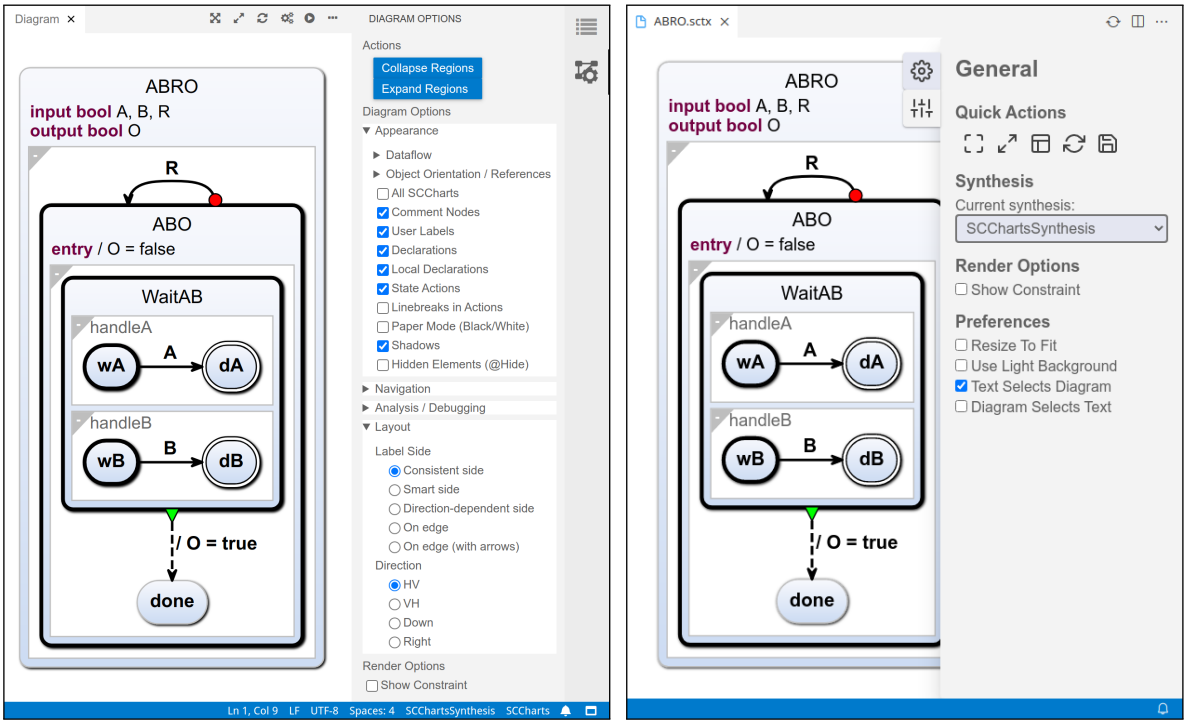
Some additional features have been added that were previously only available in the KIELER IDE. Added features from KIELER include the ability to clear persisted diagram options. The feature is implemented as a command in VS Code that can be toggled from the command palette, and as a CLI option in the standalone view that can be set when a diagram view is opened. Furthermore, the ability to refresh the diagram layout has been added. The Sprotty action for this feature was already available in KEITH but was not accessible for a user.

6.2.1 Changes to the Features

Figure 6.1 compares the diagram view in KEITH with the new diagram view in VS Code. The diagram visualization in the standalone view is identical to the content of the VS Code webview. Since Theia and VS Code are very similar, it is more meaningful to compare the implementation for both IDEs, rather than the standalone view with KEITH. The sidebar panel for general items, which is shown in Figure 6.1b, contains more features that are important for this comparison than the diagram options panel, which can otherwise be seen in Figure 4.6b. On that note, the biggest change is the location of diagram options. They have moved from a Theia widget next to the diagram view, into a sidebar panel that opens on top of the diagram. Thus, one implementation can be used for all platforms. Otherwise, they are the same with slight visual modifications, which clarify their hierarchy to improve their readability.

Most of the features that were previously located in the Theia UI in KEITH have been moved into the diagram view. In Figure 6.1a, quick actions to center, refresh, and fit the diagram to the screen are located on top of the webview. Information about the current synthesis with the ability to change the synthesis is located in the status bar.

In the new implementation in Figure 6.1b, these features have been moved into a general sidebar panel. Therefore, they are accessible in the standalone diagram view as well. The general sidebar panel contains a mix of features that do not require their own sidebar panel. The quick actions have been moved to the top of the general panel. New quick actions have been added to refresh the layout and export the diagram as SVG, since they were already



(a) Diagram view in KEITH **(b)** Diagram view in VS Code

Figure 6.1. Side by side comparison between the diagram view in KEITH and in VS Code.

available as Sprotty actions in the source code. The synthesis picker is now located underneath the quick actions and instead of an IDE-native selection popup, a select input is used to change the synthesis, because it is generally available in browser-based environments.

The editor settings in KEITH contain user preferences that allow the user to enable and disable synchronization of the selected element between the editor and diagram view. Changes to these preferences affect the diagram view immediately.

The editor settings in VS Code still contain the previously available user preferences. They have been extended with a preference to force a light background for the diagram view, and a toggle, which disables resizing of the diagram to fit the viewport after a diagram update, has been moved from the top of the webview in KEITH to the preferences. These preferences have been added to the general sidebar panel to make them accessible in the standalone view. Therefore, the preferences in the editor settings have been changed to initial preferences that are applied to new diagram views. To change them for an open diagram view, the user must change the preferences in the sidebar. Changing the preferences in the editor settings to initial preferences prevents them from getting out of sync with the preferences in the panel during usage.

Finally, KEITH has the ability to disable syncing with the editor, via a preference located in the editor toolbar overflow menu. Normally the diagram view changes its content when a

user changes the active editor. If it is not able to generate a diagram for the content of the editor, a placeholder is shown. When syncing is disabled, the diagram view continues to show the same diagram. Furthermore, it is not possible to explicitly update the diagram view. A user has to enable the preference first before the diagram can be changed.

This is the only feature that did not move into the diagram view, because it is only available in VS Code. Still, this feature behaves slightly differently in the new implementation for VS Code. If the content of an editor is not supported, it will not update the diagram view. Furthermore, the option to toggle this preference has moved to the editor toolbar, so it requires one click to toggle instead of two. When syncing is disabled, the user is now able to update the diagram view explicitly by executing the *Open in Diagram* command. This enables the user to visualize a new diagram model without having to re-enable the *Sync with Editor* preference.

With these changes, it has been possible to share all features between the VS Code extension and standalone view. If they were implemented in VS Code like they are implemented in KEITH, they would have been implemented twice. This would be against the initial requirement to avoid code duplication, which has motivated all of these changes. Supporting all platforms would have been a bigger effort and would require more maintenance work in the future.

6.2.2 Missing Behavior

The extension for VS Code feels nearly as productive as the KEITH implementation. While most diagram-related features have moved to another location, they still exist and are accessible for a user. However, one behavior could not be implemented in VS Code: When a user closes and opens KEITH again, the diagram view remains open.

The VS Code extension has not been able to implement this feature, because it is blocked by *sprotty-vscode*. To persist a diagram view, which is a webview, between reloads, an extension has to implement a `WebviewPanelSerializer` [Mic21f]. The serializer receives a webview instance and has to set its content. Further, the received instance must be stored and used to access and attach event listeners to the recreated webview. This clashes with the *sprotty-vscode* glue code, because it would be unable to attach event listeners to the received webview. Therefore, it would not be able to attach message listeners and properly manage the webview lifecycle. The ability to restore the webview has to be implemented in the glue code, to ensure that everything works together as expected.

6.3 Implementation Effort for Other Projects

The original goal and topic that was proposed for this thesis has been to port the diagram view in KEITH to VS Code and a standalone view, so the KIELER project is usable on these platforms as well. The only criterion has been that it should extend the original implementation and function with the KIELER language server.

While working on this thesis, the potential to create a general diagram view has been identified, which can be used with any language server that supports KLighD syntheses.

Therefore, the goal of this thesis has been extended to make the new diagram views usable for other projects as well. Throughout this thesis, it has been explained how the implementation is structured to achieve this goal.

This work has created integrations with the diagram view for the KIELER language server, which prove that the idea of separating the diagram views from specific a language server generally works. Rentz has been able to prove that the new implementation can indeed be used by other projects as well. He created a VS Code extension[1] that uses the *klighd-vscode* extension to visualize OSGi™ projects [RDH20]. The created language server has also been used successfully with the diagram CLI, which proves that it is possible to use the new CLI for different projects.

This section highlights the work that is required to implement a diagram view for another project. The new implementation provides three possibilities to extend the diagram view, two of them are outlined in this section. The last possibility is to extend the web server created for the standalone view. Since the web server provides an already fully working diagram view, any extension depends on the specific requirements, and cannot be generalized as required work for other projects.

6.3.1 Developing a VS Code Extension

Integrating the new *klighd-vscode* extension to create a VS Code language extension with diagram support requires only a few steps. They are documented in the overview[2] of the *klighd-vscode* extension as well. These steps assume that a language server with KLighD synthesis support has already been created.

Listing 6.1 contains a minimal implementation of the `activate` function for an extension, which is required to use *klighd-vscode*. It has to create a `LanguageClient` to integrate its language server into VS Code. The *klighd-vscode* extension has to be informed about the language server to activate the extension and to provide a connection to a KLighD synthesis. The `LanguageClient` instance is provided by executing the *klighd-vscode.setLanguageClient* command with the instance as the first argument. The second argument is an array of file endings that are supported by the language server. It ensures that the diagram view in *klighd-vscode* only updates for active editors that can actually be visualized. At last, the extension has to initialize the language server connection by calling `start`.

The command call to *klighd-vscode* returns an identifier that has to be passed to further command calls to the extension. At the moment, extensions are able to register an action handler through an additional command execution, which can intercept Sprotty actions before they are sent to the language server.

To ensure that *klighd-vscode* is installed, a dependency to the extension should be declared in the extension configuration file. Furthermore, an extension should declare contributions to allow users to open a diagram for its supported languages. Such contributions are declared in the extension configuration file as well.

[1] `https://marketplace.visualstudio.com/items?itemName=kieler.osgiviz`

[2] `https://marketplace.visualstudio.com/items?itemName=kieler.klighd-vscode`

```
export async function activate(context: vscode.ExtensionContext) {
   const clientOptions = { /* ... */ }
   const serverOptions = { /* ... */ }

   const lsClient = new LanguageClient("Language Server", serverOptions, clientOptions);

   const refId = await vscode.commands.executeCommand(
      "klighd-vscode.setLanguageClient",
      lsClient,
      ["<lang>"]
   );

   console.debug("Starting Language Server...");
   lsClient.start();
}
```

Listing 6.1. Minimal extension code required for a working *klighd-vscode* integration.

6.3.2 Integrating the Core Package

If parts of the diagram view should be replaced with custom implementations or a web-based application should be supported that cannot be supported with the VS Code extension and standalone view, the *klighd-core* package can be used as a starting point. However, the package has not been published to npm yet, which has to be done before it can be used by other projects. It provides the most flexibility but also requires the most effort to create a working implementation. An integration has to implement all services defined in Section 5.1, and potentially more while development continuous. The steps required to integrate the core package are documented in the package overview[3] as well.

The service implementations will be different for each application and project. Therefore, Listing 6.2 assumes that they are already implemented and are ready to be used. It highlights the code that is required to integrate the diagram core container. An implementation has to import and bundle the styles that are provided for the diagram view. Further, it has to create the diagram IoC container and has to provide an identifier for an element in the DOM that is used to mount the diagram view. Afterwards, the implemented services have to be registered in the container and a model request has to be dispatched to start the diagram visualization.

As it can be observed in the listings, both usage possibilities do not require much code to bootstrap a diagram visualization. The *klighd-vscode* extension hides all logic for other extensions that would otherwise be required to manage the diagram webview and message forwarding between the language client and webview. The *klighd-core* package contains all logic that is required to visualize KLighD-synthesized diagram models. Details about the diagram visualization are hidden behind API interfaces for other applications that build upon

[3] https://github.com/kieler/klighd-vscode/blob/main/packages/klighd-core/README.md

```
import "@kieler/klighd-core/styles/main.css";

import { createKlighdDiagramContainer, requestModel,
   getActionDispatcher, bindServices } from "@kieler/klighd-core";
import { ConnectionImpl } from "./services/connection";
import { PersistenceStorageImpl } from "./services/persistence";
import { SessionStorageImpl } from "./services/session";

async function init(sourceUri: string) {
   const connection = new ConnectionImpl();
   const persistenceStorage = new PersistenceStorageImpl();
   const sessionStorage = new SessionStorageImpl();

   const diagramContainer = createKlighdDiagramContainer("container-id");
   bindServices(diagramContainer, { connection, sessionStorage, persistenceStorage });

   const actionDispatcher = getActionDispatcher(diagramContainer);
   await requestModel(actionDispatcher, sourceUri);
}
```

Listing 6.2. Minimal code required to integrate the diagram core.

the core package. However, if required, they are still able to access and replace everything through the exposed IoC container.

6.3.3 Embedding the Standalone Diagram View

If the diagram visualization of the CLI is sufficient to fulfill given requirements, the new CLI can be used directly without any changes. Furthermore, it can be used to embed a visualization into other web pages, such as web-based documentation. To simplify this use case, an additional command has been added to the CLI, which starts the web server without constructing a URL for a given file and opening the diagram view. Any web page can embed the diagram view with an iframe element, which specifies a URL that points to the started web server and includes the *source* parameter that is required by the diagram view, as explained in Section 4.4.2. Diagram files that are part of a locally hosted documentation can therefore be visualized interactively in the documentation. An example of multiple diagram views that are embedded in a local documentation can be found in the source code repository[4].

While the CLI can be used for visualizations in locally hosted documentation, it does not replace the hosted service that has been described in Section 4.4.1. Such service provides URLs that can be embedded into online documentation and function for anyone that views the documentation. Furthermore, the CLI cannot be used to share a URL to a visualization with other people, which is possible with a hosted service.

[4] https://github.com/kieler/klighd-vscode/tree/main/examples/local-documentation

Chapter 7

Conclusion

This chapter summarizes the work for this thesis and outlines future work that extends the implemented applications with more functionality.

7.1 Summary

For this thesis, the diagram visualization in KEITH that has been previously developed by Rentz [Ren18] has been ported to VS Code and a standalone diagram view for a browser. While working on this thesis, the potential to extract the KIELER language server and to generalize the resulting applications for multiple language servers has been identified and applied to the implementation. The ability to support different language servers has been tested with a KIELER language server integration that has been created for this thesis and an integration for the OSGiViz project created by Rentz.

For VS Code, two extensions have been developed. The first is called *klighd-vscode* and adds general visualization support for KLighD-synthesized diagrams. The second extension, which is currently called *keith-vscode*, integrates the KIELER language server and depends on *klighd-vscode* to visualize diagrams for KIELER model languages. The standalone view has been implemented as a CLI application that starts a web server and opens the standalone view for a given file. The CLI does not have a hard dependency on any language sever and has been successfully used with the KIELER and OSGiViz language servers. GitHub Actions have been created to publish the CLI and *klighd-vscode* extension[1] automatically.

To avoid code duplication between *klighd-vscode* and the standalone view, the Sprotty IoC container developed for KEITH, which translates diagram models to SVG, has been extended to include the implementation for most diagram-related features. To properly implement the features, the container has been extended with the concept of services, registries, and a sidebar UI to create a foundational infrastructure in the IoC container for communicating, storing, and displaying application data that is not the diagram model. The sidebar has been designed to support a UI for different content and features in a coherent design through sidebar panels, which allows the addition of panels in a straightforward way.

The implementation for this thesis has created applications that can be used with other language servers, and thus can be integrated by more projects than just the KIELER project without much effort. The created core package contains almost all features, which simplifies the support for future applications that are web-based or support web content. The required

[1] https://github.com/kieler/klighd-vscode/releases

effort for an integration has been outlined in Section 6.3. This focus on reusability has opened the visualization of KLighD-synthesized models to more projects and platforms and thus to many more potential users.

7.2 Future Work

While the applications created for this thesis are already usable by users, there is still room to further improve them and to add new features.

Section 6.2.2 discusses that the diagram view is currently not restored in VS Code and has to be reopened by the user every time VS Code is opened. Restoring the webview automatically after VS Code is opened might improve the user experience, and is equivalent to the behavior of other editor tabs, such as a markdown preview, which is automatically restored as well. However, this feature should probably be implemented in *sprotty-vscode* to ensure that the library has access to the restored webview and is able to correctly initialize its class instances.

During the implementation of *klighd-vscode*, a bug has been found in *sprotty-vscode*. If a user opens a diagram, switches to another, and closes the webview, the reference to the disposed webview panel is not properly cleaned up. When the user tries to open the first diagram again, *sprotty-vscode* wants to access the disposed webview panel, which throws an error. The issue[2] has been reported with greater detail in the *sprotty-vscode* repository.

On the topic of VS Code extensions, the *klighd-vscode* extension can be used by multiple extensions. However, only one extension that depends on *klighd-vscode* can be active in a workspace at the same time. This should be fine for most cases, but gets annoying when files that are supported by different extensions exist in a workspace. In such situation, all files for one extension have to be closed and VS Code reloaded, before files for the other extension can be visualized. A solution to fix this in *klighd-vscode* has to ensure that messages from the webview can properly be routed to multiple registered language clients. Special care must be taken to not interfere with the webview management of *sprotty-vscode*, because it requires one `SprottyVscodeExtension` instance for each language client, whereas *sprotty-vscode* assumes that only one is created. A solution might remove *sprotty-vscode* and implement the webview management and message routing from scratch. The issue[3] for this problem is tracked in the source repository.

For this thesis, the diagram options have been changed from being placed next to the diagram to being placed on top of the diagram in a sidebar, as explained in Section 4.6.1. Further, they are hidden when a user does not interact with them. Future work could add a preference that toggles the sidebar between the floating mode and a docked mode. When docked, the sidebar is rendered next to the diagram and remains open when the user interacts with the diagram. This mode might be preferred by users when enough space is available to display both the sidebar and diagram comfortably next to each other. An implementation

[2] `https://github.com/eclipse/sprotty-vscode/issues/42`

[3] `https://github.com/kieler/klighd-vscode/issues/6`

should ensure that the slide-in and slide-out animations of the sidebar still exist in docked mode to continue conveying the idea of a drawer.

To improve the performance of the diagram view, future work can explore other methods for rendering shadows in the diagram visualization. As explained in Section 6.1.1, the method that is currently implemented is the main cause for rendering-performance problems in the standalone view. As the shadows are drawn across a larger area in a full-size standalone view, the browser struggles to render these intensive shadows when the diagram is panned. If applicable, an improved method might use *box-shadows* to keep the smooth look, which are CSS-based and thus may be rendered more efficiently by a browser.

Finally, future work can implement the hosted diagram service that has been discussed in Section 4.4.1. Such service should extend the existing Fastify web server to reuse the diagram-view web page and WebSocket proxy logic. The server has to be extended with a file upload plugin to make files accessible for other users and to the language server. Furthermore, a second web page has to be added to include a file upload UI on the website. After a file is uploaded, the diagram view web page can either be embedded with an iframe to visualize the diagram, or by changing the URL to the diagram view web page. Such service enables sharable diagram visualization URLs and embeddable visualizations for online documentation.

Acronyms

API	Application Programming Interface
CLI	Command Line Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IDE	Integrated Development Environment
iframe	Inline Frame
IoC	Inversion of Control
IPC	Inter Process Communication
JSON	JavaScript Object Notation
KEITH	Kiel Environment Integrated in Theia
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client
KLighD	KIELER Lightweight Diagrams
LSP	Language Server Protocol
SCCharts	Sequentially Constructive Charts
SVG	Scalable Vector Graphics
UI	User Interface
URI	Unique Resource Identifier
URL	Unique Resource Locator
VS Code	Visual Studio Code

Bibliography

[Dom18] Sören Domrös. "Moving model-driven engineering from Eclipse to web technologies". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/sdo-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Nov. 2018.

[Eff17] Sven Efftinge. *Theia — One IDE For Desktop & Cloud*. 2017. URL: https://www.typefox.io/blog/theia-one-ide-for-desktop-cloud (visited on 08/09/2021).

[Eff20a] Sven Efftinge. *Root, Docker and VS Code*. 2020. URL: https://www.gitpod.io/blog/root-docker-and-vscode (visited on 08/09/2021).

[Eff20b] Sven Efftinge. *Theia 1.0 — Finally a Good Browser IDE*. 2020. URL: https://dev.to/svenefftinge/theia-1-0-finally-a-good-browser-ide-3ok0 (visited on 07/28/2021).

[Eug20] Eugene. *My Top 13 JavaScript Diagram Libraries*. 2020. URL: https://hackernoon.com/my-top-13-javascript-diagram-libraries-g2a53z6u (visited on 07/11/2021).

[Fas21] Fastify. *The hitchhiker's guide to plugins*. 2021. URL: https://www.fastify.io/docs/v3.15.x/Plugins-Guide (visited on 08/08/2021).

[Fou21] OpenJS Foundation. *About Node.js*. 2021. URL: https://nodejs.org/en/about (visited on 09/23/2021).

[Fug20] Vincent Fugnitto. *Consuming Builtin and External VS Code Extensions*. 2020. URL: https://github.com/eclipse-theia/theia/wiki/Consuming-Builtin-and-External-VS-Code-Extensions (visited on 08/02/2021).

[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)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[Köh17] Jan Köhnlein. *sprotty — A Web-based Diagramming Framework*. 2017. URL: https://www.typefox.io/blog/sprotty-a-web-based-diagramming-framework (visited on 07/08/2021).

[Köh19] Jan Köhnlein. *Graphical VS Code Extensions with Eclipse Sprotty*. 2019. URL: https://www.typefox.io/blog/using-sprotty-in-vs-code-extensions (visited on 07/15/2021).

[Köh20] Jan Köhnlein. *Domain-Specific Languages in Theia and VS Code*. 2020. URL: https://www.typefox.io/blog/domain-specific-languages-in-theia-and-vs-code (visited on 04/07/2021).

[Mar12] Robert C. Martin. *The Clean Architecture*. 2012. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture (visited on 08/06/2021).

[Mic16a] Microsoft. *May 2016 (version 1.2)*. 2016. URL: https://code.visualstudio.com/updates/May_2016 (visited on 08/03/2021).

[Mic16b] Microsoft. *Visual Studio Code 1.0!* 2016. URL: https://code.visualstudio.com/blogs/2016/04/14/vscode-1.0 (visited on 08/03/2021).

[Mic21a] Microsoft. *Commands*. 2021. URL: https://code.visualstudio.com/api/extension-guides/command (visited on 08/04/2021).

[Mic21b] Microsoft. *Extension Guidelines*. 2021. URL: https://code.visualstudio.com/api/references/extension-guidelines (visited on 08/03/2021).

[Mic21c] Microsoft. *Language Server Extension Guide*. 2021. URL: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide (visited on 08/13/2021).

[Mic21d] Microsoft. *Language Server Protocol Specification*. 2021. URL: https://microsoft.github.io/language-server-protocol/specifications/specification-3-16 (visited on 07/21/2021).

[Mic21e] Microsoft. *The TypeScript Handbook*. 2021. URL: https://www.typescriptlang.org/docs/handbook/intro.html (visited on 08/08/2021).

[Mic21f] Microsoft. *Webview API*. 2021. URL: https://code.visualstudio.com/api/extension-guides/webview (visited on 08/04/2021).

[Ove21] Stack Overflow. *2021 Developer Survey*. 2021. URL: https://insights.stackoverflow.com/survey/2021 (visited on 08/03/2021).

[Pet19] Jette Petzold. "Intentional layout in Sprotty Diagrams: Defining User Interaction". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jet-bt.pdf. Bachelor's thesis. Kiel University, Department of Computer Science, Sept. 2019.

[RDH20] Niklas Rentz, Christian Dams, and Reinhard von Hanxleden. "Interactive visualization for OSGi-based projects". In: *2020 Working Conference on Software Visualization (VISSOFT)*. Adelaide, Australia: IEEE, Sept. 2020, pp. 84–88. DOI: 10.1109/VISSOFT51673.2020.00013.

[Rea19] React. *Virtual DOM and Internals*. 2019. URL: https://reactjs.org/docs/faq-internals (visited on 08/09/2021).

[Rea20] React. *Introducing JSX*. 2020. URL: https://reactjs.org/docs/introducing-jsx (visited on 08/09/2021).

[Ren18] Niklas Rentz. "Moving transient views from Eclipse to web technologies". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/nir-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Nov. 2018.

[Sch19] Connor Schönberner. "Intentional Layout in Sprotty Diagrams: Reevaluating Introduced Constraints". https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cos-bt.pdf. Bachelor's thesis. Kiel University, Department of Computer Science, Sept. 2019.

[Spö18] Miro Spönemann. *Visualizing npm Package Dependencies with Sprotty*. 2018. URL: https://www.typefox.io/blog/visualizing-npm-package-dependencies-with-sprotty (visited on 07/14/2021).

[Spö19] Miro Spönemann. *Architectural Overview*. 2019. URL: https://github.com/eclipse/sprotty/wiki/Architectural-Overview (visited on 08/10/2021).

[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: IEEE, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.