TU WIEN Informatics

Creating a Contextual Connector Tool for Eclipse GLSP

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software and Information Engineering

by

Tobias Pellkvist
Registration Number 12024024

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork
Assistance: Dipl.-Ing. Dr.techn. Philip Langer, CEO EclipseSource Austria
Bsc. Haydar Metin

Vienna, 20th April, 2024

Tobias Pellkvist | Dominik Bork

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Tobias Pellkvist

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. April 2024

Tobias Pellkvist

Acknowledgements

I would like to thank Dominik Bork for making this project possible, by organizing and advising during the development process as well as advising the writing of this paper and reviewing it.

I would also like to thank Philip Langer for taking the time to advise and give crucial insight into GLSP and Haydar Metin for reviewing the code, thereby improving it.

Abstract

Graph modeling is a task that is not only notoriously time-consuming, but also fallible, leading to many mistakes that are made and not corrected during its practice. One possible way to counteract aforementioned issues would be an increase to editing usability. The open source graph editor GLSP currently lacks a way to quickly create new nodes and edges without traveling a noticeable distance with the mouse or performing search operations on the keyboard to find the correct one to create the desired element. This thesis aims to implement an extendable but simple contextual menu around the relevant nodes to improve the usability of the application. This is done by conducting a simple market analysis and extracting the most helpful features by comparing their strengths and weaknesses. Then, they will be evaluated and used for the realization of the connector tool.

Contents

Abstract **vii**

Contents **ix**

1 Introduction **1**

2 Background **3**
2.1 Eclipse GLSP 3

3 Methodology **7**
3.1 Market Analysis 7
3.2 Market Analysis Evaluation 13
3.3 Summary 14

4 Design and Implementation **15**
4.1 Design Choices 15
4.2 Data structure and reusing the PaletteItem 16
4.3 Actions and ActionHandlers 18
4.4 The SelectionPalette UIExtension 20
4.5 Usage/Customization 26

5 Evaluation and Discussion **29**

6 Conclusion **33**

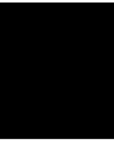
List of Figures **35**

List of Tables **37**

Acronyms **39**

Bibliography **41**

CHAPTER 1



Introduction

Recently, many new web based meta-modeling but also concrete modeling use cases such as BPMN, UML and ER graph editors have emerged, that are aiming to provide a similar, but improving experience to existing desktop applications. This trend can be explained by its many benefits, such as flexibility and availability as well as several other, more technical advantages, like server-side resource usage instead of client-side and the removal of installation issues[Gac23][REIWC18]. Currently, there is also a need to improve the usability issues that are present in existing modeling tools, as a user study by Pourali & Atlee suggests[PA18]. This study indicates that, for example, tools often do not inform users of errors in their diagram, and neither does it help much to fix these issues. Also, according to another survey amongst the modeling community, two of the topics with a Need for Action most voted for were modeling tools and usability, pointing towards a demand for improvement of the usability within tools as well as the tools themselves[MBWM24].

One web-based framework used to develop modeling tools, is the Graphical Language Server Platform (GLSP). It already provides a flexible and extensible architecture, that can be adapted by its users, while also striving to fix the current issues mentioned above. But it still lacks a fast and dynamic way to create and edit the graph[BLO24]. This paper aims to implement a solution that solves this problem by providing users with a menu that is dynamically positioned where it is needed and also provides contextual options, rather than static menus with generic options. This menu provides nodes and edges that are relevant to the current context, thereby improving the graph editing usability. Furthermore, it should be easily extensible and customizable, so that its (re)usage/adaptation is incentivized.

This thesis is structured as follows. In Chapter 2, GLSP and its most relevant features will be explanined in more detail. Chapter 3 will discuss the methodology, which includes a market analysis of similar tools in other graph editors and its evaluation by their features. In Chapter 4, the resulting implementation will be shown as well as the design

choices - partly resulting from the market analysis - which led to it. Then, in Chapter 5, this new connector tool will be evaluated using the previously inspected graph editors and finally, in Chapter 6, this paper will be concluded by reflecting and discussing potential further work.

CHAPTER 2

Background

2.1 Eclipse GLSP

The Graphical Language Server Platform[gls], also known as GLSP, is an extensible, open-source framework, which is based on Sprotty[1]. It is a web-based diagram editor that works by connecting a client, which uses the diagram editor, with a server, that handles the data which is required and used by the client, over a custom Language Server Protocol (LSP). The client handles the typical frontend tasks such as rendering, while the server is responsible for handling the graphical model as well as validating and filtering the data, amongst other tasks. The client can also be extended further by embedding or by using one of the provided integrations into IDEs such as Theia or Visual Studio Code. Similar to the integrations, there are two variants of the server, a Java and a Node version. Also, most of the project is written in TypeScript and Java.

In the following subsections, the client, server, and the protocol used to communicate between the two will be explained in more detail. Because GLSP is a very comprehensive and complex framework and explaining all of its features would not only exceed the scope of this paper but also not aid in understanding the contribution made within it, only the relevant parts will be described. For example, since the IDE integration is not a crucial part to the extension, it will not be explained any further. More details on the development of GLSP-based web modeling tools can be found in [MB23].

2.1.1 Actions and ActionHandlers

Actions and ActionHandlers are an integral part of GLSP. They are used to communicate between and within the client and the server. This is done with a so-called Action Dispatcher. When an action is triggered by, for example, a user clicking a button, the

[1] https://sprotty.org/

dispatcher first identifies whether the action is relevant for the current endpoint of the interaction (client or server) or if a remote procedure has to be used.

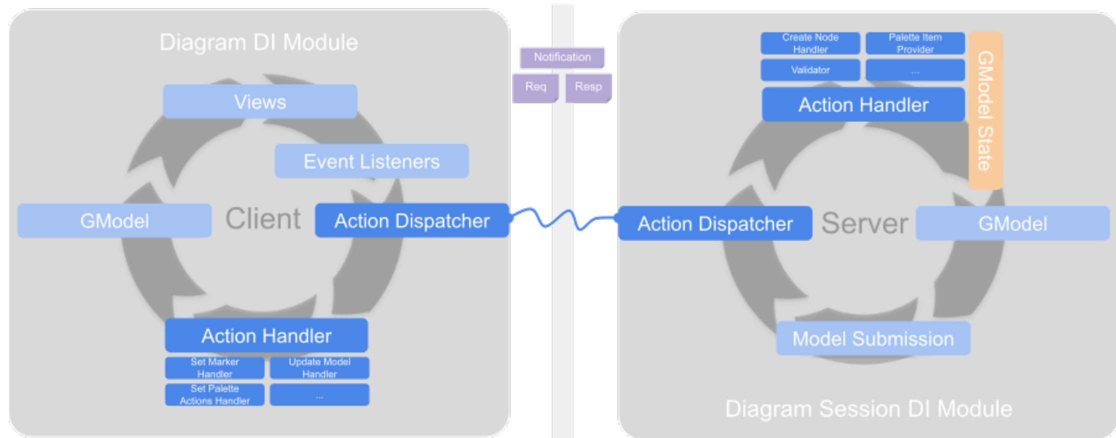


Figure 2.1: Client and Server communicating with Action (Dispatchers) and ActionHandlers [act]

The Action is then handled by the relevant ActionHandler(s) and depending on its type, might change several things. For example, creating a new node is done by sending an Action to the server, which means it has to be sent to the server to change the graphical model representation (`GModel`). The server then resends another action to the client, so it can confirm the model change and display the new node. On the contrary, changing the viewport, for example, only changes the view on the model, which does not require it to be sent to the server. This means the related ActionHandler(s) are on the client. These are only some examples of Actions and ActionHandlers that can interact between and within the client and server as displayed in Figure 2.1

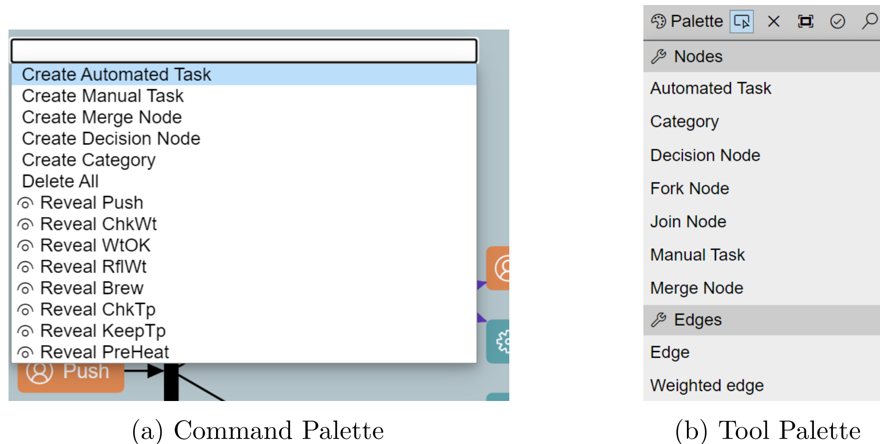
2.1.2 UI Extensions

As the name suggests, the UIExtension is an abstract class that provides the option to create a new UI element in the application. It is logically seperated from the graph, which means that it will always be in front of the graph and will not appear in contexts only relevant to the graph (like searching for an item in the following Command Palette), which makes it ideal for the task at hand. Two examples for UIExtensions are the Tool Palette and the Command Palette.

Tool Palette and Command Palette

Currently, the only existing ways to create new nodes and edges are the Tool Palette and the Command Palette. The tool palette provides the options to select a current tool (such as selection, creation, and deletion tools), to search and validate, etc. and it is

more centered on operation by mouse. On the other hand, the command palette provides similar features with the main difference being that it is centered around and created for the keyboard. This palette focuses on accessibility and is therefore currently only part of the accessibility extensions that were developed and proposed by Sarioğlu et al.[SMB23]. While similar, these palettes have distinct uses, as evident in Figure 2.2



(a) Command Palette (b) Tool Palette

Figure 2.2: Existing Palettes

The ultimate goal of the new connector tool is to implement only the creation, but allow users to do it more efficiently while also providing mouse and keyboard support. Because of the existing naming convention where menus are referenced to as "Palettes", the name that was selected for this new tool is the `Selection Palette`.

CHAPTER 3

Methodology

In this chapter, the foundations of the extension, such as UI criteria will be laid out and explained. For this, a market analysis will be conducted which will then be used and analysed. After that some additional design choices will be explained based on general practice and the existing UI of GLSP.

3.1 Market Analysis

To design this new tool, the `Selection Palette` in GLSP, a market analysis was done to extract and compare useful features as well as usability of different modeling tools with a functionally similar connector component. To have enough to compare, five were chosen from the 73 that are listed on the website of the Business Informatics Group of the TU Wien[big]. For each application, the functionality, strengths and weaknesses will be explained with the help of a flow chart, including screenshots of the application. After that, aforementioned strengths and weaknesses will be discussed in more detail, also relating to the upcoming implementation in GLSP. While none of these applications are metamodeling tools like GLSP, but rather BPMN focused, occasionally supporting UML and/or ER modelling, they were still chosen because of their popularity and the aforementioned presence of the connector tool.

3.1.1 Creately

Creately[cre] is a visual collaboration and diagram tool, which, amongst others, includes a UML editor as seen in Figure 3.1. In this case, the connector tool is, in its initial state, collapsed, while still providing some generic options, like text and edge creation. While this forces the user to make one additional click when they want to create a new node, it does not obstruct the view of other graph elements when such an action is not currently intended, thereby providing a better overview. When expanding by clicking an icon on

either side of a node, the connector tool shows a search bar, as well as the node options available, grouped together into subsections. These options are relevant to the selection, as they are in the same category of nodes and also often constrained to even less, but more relevant options. For example, a UML class as in the flow chart will, by default, primarily show UML class diagram elements while a generic square node will show other generic shapes.

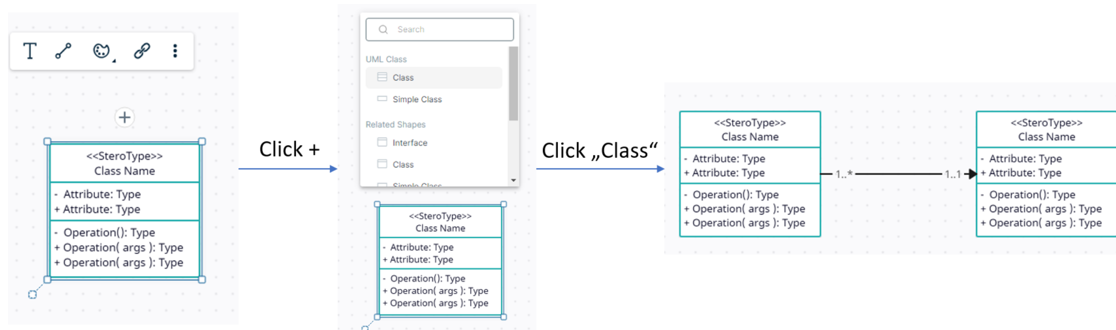


Figure 3.1: Creately flow chart

Creately provides users with good usability by including multiple functions to find the right tools to use such as a search bar and a menu that provides relevant items to the current task as well as useful grouping of said tasks. Additionally, the menu can be expanded, providing better initial visibility of the graph. However, one limitation is that edges in diagrams must be customized manually after creation, which can be time-consuming.

3.1.2 draw.io

Draw.io[dra], also known as (part of) diagrams.net, is a free diagram editing tool for flowcharts, wireframes, as well as UML diagrams, etc. It provides multiple versions of a connector tool as shown by the branching in Figure 3.2. When hovering over the arrow icon (which, as with Creately, appears in all four horizontal and vertical directions) the application displays only a few possible options to connect. While this is not clear from the flow chart, the first option shows the current node, while the remaining three are always the same. The second option to use the connector tool is to click the aforementioned arrow icon, which opens the same menu, but with four times more options where the first row is identical to the column when hovering.

Clicking one of the options creates the new node in the direction of the connector and connects the source and the new node with a directed edge. One interesting option draw.io provides is that holding the Shift key when clicking swaps out the current node to the selected one.

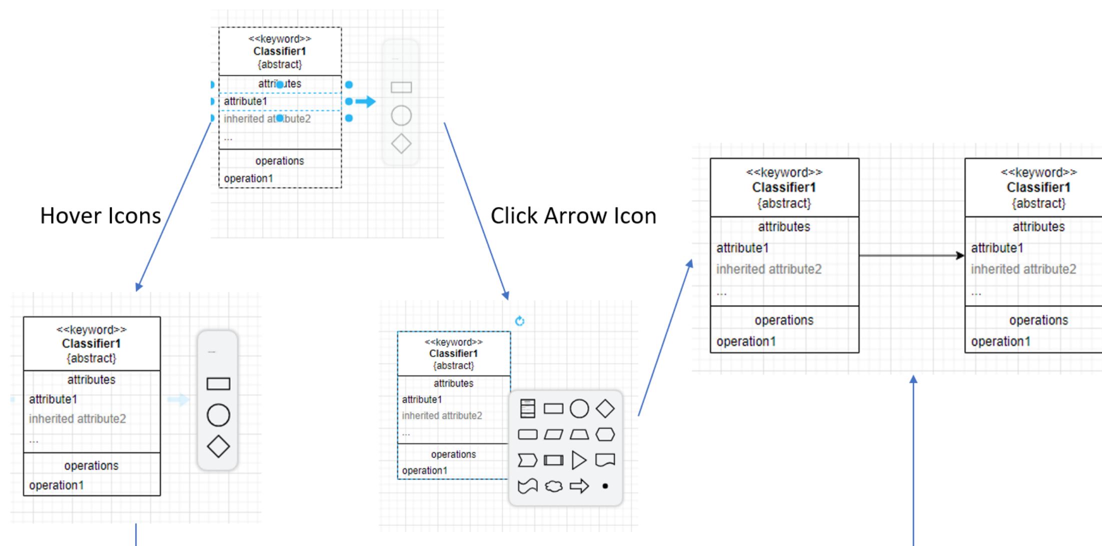


Figure 3.2: draw.io flow chart

While draw.io provides a few unique features by offering a smaller, hover-only menu and by allowing users to swap nodes, these are "nice-to-have" features, rather than the more general and flexible basics of a connector tool that GLSP aims to provide to its users. The draw.io tool does also not distinguish between its node types and does not provide edge options wheras the latter should be supported by GLSP.

3.1.3 Lucidchart

Lucidchart[luc] is, similarily to Creately, a collaboration focused charting and diagramming tool. Nodes, once again, have four directions (vertical and horizontal) to create new nodes. In this case however, the application does not make it clear that there even is a connector functionality. Hovering the orange dot on either side indicates with a tooltip (german tooltip in Figure 3.3) that dragging creates a line, but nothing else. But clicking it or drawing a new edge also opens a menu to create a new node, if the edge does not connect to an existing node.

At first glance, this tool is very similar to the one draw.io provides, since it also has a grid-like structure with four elements per row. The main differences are that the first row(s) show the most recently used nodes, chronologically from left to right. Also, there is a clear division between those nodes and the bottom nodes, which are, like in Creately, relevant.

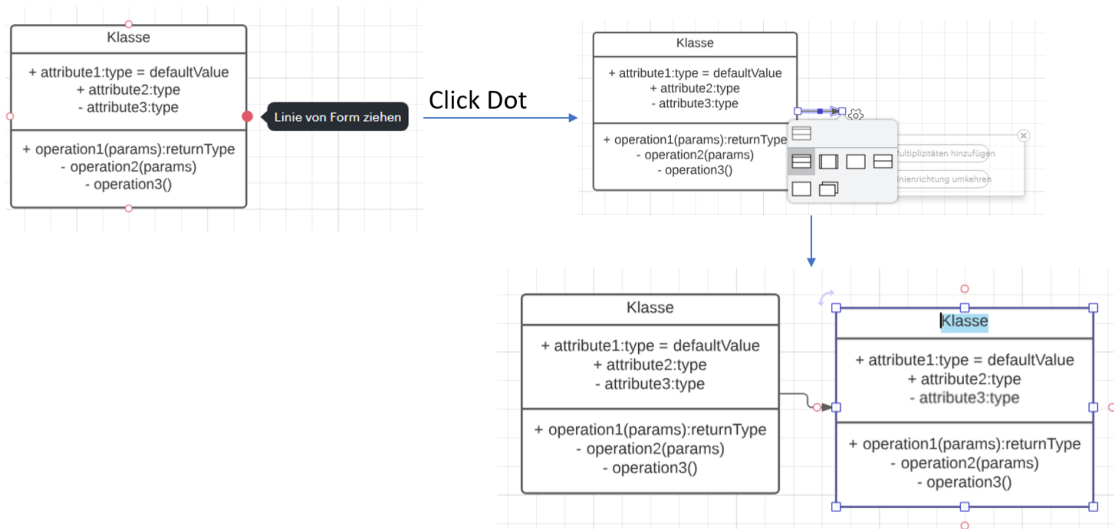


Figure 3.3: Lucidchart flow chart

The Lucidchart connector tool provides a more convenient editing experience by showing the users their most recently used nodes, as well as some recommendations for their current node. On the other hand, these recently used nodes are not labeled in any way, making it unclear what they are at first. Furthermore, the created edge has to be set beforehand, or customized after its creation, both options are only possible in a static menu. Also, while this is not as much of an issue, since the node can easily be moved afterwards, the creation of an edge, followed by a node, might also have the negative side effect of a slight misplacement of the node.

3.1.4 Camunda

While the other editors until now were mostly generic and supported multiple different types of notation, Camunda[cam] is mainly a BPMN (Business Process Management Notation) tool. Its connector tool also shows icons in a grid, but their functions are all very different, yet not grouped differently. In the flow chart, the first four options displayed are nodes, followed by a commenting tool and an expanding menu. After that, there are tools that can be used to delete elements, create new edges, etc.

Clicking the three dots opens the more detailed view, which looks very similar to the one Creately provides. It also includes a search bar and the elements that are relevant to the currently selected node. Creating a node in Camunda always creates it to the right side of the current node.

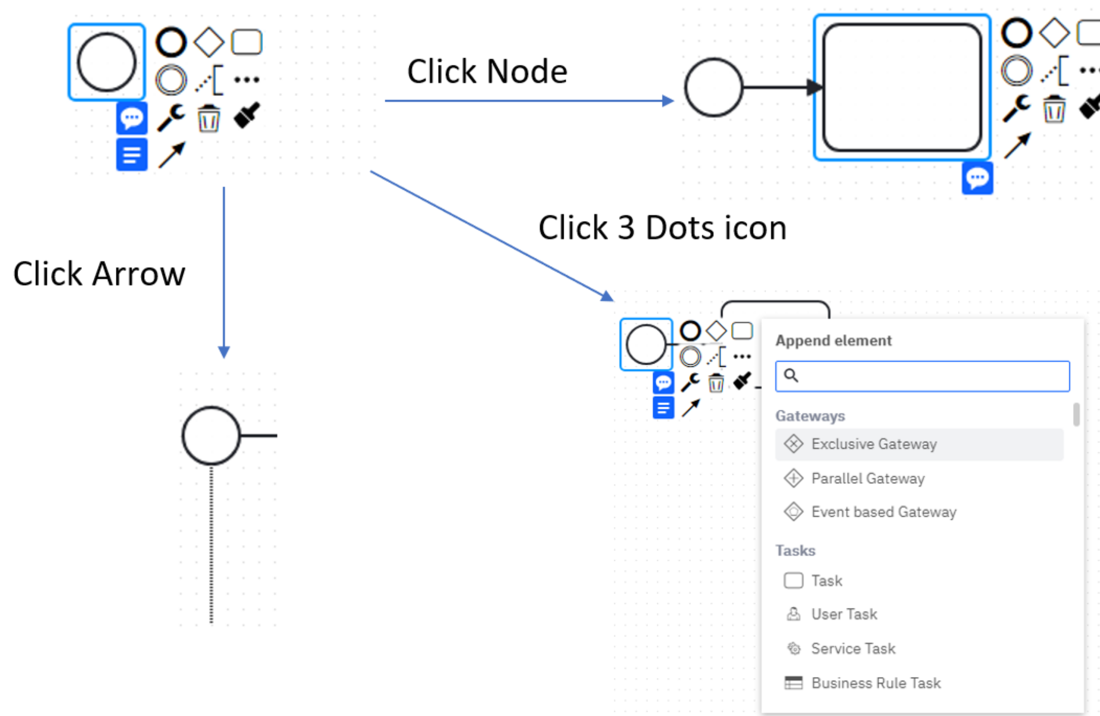


Figure 3.4: Camunda flow chart

While Camunda, on the surface, provides access to a lot of solid usability features like search and grouping based on category, its compact menu contains too many items with too many purposes as seen in 3.4. This can get confusing very quickly, especially for a novel users. Also, contrary to the previous diagram editors, this more compact, but unclear menu is already opened when selecting the node, which might impact user experience negatively, since it may hide parts of the graph unintentionally.

3.1.5 Adonis

Adonis[ado], like Camunda, is also a BPMN tool. Its connector tool contains only basic functionality. On the left side there are buttons for settings, properties, etc. while the right side contains nodes and one edge option. This might be confusing, because they are grouped together as in Camundas tool, but, since it is not as overwhelming with limited options and uses colors, this solution might be more clear. Here, the nodes are also always created to the right when clicking as shown in Figure 3.5 and the menu does not need to be expanded.

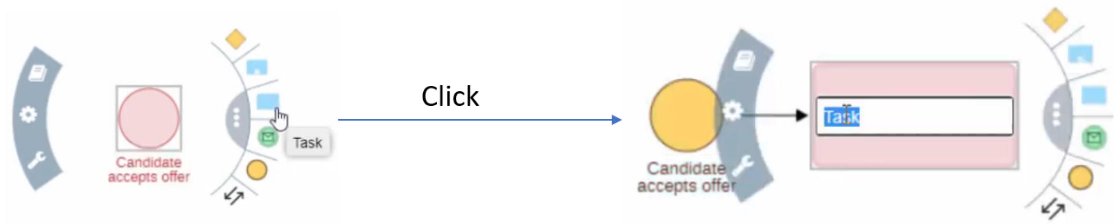


Figure 3.5: Adonis flow chart

While Adonis does not have any severe downsides, the only noteworthy one being that it might also overshadow the graph because it is always expanded, it is worth noting that this is mostly possible because of its specialized implementation. For example, using colors is not always feasable, which could reduce the clarity when selecting nodes. Adonis also provides a unique look by using a compact, circular menu instead of the rectangular ones analyzed so far.

3.2 Market Analysis Evaluation

To conclude the market analysis, I will combine the strengths and weaknesses described into a matrix as shown in Table 3.1 for comparison of their key features and then briefly list the UI/UX requirements of the connector tool to be built and why they have been chosen, based on the previous analysis and table.

	Creately	draw.io	Lucidchart	Camunda	Adonis
Search function	✓	×	×	✓	×
Grouping/ Segmentation	✓	✓	✓	×(small)/ ✓(expanded)	✓
Context-relevant suggestions/options	✓	✓	✓	✓	✓
Expandable menu	✓	✓	✓	✓	×
Node options	✓	✓	✓	✓	✓
Edge options	×	×	×	×	×
Edge customization	✓	✓	✓	✓	✓
Other strengths	None	Smaller menu on hover	Recently used nodes	Recently used nodes, deletion	compact, color usage

Table 3.1: Graph editor/feature comparison matrix

Based on Table 3.1, the connector tool should at least fulfill the following requirements:

Clear segmentation As shown with the example of Camunda, differenciating between nodes, edges, and other tools is arguably more important compared to a compact implementation. Therefore, the GLSP connector should have a segmentation between groups of options by using headers and, optionally, different locations.

Expandable menu Since creating a new object is not always the objective when selecting a node, the connector tool should only expand once a button is clicked to do so. This is done to prevent overloading of the UI.

Search bar If many possible options are available for any group (for example edges), they should be optionally filterable by a search bar. If there are less than three options however, searching is redundant and the search bar should therefore not

be displayed. Also, the search (bar) should be done for each category individually, since a user most likely knows if they are looking for either an edge or a node.

3.3 Summary

To summarize, in this Chapter five different applications containing a contextual menu which allows the creation of new elements in a graph were analyzed and compared. Their most important functional and visual features were extracted to be implemented in a similar fashion in the `Selection Palette` in GLSP. These (optional) features are clear segmentation between elements, an expandable menu on node selection and a search bar to find the desired items to create. In the following Chapter, the concrete plan to implement these criteria will be discussed before the implementation itself is shown.

CHAPTER 4

Design and Implementation

In this chapter, the design choices followed by the implementation will be explained in detail. This includes the `UIExtension`, Actions, and ActionHandlers, as well as the `ContextActionProvider` and CSS related to positioning. The remaining code, mainly arbitrary CSS choices, will be glanced over. For the implementation the node server version and Theia integration version were used, which means all of the code presented is written in TypeScript.

4.1 Design Choices

To satisfy the criterion of clear segmentation but also provide customizability and extensibility of the connector tool, a layout was chosen where different types of items, such as nodes and edges will be shown in containers that are placed around the selected node. The location of these containers can also be set (Top, Left, Bottom, Right). If the location is set to be on the same side for multiple containers, they are merged together into one container by stacking on top of each other, where they can still be seperated by headers. This was done to ensure that the distance between the node and the connector tool does not grow unnecessarily large. Also, unevenly tall containers next to each other would not look consistent.

Because containers can be stacked on top of each other, making them collapsible also makes sense, since they can otherwise grow in height to an unwanted extent.

To retain the ease of usage for current users and potentially increase the intuitiveness and the speed of the learning process of the new connector tool[DM13], the same colors for headers, sections, etc. as the existing tool palette will be used for the new one. For similar reasons, iconography and hover effects will also be retained.

Finally, since the new connector tool is always related to/associated with a selected node, it would make sense to draw an edge between it and the new node created from the

Figure 4.1: Fork Node creation button

connector. Depending on types of the graph and the nodes used, this edge should also be set to a specific default edge, limited to a certain selection. For example, in a UML activity diagram, it would not be possible to create multiple unlabeled outgoing edges, they would need to contain a condition to make sense.

Similarily, nodes that can not follow from a certain type of node, should be excluded from the available node list. For example, an end node in an activity diagram cannot point to any other nodes. More about this will be explained in detail later in this chapter.

4.2 Data structure and reusing the PaletteItem

A data structure for the communication between server and client is required. Since an existing structure already exists for a similar purpose, the tool palette, the same data structure can be used for the new connector tool. This existing data structure is called a PaletteItem and it is structured as in Table 4.1

PaletteItem Properties		
Name	**Type**	**Description**
LabeledAction Properties		
Label	`string`	Name of the PaletteItem
actions	`Action[]`	List of associated actions
icon?	`string`	optional Icon
id	`string`	Identifier
sortString	`string`	arbitrary string for sorting
children?	`PaletteItem[]`	optional list of associated PaletteItems

Table 4.1: Palette Item Properties

The `PaletteItem` interface extends the `LabeledAction`, which is an interface that contains a label, actions, and an icon. These properties will be used to create the basis of a node or edge creation button. In the case of Figure 4.1, "Fork Node" is the `label`, the `icon` string is the codicon[cod] class for the icon, in this case "symbol-property", and the `actions` are the Actions that are executed upon clicking the button. Here, clicking

the button triggers a `TriggerNodeCreationAction`, which enables the user to click a location to create the node there.

The `id` is a general use identifier, but it is mainly used as the id for the related HTML tags in the context of the selection palette and the `sortString`, by default is the first letter of the label. It is used to sort the list of nodes and edges (individually). The most important part of this data structure are the `children`. They allow a recursive use, which makes it possible to create structures such as the following:

- Nodes (PaletteItem)
  - Fork Node (PaletteItem)
  - Another Node (PaletteItem)
  - ...
- Edges (PaletteItem)
  - Normal Edge (PaletteItem)
  - Weighted Edge (PaletteItem)
  - ...

The items shown as example are taken from the Workflow Diagram example that is present in the repository of the GLSP Client[1]. In practice, these would be dependant on modeling language, a UML activity diagram could, for example, contain an Action node.

While the `PaletteItem` interface provides a good headstart, there are still more required properties for minimum functionality like the position of the containers and some optional ones to provide extensibility, like whether the menu should be extendable, if the title should be shown or if only the icon or only labels should be shown. Also, to implement the creation of a edge on node creation as explained in Section 4.1, a seperate subtype of the `PaletteItem`, the `SelectionPaletteNodeItem`, which is specifically used for nodes. All of this was implemented as shown in Listing 4.1 with two enums for icons/labels and position as well as another subtype, the `SelectionPaletteGroupItem`, which contains the necessary information mentioned above.

```
1  export enum SelectionPaletteGroupUIType {
2      Icons,
3      Labels
4  }
5
6  export enum SelectionPalettePosition {
7      Left,
8      Right,
9      Top,
10     Bottom
11 }
12
13 export interface SelectionPaletteGroupItem extends PaletteItem
       {
```

[1] https://github.com/eclipse-glsp/glsp-client

```
14      /** Position of the group */
15      readonly position: SelectionPalettePosition;
16      /** Shows the title of a group */
17      readonly showTitle: boolean;
18      /** Shows a group as a collapsed submenu if true, open if
           false */
19      readonly submenu?: boolean;
20      /** Show either only icons or labels. Show both when not
           given*/
21      readonly showOnlyForChildren?: SelectionPaletteGroupUIType;
22  }
23
24  export interface SelectionPaletteNodeItem extends PaletteItem {
25      /** default edge when creating an outgoing edge */
26      readonly edgeType: string;
27  }
```

Listing 4.1: SelectionPalette enum and structure type definitions

Now the example from before would look like this:

- Nodes (SelectionPaletteGroupItem)
  - Fork Node (SelectionPaletteNodeItem)
  - Another Node (SelectionPaletteNodeItem)
  - ...
- Edges (SelectionPaletteGroupItem)
  - Normal Edge (PaletteItem)
  - Weighted Edge (PaletteItem)
  - ...

With the structure prepared, the implementation of the client can begin.

4.3 Actions and ActionHandlers

To activate and deactivate the `UIExtension`, it was configured as a `SelectionListener` by implementing the `ISelectionListener` interface. This interface allows the `UIExtension` to listen to changes relating to selection. The related event fires when either nothing or one element is selected or when multiple elements are selected, but not when the selected elements are selected again. This listener then always triggers a function, which was called `selectionChanged` as shown in Listing 4.2. This function

shows the selection palette anytime a selection of exactly one node is made. Otherwise, it hides the selection palette. The `SetUIExtensionVisibilityAction` is used to make the `UIExtension` visible or invisible, depending of the value of its `visible` property.

```
1  selectionChanged(root: GModelRoot, selectedElements: string[]):
       void {
2      if (selectedElements.length === 1) {
3          const filteredNodes = root.children.filter(element =>
              element instanceof GNode && element.id ===
              selectedElements[0]);
4          if (filteredNodes.length === 0) {
5              return;
6          }
7          this.selectedElementId = filteredNodes[0].id;
8          this.actionDispatcher.dispatch(
              SetUIExtensionVisibilityAction.create({ extensionId:
               SelectionPalette.ID, visible: true }));
9      } else {
10         this.actionDispatcher.dispatch(
              SetUIExtensionVisibilityAction.create({ extensionId:
               SelectionPalette.ID, visible: false }));
11         this.hideAll();
12     }
13 }
```

Listing 4.2: SelectionPalette selectionChanged function

To adjust the position of the selection palette, when the node moves, several triggering Actions need to be registered, such as

- `MoveAction`, which triggers when moving the diagram elements
- `ChangeBoundsOperation`, which triggers when resizing a node
- `SetViewportAction`, which triggers when panning or zooming

After that, these `Actions` can be handled in the `handle()` function, where, depending on the `Action` subtype, the position of the `SelectionPalette` would be set according to the new values. More about positioning will be explained in Subsection 4.4.1. The `handle()` function is always triggered when the `SelectionPalette` receives an Action. These Actions were all chosen to not let the selection palette float around in seemingly arbitrary locations, since the `UIExtension` does not automatically move with the rest of the diagram. Also, when a node is deleted, the palette should not remain in the same location until a new node is selected. So when a `DeleteElementOperation`

is triggered, the `handle()` function can hide the palette like in the `else` branch in Listing 4.2

The final `Action/ActionHandler` combination is the one for item retrieval. These `Actions` and `ActionHandlers`, unlike the ones before, need to communicate with the server. For this, existing interfaces were used and implemented, namely the `RequestContextActions` Action and the `ContextActionsProvider` `ActionHandler`. These were chosen because they were, just like the `PaletteItem`, also used by the tool palette.

The `RequestContextActions` works as follows: the Action is sent with a given `contextId` (which in this case is the `ID` of the `UIExtension`) and an `editorContext` which itself contains `selectedElementIds`, which can be used to identify the selected node and can also include some optional arguments, which will be useful to set the previously discussed default edge type. An Action sent by the client for this is shown in Listing 4.3

```
1 const requestAction = RequestContextActions.create({
2         contextId: SelectionPalette.ID,
3         editorContext: {
4             selectedElementIds: [this.selectedElementId],
5             args: { nodeType: contextElement!.type }
6         }
7     });
```

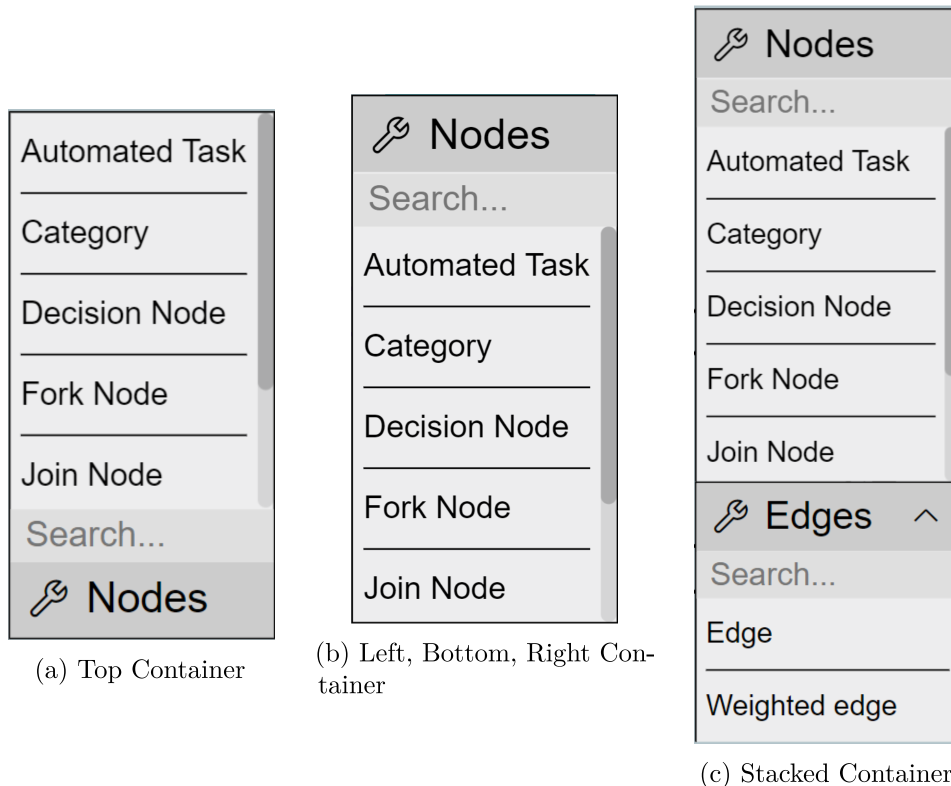
Listing 4.3: RequestContextActions for SelectionPalette

Lastly, the implementation of the `ContextActionsProvider`, the `DefaultSelectionPaletteItemProvider` (which extends the abstract `SelectionPaletteItemProvider`) receives this request and first filters by `CreateOperations` (`CreateEdgeOperation` and `CreateNodeOperation` seperately) using the `OperationHandlerRegistry`, which is a registry containing all possible Operations (special type of `Actions`). The Operations are then also filtered depending on selected node type. More about this filter will be explained in the section about Usage. After that, the items are finally returned to the client with the data structure described in the first section of this chapter.

4.4 The SelectionPalette UIExtension

Finally, the core component of this project: the `SelectionPalette` `UIExtension`. This name, as briefly mentioned at the end of Chapter 2, was chosen to fit the existing `CommandPalette` and `ToolPalette` and their related usage. When the `UIExtension` is initialized, each `SelectionPaletteGroupItem` is created as their own container with a (optional) header, a search bar, which is shown if two or more options are available and its contained children as buttons as displayed in Figure 4.1.

Because the containers can be collapsed and the headers essentially become the anchors next to the node, they should not move if the body of the container is collapsed or expanded. For this reason, the container on the top has a reversed order of items (buttons, search bar, then header, the items within the list of nodes/edges are in the usual order). As explained in Section 4.1, the containers can also be stacked on top of each other. For the containers on top, all stacked containers will be reversed. Finally, containers also have a fixed height of 200px and can therefore have a scroll bar on their right side.



(a) Top Container

(b) Left, Bottom, Right Container

(c) Stacked Container

Figure 4.2: Different Container Versions

4.4.1 Positioning and Scaling

Calculating the position of the connectors was fairly straight-forward. The first step was to calculate the center of the node that was selected by using the ID of that node and adding the halves of its width and height to its x and y positions, respectively. Since GLSP can also be used as integrations in Theia and Visual Studio Code, the width and height of the viewport had to be added as padding as well. This "main position" is an absolute position from where the single containers are spaced out. After calculating this initial position, for each container of the connector, once again use half of the node width or height and add some padding. depending on the direction of the container relative to the node (Top, Left, Bottom, Right), their own css properties with the same names need

to be adjusted. For example, if a container is above the node (position top), its bottom and left property are changed.

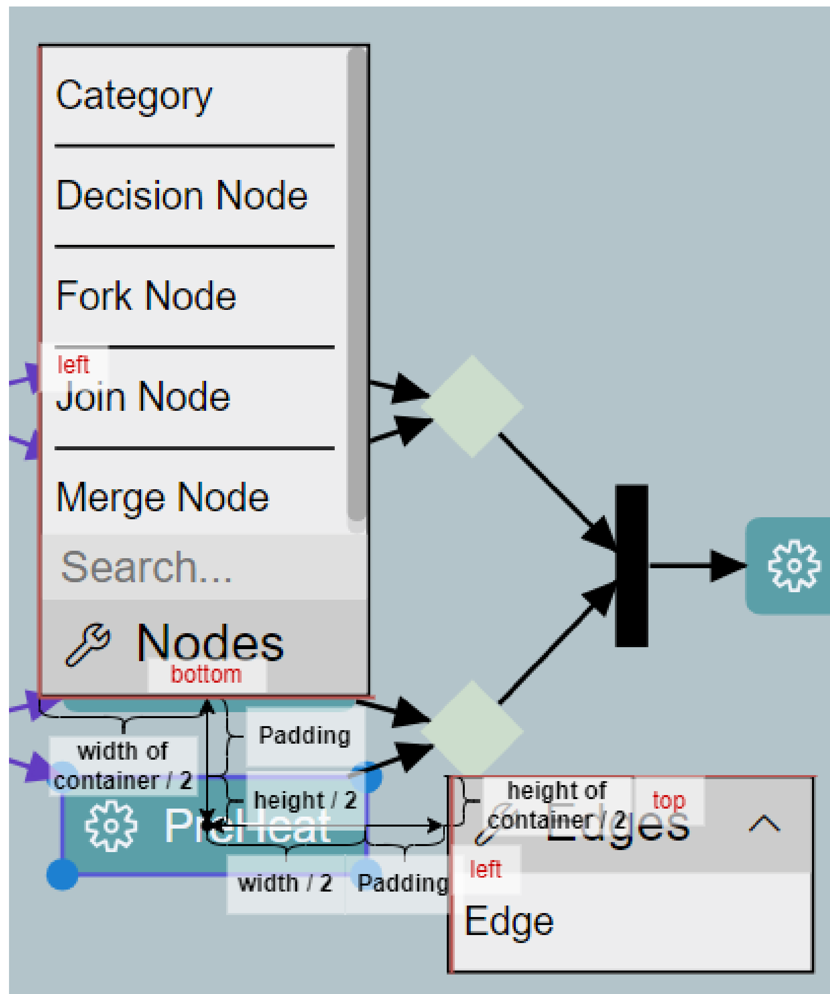


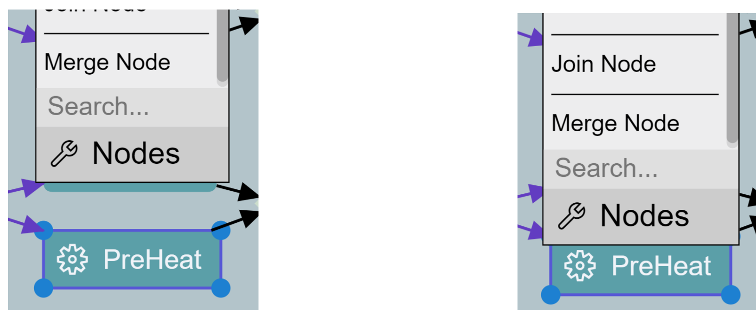
Figure 4.3: Diagram of connector positioning

What makes this a bit more complicated is the inclusion of zoom. In GLSP, the graph can be freely zoomed by scrolling. This transforms the graph with the "scale" operation in css, but leaves `UIExtensions` as they are. To counteract that, the `UIExtension` needs to be scaled as well. This creates two problems though.

1. Properties that contain the unscaled dimensions such as `HTMLElement.style.width` cannot be used for calculation
2. By default, elements in CSS are scaled from the middle, making the distance between node and container arbitrary.

To fix these issues, first of all, only rendered widths and heights can be used. They can be retrieved from functions such as `HTMLElement.getBoundingClientRect()` or properties such as `HTMLElement.offsetHeight`. To adjust the position of scaled containers, the `css` property `transform-origin` has to be used to set an anchor point for transformations. For example, to scale for the top container, `transform-origin: center bottom` has to be used to ensure that the container scales from the bottom to the top. For the sides, the anchor point was chosen to be the top right and top left corners for the left and right side, respectively, since expanding these containers would result in erratic movements when they were scaled.

For example, scaling up an element with a positive `bottom` property, results in it being lower than intended as shown in Figure 4.4



(a) Transform-origin present (b) Transform-origin missing

Figure 4.4: Comparison of transform-origin missing and present with transform:scale value larger than 1

4.4.2 Keyboard navigation

Because mouse navigation is sometimes simply not the preferred input method and because is not always feasable due to physical disabilities, injuries, or other limitations, supporting accessibility through keyboard navigation, not only in GLSP but in web modeling tools in general, is becoming a necessity[SMB23][Sar23]. In the case of the `SelectionPalette`, it has been implemented using a combination of basic `onkeydown` events and the Sprotty `KeyListener`, since the keyboard inputs are only recognized in seperate contexts.

To expand the menu when a node is selected and to close it when no elements are selected, a `KeyListener` was created, which is a class with an overridable function `keyDown` which takes a `KeyboardEvent` as input, representing the event that occured when a key was pressed. By default, pressing Space launches a `ChangeSelectionPaletteStateAction` with a value of `SelectionPaletteState.Expand`. In this case the `SelectionPaletteState` type represents the state to be changed to. Closing, which happens by pressing Escape, uses the same Action with `SelectionPaletteState.Collapse`.

```
1 export class SelectionPaletteKeyListener extends KeyListener {
```

```
2       override keyDown(_element: GModelElement, event:
           KeyboardEvent): Action[] {
3           if (matchesKeystroke(event, 'Space')) {
4               return [ChangeSelectionPaletteStateAction.create(
                   SelectionPaletteState.Expand)];
5           }
6           if (matchesKeystroke(event, 'Escape')) {
7               return [ChangeSelectionPaletteStateAction.create(
                   SelectionPaletteState.Collapse)];
8           }
9           return [];
10      }
11  }
```

Listing 4.4: SelectionPaletteKeyListener

For navigation between the nodes and edges and the containers themselves, a `onkeydown` listener was added to each button. For this, some rules were established that might at first be counterintuitive, but make sense usability-wise.

1. Collapsible groups should have focusable headers, while others should not. This was done to not focus any elements that cannot be interacted with, even though this might be inconsistent.

2. Naturally, items in collapsed groups should not be focusable.

3. Since, by default, only pressing up and down (forward and backward) is supported for navigation, deciding which container comes next depends strongly on culture and even personal preference (for example clockwise vs. counter-clockwise), the order in which the items are retrieved from the server will be the order of navigation.

4. Since using the search bar implies that an item is required from the category of the current container, navigating backward and forward will lead to the last and first items, respectively, instead of navigating to the header or a different container.

5. This also works vice-versa, entering the search bar by keyboard requires a special input (Control by default), since searching is not always required.

Here is a flow chart with an example to visualize all of these rules:

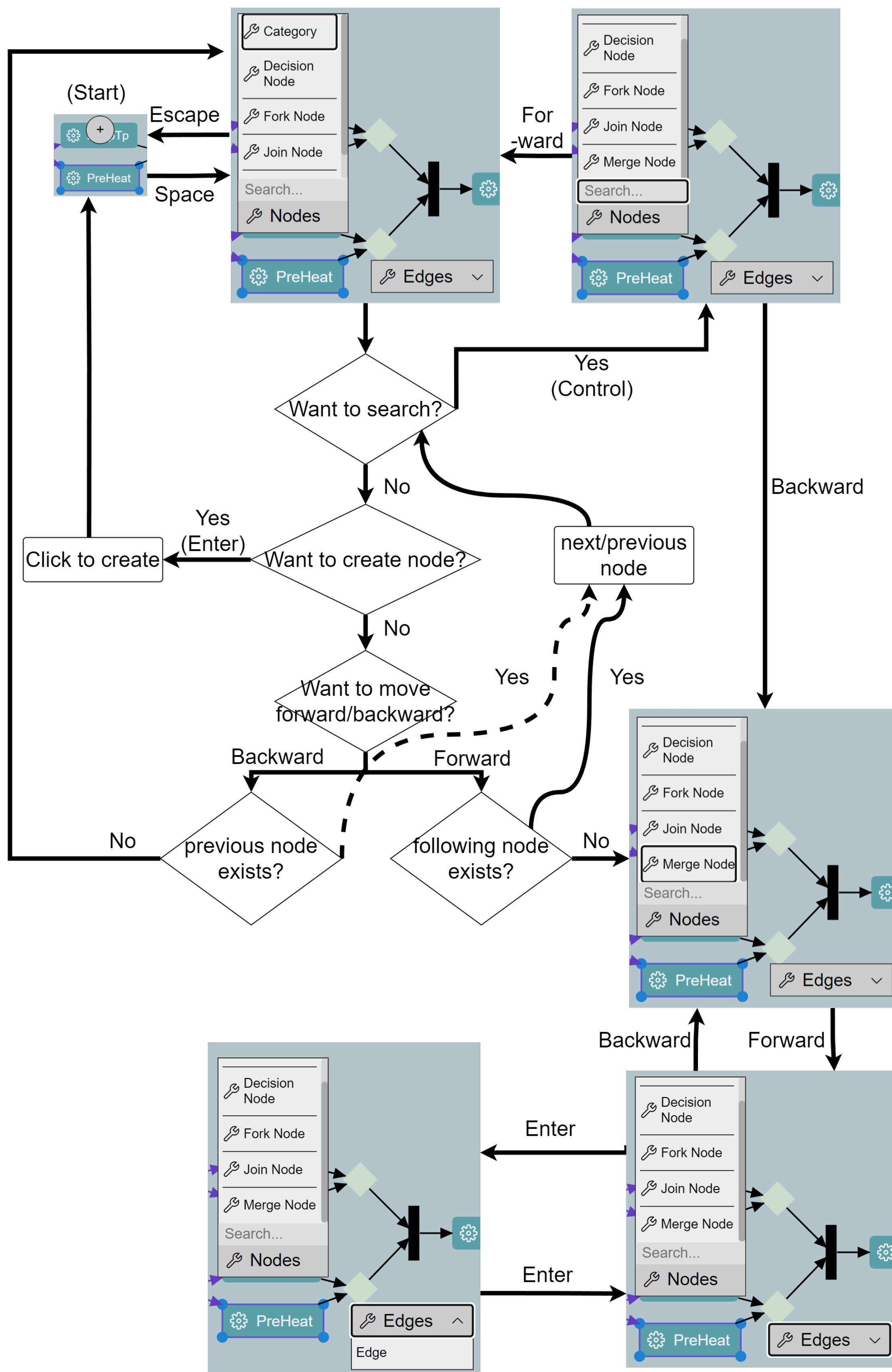


Figure 4.5: Keyboard Navigation flow chart

Finally, Table 4.2 summarizes the keyboard scheme described throughout this Chapter.

Button	Effect
Space	Expands the menu
Escape	Closes the menu
Down Arrow	moves to the next item
Up Arrow	moves to the previous item
Control	Enter search bar for current container
Enter	Toggle container state (for collipsable containers)

Table 4.2: Keyboard Controls

4.5 Usage/Customization

To close out this chapter, I will describe all of the basic customization options that are available for the selection palette. If they were described somewhere already, they will be described here again for coherence.

4.5.1 UI Settings

There are two different types of settings: server and client related. On the client, settings that are independent of context like the position of the expand button and the forward and backward button can be changed. These are typed `SelectionPalettePosition` and `KeyCode`, respectively, as shown in Listing 4.5

```
1 protected previousElementKeyCode: KeyCode = 'ArrowUp';
2 protected nextElementKeyCode: KeyCode = 'ArrowDown';
3
4 // Sets the position of the expand button
5 protected expandButtonPosition = SelectionPalettePosition.Top;
```

Listing 4.5: Keyboard navigation keys and expand button position configuration

On the server the container specific settings can be adjusted like in Listing 4.6

```
1 protected override selectionPaletteNodeSettings:
     SelectionPaletteSettings = {
2         position: SelectionPalettePosition.Top,
3         showTitle: true,
4         submenu: false,
5         showOnlyForChildren: SelectionPaletteGroupUIType.Labels
6     };
7
8 protected override selectionPaletteEdgeSettings:
     SelectionPaletteSettings = {
9         position: SelectionPalettePosition.Right,
```

```
10          showTitle: true,
11          submenu: true
12      };
```

Listing 4.6: Container specific settings

The `showTitle` value determines whether the header is shown while the `submenu` determines if the container is collapsible. Then, the `showOnlyForChildren` value, which can be either Labels or Icons, determines if only labels or icons are shown within a group.

4.5.2 Filters and Edge settings

Furthermore, on the server, the available options in the connector can be set with the `nodeOperationFilter` property of the `SelectionPaletteItemProvider`. They could for example be set like in Listing 4.7

```
1  protected override nodeOperationFilter = {
2          [ModelTypes.AUTOMATED_TASK]: [ModelTypes.WEIGHTED_EDGE,
               ModelTypes.AUTOMATED_TASK, ModelTypes.MANUAL_TASK,
3              ModelTypes.ACTIVITY_NODE],
4          [ModelTypes.MERGE_NODE]: [DefaultTypes.EDGE, ModelTypes
             .MERGE_NODE, ModelTypes.CATEGORY],
5          ...
6      };
```

Listing 4.7: nodeOperationFilter example

This filter is a list of key value pairs where the key is the selected node and the value is another list of all the options that should be filtered out. In this case, the weighted edge, automated task node, manual task node and activity node are filtered out from the automated task node and the default edge, merge node and category node are filtered out from the options for the merge node. This was implemented as a blacklist instead of a whitelist, since it was assumed that there are more likely to be exceptions present rather than the contrary.

Additionally, the edge settings can also be set in the server as shown in Listing 4.8

```
1  protected override defaultEdge = DefaultTypes.EDGE;
2
3  protected override edgeTypes = {
4      [ModelTypes.AUTOMATED_TASK]: ModelTypes.WEIGHTED_EDGE,
5      [ModelTypes.MERGE_NODE]: ModelTypes.WEIGHTED_EDGE,
6      ...
7  };
```

Listing 4.8: Edge settings

By setting the value of the `defaultEdge`, the overall default is set for the connecting edge when creating a new node. The `edgeTypes` property is, once again, a list of key value pairs where the key is the selected node and the edge is the outgoing edge that is created, overriding the `defaultEdge`.

CHAPTER 5

Evaluation and Discussion

To evaluate the created tool, the feature matrix from Table 3.1 will be expanded by one column (GLSP) to compare with the other graph editors inspected in the market analysis. This comparison is by no means a comprehensive evaluation. For that, other, more complex methods, such as user benchmarking and expert evaluation, amongst others, are required. They were not done here, since that would exceed the scope of this paper. Instead, a short discussion was written.

	Creately	draw.io	Lucidchart	Camunda	Adonis	**GLSP**
Search function	✓	×	×	✓	×	✓
Grouping/ Segmentation	✓	✓	✓	×(small)/ ✓(expanded)	✓	✓
Context-relevant suggestions/options	✓	✓	✓	✓	✓	✓(configurable)
Expandable menu	✓	✓	✓	✓	×	✓
Node options	✓	✓	✓	✓	✓	✓
Edge options	×	×	×	×	×	✓
Edge customization	✓	✓	✓	✓	✓	×
Other pros	None	Smaller menu on hover	Recently used nodes	Recently used nodes, deletion	compact, color usage	Customizable

Table 5.1: Graph editor/feature comparison matrix (including GLSP)

While this matrix supports the conclusion that the GLSP selection palette holds up compared to the other applications discussed in the Market Analysis (Section 3.1), this is still no conclusive evidence, that the created tool is sufficient for user needs. Some, for example, might not like the fact that this connector tool is strictly vertical. And besides that, it still has obvious flaws, such as the potential overlap of containers when nodes are too small in width. However, the fact that it is fully visually customizable and also supports simple extension of the provided functionality by adding additional containers, at least partially makes up for its potential usability weaknesses.

Also, while comparison might not seem entirely fair to either party, since GLSP is a metamodeling tool, while the others are primarily BPMN tools, the main functionality that is compared here is arguably not dependant on the current notation. For example, an ER and UML tool could use similar versions of the same connector tool, their differences would mainly be the contents.

Furthermore, even though most of the applications inspected support dynamic menus by reordering lists of items to list more relevant items at the top, indicating its usefulness, a study by Mitchell & Shneiderman[MS89] suggests that novel users might struggle with this type of menu, and that they might prefer a static menu, while more experienced users might be able to take advantage of a dynamic menu more easily. GLSP's `SelectionPalette` currently only displays its items according to the preset configu-

ration more akin to a static implementation. A possible compromise, considering the current rise of AI usage, might be an approach that uses AI to learn the users preferences, initially providing a more beginner-friendly entry, while also providing more advanced users with the options they are looking for. This approach, of course, would have to be evaluated with a user study on its own. For example, a recent evaluation of a concept, using the Decision Model and Notation (DMN) modeling language combined with AI, which was created to assist humans in modeling tasks, shows potential of the usefulness of AI in the domain of graph modeling[BAD23].

CHAPTER 6

Conclusion

In this thesis a new interaction method, the Selection Palette, was introducted to GLSP to quickly create new nodes and edges, enabling a more simple graph editing experience. This was done by first analyzing similar tools and extracting their desirable features. After that, they were introduced to the Selection Palette in a way that supports extensibility and finally, while not fully evaluating the result to not exceed the scope of this paper, its strengths and weaknesses were discussed.

Because this solution is certainly not perfect, further work is required. A proper evaluation, including user feedback should be conducted, while the connector should simultaneously be improved iteratively with the given feedback. Also, more customizability should be added, such as the option to enable or disable the expansion button. Many different visual default options could also be included, like a default horizontal mode to the containers or even an iconized grid system, where adaptors can easily set a preferenced layout for the containers. Additionally, even though the differences between notation types are not necessarily as important as the other features that were discussed, more default modes, adjusting small, but important details, such as edge notation in the case of, for example, 1:n connections in ER diagrams, could be included, essentially resulting in a collection of defaults for different types of notation, depending on the different needs.

List of Figures

2.1 Client and Server communicating with Action (Dispatchers) and ActionHandlers [act] 4
2.2 Existing Palettes 5

3.1 Creately flow chart 8
3.2 draw.io flow chart 9
3.3 Lucidchart flow chart 10
3.4 Camunda flow chart 11
3.5 Adonis flow chart 12

4.1 Fork Node creation button 16
4.2 Different Container Versions 21
4.3 Diagram of connector positioning 22
4.4 Comparison of transform-origin missing and present with transform:scale value larger than 1 23
4.5 Keyboard Navigation flow chart 25

List of Tables

3.1 Graph editor/feature comparison matrix 13

4.1 Palette Item Properties 16
4.2 Keyboard Controls 26

5.1 Graph editor/feature comparison matrix (including GLSP) 30

Acronyms

GLSP Graphical Language Server Platform. ix, 1, 3, 5

Bibliography

[act] Actions and actionhandlers. `https://eclipse.dev/glsp/documentation/actionhandler`. Accessed: (4.3.2024).

[ado] Adonis. `https://www.adonis-community.com/`. Accessed: (23.2.2024).

[BAD23] Dominik Bork, Syed Juned Ali, and Georgi Milenov Dinev. Ai-enhanced hybrid decision management. *Bus. Inf. Syst. Eng.*, 65(2):179–199, 2023.

[big] List of modeling tools by the business informatics group (big) of the tu wien. `https://me.big.tuwien.ac.at/tools-overview`. Accessed: (4.3.2024).

[BLO24] Dominik Bork, Philip Langer, and Tobias Ortmayr. A vision for flexible glsp-based web modeling tools. In João Paulo A. Almeida, Monika Kaczmarek-Heß, Agnes Koschmider, and Henderik A. Proper, editors, *The Practice of Enterprise Modeling*, pages 109–124, Cham, 2024. Springer Nature Switzerland.

[cam] Camunda. `https://camunda.com/`. Accessed: (23.2.2024).

[cod] Codicon icons. `https://microsoft.github.io/vscode-codicons/dist/codicon.html`. Accessed: (25.2.2024).

[cre] Creately. `https://creately.com/`. Accessed: (23.2.2024).

[DM13] Mariam Dzulkifli and M. Faiz Mustafar. The influence of colour on memory performance: A review. *The Malaysian journal of medical sciences : MJMS*, 20:3–9, 03 2013.

[dra] draw.io. `https://drawio-app.com/`. Accessed: (23.2.2024).

[Gac23] Tobias Gacko. Survey of current (web) modeling tool development platforms. Bachelor's thesis, Technische Universität Wien, 2023.

[gls] Glsp. `https://eclipse.dev/glsp/`. Accessed: (23.2.2024).

[luc] Lucidchart. `https://www.lucidchart.com/`. Accessed: (23.2.2024).

[MB23] Haydar Metin and Dominik Bork. On developing and operating glsp-based web modeling tools: Lessons learned from BIGUML. In *26th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2023, Västerås, Sweden, October 1-6, 2023*, pages 129–139. IEEE, 2023.

[MBWM24] Judith Michael, Dominik Bork, Manuel Wimmer, and Heinrich C. Mayr. Quo vadis modeling? *Softw. Syst. Model.*, 23(1):7–28, 2024.

[MS89] Jeffrey Mitchell and Ben Shneiderman. Dynamic versus static menus: an exploratory comparison. *ACM SigCHI Bulletin*, 20(4):33–37, 1989.

[PA18] Parsa Pourali and Joanne M. Atlee. An empirical investigation to understand the difficulties and challenges of software modellers when using modelling tools. In Andrzej Wasowski, Richard F. Paige, and Øystein Haugen, editors, *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2018, Copenhagen, Denmark, October 14-19, 2018*, pages 224–234. ACM, 2018.

[REIWC18] Roberto Rodriguez-Echeverria, Javier Luis Cánovas Izquierdo, Manuel Wimmer, and Jordi Cabot. Towards a language server protocol infrastructure for graphical modeling. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, pages 370–380, 2018.

[Sar23] Aylin Sarioğlu. Accessibility in web modeling tools: Systematic review and conceptualization of a keyboard-only prototype. Diploma thesis, Wien, 2023.

[SMB23] Aylin Sarioglu, Haydar Metin, and Dominik Bork. How inclusive is conceptual modeling? A systematic review of literature and tools for disability-aware conceptual modeling. In João Paulo A. Almeida, José Borbinha, Giancarlo Guizzardi, Sebastian Link, and Jelena Zdravkovic, editors, *Conceptual Modeling - 42nd International Conference, ER 2023, Lisbon, Portugal, November 6-9, 2023, Proceedings*, volume 14320 of *Lecture Notes in Computer Science*, pages 65–83. Springer, 2023.