



Integrieren von erweiterten Visualisierungs- und Interaktionsmethoden in Language Server Protocol basierenden Modellierungstools

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Giuliano De Carlo, BSc

Matrikelnummer 01526998

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Philip Langer

Wien, 13. Juli 2022

Giuliano De Carlo

Dominik Bork

Integrating Extended Visualization and Interaction Functionalities into Language Server Protocol Based Modeling Tools

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Giuliano De Carlo, BSc

Registration Number 01526998

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dipl.-Wirtsch.Inf.Univ. Dr.rer.pol. Dominik Bork

Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Philip Langer

Vienna, 13th July, 2022

Giuliano De Carlo

Dominik Bork

Erklärung zur Verfassung der Arbeit

Giuliano De Carlo, BSc

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, 13. Juli 2022

Giuliano De Carlo

Kurzfassung

Mit zunehmender Komplexität moderner Softwaresysteme und deren Entwicklung, entsteht der Bedarf nach einem effizienteren Einsatz von Modellierungssprachen. In den letzten Jahren haben Modellierungstools begonnen, sich vom traditionellen Rich-Client Ansatz in Richtung leichter und entkoppelter Systeme zu entwickeln, und damit modernere Technologie-Stacks wie die des Webs zu verwenden. Eine dieser Systeme ist die Eclipse Graphical Language Server Platform, welche das Konzept des Language Server Protocol nutzt, um eine Modellierungsumgebung in Client und Server zu unterteilen. Trotz dieser modernen Ansätze ist das Arbeiten mit räumlich großen Models oft umständlich und unproduktiv. Selbst die meisten modernen Tools bieten nur wenige Mittel, um große Models effektiv zu visualisieren oder mit ihnen zu interagieren.

Diese Arbeit befasst sich mit diesen Problemen in zwei Schritten. Der erste Schritt war, geeignete Mittel zu finden, um die Produktivität beim Arbeiten mit großen Models zu steigern. Um dies zu erreichen, haben wir uns Features und bestehende Forschungsergebnisse angesehen, die sich mit der Visualisierung und Interaktion von Informationen befassen. Darüber hinaus wird in dieser Arbeit eine Taxonomie vorgestellt, welche benutzt werden kann, um Visualisierungsfeatures zu klassifizieren und damit ihre Beurteilung zu erleichtern. Basierend auf den daraus gemachten Erkenntnissen, wurden zwei Features ausgewählt, die dann im zweiten Schritt konzipiert und in eine Graphical-Language-Server-Platform integriert wurden. Das erste Feature, *Semantic Zooming*, befasst sich mit der dynamischen Anpassung sichtbarer Informationen basierend auf dem vorherrschenden Zoom-Level. Das zweite Feature, *Visualisierung von Off-Screen-Elementen*, sorgt hauptsächlich für eine effizientere Interaktion mit Elementen die sich außerhalb des sichtbaren Teils des Bildschirmbereichs befinden. Mit Abschluss des zweiten Schrittes liefert diese Arbeit ein Konzept zur Integration beider Features in eine GLSP-basierte Umgebung. Darüber hinaus validiert es beide Konzepte, indem es eine erfolgreiche Umsetzung ihrer Integration in die Eclipse-GLSP in Form von zwei Prototypen bereitstellt.

Abstract

With an increasing complexity level of modern software systems and their development comes a need for a more efficient use of modeling languages. In the recent years, modeling tools have started to shift from the traditional rich client approach to lighter and more decoupled systems, and with that, use more modern technology stacks, such as that of the web. One of such environments is the Eclipse Graphical Language Server Platform, which utilizes the concept of the language server protocol to divide a modeling environment into client and server. Nevertheless, working with spatially large models is still often inconvenient and cumbersome. Even most modern tools offer few means to effectively visualize and interact with large models.

This work addresses these problems in two major steps. The first step was to find appropriate means that are able to increase the productivity while working with large models. In order to achieve that, we looked at features and existing research that deal with the visualization and interaction of large information. Furthermore, it presents a taxonomy which aids in the classification and evaluation of such features among three meta-characteristics. Based on these findings, two features were picked that were then conceptualized and integrated into a graphical language server platform in the second step. The first feature, *semantic zooming*, deals with the dynamic graphical adjustment of visible information based on the current zoom level. The second feature, *visualizing off-screen elements*, mainly provides a more efficient interaction with elements that are currently off screen. With the conclusion of the second step, this work provides a concept for the integration of both features into a GLSP-based environment. Additionally, it validates both concepts by providing a successful realization of their integration into the Eclipse-GLSP in the form of two prototypes.

Contents

Kurzfassung	vii
Abstract	ix
Contents	xi
1 Introduction	1
1.1 Motivation & Problem Statement	1
1.2 Aim of the Work	2
1.3 Methodological Approach	3
1.4 Structure of the Work	4
2 State of the Art	5
2.1 Conceptual Modeling	5
2.2 Language Server Protocol	7
2.3 Level-of-Detail	8
2.4 Panning & Zooming	13
2.5 Tools & Features	14
3 Taxonomy	23
3.1 Relevant Taxonomies	23
3.2 Research Approach	24
3.3 Terminology	27
3.4 Taxonomy Definition	32
3.5 Taxonomy Evaluation	42
4 Prototype	67
4.1 Theory	68
4.2 Conception	80
4.3 Evaluation & Discussion	90
5 Summary & Conclusion	99
List of Figures	103

List of Tables	105
Bibliography	111

Introduction

This chapter describes the problem that this work will be dealing with in more detail. It introduces the domain of this work and gives a short description of recent developments that provide help in solving the problem. Secondly, two research questions are given which is followed by an explanation of the methodological approach that we took to answer them. Lastly, it gives an overview about the remaining part of this work by giving a short description about the following chapters.

1.1 Motivation & Problem Statement

In the field of model engineering, the visualization of models and the interaction with them has always been an important aspect that allows for an effective communication between multiple stakeholders. Visual representation has a great impact on the ability of our brain to efficiently process information [Moo09], and good interaction methods allow for fast and efficient execution of work tasks. For this reason, a variety of different modeling tools have been developed in the past, like the Eclipse-based Sirius [VMP14] or Graphiti [eclb] tools. Because of their focus on the visualization aspect, they allow users, even those that do not have a lot of knowledge in the computer science field, to create and edit models in all kinds of domains. Regardless of how popular and widespread they are, they all lack in the following areas: (i) *visualization* of large models, and (ii) *interaction* with large models. One of the main reasons for that, is the technologies that they are based on. Many modeling tools that are used nowadays are built upon old technologies that only allow for simple interaction methods like basic zooming and panning. On top of that, only very few research publications can be found about user interface design for modeling tools [TRS21] and visualization techniques used in conceptual modeling have barely evolved in the last years [GRGS15, Gul16]. Part of this work will be to provide an overview of advanced visualization and interaction methods that can be used to improve the usability when working with large models.

In order to overcome the problem of having a foundation that is based on old technology, new approaches have to be developed that are built upon more recent technology stacks. Examples for this are systems that utilize the very recent language server protocol (LSP). LSP is a client-server approach that replaces today's mostly monolithic IDEs that are used in the field of software development. It has mainly been developed to counteract the n-to-m complexity of having to individually integrate every programming language into every code editor by decoupling the language-specific logic from the usability-centric text editor and therefore lowering the complexity to n-plus-m instead [Bün19].

The idea of a language server is not only restricted to programming languages, it can also be applied to the field of model engineering in the form of a graphical language server. A graphical language server protocol (GLSP) follows the same concept as LSP, but instead of focusing on lines of code, graphical representations of models like ERD or UML are edited on the client and transferred to the server to change the underlying models. Because of the modern technology stack, namely HTML5, that can be utilized in combination with GLSP, a lot of improvements to the user interface and interaction methods can be made that are not easily possible in traditional model engineering tools. HTML in combination with CSS and JavaScript has been used to power the web for multiple decades now. For that reason, it has become one of the best systems to easily and efficiently implement advanced interaction methods. Another major part of this work will be to provide an idea of how to integrate advanced interaction methods into the graphical language server protocol. This will be realized in the form of a prototype based on an environment that utilizes HTML, CSS, and JavaScript.

1.2 Aim of the Work

The general aim of this work is to combine advanced interaction methods with the graphical language server protocol (GLSP). The interaction methods that this work will focus on, are methods based on zooming in combination with level-of-detail functionalities. This combination is also known under the term *semantic zooming* [SKA94, MS95, Spe14]. A significant part of this work will be to, on the one hand, give a broad overview of relevant methods in other fields of work, and on the other hand, document what interaction functionalities exist and have already been evaluated in scientific environments. In order to get a broader view, this work will also look at and evaluate comparable tools outside of the scientific world. This information will then be used to create a taxonomy, which can be used to classify advanced information visualization features. This taxonomy is supposed to be of great benefit for researchers by providing a common frame to position their works, and for method engineers and modeling tool developers by sparking innovation in future modeling tools. All the functionalities and methods that were found, will then be evaluated and categorized into the taxonomy of this work. With this gained knowledge, they will then be evaluated based on how well they fit into and can be applied to a GLSP-based environment, and how much a user would benefit from them.

A set of features will then be selected and implemented in a GLSP solution as a prototype.

The GLSP implementation that this prototype will be implemented in, is the Eclipse Graphical Language Server Platform (Eclipse-GLSP) [ecla]. The initial step of this implementation will be to develop a concept to integrate the selected feature into the GLSP workflow. Because a wide range of different language servers are supposed to be implemented and supported in the future, a focus during the implementation of the prototype will be a good integration into the system. This means that the result should be a generic solution that enables efficient future use with arbitrary modeling languages. After the implementation of the prototype is finished, it will be evaluated once again. This evaluation will try to give answers to the following questions:

1. What is an appropriate means to improve the visualization of large models and interaction with them in GLSP-based modeling tools.
2. How to generalize the concept/solution towards being applicable for other modeling languages/GLSP-based modeling tools.

These topics should give an overview about whether the features should be implemented on a larger scale in existing solutions like the Eclipse Graphical Language Server Platform. Finally, general insight, related to problems, limitations, new ideas, and future work will be given.

1.3 Methodological Approach

This work mainly follows the approach of the Design Science research methodology [HMPR04] and consists of the following steps:

1. Literature Review

The current state and existing knowledge related to the topic of this thesis will be established and documented. The initial step will be to find domains that deal with large information spaces and document the techniques that they apply and why they are required. This will be followed by a review of existing modeling tools and a documentation about how well they support the visualization of large models. Finally, because it will be a major part of this work, a review of the graphical language server protocol will be given, which will establish the current state of the protocol.

2. Exploratory Study

An exploratory study will be conducted to create an overview of visualization- and interaction-based features in tools that are being used today and concepts that were found in past literature. The scientific contribution of this study will be a taxonomy. The goal of this taxonomy is to define classes which can be used to categorize advanced interaction features that are relevant to language server protocol based environments.

3. Taxonomy Development

In order to create a taxonomy that meets today's standards, it will follow the

guidelines of Nickerson et al. [NVM13] and their recent extensions by Kundisch et al. [KMO⁺21]. The main steps of the development of this taxonomy will be: *(i)* definition of ending conditions, *(ii)* definition of meta-characteristics, *(iii)* multiple empirical-to-conceptual or conceptual-to-empirical iterations, *(iv)* ex ante and ex post evaluation.

4. Analyzing Study Results

The collected data will be categorized, analyzed, and interpreted. Key points of this analysis will be the ability to implement and use these features in a GLSP environment and also the estimated gain for a user in areas like usability, efficiency and productivity. This analysis will be done in the form of a descriptive evaluation based on informed arguments, as described in [HMPR04].

5. Designing new Artifact

Based on the analyzed data, a prototype will be implemented that demonstrates state-of-the-art interaction methods in the Eclipse Graphical Language Server Platform.

6. Artifact Evaluation

The prototype will be analyzed and evaluated based on the questions that have been given in Section 1.2. This will be done in the form of a descriptive evaluation based on informed arguments and scenarios, as described in [HMPR04].

1.4 Structure of the Work

This thesis consists of four further chapters.

Chapter 2 describes the state of the art. It gives an overview of the current state of relevant domains, such as conceptual modeling and the language server protocol. Furthermore, it summarizes Eclipse-GLSP, and relevant features of existing modeling tools.

Chapter 3 presents the taxonomy that has been conducted in the course of this work. It describes the approach that has been used to develop it, all its characteristics, and its evaluation. It also summarizes all features that have been used during the evaluation.

Chapter 4 goes into detail about the developed prototypes. It describes the theoretical concept of both prototypes, followed by technical details about each one. Finally, a descriptive evaluation will be given, which addresses their strengths, weaknesses, and limitations.

Chapter 5 concludes this work with a summary and a short overview about potential future work.

State of the Art

The purpose of this section is to describe relevant domains that this work deals with in more detail. Among them are conceptual modeling, the language server protocol, the Eclipse Graphical Language Server Platform, and semantic zooming. The second part of this section deals with state-of-the-art modeling tools, and their interaction and visualization features. Finally, it provides a summary of observed features and categorizes them into groups of similar functionalities.

2.1 Conceptual Modeling

The usage of data models to abstract implementation details has been a long-known concept. It has been revolutionized in the 1970s, mainly by using it to hide implementation details of the definition of a database [Myl92]. Since then, it has gone through many adaptations and improvements, and has been used in a wide range of other domains as well. A widely agreed on definition of the term *conceptual modeling* is defined as "...the activity of formally describing some aspects of the physical and social world around us for purposes of understanding and communication." [Myl92, p. 3] in 1992. Its purpose is to abstractly describe certain real-world domains by graphical means, so that all relevant aspects of this domain can easily be understood by all stakeholders. Stakeholders are all people that are somehow involved in the described aspects, which includes people that already have a large knowledge base of the domain, but also those that are new to it with very little or even no knowledge.

Another more recent definition of a conceptual model in the domain of simulation studies is: "The conceptual model is a non-software specific description of the computer simulation model (that will be, is or has been developed), describing the objectives, inputs, outputs, content, assumptions and simplifications of the model." [Rob08, p. 283]. To further understand the process of conceptual modeling, they define three artifacts [Rob08, KR08, Rob13]: (i) *System description*, (ii) *Conceptual model*, (iii) *Computer model*. The system

description describes the problem and the system that it resides in. Through abstraction, the conceptual model is created, which helps all stakeholders understand this problem and its system. Finally, the computer model is designed and developed, which represents the conceptual model in the form of a software specific model.

One of the main reasons why conceptual modeling is rising in relevancy and going through many re-definitions, is that most real-world problems become increasingly complex over time, which causes the systems that are developed to solve them to become more complex as well. More difficult problems call for more efficient solution processes to keep the relation between increasing problem complexity and cost of time/money close to linear. On top of that, the larger a project is, the more errors are committed during the requirements- and design-phase [McC04]. Errors during the early phases of a project can be very costly, depending on the phase that they are discovered in (by average about 2-4 times more expensive than errors committed in later stages [McC04]). The usage of models for the communication between stakeholders helps to detect such errors early on.

In order to achieve this, complex systems are modeled in an abstract and simplified form, which ideally helps all stakeholders understand the system better. Conceptual models should aid in replacing some cognitive tasks (understanding) with simpler perceptual ones (seeing), and thus cause a form of computational offloading in our minds [BR21]. This is done by utilizing visual notations, which can be processed much more efficiently by our brains than text-based information [Moo09]. In his work, Moody [Moo09] raises awareness about the importance of visual representation issues in notation design, and defines certain principles for effective visual notations. All in all, little research has been conducted in the past years in the area of visualization of conceptual models. This is also reflected by the fact that many visualized models today still look very similar to their versions of decades ago [BKP18]. While this does usually not pose big problems for domain experts, because they are generally able to understand even poorly laid out diagrams [RM10], it will likely prevent people in non-technical domains from understanding them. In order to overcome that, it has been suggested to systematically combine conceptual and visual thinking by examining and picking up knowledge from other scientific fields, such as graphic design, interaction design, cognitive science, gestalt psychology, and philosophy of mind [GRGS15]. Besides the visual aspect, other factors that have a large impact on the understandability of models are their size, density, and average connector degree [RM10].

In the recent years, the idea of having models only be used by humans, as originally stated by Mylopoulos [My192], has shifted towards both, usage by humans and machines. Instead of using models purely for their representative means, they are also used as a formalized knowledge base that enables automated processing [BR21]. Model-driven engineering describes the integration of models into the process of developing software systems. This is accomplished by transforming models into first-class citizens [BCW17] in the software development processes and using them for more than just the purpose of communication between stakeholders. In combination with the correct tools, formally defined models can be transformed into other software artifacts. Bork and Fill refer to a formal model as one that is intersubjectively understandable and enables machine processing [BF14]. To

formally and precisely describe models, usually multiple meta-models are required [BKP20], which add another level of abstraction. This abstraction aids in, e.g., making the extension of existing languages possible, or allowing for the re-usability of certain transformation tools. Examples for the application of model transformation are the automatic generation of Java classes or SQL schemas from abstract models.

2.2 Language Server Protocol

Even though the traditional client-server architecture has started to gain acceptance in the late 1980's [Sch95] and has since then been applied to many major developments such as the web, only recently has it reached the world of software development in the form of language servers. The language server protocol is a very recent scientific topic that was introduced by Microsoft, RedHat, and Codeenvy in 2016 [Bün19]. The idea behind it, is to be able to split up today's heavy-weight monolithic IDE approaches into a server and a client. Between these two components, the language server protocol acts as a standardized way of communication to synchronize client and server. Currently, version 3.17 of the protocol describes about 40 different messages and has an implementation for over 100 different programming languages/technologies. [mica, mich]

Initially, the language server protocol has only been defined and used for text-based languages, but it was quickly discovered that this concept can also be applied to other areas. One of them is the area of model-driven engineering. Here, representations of models are often designed and expressed by graphical elements. Instead of specifying a protocol based on writing and editing text, a new protocol has to be created which is designed to handle graphical representations. There is currently no standardized protocol for graphical representations and the question whether LSP can and should be extended to support them does still not have a concrete answer. There exist theoretical ideas, concepts and even implementations for graphical language server protocols. Rodriguez-Echeverria et al. [REIWC18] describe how the concept of the text-based LSP can be applied to graphical-based languages. Furthermore, examples for web-based modeling tools that follow a similar concept to that of LSP have been created in the past, e.g., [LCP16], [HHFN13], [RKP12] or [TSS09]. There also exist first implementations that try to make the concept of a language server work on graphical representations. Examples are a protocol [obe], developed by the French company Obeo and the Eclipse Graphical Language Server Platform [ecla].

2.2.1 Eclipse Graphical Language Server Platform

The Eclipse Graphical Language Server Platform (Eclipse-GLSP) is an open-source framework which uses an LSP-like protocol to enable diagram editing via a client-server architecture. The server is responsible for model management, the model logic, validation, and supplies changes to the model(s). In order to do this, it supports and uses the Eclipse Modeling Framework (EMF), which already has many languages and their language-specific logic implemented. The server is mainly written in the programming language Java, and

exposes an inter-process communication interface, either via WebSockets or TCP socket connection.

The client is mainly responsible for rendering the graphical representation of a model and handling user interactions. Graphical elements are rendered as an SVG element inside a browser. This is done with the help of the Sprotty framework¹. Sprotty is written in TypeScript and its main responsibilities are rendering graphical elements, playing animations between state changes, and handling and dispatching actions. Actions are used to perform certain operations and are either handled by the client itself, or the model source. The model source is responsible for modifying the model and can either be a locally instantiated class, or a remote server. Actions could, for example, be an *UpdateModelAction* to update a model at the client, or a *CenterAction* to move the viewport so that a specific object is centered.

The communication between server and client utilizes a protocol similar to Microsoft's LSP. As mentioned above, the communication between both entities is based on actions. While many actions are reused from the Sprotty framework, e.g., for model transfer and client-local actions, many new ones are added as well. They include model-specific actions, such as *CreateNodeOperation* which adds a node to the current model, as well as editor-specific actions, such as *SetClipboardDataAction* which copies data to the clipboard, and *UndoOperation/RedoOperation*. A full list of operations can be found in the GLSP protocol specification². A more detailed description of the Eclipse-GLSP can be found in Section 4.1.2.

2.3 Level-of-Detail

The level of detail forms an integral part of advanced visualization features and will be a main topic in this work. Levels of detail can be described as multiple different representation of one or more objects. Often, depending on various factors, such as, e.g., the current zoom level, the distance, or the importance of an object, a different representation is displayed. This can help reduce the complexity of the displayed information in certain situations and help make their processing easier, not only for our brains but also for machines.

The utilization of multiple levels of detail goes back into the 1970s with works like Donelson's "Spatial Management of Information" [Don78], which describes an information management system with multiple displays. One display shows specific information of the information space and another shows a shrunken version of those information to help with navigation. Another influential work is the interface model "Pad" [PF93] from 1993. Pad is an interface shared among multiple users, which shows information in multiple levels of detail. It uses different views - or what they call portals - to show more details about specific parts of an information source. For example, instead of showing an entire text paragraph that is too small to be read, only the title is shown in a readable size. Based on these concepts,

¹<https://github.com/eclipse/sprotty> (Accessed: 06.03.2022)

²<https://github.com/eclipse-glsp/glsp/blob/master/PROTOCOL.md> (Accessed: 06.03.2022)

many other ideas were brought up. Among them are fisheye views [Fur86, BHDH95, RMG07], semantic zooming [PF93, FDB08, Kal08, LMC⁺02, OBEL10], or concepts like lenses [TAVHS06] and off-screen object visualization [ZMG⁺03].

The utilization of levels of detail is not only used in the domain of interface design. Other areas that heavily rely on multiple detail levels are computer graphics and geographic systems.

2.3.1 Level-of-Detail in Computer Graphics

Level-of-detail has always been a big topic in the field of computer graphics. The idea is to create 3D-models, mainly in domains such as video games, geographical information systems (GIS), and city modeling, in different complexity levels. A model's complexity is usually measured by the number of polygons that it consists of and the main benefits of having models with a low number of polygons are the reduced storage requirements, the reduced computational complexity, and faster transmission over the network [HD04]. One challenge with that is to find the correct balance between model complexity and realism, which becomes easier with the improving computational power that our systems displays. Besides increasing a system's performance, there are other techniques to make models look realistic but still perform well. Such techniques can usually be applied to meshes of 3D-objects and are, for example [LRC⁺03], vertex-pair collapse (connecting two unconnected vertices), triangle collapse (collapsing a triangle to a single vertex), or vertex removal (removal of vertexes and its edges, and triangulation of the resulting hole). Furthermore, they define four different level-of-detail frameworks [HD04, LRC⁺03]:

- **Discrete level of detail:** Multiple versions of an object are created in multiple different levels of detail prior to run-time. At run-time, the appropriate level is chosen and displayed.
- **Continuous level of detail:** A data structure is created which allows to dynamically extract versions of an object in different levels of detail during run-time.
- **View-dependent level of detail:** It extends the continuous level of detail. Instead of seeing an object as one single entity, different parts of an object are rendered differently. E.g., nearby parts of an object are rendered in more detail while distant parts are rendered in less detail, which is especially useful for large objects.
- **Hierarchical level of detail:** It allows to create clusters of multiple, usually small, objects. These clusters are then treated as one single object and can, e.g., be replaced all at once at a specific distance.

In order to efficiently use those frameworks, they have to be applied at the correct time. While this can be done by applying different settings and making that decision manually, for larger projects, an algorithm has to be used. Such algorithms often depend on various properties, such as [HD04]: object size, object eccentricity, object velocity, target frame rate, human eye limitation, environmental conditions, or visual importance.

2.3.2 Level-of-Detail in Geographic Information Systems

The term *geographic information system* has many definitions, one of them being "information technology which stores, analyzes, and displays both spatial and non-spatial data" [CC88, p. 1547]. Level-of-detail is also very relevant in this area. Mainly to render and show the terrain of the world's surface in different levels of detail. Especially in the past, this has been necessary because computers were not powerful enough to render the surface with a lot of detail. Although not at the same scale, this problem still persists today and will likely stay with us for a long time because of how large and detailed the earth is. Tools like Google Maps and Bing Maps still apply techniques that involve multiple levels of detail to significantly simplify the earth's surface.

When rendering terrain, the detail of the surface almost always depends on the view that a user has. For example, when the world is rendered and shown in its entirety in a digital setting (at a scale of approximately 1:40 million), its terrain can be rendered on a flat surface because differences in terrain height will barely be visible at that point. The further the user zooms in, the more details would become noticeable and should be rendered. It does not only depend on the zoom level but also on the angle that information is displayed. For example, a top-down view (90°) of the earth does not necessarily need to have the terrain rendered, because, similarly to a 2D map, it cannot be seen. When the user changes the view to be angled ($< 90^\circ$ or $> 90^\circ$), terrain becomes visible and could therefore be rendered in more detail.

Google Maps

Google Maps is a mapping tool meant to be ran inside browsers. Because of its relevancy to this work - mainly because of the fact that it works with multiple levels of detail and is ran inside a browser - we looked at it in more detail. Besides many other features, Google Maps offers to show a satellite view of the world. The user has the ability to interact with the map by scrolling and zooming. Because of how big the world is, LoD methods are necessary to deliver map tiles to the client in a manageable fashion. Without splitting up images into different levels of detail, not only would sending the images be a problem, the client/browser would also have troubles displaying them. The bandwidth needed to send/receive images would be too high for today's standards and browsers would run into performance issues trying to display all of them.

In order to show the map, Google Maps serves map tiles in the form of 256x256 pixel JPEG images. Depending on the client's current zoom level and position, they serve images in different levels of detail. There are about 22 different zoom levels available ³. The exact number depends on the position that is currently viewed. Cities, for example, usually have a higher level of detail than the sea or generally unpopulated areas. Zoom level 0 (LOD0), although it cannot be accessed through the Google Maps interface directly, shows the entire planet on one 256x256 pixel tile and zoom level 21 (LOD21) shows the surface accurate enough to easily make out cars. Every additional zoom level n is four times as

³<https://medium.com/google-design/google-maps-cb0326d165f5> (Accessed: 06.03.2022)

detailed as the previous level $n - 1$. A 256x256 pixel tile of, e.g., LOD10 can be split up into four 256x256 pixel tiles of LOD11. While there is only one tile available at zoom level 0, theoretically, there are exactly $4^{21} = 4\,398\,046\,511\,104$ tiles available at zoom level 21. This can be visualized as a pyramid of images⁴. LOD0 makes out the top, followed by LOD1 which is four times as big as LOD0 all the way down to LOD21. This so-called pyramid image is not only used by Google but in a lot of other areas related to machine vision or image processing in general [AAB⁺84].

The data structure used to save those images is called a quadtree, or more specifically, a region quadtree. Quadtrees describe a class of hierarchical data structures which are based on the principle of recursive decomposition of space. Besides region data, they are also commonly used for point data, curves, surfaces and volumes [Sam84]. The idea is to recursively split data into four quadrants. This can be visualized as a tree where every node, except the leaf nodes, has four children. In case of a region quadtree, every node represents an image, and the zoom levels are represented by the depth of the tree. E.g., images of LOD4, which are at zoom level 4, are located at nodes with depth 4 in the tree.

In order to retrieve images inside a quadtree, different approaches can be used. The Google Maps client uses simple HTTP GET requests to download the images. These requests consist of three main parameters:

- **z**: It is used to define the depth of the tree and therefore the zoom level.
- **x**: It represents the x coordinate of a tile at a specific zoom level.
- **y**: It represents the y coordinate of a tile at a specific zoom level.

If all tiles of a zoom level are represented inside a two-dimensional cartesian coordinate system, these positive coordinates reference one specific image on that zoom level. The deeper the zoom level, the higher these coordinates can be. E.g., $x = 0, y = 0, z = 0$ requests the first image of zoom level 0. Because zoom level 0 is the very top of the pyramid, a request with these parameters serves the only image that exists for this level, which is a 256x256 pixel image that contains earth in its entirety. $x = 3, y = 2, z = 2$ would request a tile of zoom level 2. The third zoom level displays earth on $4*4=16$ images. Therefore, coordinates $x = 3$ and $y = 2$ serve the image in the bottom right part of the world map, which shows Australia and the area around it.

On top of the surface tiles, metadata is added at each zoom level, if requested. Metadata can, for example, include street names, names of popular places or marker for restaurants. Served metadata also depends on the zoom level. It makes no sense and would not be feasible to, e.g., display names of all streets that are theoretically visible on LOD0 because there are simply too many of them. On LOD21 on the other hand, it makes sense to even display house numbers without cluttering the view for the user. In the recent years, Google also added 3D models that are added on specific zoom levels. Table 2.1 roughly shows what elements are added at which level of detail.

The client is responsible to request the correct tiles according to the zoom level and display them seemingly next to each other to make it look like one big map. The scene consists of

⁴<https://developers.google.com/earth-engine/guides/scale> (Accessed: 06.03.2022)

2. STATE OF THE ART

Level of Detail	Newly visible objects
LOD1	Country borders
LOD4	Country names, Oceans names
LOD6	Names of big cities
LOD7	Names of highways and other big roads
LOD9	Names of rivers
LOD10	Names of medium roads, Landscape names (forests, lakes, ...)
LOD13	Names of smaller areas (golf courses, famous buildings, ...)
LOD14	Street names
LOD15	Restaurant and other POIs
LOD19	House numbers

Table 2.1: Newly added metadata in different levels of detail in Google Maps. May differ depending on the area that is displayed.

an HTML5 Canvas object. Downloading a lot of images, as it is necessary with Google Maps, requires a lot of bandwidth. The user is able to zoom in, zoom out, and scroll the map on the same zoom level. Different techniques are used to increase the user experience and avoid a popping effect.

Zooming in: When the user zooms in, the new images are requested via an HTTP GET Request. While the new images are sent, the client keeps the current images visible and increases their size according to the new zoom level. Although this can make the map seem blurry, it stops the scene from going blanc for the duration until the new images are loaded. Once they arrived, the old images are replaced with the new ones. To avoid a popping effect between images, a quick transition effect is applied.

Zooming out: Zooming out follows the same strategy as zooming in does. When the user zooms out, the current images are kept until the new images are loaded. The difference is that the old images are already at a better quality than the new images because they are at a higher level of detail. This means that, unlike the blurry map that is shown when the user zooms in, the map stays at high detail level for the entire duration of the transition. Because of that, the popping effect of the map tiles is almost nonexistent, only the adjustment of metadata (e.g., the removal of street names) is clearly visible. This also comes with a disadvantage. It is possible that the client now has to display an area which it does not have any map tiles for because the area has not been visible so far. Here, the client has no choice but to display a blanc stage until the map tiles are fully loaded from the server. The client always loads some additional map tiles for the areas that are just outside of the viewport of the client which is the reason why this only happens when the user zooms out a significant amount.

Scrolling: When the user scrolls through the map, new areas have to be displayed by the client. As mentioned above, the client always requests additional map tiles for the border around the current viewport. This means that when the user scrolls only a small amount into a direction, the tiles are already available and can be displayed immediately without any transitions or popping effects. In the background, and ideally not noticeable for the user, new tiles are requested that cover the new border around the current viewport.

When the user scrolls further, to a point where no tiles of the current zoom level are loaded anymore, the client shows tiles of two levels below the current level. On initial page load, or when the user zooms/scrolls the map, additionally to the tiles of the current zoom level, the client also requests tiles of level $n - 2$, where n is the current level. These tiles are not displayed immediately and therefore are of low priority. Because of that, they are only requested once all the tiles that have to be displayed immediately have finished downloading. If the user now scrolls outside of the border region for which tiles are available on the client, these tiles are shown instead. They are only shown temporary until the images of the current zoom level have arrived. Because these temporary tiles are of a lower zoom level, they also look blurry. If the user scrolls a significant amount outside of the viewport, to a point that even the images of a lower zoom level are not available anymore, the client has again no choice but to display a blank map until the tiles have arrived.

2.4 Panning & Zooming

Panning and zooming are interaction techniques that have been around for a long time. Over the years it has become the norm to integrate them into almost all applications that display information which are too large to be shown all at once on a screen. For that reason, users understand their concept well and are able to instinctively apply them. Panning was one of the earliest navigation techniques. It describes techniques that are used to move the viewport around to get a different view onto a document. This can be done with the help of, e.g., scrollbars, the mouse wheel, or touch gestures. Zooming describes the magnification or demagnification of information and is often done when basic panning becomes too tedious or information is too small to be read. Zooming is normally also performed with either the mouse wheel, simple buttons, or touch gestures.

An extension of the ordinary zooming interaction is the concept of a semantic zoom. Semantic zoom describes the adjustment of the level of detail of visual object representations, based on certain conditions, such as the importance of an object or its current zoom level. The concept of semantic zoom has been around since the 1990s. The authors of the already mentioned Pad [PF93] and Pad++ [BH94] have been one of the first to integrate semantic zooming into an application. Their application consists of "portals" that can be used to see certain parts of information differently. Different portals show information at different magnification levels. The term *semantic zoom* has been defined numerous times in past literature. For example, the definition by Sengupta et al. [SKA94, p. 133], "Semantic zooming increases or decreases the level of detail by methods that depend on the function,

or semantics of programming objects", or a more recent one by Spence [Spe14, p. 141-142], "With semantic zoom, objects – or, more generally, representations of data – are now not constrained to change only their size. They can change in colour, shape, presence and texture, and they can offer a new selection and/or structure of represented data...". Over the years, the concept has also been applied in many different areas, not only in the field of model engineering with examples based on UML [FDB08], but also in various other fields like software development [SFA⁺11, YM15], parallel computing [Kal08], video editing [LMC⁺02], text documents [OBEL10, Dun09], or the medical field [KBRCn02].

2.5 Tools & Features

The last part of our state-of-the-art analysis was to explore today's modeling tools to obtain an overview about the visualization and interaction features that they offer. During this analysis, we looked at tools and libraries that can be used to create visual models and can be ran either inside a browser, or as a native desktop application. A list of all tools that were looked at can be found in Appendix 1 (all tools in the categories drawing tools, modeling tools, and meta-modeling tools). A summary of their features will be given at the end of this section. Most tools offer only the basic and widely known features, such as simple scrolling/panning/zooming, and only very few provide advanced functionalities. The most advanced one that we could find were tools provided by yWorks⁵, a company that focuses on data visualization. Of all tools that we looked at, they provided the best explanations and examples on how to integrate advanced interaction and visualization features based on level-of-detail and zooming. Because of the relevancy that it has to this work, the following section will go into more detail.

2.5.1 yFiles

yFiles is a multi-platform software library, designed to create graphs. Besides just creating graphs, they also offer functionality and algorithms to analyze, view, export, and edit graphs. It is available for five different technologies, HTML, Java, JavaFX, WPF, and .NET, and is used as the underlying layer of tools like yEd and yEd live. This section will focus on the HTML library because HTML is also utilized in GLSP.

yFiles for HTML is based on JavaScript and it is recommended to be used in combination with TypeScript. Entire diagrams can be created with either predefined or completely customized styles. It uses the SVG format by default, but it is also able to render diagrams on an HTML5 Canvas elements or even WebGL, which is recommended for larger and complex diagrams or diagrams that need special effects or graphics. It is also possible to customize or develop special user interaction methods. Besides common user interactions like zooming, panning or scrolling, other methods like adding/editing labels or drag-and-drop are supported out of the box. Interaction methods that are not available can be added by either customizing existing methods or developing their own.

⁵<https://www.yworks.com/> (Accessed: 06.03.2022)

Unlike many other libraries, yFiles for HTML comes with examples and sample source code that demonstrates level of detail features. They give three examples of applications that implement such features, along with their source code: Organization Chart⁶, Hierarchic Grouping⁷ and Collapsible Tree⁸.

These features can be summarized as:

Showing additional information based on zoom level Multiple different zoom levels can be defined. For example, a detail level, an intermediate level, and an overview level. All three levels consist of a threshold and a style. The style defines the looks of the SVG element that will be rendered. The threshold defines the maximum zoom level until which the corresponding style will be applied. When the user zooms in or out, the zoom level changes and is compared to all defined thresholds, starting from the lowest. In case the current zoom level is below or equal a threshold, the corresponding style is applied and the element is rendered on the stage. An example can be seen in Figure 2.1.



Figure 2.1: Semantic zooming in yFiles. The same object is represented in three different levels of detail. They are rendered depending on the current zoom level.

Source: <https://github.com/yWorks/yfiles-for-html-demos/tree/master/demos/03-tutorial-application-features/level-of-detail-style> (Accessed: 06.03.2022)

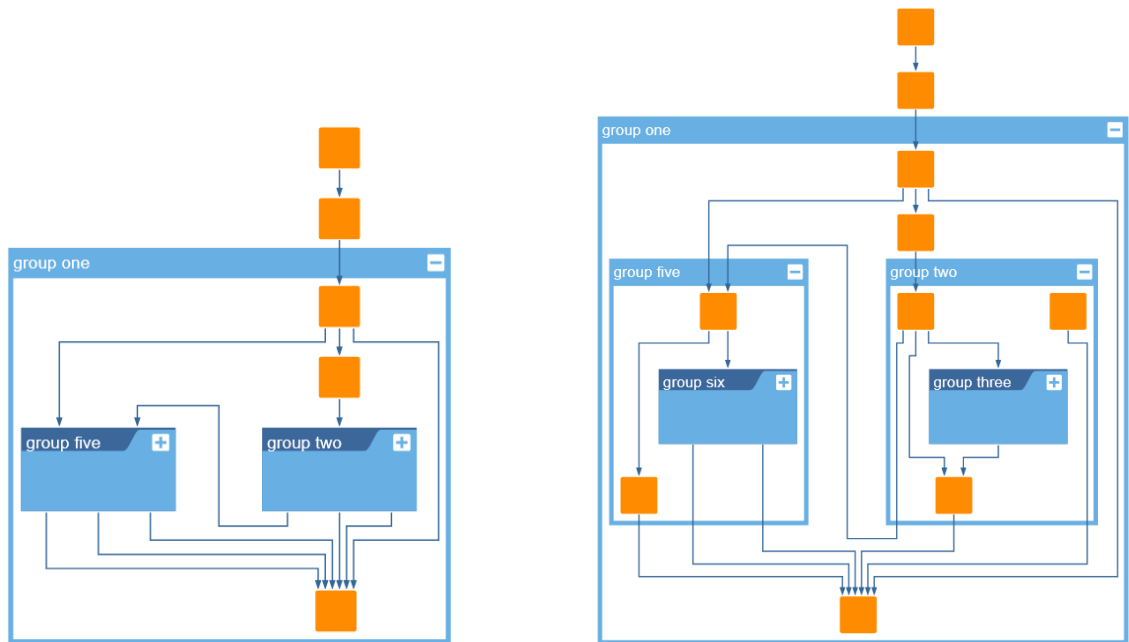
Grouping Elements Elements can be added to groups. These groups can then be collapsed and expanded, which hides/shows all elements inside them. Expanding and collapsing groups is a built-in feature in yFiles. This feature can be used out-of-the-box,

⁶<https://live.yworks.com/demos/complete/interactiveorgchart/> (Accessed: 06.02.2022)

⁷<https://live.yworks.com/demos/complete/hierarchicgrouping/> (Accessed: 06.03.2022)

⁸<https://live.yworks.com/demos/complete/collapse/> (Accessed: 06.03.2022)

but it is often not behaving in the exact way as expected. Depending on the model that it is used on, additional logic has to be added. This logic may include, for example, visual styles of groups, user interactions with a group, or the movement and position adjustment of elements inside a group. In the example given in their documentation, the logic makes sure to preserve information about edges that are visible when inside an expanded group but hidden when collapsed. Once a group is expanded from a collapsed state, this information will be used to restore the state and position of those edges and make the diagram look similar to its original state. An example can be seen in Figure 2.2.



(a) Group one expanded. Group five and two visible but collapsed. All other groups invisible.

(b) Group one, five, and two expanded. Group six and three visible but collapsed. Group four invisible.

Figure 2.2: Example for grouping elements. This diagram consists of six groups. Collapsing a group hides the elements inside it but preserves the relationships by changing their origin/destination to the group itself.

Source: <https://live.yworks.com/demos/complete/hierarchicgrouping/> (Accessed: 10.07.2021)

Expanding/collapsing branches This feature is similar to the grouping of elements and is implemented in a similar way. Instead of defining a group, all children of an element can be hidden by the click of a button which is part of the parent element. Every parent element and its children can be seen as one group that can be expanded or collapsed. The difference between this feature and the grouping feature is that here, the group is not directly visible on the stage. It is implemented by adding an event-listener to the click event of the expand/collapse button. Once the listener is triggered, the visibility states

of all children are toggled between true and false, and the style of the parent element is changed to display the collapsed or expanded state. On top of that, every time an element is expanded or collapsed, all other elements inside the diagram are updated and moved to a new position to keep the current layout small and without huge gaps in between. An example can be seen in Figure 2.3.

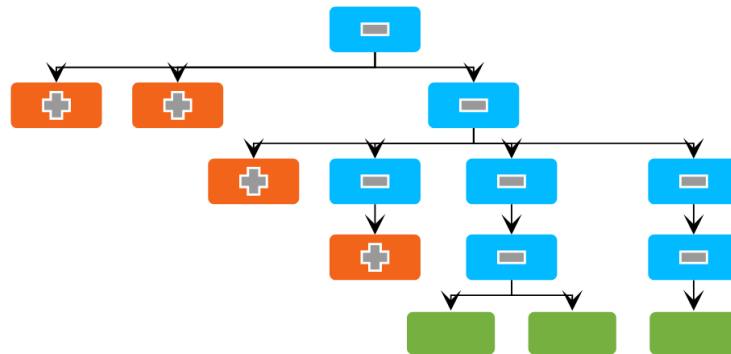


Figure 2.3: Tree-view in yFiles which can be expanded/collapsed by a user. Clicking on a plus sign expands the current node and clicking on a minus sign collapses it.

Source: <https://live.yworks.com/demos/complete/collapse/> (Accessed: 06.03.2022)

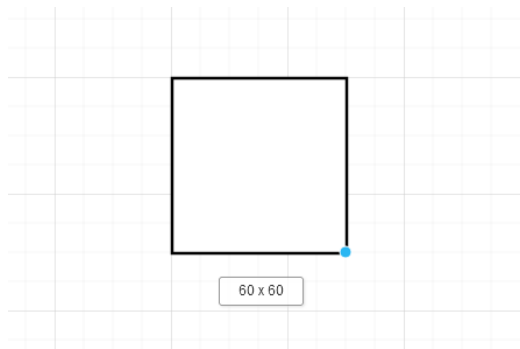
2.5.2 Summary of Zooming & Panning features

This section gives a summary of the features related to zooming and panning that were found during the evaluation of the modeling tools. A list of all tools that were looked at can be found in Appendix 1 (marked with *).

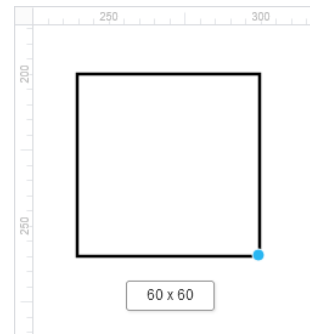
Grid and Ruler

A simple feature, which is available in almost all diagram software solutions, is the ruler and grid system. The grid splits the workspace into discrete pieces which makes it easier to consistently position shapes and forms. When the user changes the size of an element with their mouse by dragging its borders, the cursor is usually locked onto or snaps to the lines of the grid. The interval in which the workspace is split up is different from tool to tool and, in some implementation, is also dependent on the current zoom level.

The ruler is represented by a small bar which is positioned around the stage. It shows measurements, usually in pixel, that can help the user to accurately position or size an object on the stage. It is also helpful to measure distances between objects or simply get a better feeling for the scale that the stage is currently in. Selecting an element on the stage adds small indicators to the ruler. These indicators represent the edges of an element and can be used to accurately measure the element. It is directly dependent on what the viewport currently displays. Panning or zooming is directly reflected on the ruler.



(a) Grid feature. The empty space on the stage is represented by a grid. This makes positioning and aligning elements easier. One small square represents 10 pixels.



(b) Ruler feature. The Ruler can be seen on the left and top side of the stage.

Figure 2.4: Examples for grid and ruler features in the diagramming tool on <https://app.diagrams.net> (Accessed: 06.03.2022)

Size Adjustments of Elements

A feature that is not often seen in diagram/modeling software. Elements, especially text, are kept at a size that is relative to the viewport instead of the zoom level. This has the effect of, e.g., text staying at a constant size, no matter how much a user zooms in or out. It can increase the readability of elements that would otherwise become too small to be read.

This feature is essentially the opposite of the usual and expected zooming functionality, which is the increase or decrease in size of elements. Because of that, it is not often seen or only in very few parts of an application. It is relatively common in areas that are not directly part of the model or diagram, for example, buttons or metadata. It is also possible to combine the normal zooming functionality and this feature. For example, the title of a shape can stay at a constant size in a specific zoom level interval. Below or above the limits of that interval, the text increases/decreases relative to the zoom level.

Minimap

This feature is often seen in diagram/modeling software and is also popular in other domains. While the main view only displays a specific part of a document, the minimap displays the entire document inside an additional view which is usually positioned in the bottom right or left corner of the user interface. The information that is displayed is often the same as what is shown inside the main view but zoomed out by a large factor until it fits the size of the minimap view. Sometimes, minimaps also show information in a lower level of detail because showing all information would clutter the available space. Another valuable piece of information that this feature often provides is data about the area that is currently visible inside the main view in relation to the entire model. This is usually

indicated by a square that is rendered on top of the visible information inside the minimap. Borders of the square represent the edges of the viewport.

It also supports interaction by the user. Similar to a scrollbar, clicking and dragging the square to a certain location inside the document is reflected by moving the viewport of the main stage to the exact same location. Its goal is to allow for quick navigation in a document, give spatial orientation to the user by rendering the square which shows what is currently displayed, and give information about the viewport's surrounding data.

An example for a minimap can be seen in Figure 2.5.

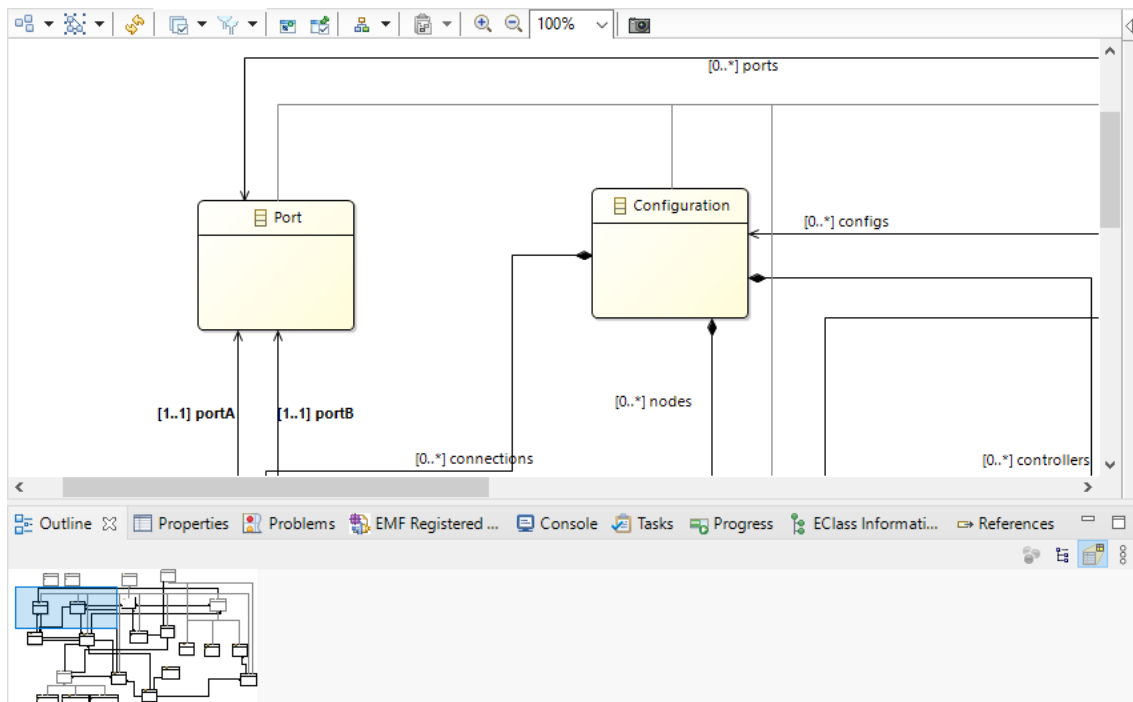


Figure 2.5: Example for a minimap in the Eclipse IDE. The main view shows only a part of a class diagram. The minimap in the bottom left shows the entire diagram with a much smaller zoom factor. The blue square indicates what is currently visible inside the main view.

Zooming to Saved Frames

This feature can be found in slightly different ways in many modeling tools (e.g., Prezi and Miro). The user can select and name specific parts of the stage. These parts are then saved in the form of frames. Frames do not save their contents but merely the zoom level and position. These frames can be seen as anchor points to specific parts of the diagram. Anchor points can then get referenced by, e.g., buttons to let the user go to parts of the diagram much faster.

Scripting

This feature is usually not only meant to be used to create special zooming functionality. Depending on how much functionality the scripting language/engine offers, this can go from simple mathematical operations to providing a turing-complete and completely customizable environment. This can be used to create all kinds of special zooming features like adjusting size/color/font of text or forms when zooming in or out, changing zoom levels/navigating with the click of a button and much more. The drawback is that it is usually much more complex to use because it has to be written and implemented first which requires knowledge of the scripting language and usually also general experience in software development. An example for such scripting capabilities is Visual Basic for Applications (VBA), which can be used in combination with Microsoft Visio.

2.5.3 Summary of Level-of-Detail Features

This section gives a summary of all features that were found during the evaluation of the modeling tools that show objects in multiple levels of detail or could be used to achieve such a behavior. It includes all features that effectively hide or show specific information about objects. A list of all tools that were looked at can be found in Appendix 1 (marked with *).

Showing Additional Information on Button Click

This feature is seen in many tools. The basic idea is to show additional information (change the level of detail) by the click of a button. It can be additional text, additional forms, shapes, or even complete models. This can be implemented in many different ways:

1. **Links:** A simple but still customizable way can be seen in tools like diagrams.net. They use links to change the content on the stage. Usually, button elements exist that reference links. When clicking on such a button, depending on what kind of link is referenced, a certain behavior is triggered. Besides the behavior that most users are already familiar with from other software like web browsers, e.g., simply opening a web link, they can lead to different positions inside a diagram, zoom in or out, highlight certain information, or even show a completely new version of a diagram that may hold additional information. Links that a button can lead to have to be defined by the user first. This can be done in different ways. Usually, the software allows to create links to one or many of the following:
 - a) **Pages:** Different pages can be created by the user. They can hold more generalized or more detailed versions of a diagram.
 - b) **Layers:** A diagram can consist of multiple layers. Each layer can hold a different version of a diagram with more or less details.
 - c) **Frames:** Frames can be defined by the user. Frames save a position and zoom level inside a diagram.

- d) **Diagram elements:** All elements that can be used inside a diagram. E.g., shapes, forms or text.
 - e) **Diagrams:** Entire independent diagrams can be linked. This can be seen in form of a built-in feature in Visual Paradigm. Instead of creating a link to another diagram, shapes can be right-clicked and decomposed. This creates a new diagram with the selected shape in it. It can then be used to add new shapes and forms, and model this part of the diagram in more detail.
2. **Expand/collapse:** Built-in LoD functionality for buttons. This can be found in many tools and is, for example, part of libraries such as yFiles, or diagram tools such as diagrams.net and yEd Live. Here, the level-of-detail functionality is directly part of the forms and shapes that are available to the user in the form of a button. There exist many different variations how this can be implemented and they are usually all optional to use. An example is a drop-down menu that can be triggered by a button inside a shape. Clicking on the button shows or hides content or information. Other examples are showing/hiding all descendants of a parent in a tree view or showing/hiding all properties of a class in a UML class diagram.
 3. **Scripting language:** A fully customizable way of achieving specific behavior, for example, with VBA in Microsoft Visio. This is by far the most powerful approach but also the hardest to use. Depending on how powerful the scripting language is and how well it works together with the overlying system, the possibilities are almost endless. Usually, listeners can be registered to the 'click' event on buttons. Once they are triggered, functions can be called that are able to modify the current state of a model. All the functionalities above could, for example, be implemented in Microsoft Visio through VBA.

Showing Additional Information on Zoom

The zoom functionality goes hand in hand with different levels of detail. For additional information, usually in form of text, additional space is required. Zooming in increases the size of individual elements which creates more space for additional text. Unfortunately, the zoom event is not used a lot in any tools as a built-in feature in combination with LoD. It is used quite often for performance reasons and to hide elements that are not relevant to the user anymore. Good examples for this are hiding elements that are outside of the current viewport or replacing text that is too small to be read with something that is simple to render, as is done by Miro and Microsoft Visio. The objective of this is to increase the framerate during the rendering process, all while keeping the user from noticing any visual differences to the original and fully detailed diagram.

Showing Additional Information on other Events

Although the click of a button is the most common event, other events exist that can also be used to add or remove details. This is not only used much less frequently by users but also implemented less often by developers for that reason. Other events can, for example,

be scrolling, hovering or typing. An example of a tool with such a feature is Lucidchart, which is able to show additional data when hovering over a shape or form through the ‘hover’ event. While scripting languages usually allow a lot more events to be used than just the click event, almost no built-in functional can be found in any tools related to events other than clicking.

Grouping of Elements

Another feature that can be found in diagram tools is the ability to group elements together and be able to show or hide them. Grouping elements together can usually be done by first creating a new group which is then visible on the stage. After a group has been created, new elements can be added to it by dragging them into this group. The reason why we see this as a feature related to LoD is because groups can be expanded and collapsed. This adds or removes details about a diagram which effectively changes the level of detail. Expanding or collapsing a group can, for example, be done with the click of a button which can be found next to the title of the group as seen in yFiles (Section 2.5.1). It can technically also be based on other events like zooming or scrolling but we could not find this behavior in any tools.

Taxonomy¹

The purpose of this section is to identify, name, and describe characteristics which are fitting for user interaction and visualization features. Although the topic of this work is about features related to the navigation of large information spaces in GLSP-based modeling tools, the categorization in this taxonomy will be kept more abstract so that it can be used in combination with other domains as well. Another reason why it makes sense for this work to look at visualization features in a broader range is, because with the introduction of modern GLSP environments, the technology stack has advanced to a point that all kinds of new features can be integrated and used, even those that have not been seen before in the domain of model engineering.

The goal of creating this taxonomy is to find similarities and to classify similar objects into the same category. This should ultimately lead to a better understanding of possible features that could be implemented in a (graphical) language server platform and should help developers of such features during the conceptualization and integration process. Because it is kept abstract, this taxonomy can also be used to achieve this goal in other domains which utilize visualization features. It will lean on relevant and existing taxonomies and complement them with extra dimensions that are fitting to the domain of this work. In order to do this, features in the general domain of information visualization and user interaction, that can be found in literature, will be looked at. This is accompanied by an evaluation of existing tools and their visualization features.

3.1 Relevant Taxonomies

The first step here is to look at existing taxonomies that could be used and applied to this work. Although we could not find any existing taxonomy that is perfectly fitting, the subsequently described works partly align with our goal.

¹A concise version of this chapter will be published at the ER conference [DCPB22b].

Shneiderman [Shn03] proposes a task by data type taxonomy with seven data types for applications with advanced graphical user interfaces. These tasks are: *Overview*, *Zoom*, *Filter*, *Details-on-demand*, *Relate*, *History*, and *Extract*. The data types are: *1-dimensional*, *2-dimensional*, *3-dimensional*, *temporal*, *multi-dimensional*, *tree* and *network*. Shapes and forms of a GLSP client can be classified as 2-dimensional data type. Relevant tasks are mainly overview, zoom, filter and details-on-demand. The remaining data types and tasks are not directly related to this work.

Similar to Shneiderman’s approach, Silva et al. [SC00] categorize temporal-data features by *visualization* and *interaction features*. Visualization features describe visual techniques of a system, as in their example, Snapshot view or Multiple Calendars. Interaction features are defined and categorized, similarly to Shneiderman’s approach, into: *Overview*, *3D Navigation*, *Time Navigation*, *Zooming*, *Filtering*, *Temporal Filtering*, *Details on demand*.

Tory et al. [TM04] categorize visualization techniques based on their design model instead of their data. They propose to categorize design models into two higher level groups: *discrete* and *continuous*. Continuous models assume that data can be interpolated, and discrete models assume that they cannot. Data can often be visualized in multiple ways and therefore it is possible to present the same data with continuous models as well as discrete models. They also propose to add the category *constrained* to the spatialization categorization. They argue that spatialization cannot only be given or chosen but also partially given or chosen. This is represented by the *constrained* class.

Cockburn et al. [CKB09] categorize graphical user interfaces into four categories: *overview-plus-detail*, *zooming*, *focus-plus-context* and *cue-based*. Overview-plus-detail represents the spatial separation of information. It splits up information into two separate views: overview-view and detail-view. Zooming represents the temporal separation of information. It allows magnification and demagnification of information. Focus-plus-context seamlessly combines a focused representation of information with its context. Cue-based techniques change how an object is displayed and rendered, and are often combined with search-criteria or off-screen elements.

3.2 Research Approach

Nickerson et al. [NVM13] propose a method to create taxonomies for information systems. They define a taxonomy as a set of n dimensions $D_i (i = 1, \dots, n)$ each consisting of $k_i (k_i \geq 2)$ mutually exclusive and collectively exhaustive characteristics $C_{ij} (j = 1, \dots, k_i)$ such that each object under consideration has one and only one C_{ij} for each D_i .

In order to create a valid taxonomy, they propose an iterative approach which is applied until all ending conditions are met. One iteration can either consist of an *empirical-to-conceptual* step, or a *conceptual-to-empirical* step. To make the correct choice between the two, one has to look at the knowledge of the researcher and the available data. An empirical-to-conceptual iteration should be chosen, when there are many objects available and the researcher is familiar with them. It consists of looking at these objects and identifying characteristics based on their qualities. A conceptual-to-empirical iteration

should be chosen when there do not exist a lot of objects and the researcher has a broad understanding and knowledge base of the relevant domain. Instead of primarily looking at objects, the researcher will identify characteristics merely based on their knowledge.

Problem Identification and Motivation

Kundisch et al. [KMO⁺21] provide further guidance on taxonomy development and evaluation by following up on the methods of the work of Nickerson et al. [NVM13]. They argue that most taxonomies of the past display an inconsistent adoption of existing methods and a non-transparent reporting of relevant design decision. In order to overcome this limitation, they present an extended taxonomy design process (ETDP) and give examples of well-written taxonomies for each step in their process. The additional steps in their ETDP focus mainly on problem identification and motivation, and taxonomy evaluation. More accurately, they add three initial steps which should be conducted before the taxonomy is designed and developed. These steps consist of specifying: (i) the *observed phenomenon*, (ii) the *target user group(s)*, and (iii) the *intended purpose* of the taxonomy. The definition of these three specifications has a high influence on the evaluation of the taxonomy. Additional evaluation steps, which were added to the model by Kundisch et al., are conducted after finishing the original method steps defined by Nickerson et al. They mainly deal with configuring and performing an *ex post* (after the building process has been terminated) evaluation which should evaluate how useful the taxonomy is in achieving the defined goals for its targeted user group. This can, for example, be done with interviews, focus groups, or experiments which could be performed with the help of people of the target user group(s).

The phenomenon that will be observed in this taxonomy are visualization and interaction features. More accurately, concrete examples, or theoretical concepts of software components (features) that allow users to modify underlying data by utilizing interaction-methods via a graphical user interface. The target user groups are researchers and developers of novel visualization features. This taxonomy is supposed to help identify defining characteristics of such features and with that, provide aid during the conceptualization and integration of them into new systems or platforms. It helps understand how features can be structured and what the defining characteristics are from the perspective of a user. Furthermore, it should give a basic understanding about requirements and limitations of the chosen characteristics, which should help make correct decision when designing new features.

Solution Objectives

Two very important qualities of a valid taxonomy were already mentioned: *mutually exclusiveness* and *collectively exhaustiveness*. This means that every object has to have exactly one characteristic in each taxonomy dimension. These two qualities form two of ten objective ending conditions [KMO⁺21, NVM13], which, together with five subjective ending conditions, are used to determine when a taxonomy is considered complete. Consequently, these ending conditions need to be applied during the iterative application of either an *empirical-to-conceptual* or *conceptual-to-empirical* step until the ending conditions are met.

The ending conditions we chose to adopt from Nickerson et al. [NVM13]. They define subjective ending conditions as follows:

- **Concise:** Does the number of dimensions allow the taxonomy to be meaningful without being unwieldy or overwhelming?
- **Robust:** Do the dimensions and characteristics provide for differentiation among objects sufficient to be of interest? Given the characteristics of sample objects, what can we say about the objects?
- **Comprehensive:** Can all objects or a (random) sample of objects within the domain of interest be classified? Are all dimensions of the objects of interest identified?
- **Extendible:** Can a new dimension or a new characteristic of an existing dimension be easily added?
- **Explanatory:** What do the dimensions and characteristic explain about an object?

Objective ending conditions are the following:

- All objects or a representative sample of objects have been examined
- No object was merged with a similar object or split into multiple objects in the last iteration
- At least one object is classified under every characteristic of every dimension
- No new dimensions or characteristics were added in the last iteration
- No dimensions or characteristics were merged or split in the last iteration
- Every dimension is unique and not repeated (i.e., there is no dimension duplication)
- Every characteristic is unique within its dimension (i.e., there is no characteristic duplication within a dimension)
- Each cell (combination of characteristics) is unique and is not repeated (i.e., there is no cell duplication)

Both types of ending conditions, subjective and objective, are important to determine when the iterative process can be stopped and the taxonomy holds enough characteristics to classify the phenomenon it is supposed to describe. For this reason, they are both used as an *ex ante* (before the building process has been terminated) evaluation.

Design and Development

As already mentioned and specified in the previous sections, according to Kundisch et al. [KMO⁺21], the first three steps are: (i) Specify the observed phenomenon, (ii) Specify target user group(s), and (iii) Specify intended purpose(s).

Following Nickerson et al. [NVM13], the next step is to define the meta-characteristic which will serve as the basis for the classification. For this taxonomy, we chose three meta-characteristics: (i) Presentation, (ii) Interaction, (iii) Data. A more detailed definition of these characteristics will follow in Section 3.4.

The next step is to look at the currently defined characteristics and iteratively apply, either an empirical-to-conceptual or conceptual-to-empirical step, until the ending conditions are met. The first iteration for this taxonomy was a conceptual-to-empirical one. In order to

get the knowledge which is necessary to apply the iteration, we looked at taxonomies and literature of the past that dealt with similar phenomena (Section 3.1). This was followed by multiple empirical-to-conceptual iterations. Initially, we looked at features of commonly used software that displays a graphical user interface and allows users to interact with it. Most of the tools we looked at were used by many people over a long period of time. They receive constant feedback and are being maintained and improved by long standing companies. For that reason, they set a good baseline for this taxonomy. Examples for such tools are: Google Maps, Microsoft PowerPoint or JetBrains IntelliJ IDEA. A summary of their features is described in more detail in Section 2. A full list can be found in Appendix 1 of this work. The derived dimensions and characteristics were then re-evaluated again with another empirical-to-conceptual iteration which considered visualization and interaction features of past literature. This iteration did not only consist of concrete examples, but also conceptual designs of features. For this reason, this iteration could also be considered a conceptual-to-empirical one.

Unlike the features of commercial tools of the previous iteration, these features provided more insight about reasoning behind design decisions and technical conditions. Examples are City Lights [ZMG⁺03], EdgeRadar [GI07] and Onion graphs [KM07] (explained in more detail in Section 3.5).

Demonstration and Evaluation

While it is not possible to consider all existing features of today's tools and literature, the sample of features which was picked was expanded until all subjective and objective ending conditions were met. The resulting dimensions and characteristics can be seen in Figure 3.1 and read up upon in Section 3.4.

It is to note that, ideally, this taxonomy should only be used to categorize concrete examples of features. During the development, we realized that conceptual designs of features can often be interpreted and implemented in many different ways. E.g., the concept of a magnifying glass feature can be implemented in a separate and independent view, or by magnifying the current view. In the first case, it would be classified as an overview-plus-detail interface, but in the second case, it would be classified under focus-plus-context. When classifying conceptual designs of features that have not been implemented yet, one has to be aware that it may include a subjective bias. Often, it is not immediately obvious how such features operate, which is why it is even more important to accurately describe them.

3.3 Terminology

Before the definition and evaluation of this taxonomy is given, it is important to define terms that are being used during it. This should prevent any miss-interpretations and confusion in the following sections.

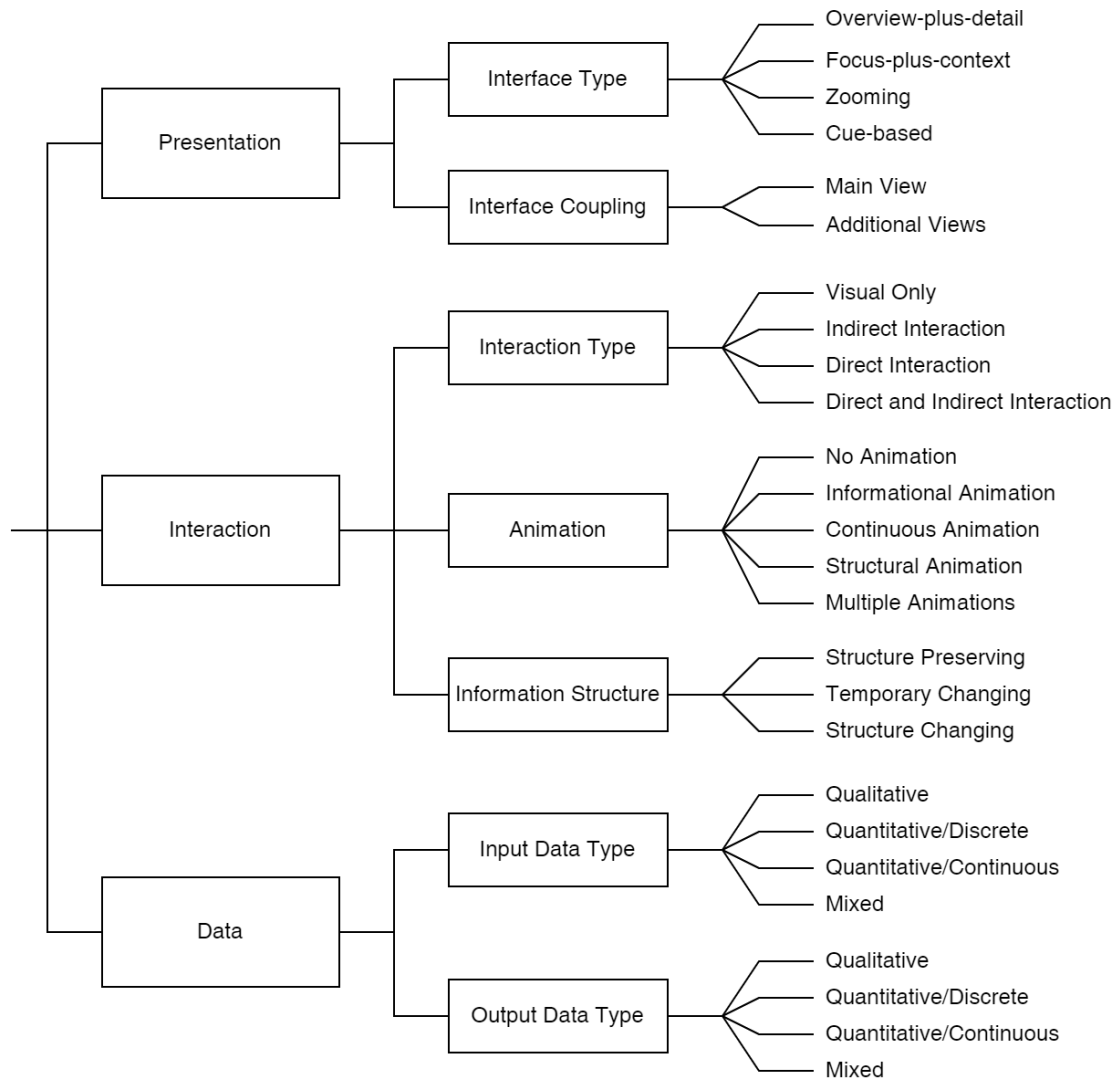


Figure 3.1: Taxonomy visualized

3.3.1 View

A view represents a certain perspective onto the information inside an application. Usually, the interface of an application consists of multiple views which all represent information in a different way. For example, a PDF reader usually always shows the PDF file contents (e.g., text or images) inside an isolated view. Besides that, there almost always exists a scrollbar, which can also be considered a separate view because it shows information in a different and isolated way. Instead of rendering text and images, it shows spatial information about the current position of another view. It is not always easy to define where the boundaries of a view lie. In case it is not immediately noticeable by a visualized spatial separation when looking at the user interface of an application, two different views

can be separated from each other by looking at the following three properties:

1. **What data they represent:** Two different views often represent different kind of information.
2. **How they represent data:** The data does not always have to be different, it can also simply be represented in a different way. E.g., Microsoft PowerPoint generally shows the same information inside the overview-view as in the detail-view in Figure 3.3 but prepared in a different way.
3. **What interaction methods they offer:** Usually, interaction methods are isolated to one view. Different views also often offer different interaction methods. But, although they are separated, they can affect other views. For example, moving the scrollbar in a PDF reader affects what information is currently shown inside the view which shows text and images of the PDF file.

3.3.2 Stage

The stage represents the area in which the main content of the application is displayed. In diagramming tools, that would be the area in which forms and shapes are displayed. New elements can usually be added by dragging and dropping them onto the stage from the tool palette. The stage is almost always part of the main view and takes up a large portion of it. In comparison to the viewport, the stage can be seen as a lower level of abstraction. The stage contains all subjects of the current workspace but only a portion of it is usually shown through the viewport.

3.3.3 Viewport

The Viewport is the area in which the stage is displayed inside an application. It acts as the layer between stage and the user and can be seen as a camera that has a specific view onto the stage. In contrast to the stage, which can technically be unlimited in size, the viewport is limited and can usually only display a fraction of the stage. It usually allows the user to interact with the content on it. Good examples are zooming and panning and figuratively “moving the camera around” to change the view of the stage. It also allows for other events, for example, drag-and-dropping new elements or modifying elements with keyboard inputs.

3.3.4 Zoom level

It represents the scale at which the stage is viewed and affects the size of the objects on it. Zoom level is usually specified in a percentage-based number. There exists a default zoom level (usually 100%) which is a defined value that shows the stage in a specific perspective. Some applications do not allow to change the zoom level or do not have the functionality to do so. Usually, this perspective is a good combination between element size and general overview. It shows and displays specific elements of a subject in a readable way but also tries to give a good overview about the entire subject. If the zoom level is changed, the

perspective that a user has onto the stage changes, and elements are rendered in a bigger or smaller way.

While it is usually the case that there exists only one global zoom level which is relevant for the entire stage, it can also be defined in a smaller scope. For example, every subject on the stage can have its own zoom level. This makes it possible to change the size of each subject individually. Defining the scope of a zoom level is often not a trivial task. We propose three broad scopes sorted from high to low:

1. **Global scope:** A zoom level that is applied to every element on the stage. This is practically the norm in most of today's application. It is used to increase or decrease the view of all subjects on the stage at the same time and with the same granularity.
2. **Subject scope:** Every subject has its own zoom level. This means that the size of each subject on the stage can be changed individually without directly affecting other subjects. This also means that the size of different subjects can be inconsistent and out of sync. Two subjects with the same semantics can have different sizes.
3. **Property scope:** Each property of a subject has its own zoom level. This means that zoom can be applied to subjects in more detail by being able to increase or decrease only specific parts of a subject.

Usage of zooming scopes is not mutually exclusive and multiple scopes can easily be used together. In fact, a more detailed scope level is usually always used in combination with all less detailed ones. It is important to note that the lower scopes do not implicitly include higher ones. It could be assumed that changing the zoom level of all elements of a scope at the same time has the same effect as changing only one level on scope above. This is not the case because of an important difference related to the way zooming works on different levels. An element's change in size only affects the element itself and nothing else. This means that if all elements at, for example, the subject scope will be increased at the same time and with the same rate, in theory they will eventually become too large and will overlap with each other. This would not happen if the global zoom level was increased instead because that would also increase the gaps between subjects.

It is not always easy to apply the scopes defined here to real examples. Sometimes it is not clear how subjects are defined or if subjects hold properties. Often subjects can hold other subjects, in which case the given zoom level scopes have to be adjusted. There is also no clear definition how other subjects should react to the change of zoom level of one specific subject. The most basic reaction is none at all which often leads to an undesired outcome for a user and therefore a bad usability. In fact, it is difficult to find a good solution for this problem and they often heavily depend on the nature of the subjects.

3.3.5 Zooming

Zooming is the change of zoom level. This effectively changes the size of the displayed objects. Increasing the size can be compared to looking through magnifying glasses or moving a camera closer or further away from a subject. Changing the zoom level in its simplest form, is often done by moving the mouse wheel up or down which steadily

increases or decreases the zoom level. It can also be implemented in a more complex way as in combination with fisheye zooming as explained in Section 3.5.3. This requires there to be more than just a global zoom level. In order to implement a fisheye-based zooming feature, often every subject needs to have its own zoom level to be able to change them individually.

3.3.6 Level of Detail

The level of detail describes the amount of information of an object that are displayed on the stage. Objects can be represented in different (usually discrete) levels of detail. The lowest level shows the least amount of information while the highest level shows the most information. Information can be directly part of the object, e.g., properties, or meta-information, like the number of relationships or dimensions of an object. Its goal is often to simplify the currently visible information and give the user an easier time to comprehend them. Other times, it is used to make elements smaller and save space, or to remove information which have become too small to be read.

A switch between levels of detail is usually directly or indirectly triggered by the user. An example for triggering a switch directly is explained in Section 3.5.3. It is done by the click of a button which shows additional information about an object. Often the level of detail is connected to other interaction methods such as zooming. A zoom event, for example, can indirectly trigger a change of details to remove elements that are now too small.

Level-of-detail-related functionalities often cannot be easily added. This is especially true in a universal setting. Every language has different properties and is represented in a different way, which makes it hard to find a universal solution. For this reason, not many features exist and they are also not often seen in existing tools.

3.3.7 Information Space

The information space is the set of abstract information that a language represents. This can also include abstract syntax or semantics of a language. Related to graphical representation, it includes only abstract and no concrete information about them. The information space of Google Maps, for example, consists, in its basic form, of information about landscapes, countries, cities, oceans, and other entities of our planet. Information about abstract graphical representations includes, e.g.: where is a country located, what are its dimensions, and what are its neighbors. It does not include information about concrete graphical representations, as for example: The color which it is represented in, the thickness of its lines which represent the border, or the font in which the country name is written. Concrete instances of an information space are usually represented and contained inside concrete files, often in a universal format as, for example, XML.

Important for this work is the abstract size of an information space, and more importantly, the size of concrete instances of an information space (workspace). While it is generally easy to determine how large an information space is, doing the same for concrete instances is often not. Technically, most instances can be infinite in size, which is why this work

will try to assume the average use case when talking about the general size of a language. It will mostly differentiate between small (e.g., Java code enumeration file), intermediate (e.g., most PDF files) and large (e.g., Google Maps).

3.4 Taxonomy Definition

As already mentioned above, this taxonomy consists of three meta-characteristics: *Presentation*, *Interaction*, and *Data*. The first meta-characteristic, Presentation, describes the interface type that is used by a feature, and how the information is coupled to either the main view or additional views. The second meta-characteristic, Interaction, gives information about those parts of a feature that directly or indirectly affect the user experience and their ability to use it. The third meta-characteristic, Data, focuses on the data that a feature utilizes or manipulates. Combined, these characteristics are supposed to help developers identify important conditions of a feature. This section will go over all characteristics in more detail.

3.4.1 Presentation

This dimension mainly describes if and how a feature utilizes one or multiple views. Additionally, it describes the dependency between views and how they represent information. It is split into **Interface Type** and **Interface Coupling**.

Interface Type

It describes how features use the space that is available to them, how they represent information to the user, and generally how a user is able to interact with them. It consists of four categories, which are based on Cockburn et al.'s work [CKB09]: **overview-plus-detail**, **zooming**, **focus-plus-context**, and **cue-based**. This categorization is thus not new and can be found frequently when browsing past literature which deals with information visualization. For this reason, the decision was made to include them in this taxonomy. It is important to note that sometimes it can be hard to categorize features into this meta-characteristic without breaking the mutual exclusivity. Some features have characteristics of multiple interface types and could therefore be assigned to multiple categories. To keep the mutual exclusivity intact, we added more constraints to some of those categories to make the categorization clearer. If, even with these additional constraints, it is still not clear where to place a specific feature, it may help to split it up into sub-features and categorize each sub-feature individually. It is also not unlikely that this category needs additional characterizations in the future to fit new interaction types and features.

Overview-plus-detail: It represents the spatial separation of the information space. The overview-plus-detail interface scheme is a widely used concept that is present in almost all applications nowadays. It splits the information space into two physically separated views. One shows information at an overview level, the other shows similar or even the same

information in greater detail. Although they are physically separated, they do semantically depend on each other and actions in one view are usually immediately reflected inside the other view. The important characteristic about the dependency between both views is that they are usually not spatially dependent on each other. If one or even both of the views were to be moved to a different location, no issues would arise. Sometimes it can be hard to make the differentiation between overview-plus-detail and focus-plus-context interfaces in which case it may help to think about this property.

An overview-plus-detail interface has usually two main purposes. Firstly, it should give the user a better feeling about what subset of information they are currently looking at in relation to the entire workspace. Secondly, they should give the user an easy way of navigating the workspace by letting them interact with the overview interface. Usually, they operate on the x- and y-axis and utilize interaction methods such as panning or scrolling. Features that work with the z-coordinate usually make use of the zooming interaction method and therefore often overlap with the zooming category. Nevertheless, features exist that can be categorized as overview-plus-detail but utilize the z-axis, for example, magnifying glasses as explained in Section 3.5.3.

A good example for a feature in this category is shown in Figure 2.5. On the stage, an UML diagram is shown in detail while in the bottom left corner, an overview is displayed which shows the entire diagram in one view. Actions like the panning or zooming inside one view are directly reflected inside the other view.

The overview interface does not always have to show the same type of information as the detail-view, it may also provide a completely different type of data, e.g., spatial information. An example for spatial information is the scrollbar. Scrollbars can be seen as the overview interface that give one-dimensional information about what the viewport currently displays in relation to the entire workspace. There also exist concepts that show more than just spatial data. For example, the text editor Sublime Text 3 which has a widened vertical scrollbar that shows additional information about the current document (shown in Figure 3.2b). It shows the content of the current document at an overview-level and even adds syntax highlighting.

Another good example for more than just spatial data representation in the overview interface can be seen in Microsoft PowerPoint in Figure 3.3. Inside the overview-view, a smaller version of the slide that is currently displayed inside the detail-view is shown, along with following and preceding slides.

Zooming: Zooming represents the temporal separation of the information space. It is similar to overview-plus-detail with the difference that only one view is provided instead of two or more. Another difference is that the user has to utilize the zooming interaction method to change the size in which information is displayed. As already explained in Section 3.3.5, the user is given the ability to change the scale (zoom level) of the view with certain actions such as a button click. Other, not so well known, ways of zooming in or out are the usage of the key-binds CTRL+mousewheel or CTRL+"+". This represents the

```

75     if(!s.includes(",")) return false;
76
77     let symbol = "o";
78     for(let k = 0; k < 2; k++) {
79         for(let i = 0; i < 3; i++) {
80             let split2 = split[(2*i)+1].split(":"+symbol+":");
81             if(split2.length === 2) {
82                 if(split2[0].length < 2) board[(i*3)] = symbol;
83                 else if(split2[0].length < 12) board[(i*3)+1] = symbol;
84                 else board[(i*3)+2] = symbol;
85             }
86             else if(split2.length === 3) {
87                 if(split2[0].length < 2) board[(i*3)] = symbol;
88                 else if(split2[0].length < 12) board[(i*3)+1] = symbol;
89                 else board[(i*3)+2] = symbol;
90             }
91         }
92     }
93 }

```

(a) Normal view (without overview-scrollbar) in Sublime Text 3

```

75     if(!s.includes(",")) return false;
76
77     let symbol = "o";
78     for(let k = 0; k < 2; k++) {
79         for(let i = 0; i < 3; i++) {
80             let split2 = split[(2*i)+1].split(":"+symbol+":");
81             if(split2.length === 2) {
82                 if(split2[0].length < 2) board[(i*3)] = symbol;
83                 else if(split2[0].length < 12) board[(i*3)+1] = symbol;
84                 else board[(i*3)+2] = symbol;
85             }
86             else if(split2.length === 3) {
87                 if(split2[0].length < 2) board[(i*3)] = symbol;
88                 else if(split2[0].length < 12) board[(i*3)+1] = symbol;
89                 else board[(i*3)+2] = symbol;
90             }
91         }
92     }
93 }

```

(b) View in Sublime Text 3 with overview-scrollbar enabled. A small additional view is added at the right of the main view which shows the content of the entire file and acts exactly like a scrollbar.

Figure 3.2: Comparison of a view with overview-scrollbar to one without in Sublime Text 3

basic concept of zooming but it can be combined with other techniques, such as fisheye zoom or semantic zoom, to make it more advanced.

A precondition to make zoomable interfaces possible, is to have information that can be magnified and demagnified. The magnification process can be categorized into *continuous* and *discrete* zooming. Continuous zooming takes place when the subject does not have a countable amount of zoom levels. The simplest form of magnification is to simply increase the size of the subject. This can be done on every subject that has some form of visual representation. Since there is no clear separation of zoom levels, and the subject can theoretically be rendered in any size, this can be considered continuous zooming. An example for discrete zooming takes place in Google Maps. When the user zooms in far enough, the level of detail is changed and a different set of tiles is served. There exists only a limited number of sets of tiles/levels as explained in Section 2.3.2 and therefore this can be considered discrete zooming. In the example of Google Maps, it is to mention that they use both, discrete and continuous zooming. Between going from one discrete level to the next, tiles are simply increased in size which is a form of continuous zooming.

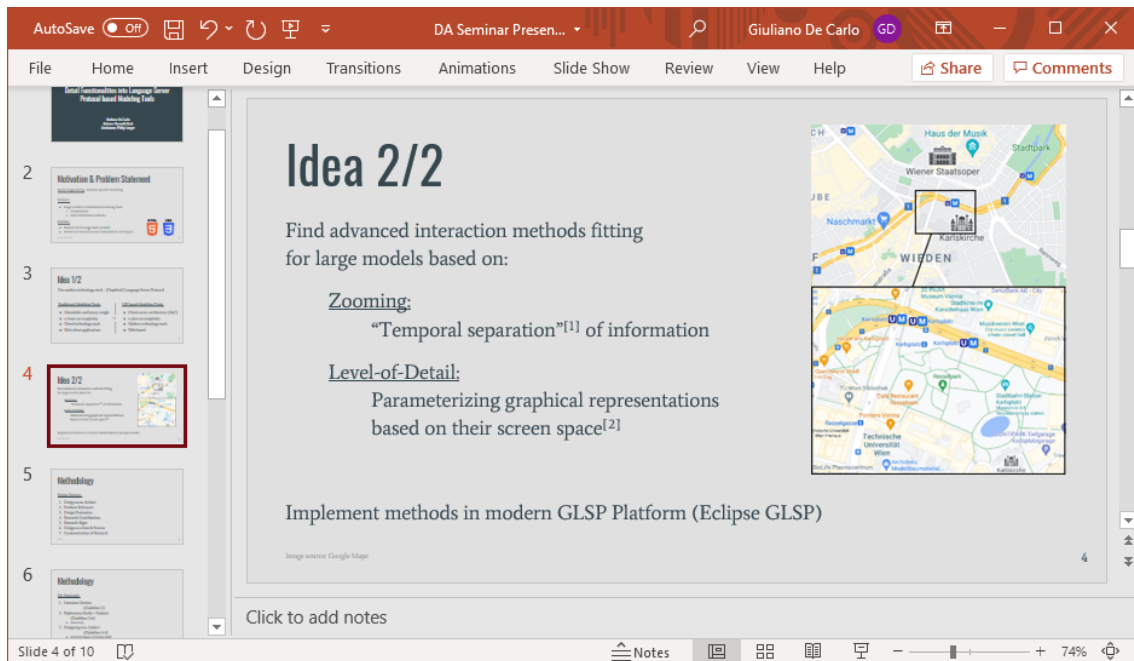


Figure 3.3: Example for an overview-plus-detail interface in Microsoft PowerPoint. The detail-view shows the current slide and covers about 80% of the interface (right side), and the overview-view additionally also shows the neighboring slides and roughly covers the remaining 20% of the interface (left side).

Focus-plus-context: It represents the distortion of the information space. The basic concept of this scheme is to let a user see specific parts of the information space in more or full detail while also getting an overview of the information around it. Unlike overview-plus-detail, both parts (overview and detail) are displayed inside the same view which is often accomplished by distorting the information space to fit the user's needs. Information that the user is interested in, the focus, is shown in greater detail and at the same time, information around it, the context, is preserved and also made visible to the user but in less detail.

The level of distortion that is applied, differs from implementation to implementation. It goes from no/infinite distortion, for example, by using the Windows 10 Magnifier app (Figure 3.5), to distortion that affects the entire view, as seen in Google Street View (Figure 3.4). The aspect of distortion is an important feature of focus-plus-context interfaces which differentiates features of this category from others. Without any form of distortion, as with the Windows 10 Magnifier app, a feature can often be categorized as a basic zooming feature or an overview-plus-detail feature instead.

Another aspect which defines focus-plus-context interfaces are the seamless integration of one view into the other. An example for this is the scrollbar. Scrollbars could be considered to be part of the focus-view, but, because they are not seamlessly integrated into it, they

3. TAXONOMY

are considered to be a separate view.



Figure 3.4: Google Street View shows a distorted view of St. Stephen's Cathedral and its surroundings. The focus lies on St. Stephen's Cathedral while surrounding buildings are visible as well. The photo is showing a very unrealistic perspective, especially noticeable by looking at and comparing parallels of the buildings to the left and right of the cathedral, which makes the distortion aspect clearly visible.

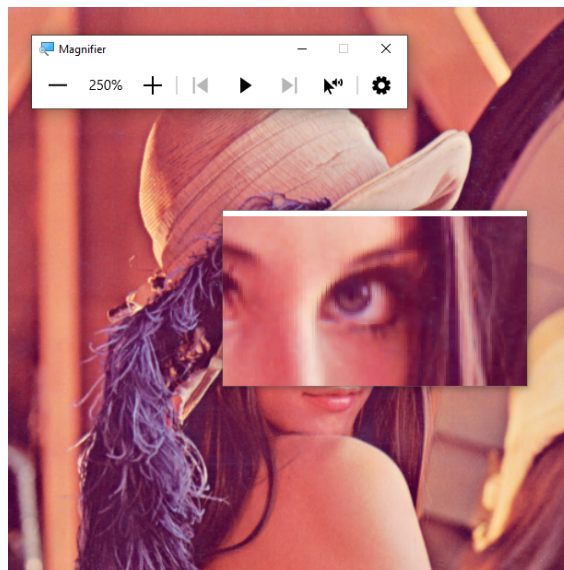


Figure 3.5: The Windows 10 Magnifier app magnifies specific parts of the view. Instead of distorting information and using, for example, fisheye techniques, the magnified part covers and effectively hides other parts of the image. Because of this, the Magnifier app is categorized as a zoom interface instead of a focus-plus-context interface.

An advantage of having only one view instead of two or more, is that the user does not have to switch focus between multiple views and can instead keep focusing on one and the same view. In a field study conducted by Baudisch et al. [BGBS02], which compares overview-plus-detail, focus-plus-context and zooming-plus-panning, all chosen tasks could

be performed faster on the focus-plus-context interface by a margin of 21-36%. They attribute the performance differences to the context switches that do not have to be made on a focus-plus-context interface, and the consistent scale that the focus-plus-context interface offers.

Cue-based: Cue-based techniques give cues that lead to other information in the information space. They often show alternative graphical representations of objects on the stage. These alternative representations, often in the form of simple labels, can then be used to offer additional functionalities, as, for example, leading to the actual object, or simply notifying the user that the object exists. Cue-based techniques are often used in combination with approaches of other interface methods. An example for cue-based techniques is the visualization of off-screen objects with, for example, lenses as seen in Section 3.5.3. They are also often used in combination with search criteria. An example for that can be seen in Figure 3.6, which shows clickable search results in Google Maps. Clicking on them pans and zooms the map to the position of the search result.

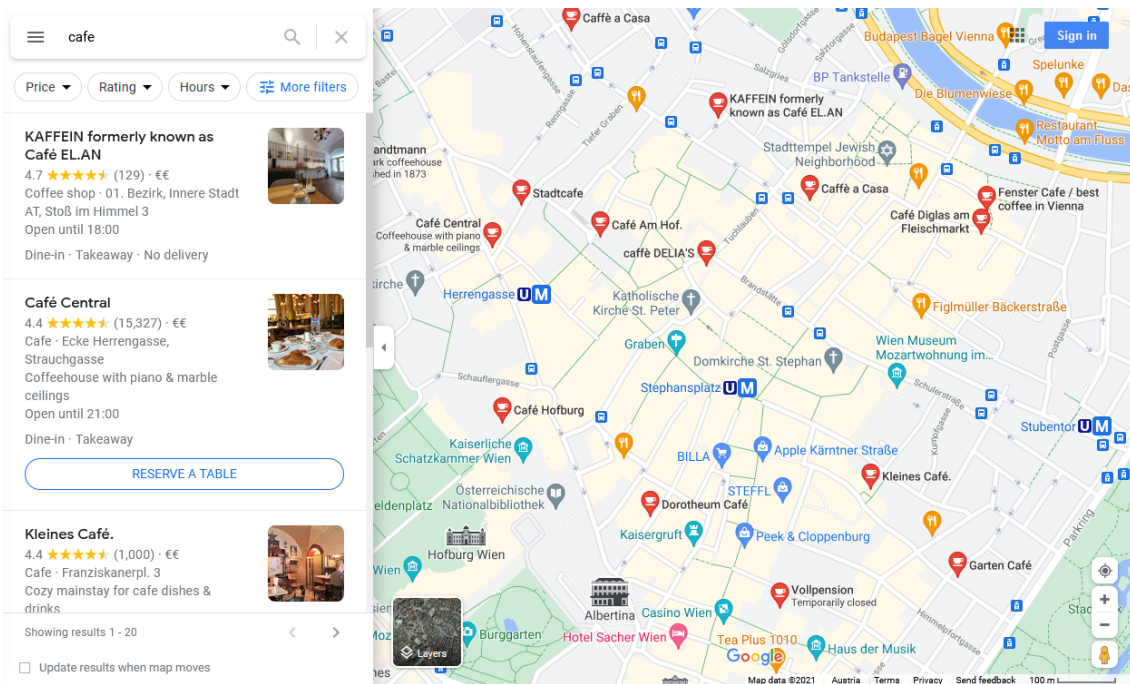


Figure 3.6: Google Maps shows search results at the left side of the view. Results act as a proxy and clicking on one zooms and scrolls the map to the position of the search result.

Interface Coupling

This category distinguishes features that require **additional views** from those operating inside the **main view**. This is an important aspect as it impacts not only how a user uses a feature, but also its implementation. From an implementation perspective, having to add

another view can become complicated, because, depending on the size and significance of the implemented feature, new space has to be found on the user interface, and, instead of managing just one view, multiple views have to be managed and be held in sync. From the perspective of a user, having more than one view available means that there are multiple separated points of interest that a user can shift their focus to. Usually, it is not possible to concentrate our focus on multiple views at once and a focus switch is required when switching between them. This focus switch does not only cost time, but also requires mental effort which can become exhausting over time. For this reason, it is often better to combine information into one view instead of separating them into multiple views. As Baudisch et al. [BGBS02] showed, tasks could be performed faster on focus-plus-context interfaces (one-view) than on overview-plus-detail (two views). This is even more important, when the user has to interact with multiple views, e.g., with a mouse. This was also confirmed by Hornbæk et al. [HBP02]. They state: "...switching between the detail and the overview window required mental effort and time moving the mouse." [HBP02, p. 382]

3.4.2 Interaction

This dimension focuses on characteristics of a feature that the user directly interacts with. The attributes in this dimension have a big impact on how a feature is being used by the target group and therefore its usability. It should give an insight about important user-related aspects of a feature and help in making design-related decisions. This section will describe the characteristics in this dimension in more detail and also give some basic guidelines on why and when certain properties are important.

Interaction Type

This characterization gives information about if and what kind of interaction a feature offers. While this category can be further extended by going more into detail and considering all kinds of interaction types and events, this taxonomy remains on a more abstract level with only three main categorizations: **visual only**, **indirect interaction**, and **direct interaction**.

Visual Only: Features which are classified as *visual only* do not give a user the ability to interact with it, and only add visual benefits. An example for a feature that does not require interactions is the grid system as explained in Section 2.5.2. A grid system is a visual feature, which helps positioning elements, but a user cannot interact with it.

Direct Interaction: Features with direct interactions are the most common. Direct interaction means that, with the addition of a feature, additional new interactions are added to the already existing ones of that tool. They are specific to a feature and would have no real use without it. They are usually intentionally performed by a user and their main purpose is to directly manipulate feature-specific data. An example for this is the peek definition feature explained in Section 3.5.3. To get a peek into the implementation of a function, the user has to directly interact with certain elements of the user interface,

by selecting a specific context menu. This interaction process is specific to the feature and the context menu would have no use if this feature did not exist.

Indirect Interaction: Indirect interactions, on the other hand, are not specific to a feature. Actions of a feature with indirect interactions can usually be triggered by performing interactions which are not part of the feature itself. Interactions, whose main purpose has nothing to do with the feature itself and could still be performed even if the feature did not exist. Actions which are controlled by indirect interactions are usually triggered either concurrently alongside actions of other features, or as a side effect of such. An example for a feature with indirect interactions is the ruler, which is visible in many text or diagram editors (Section 2.5.2). The basic functionality of a ruler is to take measurements of elements on the stage. It does not offer any direct interactions but adds small indicators, which represent edges of selected elements. Selecting an element on the stage is not an interaction method, which is specific to the ruler. Therefore, the action of adding those indicators to the ruler are performed by an indirect interaction.

Direct and Indirect Interaction: Some features offer both, direct and indirect, interactions. An example for such would be the basic scrollbar as explained in Section 3.5.3. A direct interaction would be to click and move the scrollbar. An indirect interaction would be to move the position of the current viewport by different means (e.g., with the mouse wheel). This would indirectly trigger the scrollbar to move as well.

Animation

This categorization gives an overview about the types of animations that a feature uses. Animations are important to help users understand what visual changes have just been made on the screen. The right use of animations prevents users from getting confused about these changes and increases their sense of orientation. In past literature a time span of 0.3 to 1.0 seconds has been suggested in which the animation should be played [CRM91] but it can vary depending on the type of information that is displayed.

Here, it is important to understand the difference between a separate animation that is played alongside a feature, and the feature's functionality itself. In this category, animations are considered visual techniques that physically move visual objects and are triggered by the user but not directly controlled by them. Most of the time, they are played right after the user's interaction has finished. What is not considered an animation, is physical movements of visual objects that are directly controlled by the user. For example, grabbing and moving a basic scrollbar is a continuous interaction process that can theoretically even be done in infinite detail. Practically, it is limited by the minimum amount of pixel that it can be moved in one movement, but, and this is the important aspect, it is fully controlled by the user. Although, during the interaction with the scrollbar, objects are moving, which may seem like an animation, it is considered the features functionality that is continuously executed and fully controlled by the user.

This dimension will consist of five categories: **No Animation**, **Informational Animation**, **Continuous Animation**, **Structural Animation**, and **Multiple Animation**.

No Animation: Features that do not utilize animations in any form. The basic scrollbar is an example for a feature which does not use any animations.

Informational Animation: Features, which utilize animations only in an informational way. Animations, which give the user additional information (often in the form of text), but do not directly interfere with elements on the stage. An example for this is given by Igarashi and Hinckley in [IH00, p. 142]: "..., when the user presses the mouse button, a pink slider appears." This slider gives information about the current position and scale level, but does not interfere with any data on the stage. Another example would be a small label which is transitioned in and out during a zooming interaction which shows the current zoom level. This could be done with discrete zoom levels (as demonstrated by yFiles in Section 2.5.1), or with continuous zooming actions which show the current zoom level as a percentage-based value.

Continuous Animation: Animations that take existing elements on the stage and change the way they are represented in a continuous process. An important property of those type of animations is that during the animation process, no additional information is added or removed. Examples would be a simple magnification of an element (basic zooming), or triggering a smooth transition by clicking on a proxy element (Section 3.5.3).

Structural Animation: This type of animation is used when the structure of elements or the stage is changed. Unlike continuous animations, here, new information is added, old are removed, or existing are changed. This new information could, for example, be entire elements or just properties of elements. An example for such an animation is the peek definition feature (Section 3.5.3). A transition is played, which gradually opens a box which shows information about a selected function.

Multiple Animations: Sometimes, a feature utilizes a combination of informational, continuous, and structural animations. In that case, features can be placed inside this category. *Multiple* is defined as any combination of *Informational Animation*, *Continuous Animation*, and *Structural Animation*. An example would be the already mentioned feature Speed-dependent Automatic Zooming [IH00]. It does not only utilize an informational animation in the form of a pink slider, but also a continuous animation: "When the user releases the mouse button, an animated transition gradually returns the document to the original base scale." [IH00]

Information Structure

Many tools, especially modeling tools, give the user the option to adjust the structure of the workspace. The user is able to personalize the workspace by adjusting the position of

objects inside it. Positions are saved and upon re-opening a file, objects are positioned at the same spot as the user remembered it. Not having to re-create the mental map of the workspace every time a file is opened ultimately saves a lot of the user's time and effort. Because of that, this characteristic is very valuable and should not be carelessly taken away. The larger an information space is, the more time a user requires to get used to a new structure, and therefore, the more important it is to keep the structure intact.

Nevertheless, preserving the structure is not always easily possible. For example, and, as explained in Section 3.3.4, a lot of problems arise when dealing with magnification or demagnification of individual objects. Another common cause for a change of the information structure is the automatic process of rearranging elements. E.g., some modeling tools offer a "center all elements" functionality, which automatically adjusts and centers the position of all elements. Such a functionality can be a dangerous game and finding a good algorithm that prevents the user from having to re-create the mental map of their workspace is a hard task. Because of that, it is often the best solution to stay away from implementing features that frequently change the layout of a user's workspace.

This dimension can be split up into four categories: **Structure Preserving**, **Structure Changing**, **Temporary Changing**, and **Hybrid**.

Structure Preserving: This category includes all features that do not change the structure at all, or only change it ever so slightly that the user's mental image is not destroyed. The key point of features in this category is that users do not have to re-create the mental map of their workspace. An example for such a feature is the semantic zoom in yFiles (Section 2.5.1). Zooming in adds additional information, but does not adjust the structure.

Structure Changing: Features in this category are changing the structure. This is essentially the direct opposite of structure preserving. Features in this category adjust the structure to a point that users are forced to having to re-create their mental map. An example is the grouping feature as demonstrated by yFiles (Section 2.5.1). Unlike their semantic zooming implementation, closing or opening groups automatically adjusts the structure. When opening a group, the position of elements inside the group are adjusted. When closing a group, the position of elements outside the group are changed to utilize the space that was previously populated by the expanded group.

Temporary Changing: Between structure preserving and structure changing, there exists a middle ground that only temporary changes the structure. Often, features visually change the structure, only for a limited amount of time, in order to execute a specific functionality (e.g., the peek definition feature, shown in Section 3.5.3). Such features do not persistently change the structure but merely temporally adjust it. Nevertheless, they can still destroy the mental map that a user has built over time, especially when layout changes are displayed over a long period of time. Whether it is a good idea to implement such a temporary change, or try to simply avoid it, has to be determined on a per-instance basis.

3.4.3 Data

This dimension describes the data that is used by a feature. Here, the feature-specific data is meant, which can usually be distinguished from the tool-specific data. If we use the scrollbar of a PDF reader software as an example, the tool-specific data would be the PDF itself, and the feature-specific data would be the x- and y-coordinates of the viewport. The scrollbar feature does not manipulate the PDF at all, instead it just moves the viewport to different locations by changing its coordinates. Unlike the tool-specific data, the feature-specific data is often independent of the tool itself which helps to keep this taxonomy more abstract.

This data can be split into input and output data. The input data is the data that is read by a feature in order to perform an action. The output data is the data that is modified or returned by an action of a feature. It can be compared to input and output parameter of a function in a programming language. Often, the main difficulty of this dimension is to define and understand what this data is, especially when the feature only exists in the form of a concept and has not been implemented yet. Both sub-dimensions, **Input Data Type** and **Output Data Type**, consist of the same characteristics: **Qualitative**, **Quantitative/Discrete**, **Quantitative/Continuous**, and **Mixed**.

Qualitative data is semi-structured data, such as labels, attributes, or entire domain model elements. Quantitative data can be counted or measured and is expressed as numbers. Furthermore, quantitative data exists in two variations, i.e., *quantitative/discrete* and *quantitative/continuous*. Quantitative/discrete data is countable and can only take on certain values. Continuous data, on the other hand, is measurable and can be split up into smaller parts. Some features work with more than just one data set. Such features should be classified as *Mixed*. Most of the time, it is ideal for a feature to work with quantitative/continuous data. This is because continuous data reflect user interactions more directly and responsively. It is easier to visually follow continuously rendered changes than discretely rendered ones. This is also reflected by Cockburn and Savage in [CS04]. They state: "The abrupt transitions between discrete zooming levels ... meant that the participants had to reorient themselves with each zoom action." [CS04, p. 97] For that reason, features that work with continuous data often do not require additional animations, unlike discrete data.

3.5 Taxonomy Evaluation

For the evaluation of the taxonomy, we followed a twofold descriptive evaluation strategy: First, we theoretically evaluated the contents of the taxonomy. Secondly, we implemented a prototype of one advanced information visualization feature of our taxonomy to show the practical relevance as well.

3.5.1 Theoretical Evaluation

Both of these strategies follow the "Illustrative Scenario" methods mentioned by Kundisch et al. [KMO⁺21]. More accurately, it follows both, the "Illustrative scenario with existing research" and the "Illustrative scenario with real-world objects" methods. Existing research is represented by utilizing novel features and implementations of past literature, such as Onion-graphs [KM07] or City Lights [ZMG⁺03]. Real-world objects are represented by features found in widely used tools, such as the Eclipse IDE or Microsoft PowerPoint. Even though, developers usually have a deep insight into the architecture of their features, we think that this taxonomy is abstract enough, so that this additional knowledge is not needed.

Appendix 2 shows a list of all evaluated features in the course of the ex post evaluation.

3.5.2 Practical Evaluation

For the practical evaluation, we realized a prototype that demonstrates an advanced visualization feature realized in the Eclipse Graphical Language Server Protocol Platform². At the time of writing, the Eclipse-GLSP is the most advanced and a highly maintained graphical language server platform, built on a modern technology stack which serves as a very good base for new and advanced features. Further details about this prototype can be found in Section 4. During the conceptualization of both developed features, characteristics of this taxonomy have been used to identify important aspects related to its presented user interface. Furthermore, both prototypes have been classified and added to the taxonomy, which can be found in Appendix 2.

3.5.3 Evaluated Features

The remaining part of this section will describe and give short summaries of evaluated features. These features originate either from past literature or existing software products. Some of these features were already explained in Section 2.5.2 and Section 2.5.3. Additionally, these features will be categorized and grouped into the classes of this taxonomy. While the categorization of some features was simple and straightforward, some others required more time and interpretation. Appendix 2 shows all evaluated features and their categorization. Additionally, it will briefly describe what we considered to be their input and output data.

Basic Scrollbar

The basic scrollbar is a rather old feature which is used by interfaces that need to show information that do not fit on the screen all at once. It is usually represented by a small vertical or horizontal bar which represents the full vertical or horizontal space of the entire workspace. The vertical or horizontal space which is visible inside the viewport is represented by a second bar ("handle" or "slider"). The relation between the

²<https://www.eclipse.org/glsp/>, last visited: 14.11.2021

vertical/horizontal space of the workspace and the vertical/horizontal space of information currently represented inside the viewport is the same as relation between the length of the entire scrollbar and the length of the handle. Because of that, a scrollbar gives a quick and accurate hint about how long a document is and where the currently visible information is located. The handle can be moved around by clicking and dragging it. Interactions with the scrollbar instantaneously affect the information represented inside the viewport and vice-versa.

Today, the basic scrollbar is a very widely used tool that is present in almost all graphical user interfaces on all kinds of devices. As long as some basic concepts are followed, the idea of a scrollbar is immediately recognizable by a user and they are able to use them without issues. An example for such a basic concept is the location of the scrollbar. A study by Devine and Andre [DA05] showed that users made much more cursor movement errors when the scrollbar was located at the left side of the interface instead of the right side. Besides that, scrollbars sometimes also offer additional functionality and show more than just spatial information, which can be seen in features such as the overview scrollbar.

Overview Scrollbar

The overview scrollbar follows the same concept as the basic scrollbar (explained in Section 3.5.3). The important difference is that it shows more than just spatial information. This extra information is usually first-class information directly part of the information space. An example can be seen in the text editor Sublime Text 3 (Figure 3.2b) which additionally shows the entire workspace in much smaller size.

Even though this type of scrollbar is not often seen, it is still quickly understood and easy to use because of how widely known the concept of a scrollbar is. Despite being easy to use, the negative aspect of having to switch focus often between multiple views is amplified in comparison the basic scrollbars because of extra information that is given.

Lens on Scrollbar Hover

This type of scrollbar is a mixture of a basic scrollbar and an overview scrollbar. Instead of continuously showing an overview of the workspace, like an overview scrollbar, it is only shown when the user hovers the cursor above the scrollbar. Another difference is that the overview is shown inside a pop-up window instead of continuously on the side. The pop-up window is part of the scrollbar view, because it is spatially connected to it when it is shown. In comparison to the overview scrollbar, this saves a lot of space that can be utilized by the main view instead.

An example of this type of scrollbar is present in the software development IDE Webstorm (Figure 3.7).

Magnifying Glasses

This feature is a prime example for an overview-plus-detail interface. Besides the information that is shown inside the main view, a magnified version of them is also shown

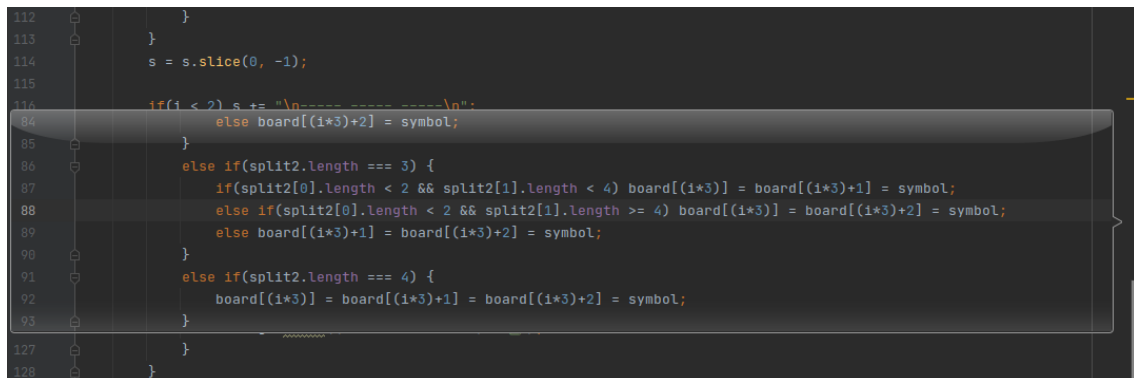


Figure 3.7: The software development IDE Webstorm opens up a pop-up window when the user hovers their cursor above the scrollbar which shows code of the corresponding location inside a document.

inside a secondary view. The magnification process usually consists of a simple zoom implementation which just increases the size of displayed information. This could also be combined with a level-of-detail functionality which shows different versions of information depending on the magnification factor.

The position of the secondary view is not defined and can be different from implementation to implementation. The two most common strategies are, either, movable and attached to the cursor, or not movable and positioned somewhere at a fixed location. Inside the Magnifier app on Windows 10, it can be switched between both of them. An example for the movable lens can be seen in Figure 3.5. The unmovable version of magnifying glasses is very similar to the minimap feature (Section 3.5.3) which is commonly seen in modeling tools with the main difference that the minimap demagnifies information instead of magnifying them.

Thumbnail Overview

The thumbnail overview follows a similar concept to the overview scrollbar. They both show similar information and give the user the option to quickly navigate to specific areas of the workspace. Inside a smaller secondary view, thumbnails of elements of the information space are shown which are linked to discrete areas inside the main view. It is often used in an information space that consists of self-contained elements. Elements of which thumbnails can easily be created. The key difference to overview scrollbars is that thumbnail overviews are usually completely independent of the main view and do not give any spatial information about it. Most of the time, these separate views of thumbnail overviews even have their own scrollbars. An Example is given in Figure 3.3 which shows thumbnails of the workspace inside a small vertical view to the left of the main view.

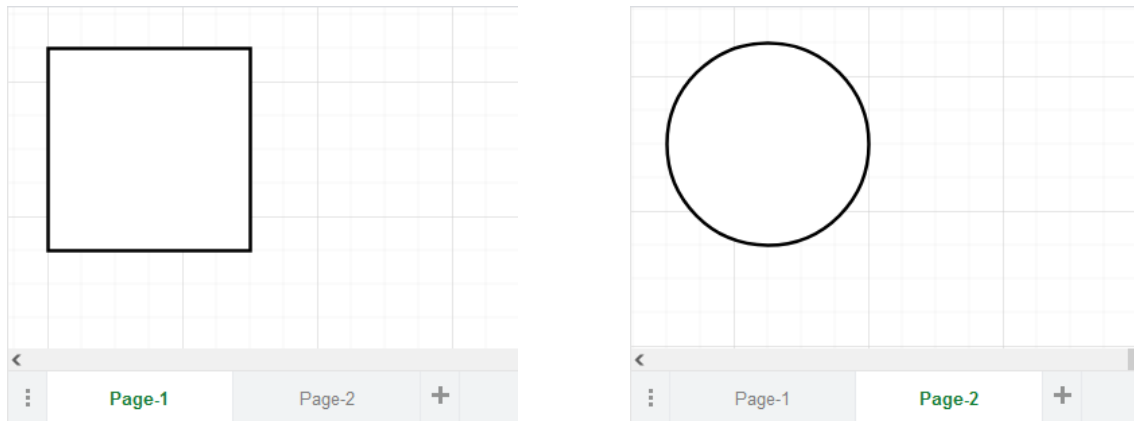
Another version of this feature is present in most modern operating systems, for example, Windows 10. With the click of a specific combination of keys (ALT+TAB in Windows 10), thumbnails of all programs that are currently open are displayed on the screen. This

can be used to get a quick overview of all open programs, and helps to quickly navigate between different programs. Unlike the other variant of this feature, this is done in the current view and is therefore classified as a focus-plus-context feature.

User-controlled View Definition

This feature can be found in some way in many diagramming tools, as explained in Section 2.5.3 1. The idea is to give the user the ability to define a specific view onto the workspace. These views are to differentiate from the more physical interface-specific views, which are used when talking about interface schemes like overview-plus-detail or focus-plus-context. The views that are definable by this feature are usually called pages, layers or frames. One of their purposes is to quickly navigate the workspace in a way that can be defined by the user. Some even add more functionality, such as grouping elements together.

An example for pages can be seen in Figure 3.8



(a) First page selected.

(b) Second page selected, which shows a new canvas with a different element.

Figure 3.8: The diagramming tool on <https://app.diagrams.net> allows to create new pages via a small view which is displayed underneath the stage. A new page creates a new empty canvas to work with.

Minimap

Minimap feature as explained on Section 2.5.2.

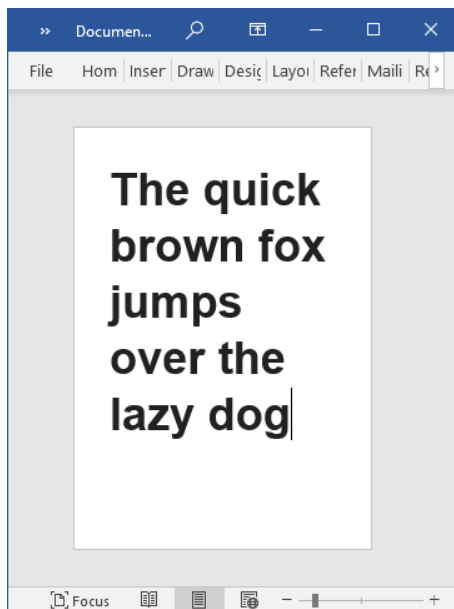
Basic Continuous Zoom

This feature is present in almost all user interfaces of today and is considered the basic "zoom" feature. It allows to magnify and demagnify information in a continuous way. This is usually done by figuratively moving the camera closer to, or further away from the stage. Elements appear larger or smaller which creates the magnification effect. A problem that sometimes arises with this technique is that objects on the stage are not perfectly scalable

which causes them to look poorly. In order to get the full benefit from a continuous zoom, it is best to use it with a scalable data structure as, for example, SVG.

Zoom events can be triggered in various ways. Usually, they are triggered by the user directly by either the press of buttons directly part of the user interface, such as a dedicated "zoom in" or "zoom out" button or a slider, or common gestures like a double click or the mouse wheel. Gestures are often not immediately intuitive to the user and, as mentioned in [CKB09], sometimes never even discovered by them. Besides being directly triggered by users, they are often also triggered and used by other features. For example, cue-based techniques like proxies can utilize them to lead users to specific parts of the workspace.

An example for a basic zoom feature can be seen in Figure 3.9.



(a) The full document is visible showing the whole sentence (zoomed out).



(b) Only the last letter is visible (zoomed in).

Figure 3.9: Example of a basic zooming event performed in Microsoft Word. Applied via the slider which can be seen at the bottom right of the user interface.

Basic Discrete Zoom

This feature is almost the same as basic continuous zooming explained in Section 3.5.3. The key difference is, as the name already tells, it works stepwise and therefore in a discrete way. This technique is often applied to discrete data that cannot be interpolated.

An example for this would be the map tiles in Google Maps as explained in Section 2.3.2. Map tiles exist in about 21 different level and interpolation between them is not easily possible. A zooming interface which only switches between these 21 levels would be a discrete zooming interface. Google utilizes a combination of a discrete and continuous

zoom in their interfaces. Between the discrete switch of levels, they simply continuously increase the size of individual tiles. Once they become too large, a discrete switch to tiles of the next level is made.

Speed-dependent Automatic Zooming

This navigation technique is proposed by Igarashi and Hinckley in [IH00]. The goal is to make navigation through large documents more efficient for a user. This is achieved by incorporating a semantic zoom into the scrolling process. The view of the document is zoomed out when a user increases their scrolling speed and zoomed back in when it is decreased again. Instead of adjusting the speed at which a document is moved through the screen, the zoom level is adjusted, and the speed stays constant. This leads to less distortion and makes it easier to locate specific parts of a document during the scrolling process.

They tested their idea on a web browser, a map viewer, an image browser, a dictionary viewer, and a sound editor, and conducted a usability study on the web browser and the map viewer. Especially the web browser yielded good results with six out of seven participants preferring automatic zooming over the traditional scrolling technique with a basic scrollbar. In another study, conducted by Cockburn and Savage [CS04], participants were observed to complete basic scrolling tasks on average 22% faster for document browsing and 43% faster for map browsing. Negative aspects are user study participants getting dizzy from the constant flow of text, the fact that it needs some time to get used to, and the requirement of a relatively good hand-eye coordination to efficiently control the zooming event.

The browser prototype used section headings and images as subjects for a semantic zoom. Headings and images are important for a user to not lose spatial orientation during a zoom event and - as can be seen in Figure 3.10 - are increased in size when the document is zoomed out. Igarashi and Hinckley suggest in their work that automatic zooming is best used on an information space of intermediate size with spatially organized information.

Adding Objects of Different Information Space on Zoom

This feature forms the basic concept of semantic zooming. Semantic zoom transforms the visual representation of objects during a zoom event. This can mean that information is added, removed, or simply changed. In this category, elements of a different information space (often of different documents or files) are seamlessly added during the zoom-in process and removed during the zoom-out process. This is often done to provide additional information to the user and save them from navigating through multiple documents in parallel. Different levels of detail are defined and their visibility is usually dependent on the current zoom factor or the remaining space. The number of existing levels is dependent on the information domain and could theoretically even be infinite with recursive detail-levels.

An example for such a feature is explained by Frisch et al. in [FDB08]. They propose to nest different UML diagrams into each other to allow for quicker navigation. They are

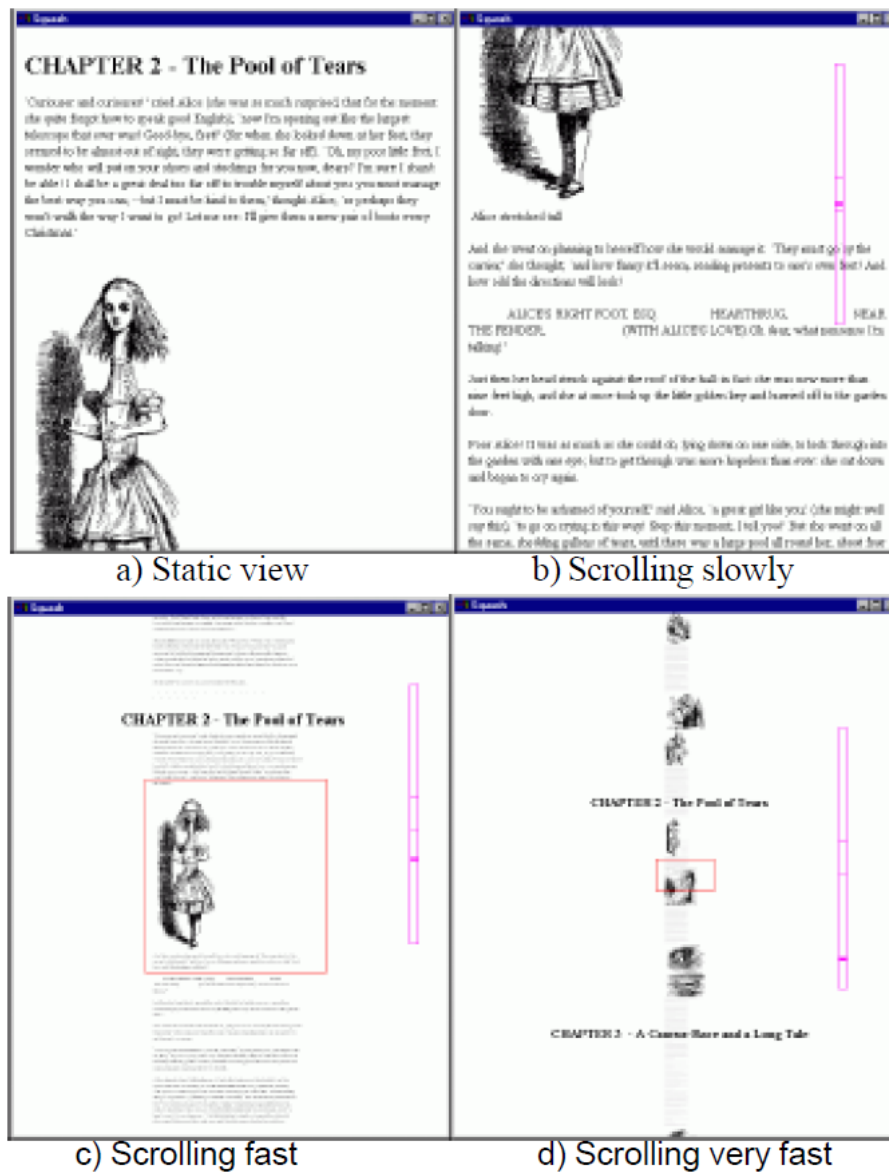


Figure 3.10: Example for speed-dependent automatic zooming in a web browser. The faster a user scrolls, the further the document is zoomed out. A semantic zoom is applied to section headings and images.

Image source: [IH00]

then seamlessly added and removed during zoom events. This can be seen in Figure 3.11 which shows a UML activity diagram nested into a use-case diagram.

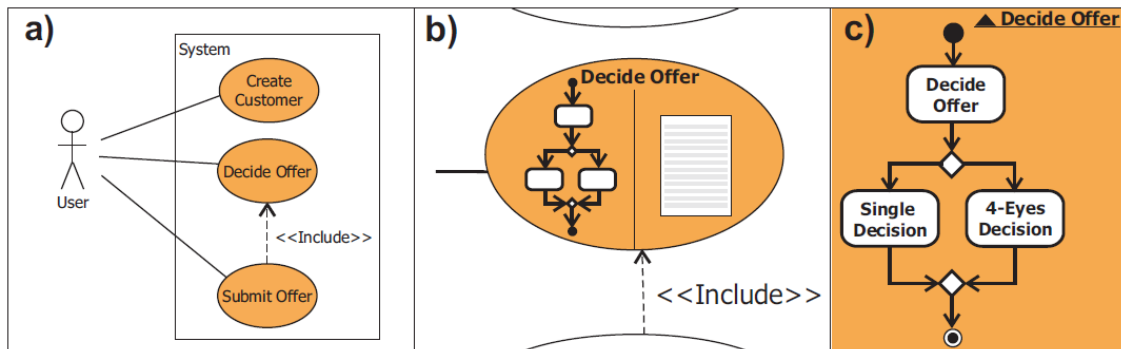


Figure 3.11: Example for semantic zooming applied to UML diagrams. In b), an activity diagram is added to the use-case diagram of a) (different information space). In c), more details are added to the diagram of b).

Image source: [FDB08]

Adding Objects of Current Information Space on Zoom

As already explained in the previous section, this feature represents the basic concept of semantic zooming. Conceptually, it is very similar to the previous section, with the difference that information which are added, changed, or removed come from the same information space. This is mainly done to effectively use the space that is available by only showing the most important information that fit onto the stage. It gives users an easier time to comprehend information by keeping the workspace clear. Besides that, this feature is sometimes also applied with the main purpose of simply increasing performance. Often, very small elements that cannot be read anymore are removed from the stage to allow for faster rendering.

An example for such a feature is explained by Frisch et al. in [FDB08]. Besides nesting different diagrams, they also propose to split up different UML diagrams into multiple levels of detail. They are then seamlessly added when there is enough space available during a zoom event. This can be seen in Figure 3.11 which shows a UML activity diagram in two different levels of detail.

Adding Details on Click

This feature hides and shows details about an object by the click of a button and is explained in more detail in Section 2.5.3 2. It is present in many graphical user interfaces nowadays, especially code editors. An example for this feature inside a modeling tool can be seen in Figure 3.12.

Traditional Fisheye Zoom

The fundamental goal of a fisheye zoom goes hand in hand with the concept of focus-plus-context interfaces. Its goal is to provide a balance between local detail and global context

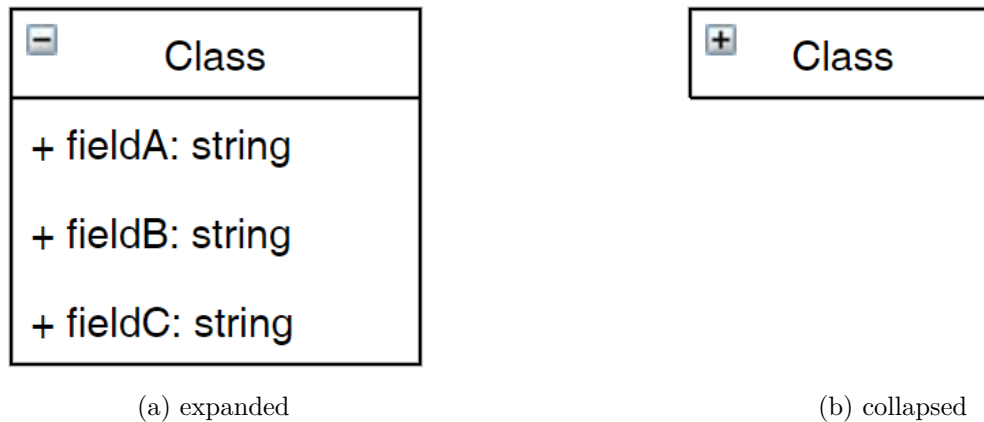


Figure 3.12: Example for showing additional information on button click in diagramming tool <https://app.diagrams.net>. A user is able to expand/collapse the object by pressing the button on the top left.

[Fur86] and they are suggested to be used in combination with large information spaces [SZG⁺96]. Unlike simple magnifying glasses, the space and information between detail and context is usually distorted in some way (as can be seen in Figure 3.4).

Over the years, fisheye zoom interfaces have been implemented in many different ways and domains. Furnas describes a generalized fisheye view in [Fur86], which calculates a degree of interest based on the *a priori* importance of an element and the distance to the current focus. Elements with a high degree of interest are then shown in focus while elements with a low degree are only shown in context or not at all.

Bartram and Dill describe an algorithm in [BHDH95] with the goal to automate sizing of notes in large information spaces. Their algorithm is also based on a degree of interest and is applied recursively to consider neighboring notes and efficiently utilize leftover space.

Reinhard et al. improve that algorithm in [RMG07] by specifically considering zoom-in and zoom-out actions. Besides that, they also made sure that a model can be edited without putting the structure of it into an unstable state.

While the algorithms mentioned above apply a fisheye zoom in a more generalized way, there also exist a lot of concrete implementations or ideas. For example, fisheye menus [Bed00], a collaborative fisheye text editor [Gre96] or the typical dock of Apple’s operating system macOS which can be seen in Figure 3.13.

City Lights

City Lights is a focus-plus-context technique by Zellweger et al. [ZMG⁺03]. They give information about the context of the focused view and are placed at the border of it. More specifically, they are able to show multiple types of information about unseen objects, e.g, existence, physical properties, positional properties, and abstract information. One goal

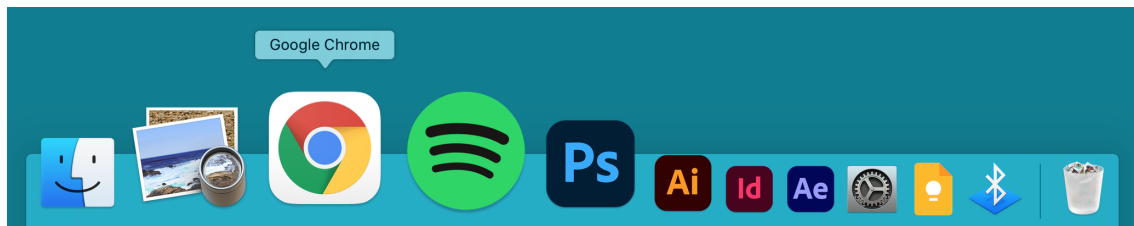


Figure 3.13: Example for a fisheye implementation in Apple’s operating system macOS. Hovering over the dock creates a focus on that particular icon by increasing its size. Neighboring icons are distorted by only slightly increasing their size.

of City Lights is to show those types of information in minimal space. Because seeing all information inside this minimal space can be overwhelming, they suggest adding a user interface that lets users decide which type of information they want to see.

Categorizing this feature is not a simple task. Although it is very similar to scrollbars, which are an overview-plus-detail interface, City Lights are classified as a focus-plus-context interface here, because of the spatial dependency between focus- and context-view. Neither view can be repositioned without affecting the other. On top of that, it can also be classified as a cue-based technique because it gives cues about off-screen elements, just like halos or wedges do. Another reason why it is listed under focus-plus-context instead is also because Zellweger et al. in their original work speak of it as giving context to a focused view.

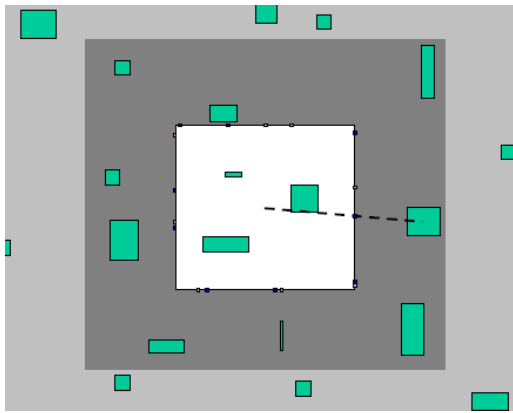
Figure 3.14 shows and explains concepts of City Lights. It also shows how enabling or disabling different types of information would look like.

EdgeRadar

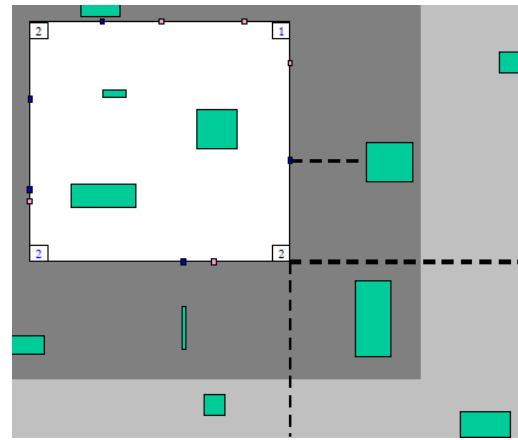
EdgeRadar is a visualization technique by Gustafson and Irani [GI07]. It works in a similar way as City Lights, explained in the previous section, and displays off-screen elements around the border of the focus-view. Key differences are that EdgeRadar displays elements in a more simplified way and with less information, and EdgeRadar is specifically designed to support moving targets. While City Lights is also able to display physical properties and abstract information, EdgeRadar merely shows information about existence and distance of objects. The advantage of EdgeRadar is that it is easier to understand, especially for new users.

An experiment was conducted by Gustafson and Irani to determine the effectiveness of tracking off-screen elements with EdgeRadar in contrast to halos. Tracking off-screen elements yielded a better result for their solution, with an 16.67% error-rate for EdgeRadar and 21.53% for halos.

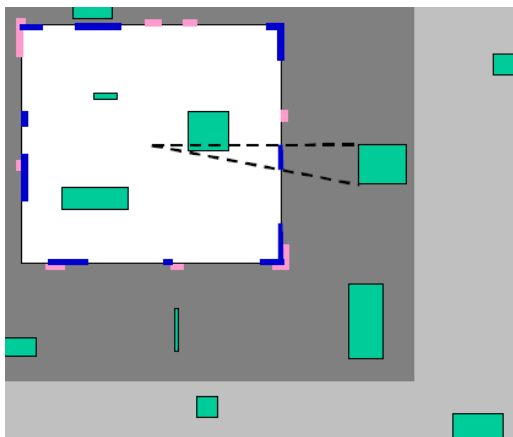
Figure 3.15 shows and explains the concept of EdgeRadar.



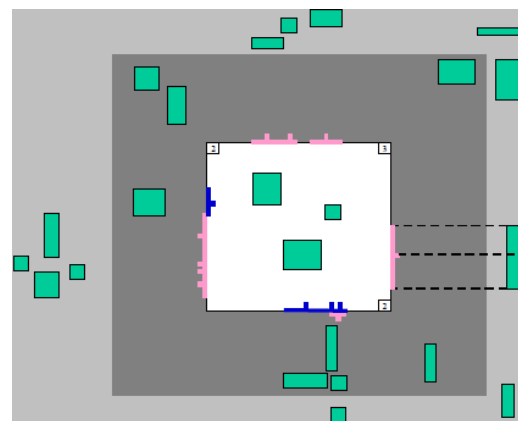
(a) "Dark/light fills indicate the near/far regions of unseen populated space around a focused view. Points show radial direction (see dotted line). Grays/colors show near/far distance." [ZMG⁺03, p. 4]



(b) "Points show orthographic direction. Gray/color shows near/far distance. Markers in the corners of the focused view indicate the number of unseen objects in the corresponding corner of unseen populated space (see dashed corner region)." [ZMG⁺03, p. 4]



(c) "Lines show radial direction. Distance is also shown if the object size is known. Grays/colors show near/far distance. Line positions inside and outside the border show near/far distance, which also reduces line overlap." [ZMG⁺03, p. 4]



(d) "Redundant points and lines show the orthographic direction of objects and clusters. Positions and grays/colors show near/far distance." [ZMG⁺03, p. 4]

Figure 3.14: Example for the focus-plus-context feature City Lights by Zellweger et al. [ZMG⁺03]

Image source: [ZMG⁺03]

Visualizing Off-Screen Elements of Node-Link Diagrams

This feature is very similar to City Lights and EdgeRadar, and is presented in [FD13] by Frisch and Dachsel. Their goal is to replace time consuming panning and zooming

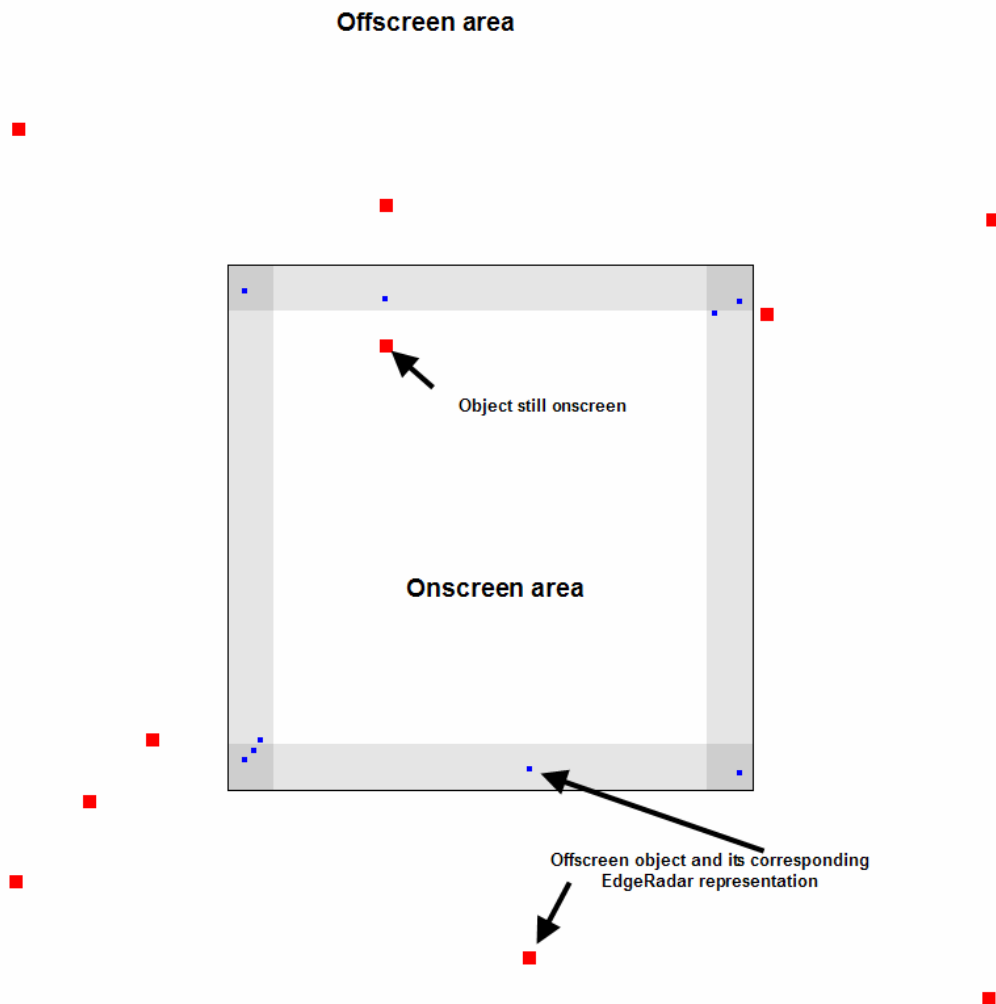


Figure 3.15: Example for focus-plus-context feature EdgeRadar by Gustafson and Irani [GI07].

Image source: [GI07]

actions in large diagrams with methods such as proxies and advanced navigation techniques. Although it is very similar to other implementations and restricted to node-link diagrams, their paper is mentioned here because of how relevant their methods are to this work.

They use proxies to display off-screen elements in an interactive border region which is

located at the edge of the viewport. They provide spatial, structural and topological information about off-screen elements which are tailored for UML diagrams. Because their work is specific to node-link diagrams such as UML, they are able to provide a better integrated environment with more features in comparison to City Lights or EdgeRadar.

Visualization of off-screen elements with the help of proxies is done with small colored indicators inside the interactive border view (Figure 3.16). Relationships from off-screen elements to on-screen elements are preserved by re-routing them to proxies. They explore many different techniques and give information about advantages and drawbacks. These techniques consist of, for example, coloring proxies without edges differently, different positioning for proxies (orthogonal and radial projection), coloring re-routed edges in different colors than original routes, different edge routing methods, and different methods for edge cases such as one off-screen node having edges to multiple on-screen nodes.

Furthermore, they create clusters of proxy elements. This is especially useful for large diagrams. These clusters are created, for example, for off-screen elements inside the corner areas, and for off-screen elements that would create overlapping proxy elements. They present two methods for clustering: Geometric and structural clustering.

For very large diagrams, they suggest defining an area of influence. This area of influence is part of the off-screen area around the viewport. To reduce cluttering, only elements which are positioned inside the area of influence are represented as proxies. This area moves with panning actions, grows/shrinks with zooming actions, and can also be dependent on the number of surrounding off-screen elements.

Besides visualization, they also mention interaction methods. This includes: Automatic panning and zooming to an off-screen element by clicking a proxy, being able to open more detailed previews of elements by hovering over a proxy, being able to edit properties of off-screen elements via their proxies, and being able to add edges from on-screen to off-screen elements.

To test their prototype, Frisch and Dachsel also conducted a pilot study. Besides being asked about opinions and comprehensibility of individual features, participants were asked to perform other tasks such as estimate directions of off-screen elements or navigate to certain classes. Following the comments and suggestions of participants, Frisch and Dachsel adjusted individual features and methods. This was then followed by another controlled experiment which showed that the exploration of relationships in their approach outperformed state-of-the-art interfaces.

An example of their proposed approach can be seen in Figure 3.16

Peek Definition

This feature is often seen in code editors, for example, Visual Studio Code and most JetBrains IDEs. It allows users to quickly navigate and see the definition of, e.g., a function. Instead of switching to the file and line number which holds this definition, it is shown inside a smaller window which is part of the main view. In the context of a

3. TAXONOMY

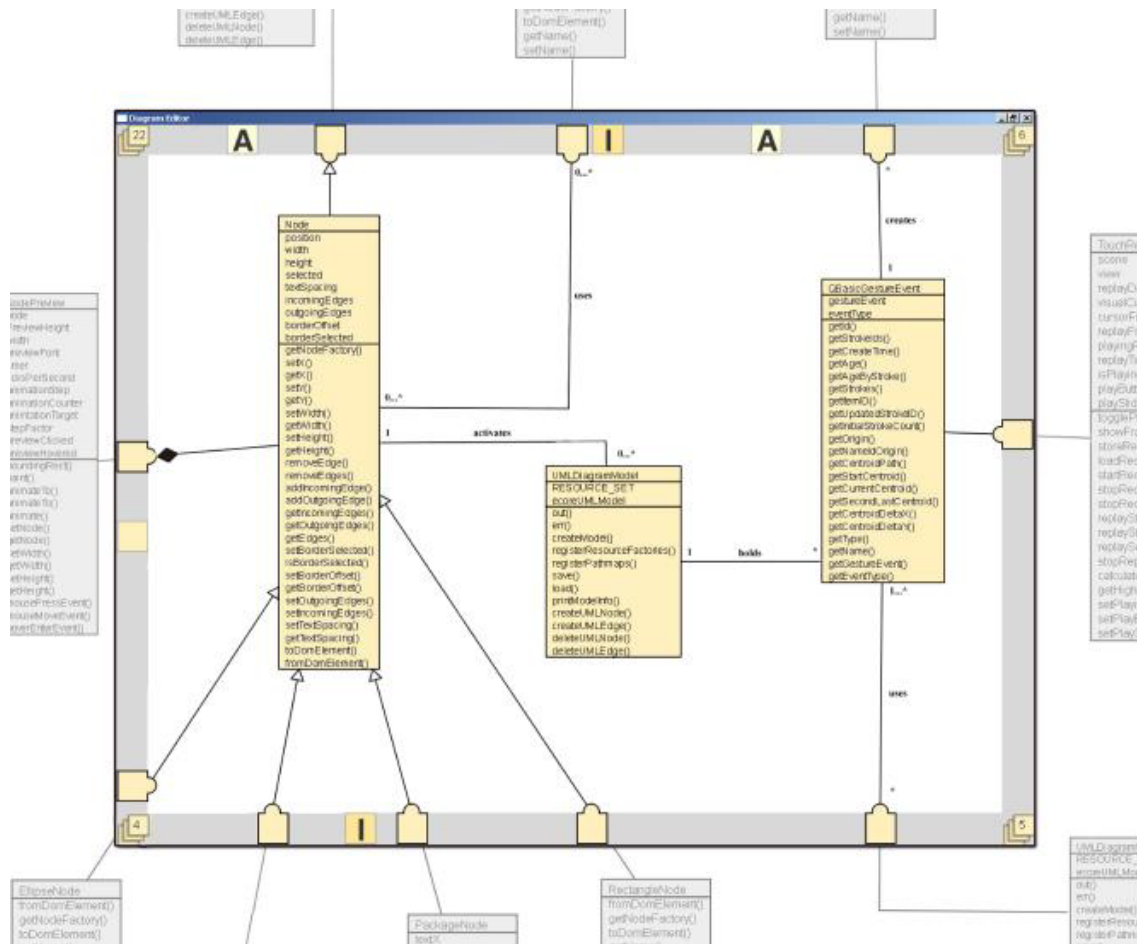
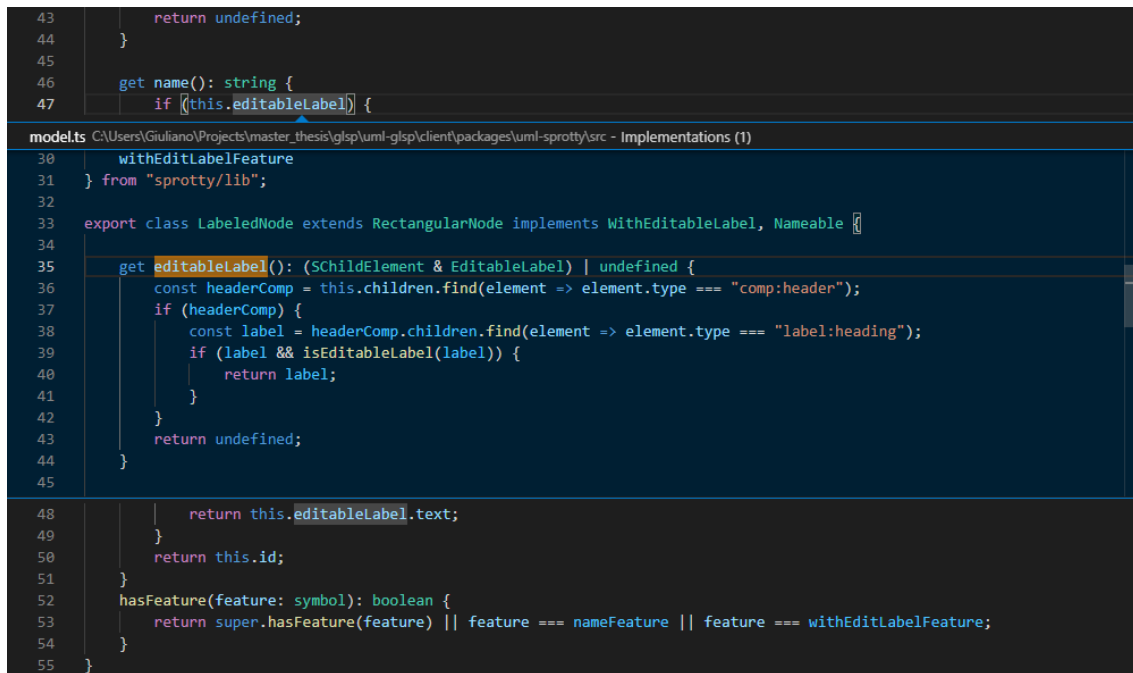


Figure 3.16: Example for off-screen element visualization by Frisch and Dachsel [FD13]. "Principle of our solution: The viewport of our UML class diagram editor with off-screen visualization (center). Classes clipped from the viewport (shown outside in gray) are represented by proxy elements located within the interactive border region." [FD13, p. 135] Image source: [FD13]

focus-plus-context interface, this small window becomes the focus-view, while the file that is currently viewed becomes the context-view. This extra window effectively creates a gap in the current file and is placed between the line which shows the function and the next line. It usually acts as its own, from the current file independent, file editor which only shows the selected function by default. The user is able to scroll further to the top or bottom to see additional content of the file. The entire file can also be changed and edited.

Independent of whether the definition is located in the current file or a different file, the behavior of this feature is always the same. It can usually be triggered via the context menu of a function or by pressing a hotkey while a function is marked.

An example is shown in Figure 3.17



```

43     return undefined;
44 }
45
46 get name(): string {
47     if (this.editableLabel) {
48
49         return this.editableLabel.text;
50     }
51     return this.id;
52 }
53
54 hasFeature(feature: symbol): boolean {
55     return super.hasFeature(feature) || feature === nameFeature || feature === withEditLabelFeature;
56 }
57
58 }
59
60 }
61
62 }
63
64 }
65
66 }
67
68 }
69
70 }
71
72 }
73
74 }
75
76 }
77
78 }
79
80 }
81
82 }
83
84 }
85
86 }
87
88 }
89
90 }
91
92 }
93
94 }
95
96 }
97
98 }
99
100 }
101
102 }
103
104 }
105
106 }
107
108 }
109
110 }
111
112 }
113
114 }
115
116 }
117
118 }
119
120 }
121
122 }
123
124 }
125
126 }
127
128 }
129
130 }
131
132 }
133
134 }
135
136 }
137
138 }
139
140 }
141
142 }
143
144 }
145
146 }
147
148 }
149
150 }
151
152 }
153
154 }
155
156 }
157
158 }
159
160 }
161
162 }
163
164 }
165
166 }
167
168 }
169
170 }
171
172 }
173
174 }
175
176 }
177
178 }
179
180 }
181
182 }
183
184 }
185
186 }
187
188 }
189
190 }
191
192 }
193
194 }
195
196 }
197
198 }
199
200 }
201
202 }
203
204 }
205
206 }
207
208 }
209
210 }
211
212 }
213
214 }
215
216 }
217
218 }
219
220 }
221
222 }
223
224 }
225
226 }
227
228 }
229
230 }
231
232 }
233
234 }
235
236 }
237
238 }
239
240 }
241
242 }
243
244 }
245
246 }
247
248 }
249
250 }
251
252 }
253
254 }
255
256 }
257
258 }
259
260 }
261
262 }
263
264 }
265
266 }
267
268 }
269
270 }
271
272 }
273
274 }
275
276 }
277
278 }
279
280 }
281
282 }
283
284 }
285
286 }
287
288 }
289
290 }
291
292 }
293
294 }
295
296 }
297
298 }
299
300 }
301
302 }
303
304 }
305
306 }
307
308 }
309
310 }
311
312 }
313
314 }
315
316 }
317
318 }
319
320 }
321
322 }
323
324 }
325
326 }
327
328 }
329
330 }
331
332 }
333
334 }
335
336 }
337
338 }
339
340 }
341
342 }
343
344 }
345
346 }
347
348 }
349
350 }
351
352 }
353
354 }
355
356 }
357
358 }
359
360 }
361
362 }
363
364 }
365
366 }
367
368 }
369
370 }
371
372 }
373
374 }
375
376 }
377
378 }
379
380 }
381
382 }
383
384 }
385
386 }
387
388 }
389
390 }
391
392 }
393
394 }
395
396 }
397
398 }
399
400 }
401
402 }
403
404 }
405
406 }
407
408 }
409
410 }
411
412 }
413
414 }
415
416 }
417
418 }
419
420 }
421
422 }
423
424 }
425
426 }
427
428 }
429
430 }
431
432 }
433
434 }
435
436 }
437
438 }
439
440 }
441
442 }
443
444 }
445
446 }
447
448 }
449
450 }
451
452 }
453
454 }
455
456 }
457
458 }
459
460 }
461
462 }
463
464 }
465
466 }
467
468 }
469
470 }
471
472 }
473
474 }
475
476 }
477
478 }
479
480 }
481
482 }
483
484 }
485
486 }
487
488 }
489
490 }
491
492 }
493
494 }
495
496 }
497
498 }
499
500 }
501
502 }
503
504 }
505
506 }
507
508 }
509
510 }
511
512 }
513
514 }
515
516 }
517
518 }
519
520 }
521
522 }
523
524 }
525
526 }
527
528 }
529
530 }
531
532 }
533
534 }
535
536 }
537
538 }
539
540 }
541
542 }
543
544 }
545
546 }
547
548 }
549
550 }
551
552 }
553
554 }
555
556 }
557
558 }
559
560 }
561
562 }
563
564 }
565
566 }
567
568 }
569
570 }
571
572 }
573
574 }
575
576 }
577
578 }
579
580 }
581
582 }
583
584 }
585
586 }
587
588 }
589
590 }
591
592 }
593
594 }
595
596 }
597
598 }
599
600 }
601
602 }
603
604 }
605
606 }
607
608 }
609
610 }
611
612 }
613
614 }
615
616 }
617
618 }
619
620 }
621
622 }
623
624 }
625
626 }
627
628 }
629
630 }
631
632 }
633
634 }
635
636 }
637
638 }
639
640 }
641
642 }
643
644 }
645
646 }
647
648 }
649
650 }
651
652 }
653
654 }
655
656 }
657
658 }
659
660 }
661
662 }
663
664 }
665
666 }
667
668 }
669
670 }
671
672 }
673
674 }
675
676 }
677
678 }
679
680 }
681
682 }
683
684 }
685
686 }
687
688 }
689
690 }
691
692 }
693
694 }
695
696 }
697
698 }
699
700 }
701
702 }
703
704 }
705
706 }
707
708 }
709
710 }
711
712 }
713
714 }
715
716 }
717
718 }
719
720 }
721
722 }
723
724 }
725
726 }
727
728 }
729
730 }
731
732 }
733
734 }
735
736 }
737
738 }
739
740 }
741
742 }
743
744 }
745
746 }
747
748 }
749
750 }
751
752 }
753
754 }
755
756 }
757
758 }
759
760 }
761
762 }
763
764 }
765
766 }
767
768 }
769
770 }
771
772 }
773
774 }
775
776 }
777
778 }
779
780 }
781
782 }
783
784 }
785
786 }
787
788 }
789
790 }
791
792 }
793
794 }
795
796 }
797
798 }
799
800 }
801
802 }
803
804 }
805
806 }
807
808 }
809
810 }
811
812 }
813
814 }
815
816 }
817
818 }
819
820 }
821
822 }
823
824 }
825
826 }
827
828 }
829
830 }
831
832 }
833
834 }
835
836 }
837
838 }
839
840 }
841
842 }
843
844 }
845
846 }
847
848 }
849
850 }
851
852 }
853
854 }
855
856 }
857
858 }
859
860 }
861
862 }
863
864 }
865
866 }
867
868 }
869
870 }
871
872 }
873
874 }
875
876 }
877
878 }
879
880 }
881
882 }
883
884 }
885
886 }
887
888 }
889
890 }
891
892 }
893
894 }
895
896 }
897
898 }
899
900 }
901
902 }
903
904 }
905
906 }
907
908 }
909
910 }
911
912 }
913
914 }
915
916 }
917
918 }
919
920 }
921
922 }
923
924 }
925
926 }
927
928 }
929
930 }
931
932 }
933
934 }
935
936 }
937
938 }
939
940 }
941
942 }
943
944 }
945
946 }
947
948 }
949
950 }
951
952 }
953
954 }
955
956 }
957
958 }
959
960 }
961
962 }
963
964 }
965
966 }
967
968 }
969
970 }
971
972 }
973
974 }
975
976 }
977
978 }
979
980 }
981
982 }
983
984 }
985
986 }
987
988 }
989
990 }
991
992 }
993
994 }
995
996 }
997
998 }
999
1000 }

```

Figure 3.17: Example for the peek definition feature in Visual Studio Code. It shows the focus-view at the center, while the context-view is visible at the top and bottom. The focus-view shows the definition of the function *editableLabel()* and the context-view shows the area at which this function is called.

Grouping Elements

This feature is explained in more detail in Section 2.5.1 and Section 2.5.3.

An example can be seen in Figure 2.2.

Onion Graphs

Onion graphs are a technique presented by Kagdi and Maletic in [KM07]. It evolves around the problem of visualizing large UML models and abstracts contextual information by utilizing the hicon onion notation [SGJ93]. The goal of their work is to abstract visual elements of a UML diagram at various levels of detail and represent them in different notations. Their proposed notations are designed to reduce edges which originates from the problem of having too many crossing edges in large information spaces.

They try to preserve semantic and structural information with their notations to help users keep their mental model of their workspace. Semantics are preserved by using a notation that is already present in UML, e.g., classes, generalizations, or associations. Structural information is preserved by ordering notations and adjusting their width and height. The width of a notation, for example, indicates how many siblings the original note has, and the height represents how many children it has.

3. TAXONOMY

This allows to reduce a UML graph as it can be seen in Figure 3.18. They also mention that semantic zooming and incremental exploration is often used in combination with onion notations and could therefore be applied here as well.

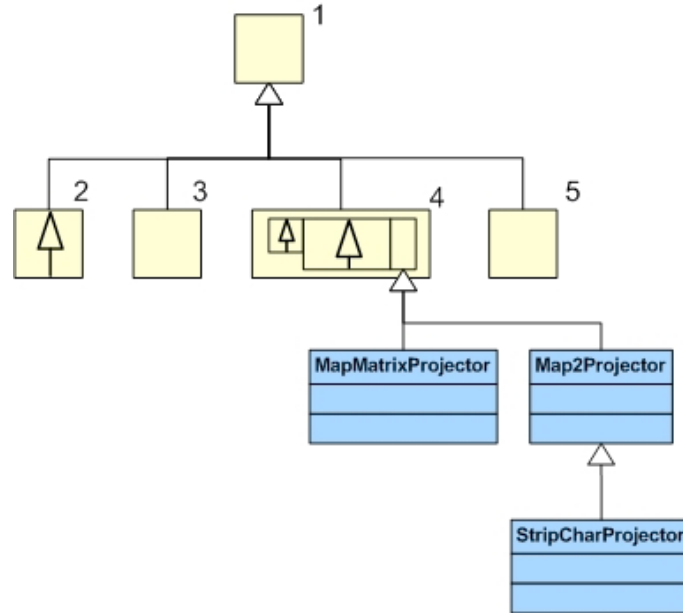


Figure 3.18: Example for onion graph notations by [KM07]. Blue elements indicate the focus-view while yellow indicate the context-view. Nodes 3 and 5 represent single abstracted individual classes. Node 2 represents multiple similar generalizations. Node 4 represents a combination of multiple generalizations with different levels and one individual generalization.

Image source: [KM07]

Grid

Grid feature as explained on Section 2.5.2.

Ruler

Ruler feature as explained on Section 2.5.2.

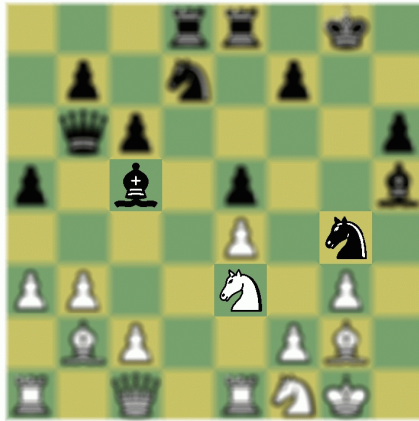
Semantic Depth-of-Field Technique

This technique is explained by Miksch and Hauser in [MH01]. The general idea is to adjust the sharpness of elements in relation to their relevance to the user. Blur is used to de-emphasize elements and therefore make them be part of the context while un-blurred objects represent the focus. This technique is often used in the field of photography by blurring parts of an image based on their distance to the camera. Blurred parts of an image are discarded by our brains in a very short time (within about 10 milliseconds)

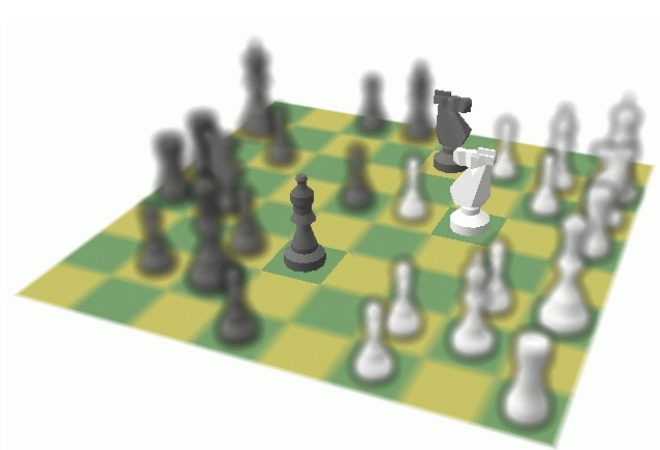
which allows us to quickly shift the focus to the sharp parts [MH01]. In comparison to photography, here it is mainly used to make it easy to quickly differentiate between more important and less important parts of an information space.

In their paper, they define two functions: a relevance function and a blur function. The relevance function is used to define how relevant an object is and is based on the fulfillment of certain criteria. The blur function is used to map relevance values to blur levels and defines how sharp elements with a specific relevance are.

Miksch et al. distinguish between three different classes of the semantic depth-of-field technique: *2D SDOF*, *layered 2D SDOF* and *3D SDOF*. As the names already reveal, 2D SDOF blurs independent 2-dimensional objects (e.g., a 2D chess board), layered 2D SDOF blurs entire 2-dimensional layers (e.g., layers on a map), and 3D SDOF blurs 3-dimensional objects (e.g., a 3D chess board). Examples for 2D and 3D chess board and the application of SDOF can be seen in Figure 3.19.



(a) 2-dimensional semantic depth-of-field



(b) 3-dimensional semantic depth-of-field

Figure 3.19: Example for the application of 2D and 3D semantic depth-of-field technique. The focus lies on the white knight on e3 and all black pieces which threaten it. All other pieces are blurred.

Image sources: [MH01]

Lenses

Lenses provide a better way to navigate and explore cluttered graphs. They follow a similar concept as magnifying glasses (Section 3.5.3), but, instead of simply magnifying information, they can add additional interactive behavior and only show relevant information. They can interactively be moved around as well and can therefore easily be combined with fisheye techniques. Most of the time, the area which is displayed inside the lens is considered the focus area, while everything outside it is considered the context area.

Examples for lenses are explained by Tominski et al. in [TAVHS06]. They propose multiple types of lenses, combined with fisheye techniques, to support users in exploring unfamiliar and cluttered graph layouts. They propose three types of lenses: Local Edge Lens, Bring Neighbors Lens, and Composite Lens.

The Local Edge Lens is supposed to help users identify edges which are connected to objects inside the focus area. It does this by only showing those edges that are connected to vertices inside the focus area. The context area still shows all edges. An example can be seen in Figure 3.20b.

The Bring Neighbors Lens is designed to not only help identify edges, but vertices as well. This is done by moving all off-screen objects, which have a neighbor with a specific vertex of interest, into the focus area. In comparison to the Local Edge Lens, this lens adjusts the layout of the information and is therefore not structure preserving. An example can be seen in Figure 3.20c.

The Composite Lens is a combination of the previous two lenses and a fisheye lens. It applies the Bring Neighbors Lens first, the Local Edge Lens second, and a fisheye lens last. The fisheye lens is added to counter a problem of the Bring Neighbors Lens by further spreading neighbors. The problem arises when one neighbor from far away and multiple neighbors from close by are brought into the focus area. This leads to neighbors being indistinguishable from each other as seen in Figure 3.20c. An example for the Composite Lens can be seen in Figure 3.20d.

Another implementation of lenses is presented by Karnick et al. in [KCJ⁺09]. They propose "detail lenses" to make following static route maps easier. Besides only a textual description of routes, they added lenses that show points of interest along the path in a more detailed way. In comparison to lenses described above, they do not follow the cursor. Instead, multiple lenses are placed along the edges of the main view. An example can be seen in Figure 3.21.

Halos

Halos are proposed by Baudisch and Rosenholtz in [BR03]. They are a visualization technique that is used to show off-screen elements. Relevant elements which are outside the viewport are marked with a circle around them. This circle is then expanded equally into all directions until it reaches the edge of the currently visible area. A small portion of those circles can then be seen by the user inside the viewport. Elements that are far away produce a larger arc than elements that are close by. This property allows users to roughly determine the distance between the off-screen elements and the currently visible area. This is the main advantage of this technique, in comparison to arrow-based solutions such as City Lights. They provide information about both, distance and location, without additional annotations. To better illustrate the functionality of halos, an example can be seen in Figure 3.22.

This technique works very well when only a few off-screen elements are displayed. Showing a lot of elements quickly leads to a cluttered and confusing view. In order to mitigate

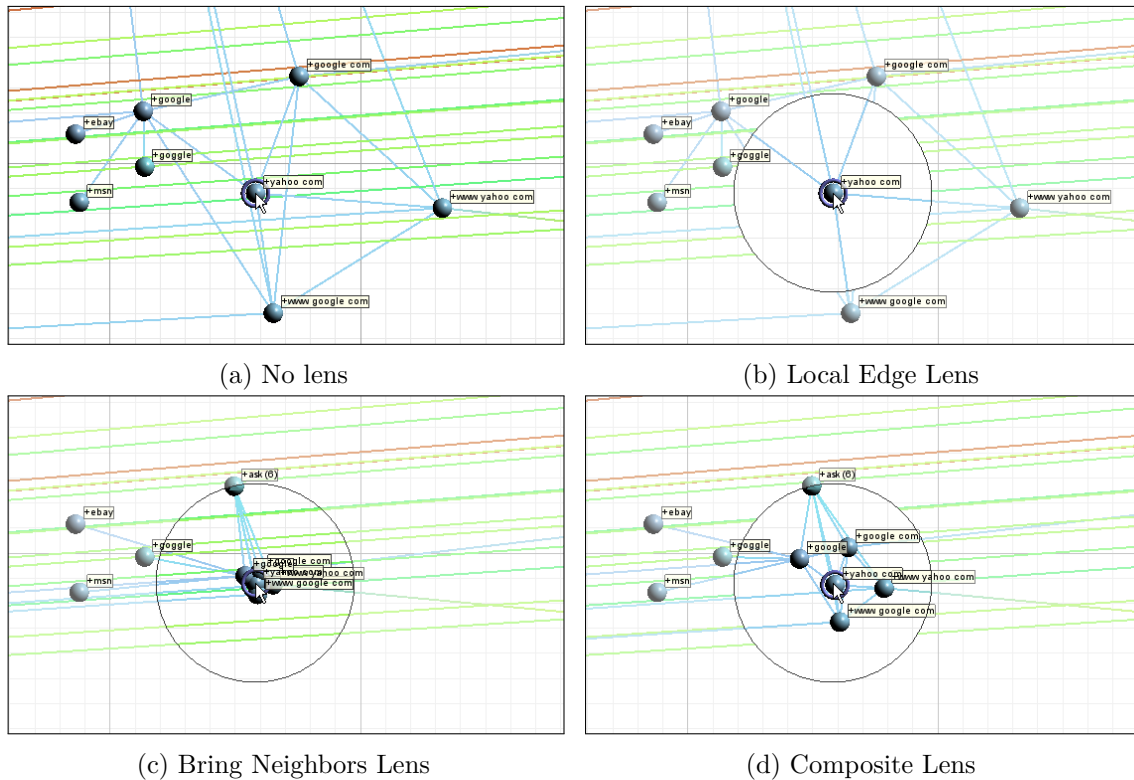


Figure 3.20: Example for lenses proposed by Tominski et al. in [TAVHS06].
Image sources: [TAVHS06]

this, overlapping arcs which have a similar size are combined into a single multi-arc. Multi-arcs are represented by multiple thinner lines which originate from their average origin. Furthermore, groups of four or more elements are combined and represented by thicker arcs.

In a user study conducted by Baudisch and Rosenholtz in the same paper, they compared their halo approach to an arrow-based solution. Users were supposed to complete tasks such as, selecting the closest objects or finding the shortest path, as fast as possible while maintaining reasonable accuracy. Users were able to complete tasks 16-33% faster without a significant difference in error rate.

Wedges

Wedges follow the same concept as halos of the previous section. They are proposed by Gustafson et al. in [GBGI08] and, just like wedges, provide information about location and distance. The difference to halos is that wedges have a better way of dealing with clutter and overlapping. Instead of circles, triangles are used to point to off-screen elements. One corner is placed at the origin of the off-screen element while the other two are extended and placed at the edge of the viewport. An example can be seen in Figure 3.23.

3. TAXONOMY



Figure 3.21: Example for detail lenses by Karnick et al. [KCJ⁺09]. Lenses around the edges of the main view show points of interest along the generated path.

Image source: [KCJ⁺09]

Wedges possess three degrees of freedom: rotation, aperture (width), and intrusion (length). All three of them are used to convey specific information about an element, or deal with clutter or overlap. Although, they can also be used for different purposes, in their algorithm, they adjust the rotation to keep wedges from overlapping, and the intrusion plus aperture to indicate distance.

An example of wedges with additional information is displayed by Gladisch et al. in [GST13]. They give additional information about why a specific off-screen element is relevant by adding a bar chart to the wedge. The bar chart represents individual parameters of their degree-of-interest function and is directly displayed inside the wedge. An example, along with a good comparison of different off-screen visualization techniques, can be seen in the form of an image in their work (Figure 3.24).

Gustafson et al. also conducted a user study which compares wedges with halos. Participants had to perform the same tasks as those of the user study that was conducted in [BR03]. The result showed that wedges significantly increase the accuracy of judging where an off-screen element is located. All other tasks did not yield significant differences.



(a) Red arcs around the edges of the viewport indicate locations of off-screen elements.

(b) The center of each circle is located outside of the viewport and determines the exact position of off-screen elements.

Figure 3.22: Example for halos. A visualization technique by Baudisch and Rosenholtz [BR03] for off-screen element visualization.

Image sources: [BR03]

Proxies

Proxies are a cue-based technique which are often used in combination with Halos or Wedges. A proxy is an alternative visual representation of an actual object of the information space. Usually, they are simplified versions of the actual object and not tied to a specific location. They provide a quick way to navigate to a specific object, usually by simply clicking on the proxy.

An example for a proxy-based technique is explained by Irani et al. [IGY06]. In their paper, they present a technique they call "hop", which combines halos and proxies. They

3. TAXONOMY



Figure 3.23: Example for wedges. A visualization technique by Gustafson et al. [GBGI08] for off-screen element visualization. Red triangles indicate existence of off-screen elements. Visible legs of triangles give information about distance. Image source: [GBGI08]

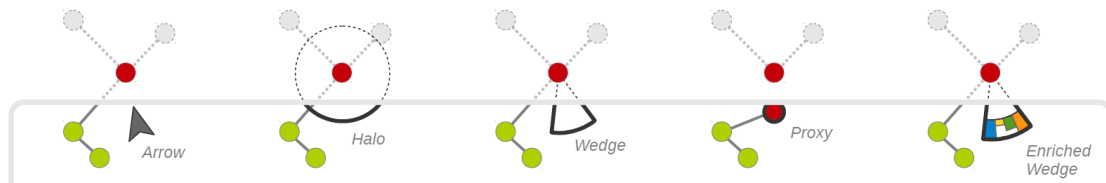


Figure 3.24: Example for enriched wedges in comparison to other off-screen visualization techniques by Gladisch et al. [GST13] Image source: [GST13]

use halos to make users aware of off-screen elements, proxies to bring targets closer to the cursor, and a teleportation mechanism to move the view to these elements. In order to do that, they thought of a new interaction method which they call the "laser beam". The

laser beam can be triggered by the user by clicking and holding at some point inside the current view. Now the user is able to move their cursor around which creates the laser beam, starting from the original clicking position and going through the current mouse position towards the edge of the viewport. Elements of halos that intersect with this laser beam are now moved closer to the user's cursor in the form of a proxy. If the user releases the mouse button while having a proxy selected, an animation is played, which moves the view to the element represented by that proxy. To better illustrate how hop works, they provide an image in their paper which can be seen in Figure 3.25.

They also conducted a user study which compared their hop technique with standard panning and two-level zooming. Users were able to select off-screen elements about twice as fast as with the standard interaction methods.

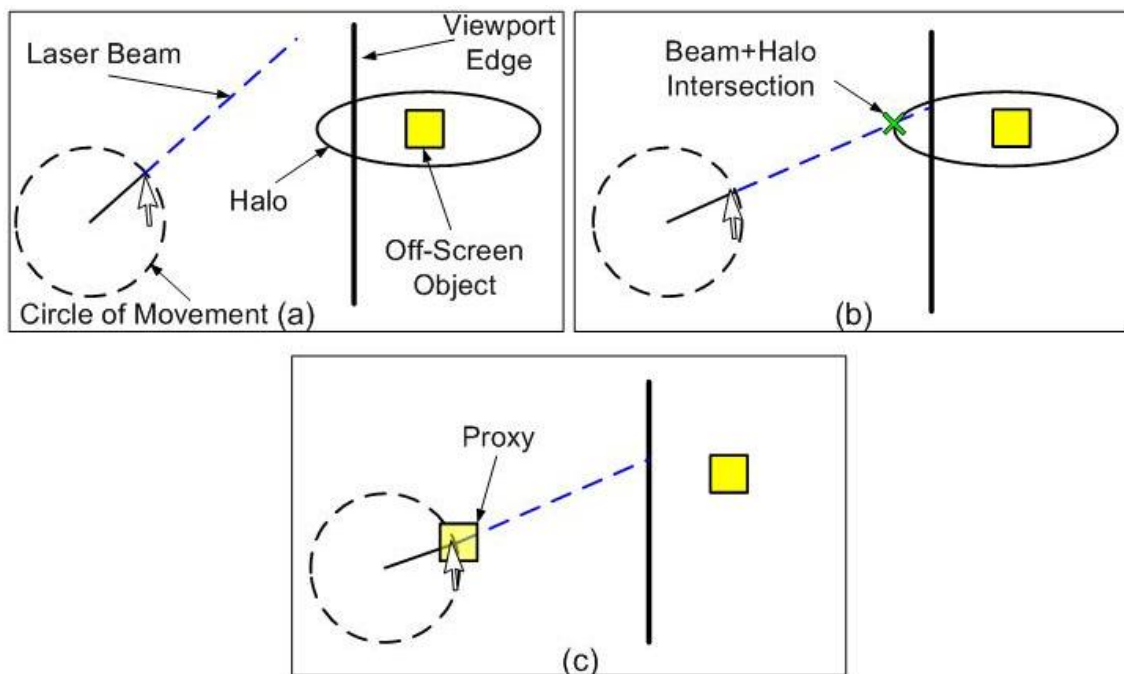


Figure 3.25: Example for proxy-based technique "hop" by Irani et al. [IGY06]. "Invoking proxies with the laser beam: a) beam is created; b) beam intersects halo; c) proxy created." [IGY06, p. 302]

Image source: [IGY06]

Scrollbar with Indicators

This feature is closely related to other scrollbar features and could also be classified as an overview-plus-detail feature. Here, the scrollbar is not only used to let the user navigate the workspace more quickly, but also to give cues about off-screen elements or meta-information about those elements. These cues are often simple indicators in the form of colored stripes which are rendered on top of the scrollbar but underneath the slider.

3. TAXONOMY

Colors of the indicators are used as an additional degree of freedom. Often they convey information that lets a user differentiate between specific elements of the information space.

This feature is often used in text-based languages. For example, the software development IDE Webstorm which displays information about errors or warnings via the scrollbar. Hovering over an indicator brings up a context menu with more details about an indicator. An example can be seen in Figure 3.26.

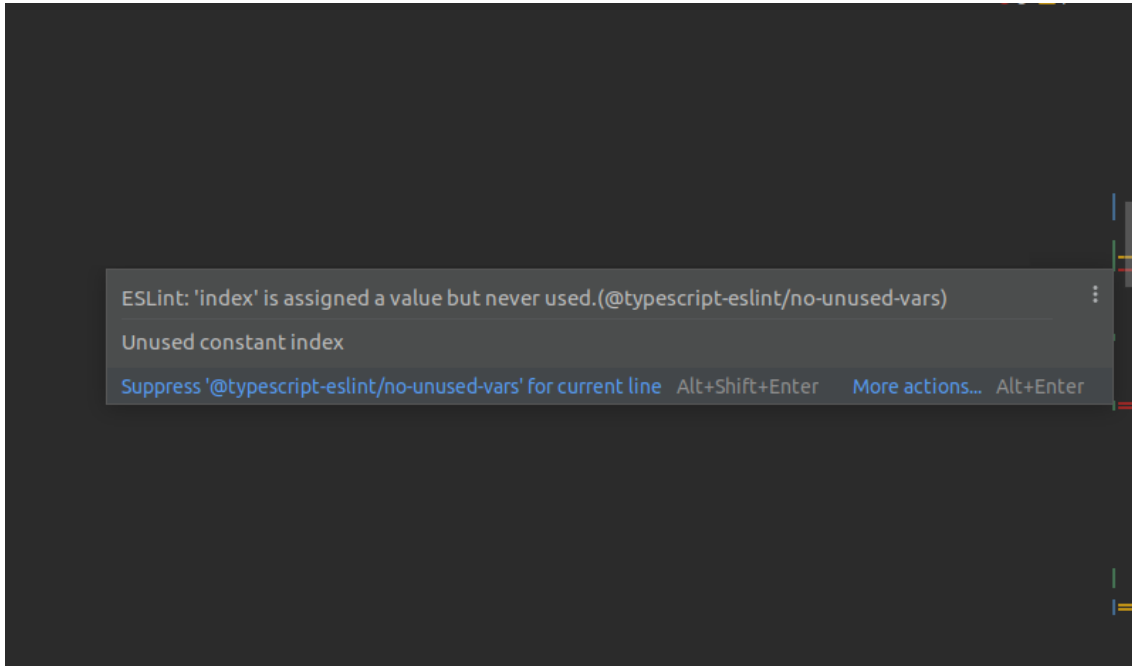


Figure 3.26: Example for a scrollbar with indicators in software development IDE Webstorm. Horizontal red/yellow lines indicate errors/warnings. Hovering over them opens up a context menu with additional information (indicated by the box with a gray background). Vertical green/blue lines indicate added/modified lines.

Prototype¹

The goal of this prototype is to add features to a graphical language server platform that help reading and navigating large information spaces. Although, such features exist already in other domains, they have not been used in combination with a GLS platform yet. The initial step, which has been done during the writing of the taxonomy, is to find and evaluate features that could be integrated into the GLS protocol. The graphical language server platform which was chosen to implement this prototype in, is the Eclipse Graphical Language Server Platform. At the time of writing, the Eclipse-GLSP is the most advanced and a highly maintained graphical language server platform. Because of its recency, it is built on a modern technology stack which serves as a very good base for such features.

The difficulty in implementing new features in a GLS platform, compared to implementing it in other environments, is the clear separation of concerns (server and client) that a GLS platform displays. In GLSP, a new feature can usually not only be implemented on the client side, it has to be able to understand data which is coming from the server in some sort of data interchange format (e.g., JSON), and send data back in a format that can be understood by the server. This is not always possible and often, dealing only with the data formats that are already part of the GLS platform is not sufficient. New feature-specific data is required which means that, in order for the server to understand this data, new feature-specific logic has to be added to it. This shows that, even though a feature is labeled as a *client-side* feature, it often requires additional functionality on the server-side as well.

This chapter will describe the implemented features in more detail.

Section 4.1, Theory, will go over its concept and the reasoning behind the selection of the features that will be implemented. Furthermore, it will describe theoretical aspects of its functionalities and details about its integration into the Eclipse-GLSP. Section 4.2

¹A concise version of this chapter will be published at the MODELS conference [DCPB22a].

will be the practical counterpart to Section 4.1 and will be dealing with details about its implementation and its architecture. Section 4.3 will go into detail about encountered problems during its conceptualization and implementation, and strengths and weaknesses of the feature.

4.1 Theory

Currently, the Eclipse-GLSP offers very little features that make the navigation and comprehension of large models easier. The first step of the prototype development is to determine what feature(s) should be implemented. In order to do this, we used the knowledge that we gained during the State-of-the-Art research, and during the development of the presented taxonomy. Especially during the ex-post evaluation, a lot of related features were looked at and categorized which ultimately helped make this decision. The following section will give a general descriptive evaluation of relevant features and give insight about the decision-making process that led to the concept of the prototype.

4.1.1 Feature Evaluation

The performed State-of-the-Art research helped to get a general idea of where current modeling tools stand in terms of visualization and navigation of large models. It clearly shows that most modeling tools currently do not have good support for it. Related features were only found in very few modeling tools and mostly only in very basic or abstract ways. Other modeling tools, like yEd or Microsoft Visio, give ways of adding related features by means of their integrated scripting languages or frameworks. While this offers great flexibility, it is not very convenient for users that do not have the required experience to implement them.

Even though this prototype will be part of a modeling tool, limiting the research of related features to only other modeling tools did not lead to satisfying results. We expanded our search to the much more abstract domain of general information visualization. In this domain we looked at tools that work with large information spaces, such as mapping platforms or software development IDEs. Tools like Google Maps use many relevant features, for example, proxies and different levels of detail. Although, in this broadened domain, we could find more related features than just in the model engineering domain, most of them were tailored to specific use cases and could not be applied to modeling tools in the exact same fashion. Many of them are so tightly coupled with the underlying software tool or domain that it would not make sense to use the same feature in the domain of model engineering. For example, overview scrollbars (Section 3.5.3) which are used in the domain of software development. They rely on the fact that text files holding source code usually require a lot of vertical space and relatively little horizontal space. Because of that, the vertical scrollbar can be used very well to display contextual information without them becoming too small to be useful. In visual model engineering tools, this would usually not be the case because models are often expanded in all directions equally.

The third step was to look at research that has been done in the area of visualizing and navigating large information spaces. While this led to very interesting ideas and concepts, such as City Lights [ZMG⁺03], and speed-dependent automatic zooming [IH00], it had similar problems as most tools: features were too specific to certain domains.

When comparing objects of those three steps to each other, it becomes clear that many of these features follow similar ideas and concepts. For example, the Windows 10 Magnifier app (Figure 3.5) is comparable to the minimap feature (Section 3.5.3). Both follow the same concept of magnifying information with the difference that one can be moved around while the other one is stationary. Another example is the scrollbar with indicators (Section 3.5.3) of Visual Studio Code or JetBrains' WebStorm. This is a very similar concept to EdgeRadar [GI07] (Section 3.5.3). Both visualize off-screen elements with very simple geometric graphical elements, such as dots or squares.

Looking at all the tools and research, the two most common concepts related to large information spaces were the following: Adjusting the visual representation of objects and visualizing off-screen elements. They can not only be found in existing modeling tools/frameworks, like yFiles, but also in other tools, such as Google Maps or the JetBrains' IntelliJ IDEA Java IDE, and research, e.g., [BR03]. Considering tools that do not offer any such features yet but want to add them, these two concepts represent a good starting point. They portray a good mixture between genericity, development complexity, and added usability. They are generic enough to be used in some way in a wide range of different tools, but can also be made more specific by tailoring them to specific use cases and domains. This is a very valuable characteristic for the development of a prototype, because it can be developed in iterative steps. The early steps can be kept simple, and complexity can be added gradually after successful evaluation of previous steps. At the same time, even if they are kept simple and generic, we think that they improve the usability immensely. They are self-explanatory and easy to understand, not very intrusive or dependent on other features, but still very customizable. For these reasons, these two concepts will also form the foundation of the prototypes developed in the course of this work.

To cover both of the explained concepts, we decided to split up the implementation into two independent prototypes. Prototype 1 will be dealing with the adjustment of visual object representations. More accurately, it will introduce the ability to define multiple global discrete levels of detail. Objects can have different visual representations, depending on the level that is currently active. Each level of detail is active at specific zoom levels. In other words, the currently active level of detail can be switched by zooming in or out (semantic zooming). This concept is very closely related to the ideas of Frisch et al. [FDB08].

Prototype 2 will add indicators for off-screen elements. Similarly to [FD13], these indicators will be rendered at the border of the stage. Ideally, they will act as proxies of the original element. This means that they can be used as a replacement for the original elements for all interactions. For example, edges leading to the original element should be leading to the proxy element instead. Additionally, graphical representations of indicator elements should be customizable for each model element.

Both prototypes have been implemented with universality in mind and they are supposed to

be able to be used with multiple languages. Adding and using the proposed functionalities in combination with other languages should be able to be done with only small modifications to the existing code. These necessary code changes are mainly language-specific information, such as information about the visualization of elements of a language.

4.1.2 Eclipse Graphical Language Server Platform

In order to understand the concept and the implementation details of the prototypes, one has to have a basic understanding of the underlying system. While Section 2.2.1 gives a brief overview of the general concept of the Eclipse Graphical Language Server, this section will describe the architecture of it in more detail. Bear in mind that the system is still in active development and the current version may differ from the version that is used and explained here. Furthermore, since the complete source-code is open-source, it may also be the case that different language servers use different architectures and implementations. This work uses the relatively simple *Workflow* example language server, which is described in more detail in Section 4.1.3. For a more accurate explanation, please visit the official website² or code repository³.

Server

The main responsibility of the server is to manage the model. It uses the powerful and established Eclipse Modeling Framework (EMF) to do this. With EMF, a meta-model (Ecore) can be created which can be used to describe models. With elements, such as *EClass* or *EAttribute*, entire models can be defined. Once all objects of a model have been described, EMF can be used to translate these meta-models into models in the form of interfaces and classes with the included code-generator. The resulting Java-code is then used throughout the language server to manipulate and save models.

Besides the model which describes the language itself, there exists another model (GModel), which describes the graphical representation of a language. Before a model is sent to the client, it is transformed into its GModel representation. This GModel representation is then sent to the client in the form of JSON objects. Most languages require similar graphical representations, such as edges, nodes, or icons. While the model which describes the language is specific to the language, elements of the GModel can usually be reused by many languages. Some important and relevant GModel elements are:

- **GModelElement:** Describes an element of a language with a unique id and a type. Furthermore, it can have a parent, children, and CSS classes.
- **GShapeElement:** Descendent of *GModelElement*. Describes a visible element, which has a size and a position.
- **GNode:** Descendent of *GShapeElement*. Describes a node-like element, which can have edges connected to it.
- **GLabel:** Descendent of *GShapeElement*. Describes a text label.

²<https://www.eclipse.org/glsp/> (Accessed: 19.12.2021)

³<https://github.com/eclipse-glsp/glsp> (Accessed: 19.12.2021)

- **GEdge:** Descendent of *GModelElement*. Describes an edge with a source and target element.

As already mentioned, the communication between server and client is done via actions. Most of these actions are initiated by the client and answered by the server, but there also exist situations, where the initiation of an action comes from the server. Important and relevant actions are:

- **RequestModelAction:** It is usually the first action which is sent from the client to the server. It requests the graphical representation of a model. It is answered by the server with either an *SetModelAction*, or an *UpdateModelAction*.
- **SetModelAction:** It is sent from the server to the client. It sends a graphical model to the client. The old model, if one exists, is discarded.
- **UpdateModelAction:** It is sent from the server to the client. It updates the graphical representation of a model on the client.
- **SaveModelAction:** It is sent from the client to the server. It signals the server to persist the current model.
- **RequestBoundsAction:** It is sent from the server to the client. It signals the client to pre-compute the layout which can then be used by the server to compute the finalized layout. This pre-computed layout is necessary, if the layout process of the graphical representation also depends on the client. E.g., if the client has font-size settings, which indirectly affect the sizes of GModel elements. It is rendered by the client invisibly and its results are then sent back to the server in a *ComputedBoundsAction*. This can be enabled or disabled with the *needsClientLayout* setting on the client.
- **ComputedBoundsAction:** It is sent from the client to the server, as a response to a *RequestBoundsAction*. It is usually followed by a *SetModelAction* or an *UpdateModelAction*.
- **CreateNodeAction:** It is sent from the client to the server. It signals the server to add a new node to the model. Once this action was performed successfully by the server, an *UpdateModelAction* is triggered, which is sent from the server to the client to update the model on the client as well.
- **CreateEdgeAction:** It is sent from the client to the server. It signals the server to add an edge to the model and, similarly to *CreateNodeAction*, indirectly triggers an *UpdateModelAction*.

Client

The client is responsible for rendering the supplied GModel. This is done with the help of the Sprotty framework. Sprotty handles the rendering and dispatching of events/actions. The model that Sprotty works with is called the SModel (Sprotty Model). Its design is very similar to that of the GModel which is generated on the server. This makes the conversion from the GModel to SModel simple and fast. All JSON elements that were received by the client are translated into TypeScript objects and represent the SModel.

In order to fully understand and render all graphical elements received by the server, a

view has to be defined and assigned to all received types. As already mentioned above, all *GModelElements* have an assigned type. This type is used by the client to look up its view. Each view on the client has a *render()* function, which defines how an element is supposed to look like, once it is displayed in the browser. While the client has a lot of predefined views, such as the *RectangularNodeView*, which simply renders the SVG element *rect*, new views can be defined for each language. All that is necessary on the client, is the definition of a view element with its *render()* function, and assigning a type to it.

After the translation of the received JSON objects into TypeScript objects, they are handled by the viewer. The viewer uses the SModel to generate a virtual document object model (VDOM) from it. Similar to popular frameworks like React or Angular, this virtual DOM represents a layer above the actual DOM and can easily be accessed and manipulated by the rest of the application. During the rendering process, this virtual DOM is taken and converted into real DOM elements that are understood by browsers. Furthermore, the viewer is also responsible for adding and removing event listeners that are required by some features.

Another large component of the client is the ActionDispatcher. The ActionDispatcher is responsible for making sure that actions are handled correctly. These actions can come from the server, but also other sources, such as the viewer. In order to handle an action, it has to be dispatched to the correct handler. This can either be the client itself, in which case it is converted to a command, or the model source, in which case it is sent to the server. Commands implement an *execute()* function that execute the command, and *undo()* and *redo()* functions that are called when the user reverts the last changes (e.g., by pressing the "CTRL+Z" keys)

A visual representation of the client's life-cycle can be seen in Figure 4.1.

4.1.3 Workflow Language

Both prototypes have been implemented in the workflow example language server. This section will briefly describe the workflow language and serve as a base for the following sections. The workflow language server is similar to a UML activity diagram and is the implementation that is used to demonstrate the general functionality of Eclipse-GLSP. Its purpose is to describe the order and flow of activities necessary to finish a specific task. It is a relatively simple language and, at the time of writing, consisted of the following objects:

- **Automated Task:** Represents a task in the workflow that is performed automatically. Has a duration.
- **Manual Task:** Represents a task in the workflow that has to be performed manually. Has a duration.
- **Fork Node:** Splits up the workflow into two or more different paths. All paths are mandatory and have to be followed.
- **Decision Node:** Splits up the workflow into two or more different paths. Represents a point in the workflow where a decision has to be made. Depending on the decision,

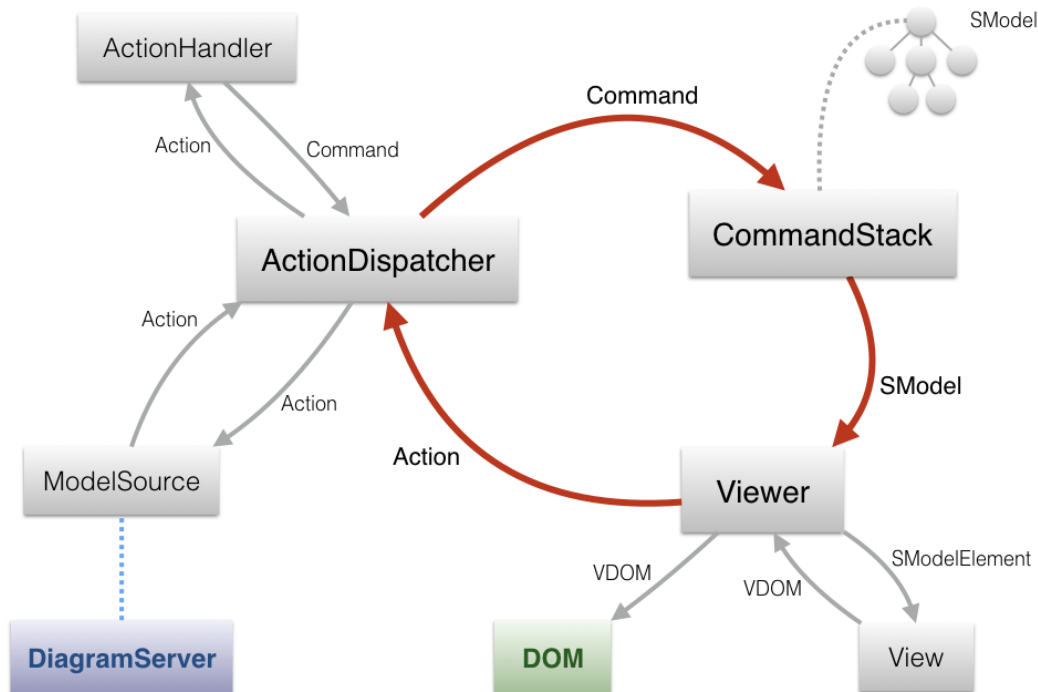


Figure 4.1: Visual representation of the life-cycle and responsibilities of the Eclipse graphical language server platform client.

Image source: <https://github.com/eclipse/sprotty/wiki/Architectural-Overview> (Accessed: 06.03.2022)

some paths are optional and only one has to be followed.

- **Join Node:** The counterpart to a fork node. It connects multiple mandatory paths.
- **Merge Node:** The counterpart to a decision node. It connects multiple optional paths.
- **Edge:** An edge between two objects which describes the flow of the work. It can connect tasks and nodes with each other.
- **Weighted edge:** An edge which additionally has a probability. It can only be used as an outgoing edge of decision nodes and represents the probability of its occurrence.

An example of a simple workflow diagram can be seen in Figure 4.3a.

4.1.4 Prototype 1 - Semantic Zooming

Prototype 1 features semantic zooming. It allows a user to change the graphical representation of a model by zooming in or out. Depending on the current zoom level, an object

can either show more or less details about itself. With semantic zooming, the information that a specific object exists is not taken away, instead, each object only shows the most important information about itself. At the lowest level, this could, e.g., only be their title or outlines. The further the user zooms in, the more details are seamlessly added, until, at the highest level, all details are shown. An example can be seen in Figure 4.2, which shows the same model element in four different levels of detail. The goal of this feature is to make it easier to comprehend large models by showing less information about objects when zoomed out, and more information when zoomed in. This prototype will only add/remove details about objects of the current information space (as explained in Section 3.5.3) An addition to this prototype would be to also add details of other information spaces if applicable (as explained in Section 3.5.3).

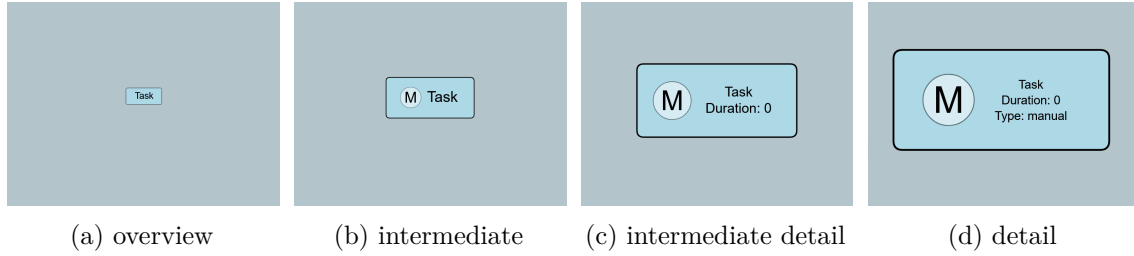


Figure 4.2: Screenshots of the developed prototype 1, showing the model at four different zoom levels with automatically adjusted information visualization.

Level of Detail

This is implemented by giving language server developers the opportunity to define an arbitrary number of discrete levels of detail on the server. These levels can then be used throughout the server and the client. They consist of a name (*name*) and a zoom level range (*from* and *to*) defined as $[from, to[$. This range is used to determine when a discrete level of detail is active. Both values, *from* and *to*, are optional. If *from* is omitted, it is treated as $-\infty$, if *to* is omitted, it is treated as ∞ . Developers are free to use their own interpretation of a zoom level and these values. However, because the zoom level is transferred between server and client, both of them have to have the same interpretation of it. In the implementation of this prototype, the zoom level is a number $x > 0$ where 1 is considered the default zoom level. Everything above 1 is zoomed out and everything below 1 is zoomed in. This is done because Sprotty uses the natural exponential function on the δY value of the standardized UI DOM events to calculate the current zoom level of the viewport. The defined discrete levels of detail (as seen in Figure 4.2) are the following: detail $[0, 0.25[$, intermediate detail $[0.25, 0.5[$, intermediate $[0.5, 1.25[$, overview $[1.25, \infty[$. Developers are responsible for the correct and full coverage of this range. Gaps or overlapping levels may lead to undefined behavior in the current implementation.

All defined discrete LoDs can be requested by the client with the *RequestDiscreteLevelOfDetail* action. Because they are needed in order to render the model, this action is usually one of the first that is dispatched to the server. The server responds with a

SetDiscreteLevelOfDetail action, which includes all discrete LODs in the JSON format. This information only has to be requested once and can then be cached by the client because it is not subject to change.

Rules

Level of detail rules are used to trigger specific behavior on certain LoD levels. These rules describe how specific graphical representations should be adjusted when the client enters a specific level of detail. All rules consist of at least a type and information about when it is supposed to be applied. Furthermore, depending on the type of the rule, they can include additional rule-specific parameters.

Rule types The type of a rule defines the behavior of a rule. It is transferred in the *type* field and, currently, there exist three types of rules:

- **CssStyleRule:** It allows to add certain CSS-styles to objects. Styles can be given with the additional parameter *styles*. This rule alone is very powerful and can accomplish most of the graphical adjustments. It can, for example, be used to increase the font-size of text when a user zooms out, change the background-color, or add transparency to elements. An example can be seen when comparing Figure 4.2a to 4.2b, which shows the same information about an object, but Figure 4.2a has an increased font-size, so that the title of the task is still readable, even when zoomed out. Additionally, the value of a given CSS-style can include the keyword '*\$level*'. On the client, all occurrences of this keyword are automatically replaced with the current zoom level. This allows to make values be dynamically dependent on the current continuous zoom level. This can, for example, be used to increase the font-size of an element dynamically with every zoom event, instead of just once to a static value.
- **VisibilityRule:** It allows to hide specific objects with the additional boolean parameter *setVisibility*. This rule is important and used often to completely remove specific elements of an object. Its most common use case is to add certain details, e.g., properties, of a model when the user zooms further in. An example can be seen in Figure 4.2b and 4.2d, where two properties are added.
- **LayoutRule:** It allows to modify an element's layout. The layout of an element is usually defined by the server and can consist of properties such as padding, horizontal/vertical gap, or a minimal width/height. It can, for example, be used to increase the padding of certain elements at zoom levels that offer a lot of space. An example can be seen when comparing Figure 4.2a to 4.2b. Figure 4.2a has a smaller margin in comparison to Figure 4.2b, so that there is still enough space inside the object to fit the text which has an increased font-size at that level.

Rule trigger The information about when a rule is supposed to be applied is stored inside the *trigger* field. It can hold either a *triggerDiscreteLevel*, or a *triggerContinuousLevel*, depending on whether it should be triggered on a discrete level of detail, or a continuous

zoom level. Discrete levels of detail are references to the LoDs which were transferred initially with the *SetDiscreteLevelOfDetail* action, and continuous levels are two double values which specify a range in which the rule is supposed to be applied. Most of the time, rules reference a discrete level of detail. This keeps all rules grouped together under the label of a discrete LoD, which allows to change settings of multiple rules at once. Triggering a rule by continuous zoom level should only be used in rare cases. Cases in which it does not make sense to add an extra discrete LoD, such as when a single element has to be adjusted in a range between existing discrete LoDs.

Rule application Additionally to the information about the type of a rule and its trigger, a reference to elements that it is applied to has to be supplied. This can be done on the server by instantiating rules and assigning them to elements by their type. Each GModel element type can have zero or more rules which will be applied on their specified discrete or continuous trigger. Often it is not enough to only use the type of a GModel element as a reference. For example, the GModel representation of an icon can be used in automated tasks and manual tasks in the workflow model. Although they have the same type in both tasks, one may want to assign different rules to icons of automated tasks than to those of manual tasks. This could not be done if references only consisted of GModel element types. As a solution, a reference can also consist of a type selector, similar to CSS selectors. More accurately, three selectors are currently implemented and behave the same as their equivalent CSS selector: selector group (","), descendant selector (" "), and child selector (">").

- **Selector group (",")**: Applies a rule to all selected elements in a group. E.g.: "task:automated, task:manual" applies the specified rules to all manual and all automated tasks,
- **Descendant selector (" ")**: Applies a rule to all elements specified on the right-hand side that are descendants of the elements specified on the left-hand side. E.g.: "task:automated label:icon" applies the specified rules to all icons inside automated tasks
- **Child selector (">")**: Applies a rule to all elements specified on the right-hand side that are direct descendants (children) of the elements specified on the left-hand side. E.g.: "task:automated > label:text" applies the specified rules to all text labels on the first level of all automated tasks.

A further distinction between rules is whether they are applied on the server or the client. There exist two types of rules, those that are applied to the GModel at server-level, and those that are applied to the SModel at client-level.

Client-rules Most rules are executed on the client because the client has the information about the current zoom level, and it is the client's responsibility to render objects accordingly. Nevertheless, the assignment of rules is language-specific information which is the reason why they are defined and stored on the server. Since the client only has information about how to apply specific rules but not about their assignments, these assignments have to be transferred to the client, along with the model itself. This is done with a new action,

right after the discrete LoDs have been transferred, but still before the model is requested. Similarly to requesting the discrete LoDs, the client requests all level of detail rules and their assignments with the *RequestLevelOfDetailRules* action, and the server response with a *SetLevelOfDetailRules* action. The response contains a list of all element types that have rules assigned, along with an array of all the assigned rules. This means that this request only has to be done once and can then be cached and reused by the client.

Client-rules are applied during the rendering process on the client. The logic of a client-rule is part of the client itself, the server merely provides information about what rule to apply and when. This logic can be changed at will by language server developers, and new rules can easily be added to the client. During the rendering process, the client also has access to the current continuous zoom level. This means that a rule can integrate the current zoom level into its logic and continuously adjust certain elements. An example for this would be to continuously adjust the font-size of a label to the current zoom level. The size of the font would then be adjusted whenever the user zooms in or out, instead of being adjusted once by a fixed amount every time the discrete LoD changes.

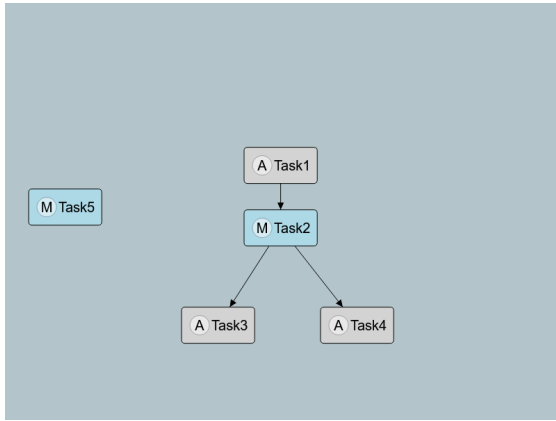
Server-rules Server-rules are used in cases where a rule has to be applied on the GModel before it is sent to the client. Cases where decisions made by the server directly affect the rendering process of the client. Usually, these are fundamental changes to the layout that cannot be done by the client. One disadvantage of server-rules is that the server does not have information about the current zoom level of the client. Because of that, the server does not know when to apply certain server-rules. Only the client knows the current zoom level and has to provide it to the server. By default, the server assumes a zoom level of 1 because that is also the default zoom level of the client. Additionally, the current zoom level can be provided by the client as an optional field in the *requestModel* action, which makes the server apply all rules for that zoom level in the resulting *setModel* action. Not only does the client have to provide the current zoom level, it also has to notify the server whenever a server-rule is supposed to be applied. For this reason, all server-rules are also transferred to the client, along with all client-rules. Whenever the client encounters a server-rule, it has to request the model again to make the server apply the server-rule. This shows why server-rules should be avoided when possible: They require an additional server round trip whenever they are triggered.

An example for a situation where a server-rule would be required, is the adjustment of the layout option *resizeContainer*. It tells the client whether to resize the parent of an element, if the element becomes too large. If this was switched from *false* to *true* via a client-rule, the client would not have correct bounds for this element because these are calculated on the server.

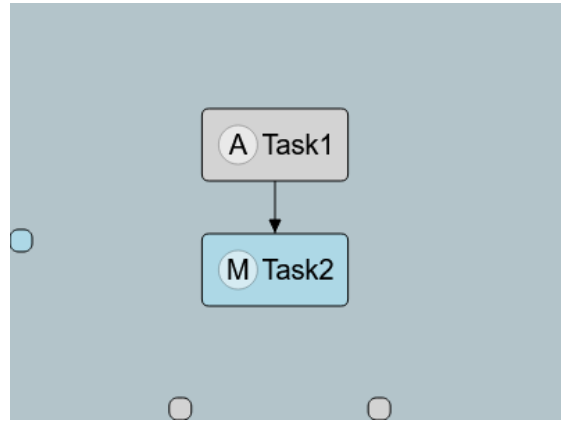
4.1.5 Prototype 2 - Visualizing Off-screen Elements

Prototype 2 features the visualization of off-screen elements. It allows a user to see elements, even when they are not positioned inside the viewport anymore. This is especially useful when working with larger models or models that show a lot of details about its elements,

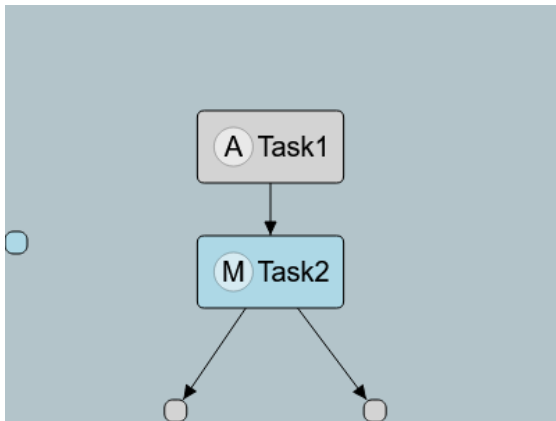
as users tend to zoom in further. Zooming in narrows the view that a user has on a model, which effectively pushes elements off-screen. By showing small indicators around the border, similarly to the solution of Frisch and Dachsel [FD13], the context around an element in focus is kept visible. As soon as an element becomes completely invisible because of, e.g., a zooming or panning event, they are replaced by smaller indicator elements which are pinned to the border of the viewport. This increases the sense of orientation of users, even with larger models or while zoomed in.



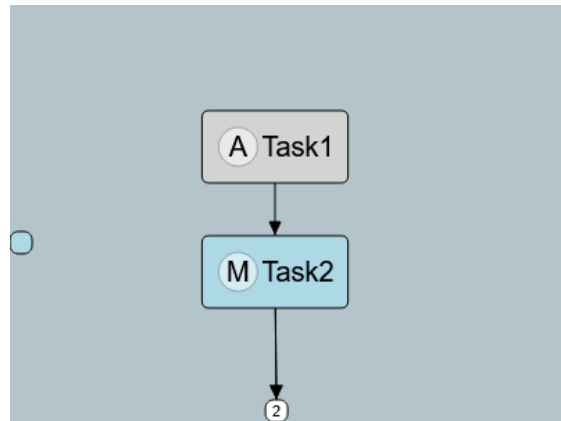
(a) Zoomed out version of the model as reference, showing three automated tasks and two manual tasks.



(b) Visualization of three off-screen tasks in the form of off-screen indicators. The color of the indicators gives information about their type (automated or manual).



(c) Visualization of off-screen elements with edges connected to on-screen elements.



(d) Visualization of multiple off-screen elements merged into one indicator to reduce cluttering. The number represents the amount of combined off-screen element indicators.

Figure 4.3: Screenshots of the second developed prototype

Nodes All nodes (tasks) that are moved off screen are replaced by smaller indicators that are pinned to the border of the viewport. As soon as the original elements become

visible again, its indicator is removed, and the original element is shown instead. Each type of GModel element can have its own individually designed indicator. This can be used to encode additional information into the indicators by, e.g., changing their form or color accordingly. A visual example of three off-screen indicators can be seen in Figure 4.3b. The color of each indicator is used to encode the information about whether it is an automated or manual task.

Multiple indicators at the same place can be merged together. This stops the border of the viewport from getting cluttered and prevents multiple indicators from overlapping. The visual representation of merged indicators differs from the others. An example can be seen in Figure 4.3d. Instead of the colors of their original elements, they are painted in white and include the number of merged indicators in their center. Being able to differentiate between normal and merged indicators is an important aspect. This will prevent users from looking for certain elements and thinking they disappeared even though they were only merged with another indicator.

Edges Nodes in diagrams are not only identified by their name, but also by their position and relationships that they have. For this reason, edges between elements play an important role in combination with off-screen element visualization. They are vital to identifying certain elements and keeping the mental map of a workspace intact. Although edges are also considered elements of a model, they do not have off-screen indicators. In this prototype, when an element disappears from the viewport, its indicator element serves as the new port for all incoming and outgoing edges. In combination with their position, this helps the user to quickly identify an off-screen element, even without the visualization of their name. An example can be seen in Figure 4.3c, which shows two edges connected to off-screen indicators.

Proxies All indicators also act as proxies for the elements they represent. Ideally, all actions that can be performed on the original elements should also be able to be performed on their proxy representations. For example, connecting an edge from an on-screen element to a proxy of an off-screen element, should create an edge from the on-screen element to the off-screen element that is represented by the proxy. This decreases the number of actions that have to be performed to achieve a specific goal in many cases. In the above example, a user would not have to zoom out first to make both elements visible to connect them with an edge.

Navigation All indicators do not only act as a visual aid, they also help navigate the model. Clicking on an indicator will automatically move the viewport to the element that is represented by the indicator and center it. Additionally, the element that was clicked on will also be selected. This does not only let the user start working with it immediately, it also gives it a slightly different look which helps identifying it after the zooming/panning event. This makes traversing a model much easier and faster because the user does not have to manually perform zooming and panning actions. This functionality can also be applied on merged indicators. When the user clicks on a merged indicator, all elements are

selected, and the client automatically calculates and sets the zoom level of the viewport to a value, in which all selected elements are visible.

4.2 Conception

This section will go over implementation details of both prototypes. It will describe each added functionality and the files they are in. Furthermore, it will go into further detail about the design, and it will bring up limitations and problems that were encountered during the implementation.

4.2.1 Prototype 1 - Semantic Zooming

Prototype 1 required adjustments on the server-side and the client-side. It also required additions to the protocol in the form of new actions. A sequence diagram of its most important operations can be seen in Figure 4.4.

Server

The server is built exclusively in Java. Because of it being a prototype, the goal was not to end up with a production-ready implementation. Nevertheless, during its conception and implementation care was taken to allow for easy additions, e.g., in the form of new rules. The following section will give an overview of the most relevant modules, interfaces and classes on the server.

Discrete Levels of Detail All discrete levels of detail are defined in the file: *DiscreteLevelOfDetailEnum.java*, which holds a Java enumeration file. Each enumeration entry consists of an LoD name (enumeration name), and two double values *from* and *to*. Additionally, the static function *getDiscreteLevelForContinuousLevel* exists, which can be used to convert a continuous zoom level to a discrete level of detail. This function is needed to apply server-rules to the GModel and is used to convert the discrete zoom level, which is sent by the client, to a discrete one.

Actions As already mentioned and explained in Section 4.1.4, this prototype introduces two new actions that are handled by the server: *RequestDiscreteLevelOfDetail*, and *RequestLevelOfDetailRules*. Both actions are handled by their respective handler class:

- ***RequestDiscreteLevelOfDetailActionHandler***: It fetches all discrete LoDs that were defined and converts them into a JSON object. This object is then sent to the client in an *SetDiscreteLevelOfDetail* action.
- ***RequestLevelOfDetailRulesActionHandler***: It fetches all registered rules, along with the ids or selectors of the model elements that they were assigned to, and converts them into a JSON object. This object is then sent to the client in an *SetLevelOfDetailRules* action.

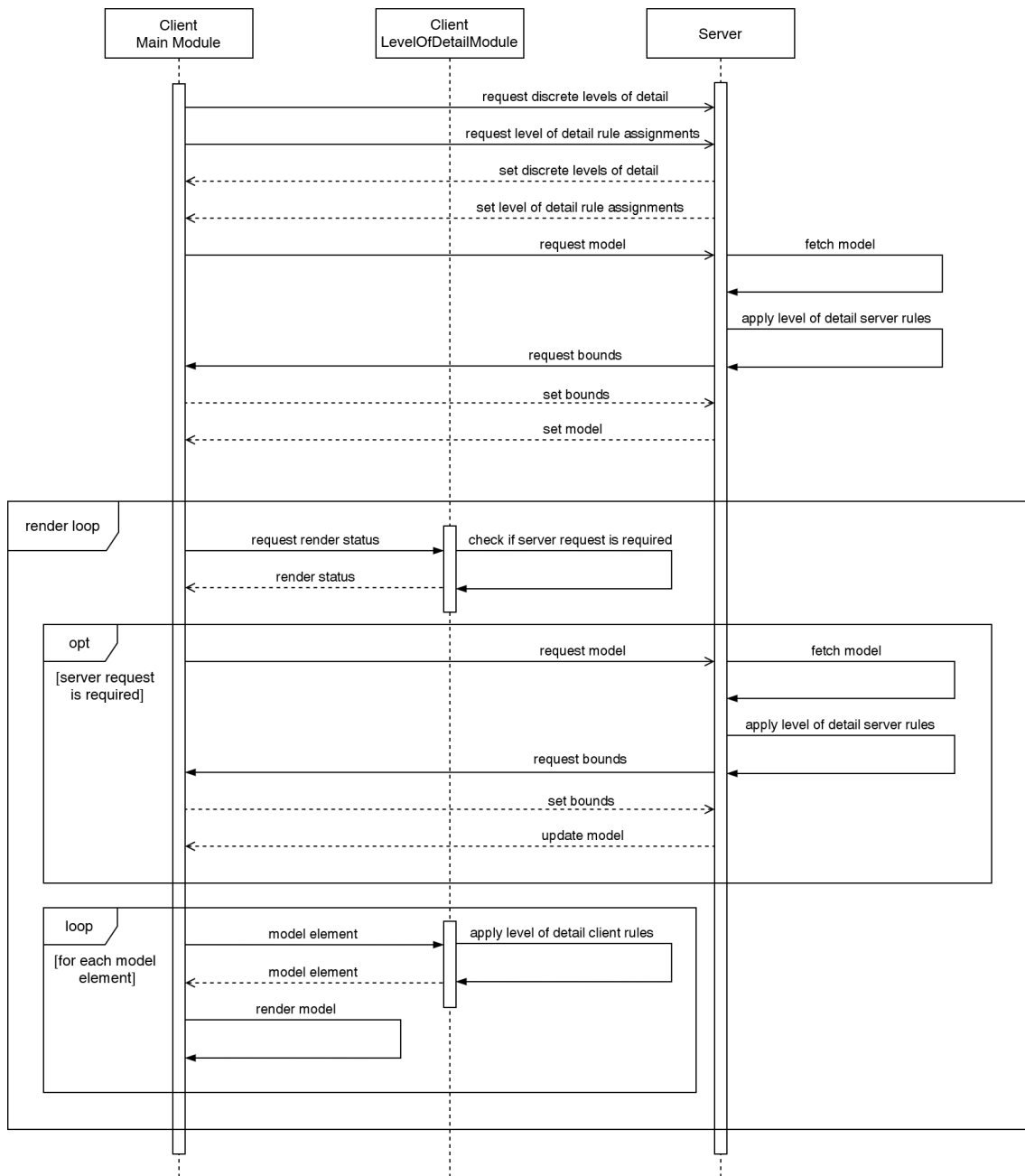


Figure 4.4: Most important operations of prototype 1 visualized in the form of a sequence diagram. It shows the initialization process of the client which requests all required information from the server, followed by operations performed during the rendering process of a model.

Rule Registry Before the server can send rules to the client, they have to be registered. The file where this is done, is called *WorkflowLevelOfDetailRuleRegistry.java*. It includes a

class which inherits from the default rule registry *DefaultLevelOfDetailRuleRegistry*. The default rule registry exposes a function *registerRule* that has to be used to register new rules. As a parameter, it takes the type of a GModel element or a selector as explained in Section 4.1.4, and an instance of a rule. An example of a rule registration can be seen in Listing 4.1. Calling the *registerRule* function assigns a rule to the specified element, which is then transferred to the client with the *SetDiscreteLevelOfDetail* action. By using selectors, this JSON object, and therefore the response, can be kept relatively small, because even if the GModel consists of, e.g., 100 automated tasks, each rule is only included once in the response. Another solution would have been, to reference each GModel element by its unique id instead of with selectors. While this causes the response to increase in size by a lot in certain cases, it would allow two elements of the same type to have two completely different rules. Usually, all elements of a type are supposed to behave the same which is why there is no need to assign different rules to the same model types. The advantage of this solution would be that the client does not have to evaluate any selectors.

Currently, only child (" $>$ ") and descendants (" ") selectors are supported. These two are already sufficient to cover most use cases. In cases where these two selectors are not enough, a new GModel type has to be defined for just those elements that need additional rules. While the logic of these selectors is currently part of the code of this prototype itself, in the future, the support for more selectors could be added by integrating entire selector engines, similarly to a CSS selector engine.

```
1 LayoutRule rule = new LayoutRule(  
2     new GLayoutOptions()  
3         .paddingBottom(3D)  
4         .paddingTop(3D)  
5 );  
6  
7 rule.addLevelOfDetailRuleTrigger(  
8     new LevelOfDetailRuleTriggerDiscrete()  
9         .addDiscreteLevelOfDetail(DiscreteLevelOfDetailEnum.OVERVIEW)  
10 );  
11  
12 registerRule(  
13     ModelTypes.AUTOMATED_TASK + ", " + ModelTypes.MANUAL_TASK,  
14     rule  
15 );
```

Listing 4.1: Java code that shows the registration of a new rule which changes the size of the bottom and top border. The rule is registered for all automated and manual tasks of the workflow language by using the ", " selector. Furthermore, a trigger is added to the rule which activates it when the client enters the discrete level of detail "Overview".

Rules As already mentioned in Section 4.1.4, three concrete rules currently exist: *CssStyleRule*, *VisibilityRule*, and *LayoutRule*. Each rule is a descendant of the interface

LevelOfDetailRuleInterface and the abstract class *LevelOfDetailRule*.

- ***LevelOfDetailRuleInterface***: Each rule, independent of whether it is applied on the server or client, inherits from this interface. It exposes functions that are required by all rules, namely getter and setter functions for the trigger information.
- ***LevelOfDetailRule***: It implements all functions of the *LevelOfDetailRuleInterface* interface. Concrete rule classes inherit from this class.

Information about when a rule is supposed to be applied is stored inside either a *LevelOfDetailRuleTriggerContinuous* or a *LevelOfDetailRuleTriggerDiscrete* object. Both classes inherit from the *LevelOfDetailRuleTrigger* interface and new concrete trigger classes could easily be added.

- ***LevelOfDetailRuleTrigger***: It has a type, which is used to identify the concrete trigger class. It exposes a function *isTriggered(double continuousLevel)*, which returns a boolean that can be used to check whether a trigger is currently active. It has to be implemented by all concrete rule trigger classes.
- ***LevelOfDetailRuleTriggerContinuous***: A concrete implementation of a trigger for a continuous level of detail. It requires two parameters *from* and *to* and is considered active when the current zoom level is between *from* and *to*.
- ***LevelOfDetailRuleTriggerDiscrete***: A concrete implementation of a trigger for a discrete level of detail. It requires one parameter of the type *DiscreteLevelOfDetailEnum* and is considered active, when the current zoom level is between the defined *from* and *to* values of the given discrete LoD.

Classes that inherit from the abstract *LevelOfDetailRule* class can be used as client-rules. Their structure is understood by the *RequestLevelOfDetailRulesActionHandler* and, once registered in the registry, they are sent to the client if requested. The client interprets them as a normal client-rule.

Server-rules require a slightly different treatment. Not only does the server require additional logic to apply them, they also have to be transferred to the client in a different form than client-rules. Unlike client-rules, all the client needs to know about a server-rule is the fact that is a server-rule, and its trigger information. No additional rule-specific parameters are needed, because the client does not have to execute any rule-specific logic. On the server, there exist two additional classes for server-rules:

- ***LevelOfDetailServerRule***: It is an interface that inherits from the *LevelOfDetailRuleInterface*. It exposes a function *handle(GModelElement element)* which takes a GModel element as parameter and applies the logic of a concrete server-rule to the element. All server-rules have to implement this function.
- ***LevelOfDetailServerRuleTransfer***: It is used as an envelope to transfer server-rules to the client and exposes nothing but the type of a rule and its trigger information. When the client requests all rules, the *RequestLevelOfDetailRulesActionHandler* automatically creates new instances of this class for each registered server-rule and converts them into JSON objects.

Server-rules are handled inside the already existing *ModelSubmissionHandler*. It is responsible for sending *setModel* and *updateModel* actions to the client. Before this is done, a call to the function *applyLevelOfDetailRules()* of the class *LevelOfDetailHandler* is done.

- ***LevelOfDetailHandler***: It exposes a function *applyLevelOfDetailRules()* and is responsible for applying server-rules to the current GModel. The function traverses the entire GModel tree and checks each element for referenced rules. If rules exist for an element, it is checked whether the rule is currently triggered. In case it is triggered, the *handle()* function is called and the rule is applied to the element.

Changes to the Language Besides changes to the protocol, the workflow language had to be changed as well. This was mainly necessary to add additional compartments for properties that can then be enabled and disabled with rules depending on the zoom level. Two labels have been added: One to display the duration and one to display the type of the task. Both of them are hidden on the default zoom level and displayed when the user zooms in. All other defined rules are applied to graphical elements that already existed in the original workflow language.

Client

The architecture of the client is very similar to that of the server. Most classes and interfaces defined on the server can also be found on the client.

Discrete Levels of Detail Discrete levels of detail are fetched via an action and then stored on the client for the entire remaining session. Each discrete LoD is stored inside an instance of the class *DiscreteLevelOfDetail*:

- ***DiscreteLevelOfDetail***: It consists of three variables: *from*, *to*, and *name*. Just as on the server, *from*, and *to* are used to store the range in which a level is supposed to be applied. The variable *name* is used to store the name of a level.

Actions Two new actions were added that are handled by the client: *SetDiscreteLevelOfDetail* and *SetLevelOfDetailRules*. Both actions are handled by their respective handler classes:

- ***SetDiscreteLevelOfDetailActionHandler***: It takes the received JSON object and converts all discrete LoDs into *DiscreteLevelOfDetail* TypeScript class objects. These objects are then stored inside the *LevelOfDetail* class object, which has to access and use them throughout the remaining session. Furthermore, when this action is received, the current discrete level of detail is determined and stored by fetching the current continuous level of detail of the stage and converting it into a discrete one.
- ***SetLevelOfDetailRulesActionHandler***: It takes the received JSON list of all rules that are defined on the server, and converts them into their respective TypeScript classes. Just as the discrete LoDs, they are then stored inside the *LevelOfDetail* class object, which needs to access them throughout the remaining session.

Rules In order for the client to understand all rules that were received during the *SetLevelOfDetailRules* action, they have to have an implementation on the client. All rules inherit from the interface *LevelOfDetailRuleInterface* and the abstract class *LevelOfDetailRule*:

- ***LevelOfDetailRuleInterface***: It exposes the basic fields which are required in all rules. Namely *trigger*, which holds information about when this rule is supposed to be applied, *type*, which holds the unique type of the rule, and *isServerRule*, which is used to tell if a rule is a server-rule.
- ***LevelOfDetailRule***: Each rule inherits from this abstract class. It inherits from *LevelOfDetailRuleInterface* and adds additional logic that is required by every rule. Among them are implementations for two functions: *isTriggered()*, which is used to determine if the rule is currently active, and *isNewlyTriggered()*, which is used to determine if a rule has become active in the most recent zoom event.

Just as on the server, information about when a rule is supposed to be applied is stored inside either a *LevelOfDetailRuleTriggerContinuous* or a *LevelOfDetailRuleTriggerDiscrete* class. Both of them inherit from the abstract class *LevelOfDetailRuleTrigger*, possess the same fields, and have the same function implementations as their equivalent class on the server.

All concrete rules on the client require an implementation of the *handle()* function. This function takes a graphical element that is about to be rendered on stage as an argument. Rule-specific logic can then be applied on the model inside this function, before it is returned and rendered. Currently, the following logic is applied on existing rules:

- ***CssStyleRule***: It appends all CSS styles of the rule to the received graphical element.
- ***VisibilityRule***: Depending on whether the rule-specific parameter *setVisibility* is *true* or *false*, this rule adds or removes the CSS class *hidden* to the graphical element. This CSS class sets the CSS *display* property to *none*, which removes the element from the stage.
- ***LayoutRule***: It appends all layout options of a rule to the received graphical element.

Each type of rule or rule-trigger has to be registered in order for the client to understand and use it. This can be done with the *registerLevelOfDetailRule()*, and *registerLevelOfDetailRuleTrigger()* function. Unlike the rule registry on the server, registering a rule on the client only makes the client aware that a rule exists by creating a connection between a unique type and its concrete class. It does not create connections between rules and model elements. This is done only on the server and then fetched by the client. Once a rule or rule-trigger is registered on the client, the client is able to initialize instances of it when necessary.

The class *LevelOfDetail* is another important class related to rules which is injected into many other classes throughout the client.

- ***LevelOfDetail***: It is a singleton instance which is instantiated during the initialization of the client. Among its responsibilities are the conversion of rule JSON objects

to TypeScript objects, conversion of continuous levels of detail to discrete levels, storing rules and their assignments, and looking up and returning all assigned rules for an object id which involves the evaluation of selectors.

Zoom Listener The zoom listener (*class ZoomListener*) is another new addition of this prototype. It inherits from the already existing *MouseListener* and is an event-listener that listens to the standard "wheel" DOM event. Every time, a *wheel* event is triggered, the listener fetches the current zoom level of the viewport. In the current implementation, the zoom level is calculated with the event's *deltaY* value, which represents the vertical scroll amount of the performed event. The zoom listener keeps a copy of the last zoom level in memory. This copy always represents the current zoom level of the viewport and can be fetched by other modules in case they need it. Currently, only the mouse wheel can be used to increase or decrease the zoom level.

Rendering Once the client has all information about LoDs, rules, and rule assignments, they can be applied to the SModel. This is done during the rendering process of the elements in the class *LevelOfDetailModelRenderer*, which inherits from the class *ModelRenderer*. Furthermore, the class *LevelOfDetailRenderer* is used to apply rules.

- **ModelRenderer:** It is the default implementation which is used to render model elements. It consists of two functions *renderChildren()* and *renderElement()*. The *renderChildren()* function is used to call *renderElement()* on all children of an element, and *renderElement()* calls the *render()* function of the given view-element.
- **LevelOfDetailModelRenderer:** It overwrites the default *renderElement()* function and adds additional functionality. In the function, when the root element is about to be rendered, it is checked if a new server round trip has to be made. This can occur on two occasions: (i) the client encountered a server-rule which has to be applied, and (ii) the client encountered a rule which requires the server to re-create the layout of the model. (ii) occurs when the server is responsible for sizes of objects and the client tries to apply a rule which changes this size. In both cases, the client sends a new *RequestModel* action to the server. The server applies all server-rules, recalculates the sizes with the help of a *RequestBounds* action, and sends the model back to the client. Currently, a switch between discrete levels of detail always triggers a new server round trip to recalculate sizes. Another new functionality added by the *LevelOfDetailModelRenderer* is a call to the class *LevelOfDetailRenderer*, which applies all relevant rules to model elements.
- **LevelOfDetailRenderer:** This renderer consists of two functions *checkForRerender()*, and *prepareNode()*. The function *checkForRerender()* is used to check if a new server round trip is required. It does this by using the *isNewlyTriggered()* function of all rule instances. If any of the rules have become active since the last time this was checked, this function returns *true*. The *prepareNode()* function is used to apply all relevant rules on an SModel element by calling the *handle()* function of each rule.

4.2.2 Prototype 2 - Visualizing Off-screen Elements

Unlike the first prototype, this prototype only consists of client-side functionalities. For this reason, no additions or changes to the server were needed. With the exception of small additions in HTML and CSS, the prototype is built entirely in TypeScript. The following section will give an overview of the most relevant algorithms, interfaces and classes of the implementation. A sequence diagram of its most important operations can be seen in Figure 4.5.

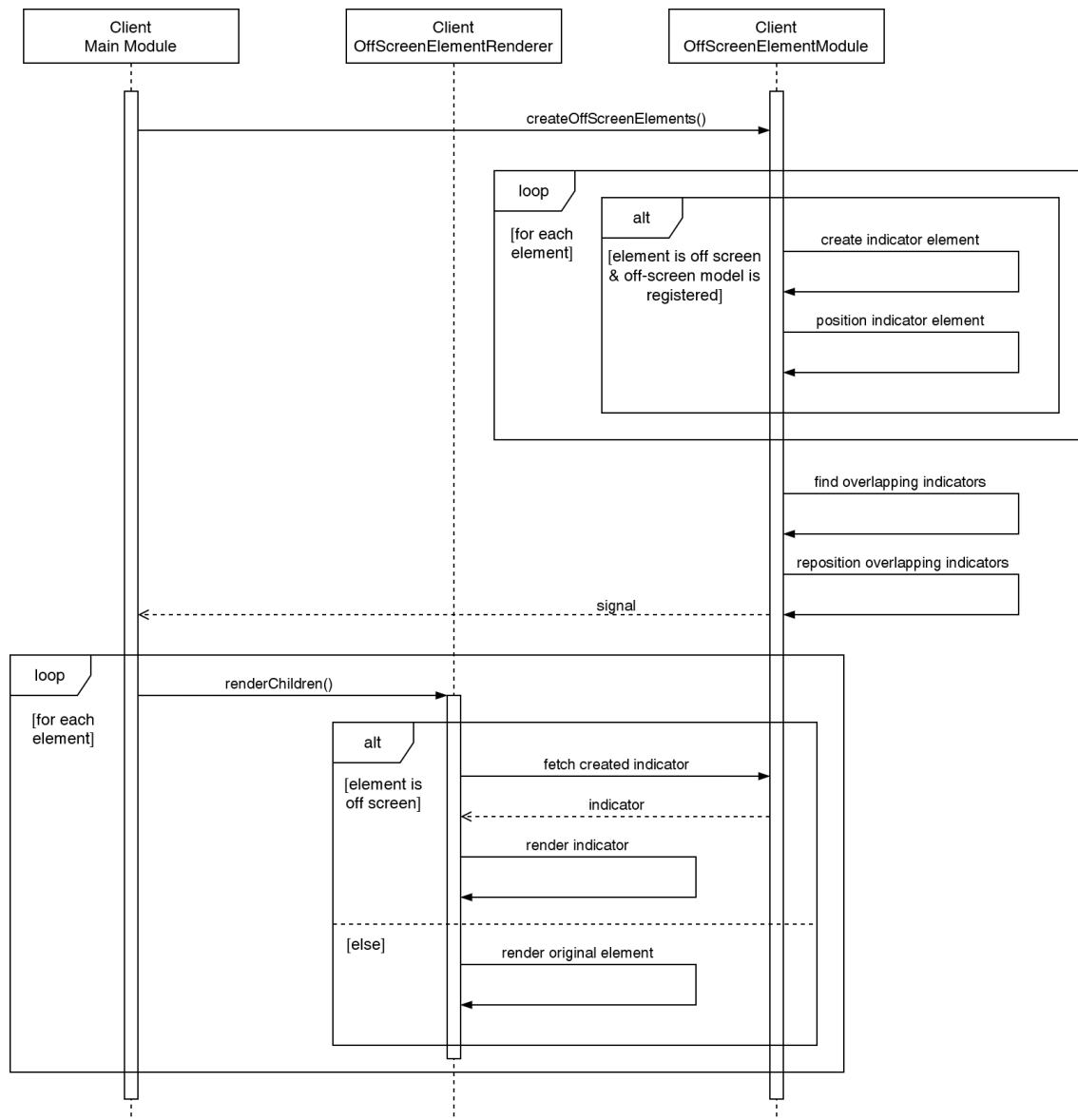


Figure 4.5: Most important operations of prototype 2 visualized in the form of a sequence diagram. It shows operations performed during the rendering process of a model.

Indicators Each original model element, which is supposed to have off-screen indicators, needs to have a defined model and view. The model is defined in the file *models.ts* and acts as the SModel element which is going to be displayed. The view is used to render the SModel element and is defined in the file *off-screen-views.tsx*. Each model then has to be registered for the client to be aware of it. This is done by calling the public *registerOffScreenModelElement()* function. It creates a connection between the original SModel type, the SModel indicator element, and the indicator view definition. Each registration tells the client to replace all elements of that type with the specified off-screen indicator once they are moved off screen.

Positioning Indicators Indicators are always positioned at the border of the viewport. To not cause confusion when an element disappears from the viewport and the indicator element becomes visible, the indicator has to be at a position close to the element it represents. The algorithm calculates the position by checking on which side of the viewport the invisible element currently is. Here it differentiates between, either left or right, and top or bottom. If the element is at the left or the right, the y-coordinate of the indicator is set to the same value as the y-coordinate of the element. If it is at the top or bottom, the x-coordinate is set instead. Furthermore, they will be restricted from going below or above the bounds of the viewport. This will prevent the indicator from disappearing off-screen, which can happen when an element is at a corner of the off-screen area (e.g., both, top and left of the viewport). The remaining coordinate will be set to the same value as that of the viewport's border at the respective side. E.g., if the off-screen element is to the left of the viewport, the indicators x-coordinate will be set to the x-coordinate of the left border of the viewport, and its y-coordinate will be set to that of the element. With this algorithm, the indicator is always rendered at the position on the border of the viewport that is closest to the element that it represents.

Overlapping Indicators As already mentioned, indicators that overlap with other indicators are merged into one. The merging algorithm is kept simple and just checks for groups of overlapping indicator elements and combines them. A group of overlaps is considered two or more indicator elements that do not have empty space between them. If a group is formed, the average position of all represented elements is calculated, which is going to be used to calculate the position of the new merged indicator element.

During manual tests of the overlapping implementation, we realized that, while moving the viewport around, elements were popping in and out a lot. This led to confusion and loss of track of off-screen elements. The reason for this behavior was that during panning events, the calculated position of the indicator elements is constantly adjusted. The adjustment of an indicators position could cause new overlaps to appear or old ones to disappear. In many situations where a lot of indicators are close together, this adjustment causes them to merge and separate often. To mitigate this behavior, a configuration variable *OVERLAP_SIZE_MULTIPLIER* was added, which can be used to increase or decrease the size of an indicator element only during the calculation of existing overlaps. For example, a value of 2 would cause all indicators to appear twice as large to the algorithm during the

calculation, which causes indicators to form larger groups and be merged together more easily. A value of 4 seemed to be a good compromise between having less elements pop in and out, and not having indicators merge too quickly.

Proxies Each registered off-screen indicator acts as a proxy for the original element it represents. Most interactions that can be carried out on the original elements, such as connecting edges, context menus, or tooltips, can also be carried out on the indicator element. This behavior is implemented by taking the original element and changing its appearance and position to that of the indicator element. Instead of having to re-implement existing interactions to also work on indicator elements, this keeps most of the original interactions intact and working out-of-the-box.

Click Listener Additionally to all existing interactions, one new interaction was added as well. Clicking on an indicator moves the viewport to the original element. This is done by implementing a new mouse listener in the class *OffScreenElementMouseListener*.

- ***OffScreenElementMouseListener***: It inherits from the existing *MouseListener* class and listens to the *mouseUp* event on SModel elements. When it is triggered, it is checked whether the element is an off-screen indicator element or not. Next, the number of overlapping indicators is counted. In case of only one element, the action *CenterAction* is applied, which moves the viewport to center one element (the original SModel element) on the stage. In case of multiple overlaps, the action *FitToScreenAction* is applied, which changes the zoom level and position of the viewport to a point in which all elements (all SModel element that the overlapping indicator represent) are visible. Furthermore, it applies a *Select* action, which selects all original elements that are represented by the indicators.

Edges Edges are usually represented by arrows pointing from one element to the other. Even when the source or target element is off screen, these arrows are still rendered and visible. In this prototype, the origin or end position of an arrow pointing to or from an off-screen element is adjusted to the position of its off-screen indicator element instead. This is done in the class *WorkflowEdgeView*.

- ***WorkflowEdgeView***: It inherits from the existing view implementation of edges and overwrites them. The difference to the original implementation is that, before the edge is rendered, the source and target elements are replaced by their indicator elements, if they currently exist.

Another change to the default implementation of edges is the calculation of the exact positions of their beginning and end. Because indicator elements are not part of the main information content of the editor, they are always kept at a relatively small size at the border of the viewport. This is implemented by using the SVG *scale* transformation to always stay at the same size, independent of the current zoom level. The existing implementation of the calculation of their start and end position did not consider the *scale* attribute, which resulted in arrows pointing to wrong positions when the current zoom

level was anything other than 1. For this reason, a new implementation of anchor points was added inside the class *RectangleScaledAnchor*.

- ***RectangleScaledAnchor***: It inherits from the class *RectangleAnchor* which contains the original implementation and overwrites the function *getAnchor()*. This function is used to calculate the position at the border of an element (edge start position) relative to another point on the stage (edge end position). It reuses the original implementation but multiplies the height and width of the received element by the value of its *scale* attribute.

Rendering Before all indicator elements can be rendered, they have to be prepared. This preparation consists of the following steps: (i) instantiating indicator elements for each element which is currently off screen, (ii) calculating their position around the viewport's border (iii) finding overlapping indicators, and (iv) calculating the new position of merged indicators. All these tasks are triggered every time the stage is rendered inside the root view of the workflow language, and are performed by the class *OffScreenElements*.

- ***OffScreenElements***: It stores necessary information related to off-screen elements. Among them are all registered off-screen views and their models, and the most recent indicators with all their calculated positions and overlaps. Furthermore, it provides the most important functions related to off-screen elements. Among them are functions to calculate indicator border positions and find overlaps. Additionally, it provides a *renderOffScreenElement()* function, which looks up the correct view for the received type, and calls the *render()* function of it.

Once all indicator elements are instantiated and ready to be used, they are rendered with the help of the class *OffScreenElementRenderer*.

- ***OffScreenElementRenderer***: This class inherits, similarly to *LevelOfDetailRenderer*, from the *ModelRender*, which is explained in more detail in Section 4.2.1. This renderer adds small changes to the original *renderElement()* function. Before an element is rendered, it is checked if it is currently visible. If it is not, the rendering is delegated to the *renderOffScreenElement()* function inside the *OffScreenElements* class.

4.3 Evaluation & Discussion

Both prototypes have been successfully integrated into the Eclipse Graphical Language Server Platform, and are fully functional and usable with the current version of the workflow language. Prototype 1 represents an effective way of condensing visible information based on the current zoom level. It uses a rule-based approach to define descriptions for the adaptation of graphical elements that consists of client- and server-rules. Its logic is split up into rule-definition, rule-assignment, and rule-application, which are either executed on the server- or client-side. With that, the prototype demonstrates how to integrate a semantic zooming functionality into a client-server architecture by providing a clear

separation of concerns. Furthermore, it provides an extendable architecture, e.g., in the form of interfaces for the definition of new rules, which can be used by developers to easily add new functionalities. During the conceptualization, care was taken to keep the current version of the graphical language server protocol intact, and the additions/changes at a minimum. This was accomplished by adding two new optional actions and their respective responses, and one change in the form of an additional parameter for the *requestModel* action.

Prototype 2 adds the visualization of off-screen elements to the client. It extends the already existing functionality of the client implementation to demonstrate how to add cues about off-screen elements at the border of the viewport. This helps users, not only to create a strong mental image of their workspace, but also to increase the contextual awareness that they have, by providing contextual information about elements surrounding the viewport in the form of off-screen indicators. Furthermore, it provides new ways of navigating the workspace, which decrease the time that is required to reach off-screen elements. What previously had to be done with potentially multiple zooming and panning actions can now be done with a simple click on an off-screen indicator. It also eliminates the need of scrolling and panning actions for other actions, e.g., connecting edges from on-screen to off-screen elements, by making each indicator act as a proxy for the respective off-screen element. Unlike prototype 1, it operates entirely on the client and does not require any new server functionalities. While it is integrated into the workflow language, its functionality is kept universal, and it can therefore also be used in combination with other node-like languages.

Ultimately, the goal of the implementation of these prototypes is to answer the research questions given in Section 1.2. The first question deals with the choice of their functionalities and tries to answer the following:

1. What is an appropriate means to improve the visualization of large models and interaction with them in GLSP-based modeling tools.

This question is answered with the help of the taxonomy (Section 3) and the knowledge that was gained during its development. The reason why these two functionalities were chosen, was because they represent a good mixture between genericity, development complexity, and increase in usability. Both chosen features represent functionalities that are generic enough to act as a good foundation for a prototype in a new environment but are also specific enough to be used in the domain of modeling tools. More details about the reasoning can be found in Section 4.1.1.

The second research question deals with the conceptual decisions of both prototypes:

- 2 How to generalize the concept/solution towards being applicable for other modeling languages/GLSP-based modeling tools.

With the successful integration of both prototypes into the Eclipse Graphical Language Server Platform, we provided an exemplary solution for the integration of advanced

visualization and navigation features in the currently largest GLSP-based modeling tool. During the conceptualization of both prototypes, a focus was put on keeping it universal and abstract. The detailed descriptions of their concepts in Section 4.1.4 and Section 4.1.5 are kept mostly independent of any concrete platforms. For this reason, they provide a good basis for integration of such features into other tools and platforms. Sections 4.2.1 and 4.2.2 describe their implementations in more detail and are therefore specific to Eclipse-GLSP. Because Eclipse-GLSP currently represents the most advanced GLSP environment, we believe that the developed concept of the prototypes can also be applied to other environments, and potentially also to any future versions of Eclipse-GLSP or other advanced GLSP environments without complex modifications. The main logic of the prototype implementations can be reused in other language servers that are based on Eclipse-GLSP without much effort. What cannot be avoided is additional configuration and adaptation to new graphical elements that may be required in other languages. All in all, we believe that the given concepts provide a good description for an integration of the chosen features into GLSP-based modeling tools. Furthermore, the implementations of both concepts prove their validity and provide a solid foundation for further additions or improvements in future works.

During the development of both prototypes, many things were learned and new ideas came up. The remaining parts of this chapter will give insight about limitations and improvements that could be made.

4.3.1 Limitations & Improvements

While both prototypes are fully functional and usable, they still display some limitations that ideally have to be addressed before they can be used at a larger scale. The following sections will go into further detail about limitations and potential solutions to them, and other improvements that can be made to both prototypes.

Prototype 1 - Semantic Zooming

The first prototype was not only more sophisticated to conceptualize and develop, it was also the one that more time was invested in. It required a slight adjustment of the existing protocol (addition of a zoom level parameter in the *requestModel* action) and two new server-side actions. Both of these changes are optional and do not have to be called or used. The same goes for the zoom level parameter, which does not have to be set either. This means that the server can still be used, even with clients that do not actively use semantic zooming. Depending on the changes that had to be made to the GModel of a language, a client without semantic zooming would usually always display the model at the lowest level of detail.

Animations Each time the discrete level of detail is changed, an animation is played that transitions elements from one state to the next. A limitation in the current implementation is that during this animation, no new user interaction events are processed. This is especially noticeable and problematic when zooming in or out in one continuous motion

with, e.g., the mouse wheel. During the animation, the zoom event is not recognized, which causes the continuous zoom event to be stopped for the duration of the animation. This seems to be a limitation of the Sprotty framework, which is the reason why it was only solved with a workaround instead of a fix. The workaround was to change the interval in which animations are played to a smaller amount ($\sim 200\text{ms}$). This caused the animation to be played long enough to still be visible, but fast enough to not noticeably disrupt zooming events.

Server Round Trips Another limitation is the additional server round trips that are required during every change in discrete level of detail. Because the server is responsible for the calculation of some values during the rendering process, such as the size of elements, the client needs to make a request to the server before it can re-render the diagram. While this is usually not a problem with discrete levels of detail, working with rules that are triggered continuously would require an extra server round trip every time a zoom action is performed by the user. This can also be a problem when using the *\$clevel* keyword in client-rules. If, e.g., the continuous zoom level is used to dynamically change the size of an element on the client, the server needs to be notified to recalculate the sizes of other elements that are affected by it. A reduction of required server round trips would increase the responsiveness of the client and save resources, such as bandwidth and processing power.

GModel Selection As already mentioned before, sometimes the existing functionality to select elements that are assigned a rule is too vague. Often, the same element type is used multiple times in a diagram but different rules depending on where the element is located are required. In some cases, this cannot be realized properly with the current solution, even with the implemented child and descendant selectors, and a workaround is needed. This workaround consists of the type of a specific element having to be replaced with a unique one, which can then be used to directly reference these elements. This could be prevented with integration of a full selector engine, similar to the CSS selector engine. It would allow for a much bigger variety in element selection and greatly reduce the occurrence of this problem.

Dynamic Size Adjustments Currently, the implementation automatically adjusts the size of each diagram element to the size of its content. This means that, e.g., whenever a property is made visible by a rule, the parent's size is adjusted to fit the new property. While this is intended behavior in the current implementation, it can, in very rare cases, be the cause for a structural change which destroys the user's mental map, as described in Section 3.4.2. In most cases, an element is only adjusted very slightly, which does not change the overall structure at all. A potential solution to this problem would be to only show further elements once there is enough space for it instead of at a fixed zoom level. E.g., when the user zooms in, the newly created space inside model elements is calculated and new properties are added if they fit into this space. This would keep all elements at a constant size instead of making small adjustments on every change of discrete level of detail.

The problem here is that, in the current implementation, the client cannot calculate the correct sizes for each element, because part of that responsibility is also on the server-side. In order to calculate the correct size, the server would have to be asked each time an element is rendered. This requires some fundamental changes which is why this feature has not been added in this prototype. Furthermore, this introduces other problems such as two tasks of the same type adding a property at different zoom levels because one has a longer value than the other.

Prototype 2 - Visualizing Off-screen Elements

The second prototype is currently only implemented on the client which is why the protocol has not needed to be adjusted. This brings us to its first limitation.

No Integration of the Server The questions, whether the server is supposed to be involved in the visualization of off-screen indicators, is not answered easily. On the one hand, indicator elements are not directly part of the model, do not have to be persisted, and are directly dependent on information that only the client has (e.g., position and bounds of the viewport). All of these reasons speak for an implementation of the client side. On the other hand, being able to control the visualization of off-screen elements from the server speaks in favor of the goal of LSP, which is to prevent having to implement the same functionality multiple times for different clients.

With the current workflow of GLSP, it is hard to shift the entire feature to the server-side for mainly two reasons: (i) the server does not have necessary information that are needed to determine the position of indicator elements. The positioning of indicators depends on the position and size of the viewport which have to be sent to the server. (ii) the positions of all indicators have to be adjusted during all zooming or panning events. If the server is responsible for the positioning of indicators, this would require a server round trip during and after every event. This would increase the number of required round trips significantly.

Event though, in this prototype, the complete functionality lies on the client, ideally, it should be distributed among the server and the client. Because only the client has all necessary information, the logic which determines the position and size of indicators should be kept on the client. The server should still be able to have a saying in multiple configurational matters. Examples are: which elements should have indicators and which elements do not need any, CSS classes of indicators of different elements, configuration about when to merge which elements together, or the content inside indicator elements. All clients can then be implemented to understand the configuration, which is supplied by the server.

Additional Configuration Parameter Another improvement that can be made is the addition of configuration parameter to the client. Currently, the only parameter is the value of `OVERLAP_SIZE_MULTIPLIER`, which determines how close elements have to be together for them to merge. Another important parameter that is not implemented currently, is the maximum distance to the center of the viewport that elements can have,

before their indicators are not rendered anymore ("area of influence" [FD13, p. 143]). Elements that are too far away or have no relationship with the currently visible elements are often not relevant, and only decrease the performance or clutter the workspace. For this reason, and with the correct algorithm, they could potentially be omitted.

Identification of Indicators Knowing which indicator represents which model element can often be difficult but is required for an efficient utilization of this prototype. Currently, there exists three parameters which give information about the model it represents: (i) its position, (ii) its color, and (iii) its edges. This is often not enough and can be further improved, e.g., with ideas given in [FD13]. They visually adjust indicators with, e.g., a stacking effect for multiple indicators at the same location, or 2D representation of off-screen elements and their relationships to other elements. Furthermore, they also propose to add specialized interactions for indicators. Among them are tooltips, which show the full name of all elements represented by indicators, or even a full visual representation of their layout with relationships.

Positioning of Indicators Originally, we implemented a different algorithm to position indicator elements. The algorithm can be explained as follows: An imaginary line is drawn from the center of the element to the center of the viewport. Now, the intersection point of that line with the border of the viewport is calculated and the indicator is rendered at that point. While this represents a good alternative strategy to position indicator elements, we decided to change it to the current algorithm, because it sometimes caused indicator elements to appear at unexpected locations. For example, when an element with a large horizontal size is moved off screen at the bottom of the left border of the viewport, its indicator will appear further above than most would expect. The reason for this is that the center of the element, which is used as a reference point for the calculation of the indicator's position, is already much further off screen than the remaining visible part of the element, but the indicator is only shown once the original element is completely off screen. This causes the intersection point with the center of the viewport to be further above than the position where the element was last seen by the user. As a solution, we changed the algorithm to the one that is explained in Section 4.2.2, which does not have this problem.

4.3.2 Performance

For both prototypes, a small internal test has been conducted. The goal of this test was to find out how the implementation behaves with a diagram which consists of a large number of elements. The test was conducted by performing simple zooming and panning interactions on a diagram which consists of a defined number of tasks and edges, and checking at what point there is a noticeable delay for these interactions. As a reference point, the original implementation of the original workflow language server has been taken. Furthermore, the time that the rendering of all elements takes has been measured for each iteration. The results were as follows:

4. PROTOTYPE

	Elements	Event	Render Time
Original	10 Tasks/10 Edges	Zooming	~1ms
		Panning	~1ms
	100 Tasks/100 Edges	Zooming	~5ms
		Panning	~5ms
	500 Tasks/500 Edges	Zooming	~33ms
		Panning	~32ms
Prototype 1	10 Tasks/10 Edges	Zooming	~1ms
		Panning	~1ms
	100 Tasks/100 Edges	Zooming	~13ms
		Panning	~11ms
	500 Tasks/500 Edges	Zooming	~68ms
		Panning	~57ms
Prototype 2	10 Tasks (0 Indicators)/10 Edges	Zooming	~1ms
		Panning	~1ms
	100 Tasks (0 Indicators)/100 Edges	Zooming	~7ms
		Panning	~6ms
	100 Tasks (99 Indicators)/100 Edges	Zooming	~8ms
		Panning	~6ms
	500 Tasks (0 Indicators)/500 Edges	Zooming	~38ms
		Panning	~39ms
	500 Tasks (499 Indicators)/500 Edges	Zooming	~57ms
		Panning	~61ms

Table 4.1: Performance test results

Throughout the entire test, both events, zooming and panning, showed very similar rendering times. Furthermore, the measured times of both prototypes were very similar to each other, and roughly twice as high than those of the original implementation. This can be led back to the additional logic that is executed during each iteration of the rendering function in both prototypes. It is to say that, while the performance aspect was not completely neglected during the development of the prototypes, a good performance was not a primary focus. For this reason, the performance can probably be improved in many parts of the prototypes, especially by caching calculations and reusing them.

For the subjective evaluation, a slight unresponsiveness could be felt during the execution of the test with 100 Tasks. This delay was barely noticeable and could only be felt when a direct comparison to a diagram with fewer tasks was given. During the execution of the test with 500 Tasks, the delay became very noticeable. This could be felt even more

during the execution of prototype-specific functionality. For prototype 1, during the switch between discrete levels of detail, and for prototype 2, while displaying multiple indicators. Especially the switch between discrete levels of detail added a noticeably long delay to the interaction, which is not reflected in the rendering times. It can probably be attributed to the additional server round trip that has to be made in prototype 1. Besides that, the additional prototype-specific logic of both prototypes also plays a major role in the differences in responsiveness.

It is to note, that we intend both prototypes to be used in combination with models created by humans (see definition of conceptual modeling by Mylopoulos [Myl92]). With a large number of entities, a model often becomes too complex, unmanageable, and difficult to understand [Moo97]. For this reason, we do not expect the number of entities of models that utilize either of the two prototypes to go as high as 500 or even 100. This performance test was merely conducted to get a feeling for the current technical limit of the prototypes.

Summary & Conclusion

The final goal of this work was to successfully combine language server protocol driven modeling tools with advanced visualization and interaction tools. Visualization and interaction features that ultimately provide ways to make the processes of working with large models more efficient and increase productivity. In order to achieve this, this work was split up into three major parts: (i) Establishing the current state of such features in today's modeling tools and diagramming software, (ii) creating an overview of features that have the potential to achieve our set goals, and (iii) integrating chosen features into a suitable environment in the form of a prototype.

Part (i) has been conducted in the course of our state-of-the-art analysis. We looked at multiple known modeling tools, such as Eclipse Modeling Tools and Microsoft Visio, and came to the conclusion that only a surprisingly small number of tools offer support for advanced interaction and visualization features. Most features that we could find, for example, a basic minimap or simple buttons to show/hide additional information of certain elements, were still rather basic and known concepts. In order to utilize more advanced features, such as semantic zooming, they would have to be implemented first in one of the tools' supported scripting languages, such as VBA or JavaScript. Even though some tool providers offer help in that regard with guides and tutorials, this still requires model engineers to possess knowledge in the field of software development.

Part (ii) has been conducted with the help of a taxonomy that was created in the course of this work. We presented a generic taxonomy of advanced information visualization with applications to conceptual modeling. The taxonomy structures visualization features along the three higher-level dimensions *Presentation*, *Interface*, and *Data*. On lower levels, the taxonomy further defines seven dimensions, each of which with specific characteristics. We evaluated the usefulness of the taxonomy with ex-ante and ex-post taxonomy evaluation methods by relating it to existing research and by applying our taxonomy on real-world objects, among them a self-developed prototype, respectively. This taxonomy combines established categorizations with new and original ones. From a scientific viewpoint,

combining those two forms a novel taxonomy with an expressivity in classifying visualization features lacking in past literature. From a practical viewpoint, the presented taxonomy can facilitate a new feature's ideation and conceptualization phases. The taxonomy can push method engineers and tool developers towards rethinking model representation, and designing and implementing advanced information visualization features. Furthermore, we used the knowledge that we gained during the development and evaluation of the taxonomy to come up with a set of features that best fulfil our initial goal and are suitable for a prototype in a graphical language server protocol based environment.

With the help of Part (i) and (ii), our first research question could be answered:

1. What is an appropriate means to improve the visualization of large models and interaction with them in GLSP-based modeling tools.

We decided on two features, **semantic zooming**, and **visualization of off-screen elements**. Semantic zooming significantly improves the readability of larger models by only showing information that are relevant to the user based on the current zoom level. The visualization of off-screen elements increases spatial awareness and improves interaction processes such as the navigation between model elements. Both features are integrated into the main view of the user interface, which makes them very space-efficient, and reduces the amount of focus switches that are required to effectively use them. Another reason why they were chosen was because they represent a good mixture between genericity, development complexity, and increase in usability. Both chosen features represent functionalities that are generic enough to act as a good foundation for a prototype in a new environment but are also specific enough to add benefits to modeling tools in the field of model engineering. Further details about both prototypes can be found in the following section and in Chapter 4.

Part (iii) of this work consisted of the prototype conceptualization and development. In the course of this work, two prototypes were successfully integrated into the Eclipse Graphical Language Server Platform. The first prototype adds support for semantic zooming to the server and client. Elements are able to dynamically change their detail level and show more information when zoomed in, and less when zoomed out. This is implemented by being able to define rules on the server, which are then triggered on discrete or continuous zoom levels. When they are triggered, they are executed either on the client or the server, and are able to, e.g., toggle the visibility or change the size of certain model elements. To realize prototype 1, only slight modifications and additions had to be made to the existing protocol. They consisted of, firstly, one additional optional parameter, the current zoom level, which has to be appended to the *requestModel* action when sent to the server by the client. Secondly, two new actions have been added that are both initiated by the client. The first one requests all discrete levels of detail, and the second one requests all defined rules for all elements from the server.

The second prototype deals with the visualization of off-screen elements and improves the process of interacting with them. Elements that are usually not visible because they

are off-screen are instead rendered in a different graphical representation at the border of the viewport. To prevent cluttering the available space, these graphical representations are more compact than their original representation and are combined when there are too many too close to each other. Additionally, they act as a proxy which represents the original off-screen element. This allows most interactions that can be performed on the original elements to also be carried out on their proxy representation. It replaces the usual zooming and panning that has to be performed to get off-screen elements back on-screen before they can be interacted with. On top of that, actions like connecting edges or simple hover effects can be done directly on their proxies. Besides the improved means of performing interactions, this prototype is also meant to increase the spatial orientation of users. Because off-screen elements are now permanently visible inside the viewport, users are constantly and mostly subconsciously reminded of their whereabouts. With this condition, users should not only be able to create a stronger mental image of their workspace but also speed up its creation process.

With these prototypes, the second research question can be answered:

- 2 How to generalize the concept/solution towards being applicable for other modeling languages/GLSP-based modeling tools.

Both prototypes were successfully integrated into a graphical language server protocol based environment with minimal changes to the underlying protocol. They provide an exemplary solution for advanced visualization and interaction features in a GLSP environment. With the Eclipse-GLSP being the largest GLSP environments, their concept and implementation serve as a good base for their integration into other platforms as well. We believe that the developed concept of the prototypes can also be applied to other environments, and potentially also to any future versions of Eclipse-GLSP or other advanced GLSP environments without complex modifications. Furthermore, the implementations also provide a solid foundation for further additions or improvements to these features in the future. Additional implementation details, evaluation, and limitations of both prototypes can be found in Chapter 4.

Future Work

This work has provided fully functional prototypes of advanced visualization and interaction features in the Eclipse-GLS Platform, and with that, has taken the first steps towards an integration of such features into GLSP-based environments. Nevertheless, it has only scratched the surface of the never-ending topic of information visualization and interaction. Regarding the theoretical part of this work, there are too many ideas and concepts of features out there already to be covered in one master's thesis, and likely even more to come in the future. A significant part of future work will be to keep track and find more features that can be used to improve processes in the field of model engineering. This could be done by digging even deeper into the nearly endless number of existing information visualization tools, or even coming up with completely new features by working together

with experts in other scientific fields, such as, for example, cognitive science. In this search for more features, the taxonomy that was developed in the course of this work can be of assistance. Related to the practical part of this work, some individual improvements, such as reducing the number of required server round trips or adding additional configuration, can be made, which are described in more detail in Section 4.3.1. Furthermore, a more extensive evaluation can be performed. The main focus of this work was to successfully implement specified features into the system. Their evaluation was merely a descriptive one so far, and can be accommodated with user studies, similarly to those that have already been performed for other features in past literature. This would bring up strengths and weaknesses of the designed features which could then be used in a future development iteration to further improve them. Another future step for both prototypes is to start the integration process of them into production systems. This would be an excellent way of finding potential unsolved technical issues and collecting user feedback, similarly to user studies. Before this can be done, both features would have to be integrated into language servers of other languages. Both prototypes have been developed with the workflow language, and currently it is the only supported language. A further goal of future work would be to integrate the prototype into other language servers as well.

Furthermore, we plan to share our findings with more scientific communities. Parts of this work have been submitted and accepted at international scientific conferences. Our taxonomy [DCPB22b] will be presented at the International Conference on Conceptual Modeling (ER 2022) and our prototypes [DCPB22a] will be part of the International Conference on Model Driven Engineering Languages and Systems (MODELS 2022). This should help to reach the target user groups of the taxonomy and the developed prototypes, and serve as an additional round of feedback for further improvements in future works.

List of Figures

2.1	Semantic zooming in yFiles	15
2.2	Example for grouping elements	16
2.3	Collapsible tree-view in yFiles	17
2.4	Examples for grid and ruler features in the diagramming tool on https://app.diagrams.net (Accessed: 06.03.2022)	18
2.5	Example for a minimap in the Eclipse IDE	19
3.1	Taxonomy visualized	28
3.2	Comparison of a view with overview-scrollbar to one without in Sublime Text 3	34
3.3	Example for an overview-plus-detail interface in Microsoft PowerPoint	35
3.4	Example for focus-plus-context distortion	36
3.5	Example for no distortion/basic zooming in the Windows 10 Magnifier app	36
3.6	Example for cue-based techniques in Google Maps	37
3.7	Example for a scrollbar with hover functionality in the software development IDE Webstorm	45
3.8	Example for user-defined views in the form of pages in diagramming tool on app.diagrams.net	46
3.9	Example for a basic zoom event in Microsoft Word	47
3.10	Example for speed-dependent automatic zoom [IH00]	49
3.11	Example for semantic zooming applied to UML diagrams [FDB08]	50
3.12	Example for showing additional information on button click in diagramming tool app.diagrams.net	51
3.13	Example for a fisheye implementation in Apple's operating system macOS	52
3.14	Example for City Lights by Zellweger et al. [ZMG ⁺ 03]	53
3.15	Example for EdgeRadar by Gustafson and Irani [GI07]	54
3.16	Example for off-screen element visualization by Frisch and Dachsel [FD13]	56
3.17	Example for the peek definition feature	57
3.18	Example for onion graph notations by [KM07]	58
3.19	Example for semantic depth-of-field technique by Miksch et al. [MH01]	59
3.20	Example for lenses by Tominski et al. [TAVHS06]	61
3.21	Example for detail lenses by Karnick et al. [KCJ ⁺ 09]	62
3.22	Example for halos by Baudisch and Rosenholtz [BR03]	63
3.23	Example for wedges by Gustafson et al. [GI07]	64

3.24	Example for enriched wedges in comparison to other off-screen visualization techniques by Gladisch et al. [GST13]	64
3.25	Example for proxy-based technique "hop" by Irani et al. [IGY06]	65
3.26	Example for a scrollbar with indicators in software development IDE Webstorm	66
4.1	Eclipse Graphical Language Server Platform Client Life-cycle	73
4.2	Screenshots of the developed prototype 1	74
4.3	Screenshots of the second developed prototype	78
4.4	Prototype 1 operation sequence	81
4.5	Prototype 2 operation sequence	87

List of Tables

2.1	Newly added metadata in different levels of detail in Google Maps	12
4.1	Performance test results	96

Appendix 1: Software and tools that was/were looked at in the course of the ex ante evaluation.

Commercial Software/Tools			Reference (Accessed on 20.11.2021)
#	Title	Company	
1	Windows 10	Microsoft	
2	Microsoft PowerPoint	Microsoft	https://www.microsoft.com/en-us/microsoft-365/powerpoint
3	Microsoft Word	Microsoft	https://www.microsoft.com/en-us/microsoft-365/word
4	MS Visio*	Microsoft	https://www.microsoft.com/en-us/microsoft-365/visio/flowchart-software
5	MS Visio Online*	Microsoft	https://office.live.com/start/visio.aspx
6	Visual Studio Code	Microsoft	https://code.visualstudio.com/
7	Bing Maps	Microsoft	https://www.bing.com/maps/
8	macOS	Apple	
9	Google Maps	Google	https://www.google.com/maps
10	JetBrains IntelliJ IDEA	JetBrains	https://www.jetbrains.com/idea/
11	JetBrains WebStorm	JetBrains	https://www.jetbrains.com/webstorm/
12	JetBrains MPS*	JetBrains	https://www.jetbrains.com/mps/
13	Sublime Text 3	Sublime HQ	https://www.sublimetext.com/3
14	Eclipse Papyrus*	Eclipse Foundation	https://www.eclipse.org/papyrus/
15	Eclipse Graphiti*	Eclipse Foundation	https://www.eclipse.org/graphiti/
16	Eclipse Modeling Tools*	Eclipse Foundation	https://www.eclipse.org/downloads/packages/release/2020-06/r/eclipse-modeling-tools
17	EdrawMax*	Edrawsoft	https://www.edrawsoft.com/edraw-max/
18	ConceptDraw DIAGRAM*	CS Odessa	https://www.conceptdraw.com/products/drawing-tool
19	StarUML*	MKLabs	https://staruml.io/
20	Visual Paradigm*	Visual Paradigm	https://www.visual-paradigm.com/
21	XMind*	XMind	https://www.xmind.net/desktop/
22	yEd Desktop*	yWorks	https://www.yworks.com/products/yed
23	yEd Live*	yWorks	https://www.yworks.com/products/yed-live
24	yFiles*	yWorks	https://www.yworks.com/yfiles-overview
25	diagrams.net*	JGraph	https://www.diagrams.net/
26	Gliffy Online*	Perforce	https://www.gliffy.com/products/gliffy-online
27	Lucidchart*	Lucid Software	https://www.lucidchart.com
28	creately*	Cinergix	https://creately.com/
29	cacoo*	Nulab	https://cacoo.com/
30	miro*	Miro	https://miro.com/
31	GenMyModel*	Axellience	https://www.genmymodel.com/
32	Prezi*	Prezi	https://prezi.com/

* evaluated during the state-of-the-art analysis

#	Title	Author(s)	Literature	Reference
33	Speed-dependent automatic zooming for browsing large documents	Takeo Igarashi and Ken Hinckley	Takeo Igarashi and Ken Hinckley. Speed-dependent automatic zooming for browsing large documents. In Proceedings of the 13th annual ACM symposium on User interface software and technology, pages 139–148, 2000.	
34	Generalized fisheye views	George W Furnas	George W Furnas. Generalized fisheye views. Acm Sigchi Bulletin, 17(4):16–23, 1986.	
35	Fisheye menu	Benjamin B Bederson	Benjamin B Bederson. Fisheye menus. In Proceedings of the 13th annual ACM symposium on User interface software and technology, pages 217–225,2000.	
36	A fisheye text editor for relaxed-wysiwiw groupware.	Saul Greenber	Saul Greenber. A fisheye text editor for relaxed-wysiwiw groupware. In Conference Companion on Human Factors in Computing Systems, pages 212–213, 1996.	
37	An improved fisheye zoom algorithm for visualizing and editing hierarchical models.	Reinhard et al.	Tobias Reinhard, Silvio Meier, and Martin Glinz. An improved fisheye zoom algorithm for visualizing and editing hierarchical models. In Second International Workshop on Requirements Engineering Visualization (REV2007), pages 9–9. IEEE, 2007.	
38	City lights: contextual views in minimal space.	Zellweger et al.	Polle T Zellweger, Jock D Mackinlay, Lance Good, Mark Stefik, and Patrick Baudisch. City lights: contextual views in minimal space. In CHI'03 extendedabstracts on Human factors in computing systems, pages 838–839, 2003.	
39	Comparing visualizations for tracking off-screen moving targets.	Sean G Gustafson and Pourang P Iran	Sean G Gustafson and Pourang P Iran. Comparing visualizations for tracking off-screen moving targets. In CHI'07 Extended Abstracts on Human Factorsin Computing Systems, pages 2399–2404, 2007.	
40	Visualizing offscreen elements of node-link diagrams	Mathias Frisch and Raimund DachseIt	Mathias Frisch and Raimund DachseIt. Visualizing offscreen elements of node-link diagrams. Information Visualization, 12(2):133–162, 2013.	
41	Towards seamless semantic zooming techniques for uml diagrams.	Frisch et al.	Mathias Frisch, Raimund DachseIt, and Tobias Brückmann. Towards seamless semantic zooming techniques for uml diagrams. In Proceedings of the 4thACM Symposium on Software Visualization, pages 207–208, 2008.	
42	Onion graphs for focus+context views of uml class diagrams.	Huzefa Kagdi and Jonathan I Maletic.	Huzefa Kagdi and Jonathan I Maletic. Onion graphs for focus+context views of uml class diagrams. In 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, pages 80–87. IEEE,2007.	
43	Edge: An extendible graph editor.	Frances Newbery Paulisch and Walter F Tichy.	Frances Newbery Paulisch and Walter F Tichy. Edge: An extendible graph editor. Software: Practice and Experience, 20(S1):S63–S88, 1990.	
44	Semantic depth of field	Silvia Miksch and Helwig Hauser.	Silvia Miksch and Helwig Hauser. Semantic depth of field. In Proc. IEEE Symp. Information Visualization, pages 97–104. Citeseer, 2001.	

45	Fisheye tree views and lenses for graph visualization.	Tominski et al.	Christian Tominski, James Abello, Frank Van Ham, and Heidrun Schumann. Fisheye tree views and lenses for graph visualization. In Tenth International Conference on Information Visualisation (IV'06), pages 17–24. IEEE, 2006.
46	Route visualization using detail lenses.	Karnick et al.	Pushpak Karnick, David Cline, Stefan Jeschke, Anshuman Razdan, and Peter Wonka. Route visualization using detail lenses. IEEE transactions on visualization and computer graphics, 16(2):235–247, 2009.
47	Halo: a technique for visualizing off-screen objects.	Patrick Baudisch and Ruth Rosenholtz.	Patrick Baudisch and Ruth Rosenholtz. Halo: a technique for visualizing off-screen objects. In Proceedings of the SIGCHI conference on Human Factors in computing systems, pages 481–488, 2003.
48	Wedge: clutter-free visualization of off-screen locations.	Gustafson et al.	Sean Gustafson, Patrick Baudisch, Carl Gutwin, and Pourang Irani. Wedge: clutter-free visualization of off-screen locations. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 787–796, 2008.
49	Navigation recommendations for exploring hierarchical graphs.	Gladisch et al.	Stefan Gladisch, Heidrun Schumann, and Christian Tominski. Navigation recommendations for exploring hierarchical graphs. In International Symposium on Visual Computing, pages 36–47. Springer, 2013.
50	Improving selection of off-screen targets with hopping.	Irani et al.	Pourang Irani, Carl Gutwin, and Xing Dong Yang. Improving selection of off-screen targets with hopping. In Proceedings of the SIGCHI conference on Human Factors in computing systems, pages 299–308, 2006.
51	The continuous zoom: A constrained fisheye technique for viewing and navigating large information spaces.	Bartram et al.	Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. The continuous zoom: A constrained fisheye technique for viewing and navigating large information spaces. In Proceedings of the 8th annual ACM symposium on User interface and software technology, pages 207–215, 1995.

Appendix 2: Features that were looked at in the course of the ex post evaluation.

Feature	Presentation			Interaction		Data		
	Interface Type	Interface Coupling	Interaction Type	Animation	Information Structure	Input Data Type	(Input Data)	Output Data Type (Output Data)
Basic scrollbar [11][2][3]+	Overview-plus-detail	additional views	direct + indirect	no animation	structure preserving	quantitative/discrete	x-y coordinates of scrollbar	quantitative/discrete
Overview scrollbar [2]	Overview-plus-detail	additional views	direct + indirect	no animation	structure preserving	quantitative/discrete	x-y coordinates of scrollbar	x-y coordinates of detail-view
Lens on scrollbar hover [13]	Overview-plus-detail	additional views	direct	structural	structure preserving	quantitative/discrete	x-y coordinates of scrollbar	section of detail-view
Thumbnail overview [2]	Overview-plus-detail	additional views	direct + indirect	continuous	structure preserving	qualitative	reference to model element (side)	discrete section of detail-view
User-controlled view definition [30][32]	Overview-plus-detail	additional views	direct	continuous	structure preserving	qualitative	reference to section of detail-view	discrete section of detail-view
Minimap [16]	Overview-plus-detail	additional views	direct + indirect	no animation	structure preserving	quantitative/discrete	current viewport dimensions (size + position + zoom level)	section of detail-view
Basic continuous zoom [11][2][3]+	zooming	main view	direct	no animation	structure preserving	quantitative/continuous	abstract zoom-factor (e.g. mouse wheel scroll distance)	continuous zoom-level
Basic discrete zoom [9]	zooming	main view	direct	continuous	structure preserving	quantitative/continuous	abstract zoom-factor (e.g. mouse wheel scroll distance)	discrete zoom-level
Speed-dependent automatic zooming [33]	zooming	main view	indirect	multiple (continuous + informational)	temporary changing	quantitative/continuous	scroll-speed	quantitative/continuous
Adding objects of different information space on zoom [41]	zooming	main view	indirect	multiple (structural + informational + continuous)	structure preserving	quantitative/continuous	zoom-factor	qualitative
Adding objects of current information space on zoom [41]	zooming	main view	indirect	multiple (structural + informational + continuous)	structure preserving	quantitative/continuous	zoom-factor	qualitative
Adding details on click [5][10][23]+	focus-plus-context	main view	direct	structural	temporary changing	qualitative	references to model elements (e.g. properties)	mixed
Magnifying glasses [1]	focus-plus-context	main view	direct	continuous	structure preserving	quantitative/discrete	x-y coordinates of cursor	quantitative/discrete
Fisheye zoom [9][35][36]+	focus-plus-context	main view	direct	continuous	structure changing	quantitative/discrete	x-y coordinates of cursor	mixed
City Lights [38]	focus-plus-context	main view	indirect	structural	structure preserving	quantitative/discrete	position of off-screen model elements + current viewport dimensions	mixed
EdgeRadar [39]	focus-plus-context	main view	indirect	no animation	structure preserving	quantitative/discrete	position of off-screen model elements + current viewport dimensions	mixed
Visualizing Off-Screen Elements [40]	focus-plus-context	main view	indirect + direct	multiple (structural + continuous)	structure changing	quantitative/discrete	position of off-screen model elements + current viewport dimensions	mixed
Peek definition [6][10][11]+	focus-plus-context	main view	direct	structural	temporary changing	qualitative	reference to domain model element	quantitative/discrete
Grouping elements [22][23][24]	focus-plus-context	main view	direct	structural	structure changing	qualitative	references to domain model elements	mixed
Onion graph [42]	focus-plus-context	main view	visual only	no animation	structure preserving	mixed	graph	mixed
Grid [14][26][31]+	focus-plus-context	main view	visual only	no animation	structure preserving	quantitative/discrete	current viewport dimensions (size + position + zoom level)	mixed
Ruler [3][17][25]+	focus-plus-context	main view	visual only	no animation	structure preserving	Continuous	current viewport dimensions (size + position + zoom level)	mixed
Semantic depth-of-field technique [44]	focus-plus-context	main view	indirect	no animation	structure preserving	mixed	algorithm specific input (relevancy of domain model elements)	quantitative/continuous
Lenses [46]	focus-plus-context	main view	visual only	no animation	structure preserving	mixed	algorithm specific input (relevancy of domain model elements)	quantitative/discrete
Thumbnail overview [1]	focus-plus-context	main view	direct	continuous	temporary changing	mixed	information about currently open programs	qualitative
Search results [7][9][10]+	cue-based	additional views	direct	continuous	structure preserving	qualitative	references to model elements (e.g. POIs)	qualitative
Halos [47]	cue-based	main view	visual only	no animation	structure preserving	quantitative/discrete	position of off-screen model elements	quantitative/continuous
Wedges [48][49]	cue-based	main view	visual only	no animation	structure preserving	quantitative/discrete	position of off-screen model elements	quantitative/continuous
Proxies [7][9][50]+	cue-based	main view	direct	continuous	structure preserving	quantitative/discrete	position of off-screen model elements	quantitative/discrete
Scrollbar with indicators [6][10][11]+	cue-based	additional views	direct	multiple (structural + continuous)	structure preserving	quantitative/discrete	position of off-screen model elements	qualitative
Semantic zooming in GLSP (Prototype 1)	zooming	main view	indirect	multiple (structural + continuous)	structure preserving	quantitative/continuous	zoom-factor	qualitative
Visualizing off-Screen elements in GLSP Prototype 2)	focus-plus-context	main view	indirect + direct	continuous	structure changing	quantitative/discrete	position of off-screen model elements + current viewport dimensions	mixed

Bibliography

- [AAB⁺84] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. Pyramid methods in image processing. *RCA engineer*, 29(6):33–41, 1984.
- [BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis lectures on software engineering*, 3(1):1–207, 2017.
- [Bed00] Benjamin B Bederson. Fisheye menus. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 217–225, 2000.
- [BF14] Dominik Bork and Hans-Georg Fill. Formal aspects of enterprise modeling methods: A comparison framework. In *47th Hawaii International Conference on System Sciences, HICSS 2014, Waikoloa, HI, USA, January 6-9, 2014*, pages 3400–3409. IEEE Computer Society, 2014.
- [BGBS02] Patrick Baudisch, Nathaniel Good, Victoria Bellotti, and Pamela Schraedley. Keeping things in context: a comparative evaluation of focus plus context screens, overviews, and zooming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 259–266, 2002.
- [BH94] Benjamin B Bederson and James D Hollan. Pad++ a zooming graphical interface for exploring alternate interface physics. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, pages 17–26, 1994.
- [BHDH95] Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. The continuous zoom: A constrained fisheye technique for viewing and navigating large information spaces. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, pages 207–215, 1995.
- [BKP18] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. Systematic analysis and evaluation of visual conceptual modeling language notations. In *12th International Conference on Research Challenges in Information Science, RCIS 2018, Nantes, France, May 29-31, 2018*, pages 1–11. IEEE, 2018.

- [BKP20] Dominik Bork, Dimitris Karagiannis, and Benedikt Pittl. A survey of modeling language specification techniques. *Inf. Syst.*, 87, 2020.
- [BR03] Patrick Baudisch and Ruth Rosenholtz. Halo: a technique for visualizing off-screen objects. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 481–488, 2003.
- [BR21] Dominik Bork and Ben Roelens. A technique for evaluating and improving the semantic transparency of modeling language notations. *Softw. Syst. Model.*, 20(4):939–963, 2021.
- [Bün19] Hendrik Bänder. Decoupling language and editor-the impact of the language server protocol on textual domain-specific languages. In *MODELSWARD*, pages 129–140, 2019.
- [CC88] GI Cooperative and Fort Collins. The unique qualities of a geographic information system: a commentary. *Photogrammetric Engineering and Remote Sensing*, 54(11):1547–1549, 1988.
- [CKB09] Andy Cockburn, Amy Karlson, and Benjamin B Bederson. A review of overview+ detail, zooming, and focus+ context interfaces. *ACM Computing Surveys (CSUR)*, 41(1):1–31, 2009.
- [CRM91] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181–186, 1991.
- [CS04] Andy Cockburn and Joshua Savage. Comparing speed-dependent automatic zooming with traditional scroll, pan and zoom methods. In *People and Computers XVII—Designing for Society*, pages 87–102. Springer, 2004.
- [DA05] Heather Devine and Anthony D Andre. Effect of scroll bar and navigation menu co-location on web performance. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 49, pages 1454–1458. SAGE Publications Sage CA: Los Angeles, CA, 2005.
- [DCPB22a] Giuliano De Carlo, Langer Philip, and Dominik Bork. Advanced visualization and interaction in glsp-based web modeling: Realizing semantic zoom and off-screen elements. In *International Conference on Model Driven Engineering Languages and Systems*, pages –in press–. Springer, 2022.
- [DCPB22b] Giuliano De Carlo, Langer Philip, and Dominik Bork. Rethinking model representation - a taxonomy of advanced information visualization in conceptual modeling. In *International Conference on Conceptual Modeling*, pages –in press–. Springer, 2022.

- [Don78] William C Donelson. Spatial management of information. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 203–209, 1978.
- [Dun09] Dustin Dunsmuir. Selective semantic zoom of a document collection. *Available at, (Oct. 30, 2009)*, pages 1–9, 2009.
- [ecla] Eclipse graphical language server platform. <https://github.com/eclipse-glsp/glsp>. Accessed: 23.04.2021.
- [eclb] Graphiti - a graphical tooling infrastructure. <https://www.eclipse.org/graphiti/>. Accessed: 23.04.2021.
- [FD13] Mathias Frisch and Raimund Dachzelt. Visualizing offscreen elements of node-link diagrams. *Information Visualization*, 12(2):133–162, 2013.
- [FDB08] Mathias Frisch, Raimund Dachzelt, and Tobias Brückmann. Towards seamless semantic zooming techniques for uml diagrams. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 207–208, 2008.
- [Fur86] George W Furnas. Generalized fisheye views. *Acm Sigchi Bulletin*, 17(4):16–23, 1986.
- [GBGI08] Sean Gustafson, Patrick Baudisch, Carl Gutwin, and Pourang Irani. Wedge: clutter-free visualization of off-screen locations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 787–796, 2008.
- [GI07] Sean G Gustafson and Pourang P Irani. Comparing visualizations for tracking off-screen moving targets. In *CHI’07 Extended Abstracts on Human Factors in Computing Systems*, pages 2399–2404, 2007.
- [Gre96] Saul Greenberg. A fisheye text editor for relaxed-wysiwis groupware. In *Conference Companion on Human Factors in Computing Systems*, pages 212–213, 1996.
- [GRGS15] Jens Gulden, Hajo A Reijers, J Grabis, and K Sandkuhl. Toward advanced visualization techniques for conceptual modeling. In *CAiSE Forum*, pages 33–40. Citeseer, 2015.
- [GST13] Stefan Gladisch, Heidrun Schumann, and Christian Tominski. Navigation recommendations for exploring hierarchical graphs. In *International Symposium on Visual Computing*, pages 36–47. Springer, 2013.
- [Gul16] Jens Gulden. Recommendations for data visualizations based on gestalt patterns. In Gang Li and Yale Yu, editors, *International Conference on Enterprise Systems*, pages 168–177, 2016.

- [HBP02] Kasper Hornbæk, Benjamin B Bederson, and Catherine Plaisant. Navigation patterns and usability of zoomable user interfaces with and without an overview. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4):362–389, 2002.
- [HD04] Tan Kim Heok and Daut Daman. A review on level of detail. In *Proceedings. International Conference on Computer Graphics, Imaging and Visualization, 2004. CGIV 2004.*, pages 70–75. IEEE, 2004.
- [HHFN13] Shuhei Hiya, Kenji Hisazumi, Akira Fukuda, and Tsuneo Nakanishi. clooca: Web based tool for domain specific modeling. In *Demos/Posters/StudentResearch@ MoDELS*, pages 31–35. Citeseer, 2013.
- [HMPR04] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.
- [IGY06] Pourang Irani, Carl Gutwin, and Xing Dong Yang. Improving selection of off-screen targets with hopping. In *Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 299–308, 2006.
- [IH00] Takeo Igarashi and Ken Hinckley. Speed-dependent automatic zooming for browsing large documents. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, pages 139–148, 2000.
- [Kal08] Banda KalyanaChakravarthy. Visualizing the mpi programs: Using continuous semantic zooming. *Computer Science Technical Reports*, 2008.
- [KBRCn02] Maged N Kamel Boulos, Abdul V Roudsari, and Ewart R Carso n. Health-cybermap: a semantic visual browser of medical internet resources based on clinical codes and the human body metaphor. *Health Information & Libraries Journal*, 19(4):189–200, 2002.
- [KCJ⁺09] Pushpak Karnick, David Cline, Stefan Jeschke, Anshuman Razdan, and Peter Wonka. Route visualization using detail lenses. *IEEE transactions on visualization and computer graphics*, 16(2):235–247, 2009.
- [KM07] Huzefa Kagdi and Jonathan I Maletic. Onion graphs for focus+ context views of uml class diagrams. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 80–87. IEEE, 2007.
- [KMO⁺21] Dennis Kundisch, Jan Muntermann, Anna Oberländer, Daniel Rau, Maximilian Roeglinger, Thorsten Schoormann, and Daniel Szopinski. An update for taxonomy designers - methodological guidance from information systems research. *Business & Information Systems Engineering*, 10 2021.
- [KR08] Kathy Kotiadis and Stewart Robinson. Conceptual modelling: knowledge acquisition and model abstraction. In *2008 Winter Simulation Conference*, pages 951–958. IEEE, 2008.

- [LCP16] José Paulo Leal, Helder Correia, and José Carlos Paiva. Eshu: An extensible web editor for diagrammatic languages. In *5th Symposium on Languages, Applications and Technologies (SLATE'16)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
- [LMC⁺02] A Chris Long, Brad A Myers, Juan Casares, Scott M Stevens, and Albert Corbett. Video editing using lenses and semantic zooming. 2002.
- [LRC⁺03] David Luebke, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson, and Robert Huebner. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.
- [McC04] Steve McConnell. *Code complete*. Pearson Education, 2004.
- [MH01] Silvia Miksch and Helwig Hauser. Semantic depth of field. In *Proc. IEEE Symp. Information Visualization*, pages 97–104. Citeseer, 2001.
- [mica] Microsoft language server protocol implementations. <https://microsoft.github.io/language-server-protocol/implementors/servers/>. Accessed: 23.04.2021.
- [micb] Microsoft language server protocol specification. <https://microsoft.github.io/language-server-protocol/specifications/specification-current/>. Accessed: 23.04.2021.
- [Moo97] Daniel Moody. A multi-level architecture for representing enterprise data models. In *International Conference on Conceptual Modeling*, pages 184–197. Springer, 1997.
- [Moo09] Daniel Moody. The “physics” of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779, 2009.
- [MS95] Jeyakumar Muthukumarasamy and John T Stasko. Visualizing program executions on large data sets using semantic zooming, 1995.
- [Myl92] John Mylopoulos. Conceptual modelling and telos. *Conceptual modelling, databases, and CASE: An integrated view of information system development*, pages 49–68, 1992.
- [NVM13] Robert C Nickerson, Upkar Varshney, and Jan Muntermann. A method for taxonomy development and its application in information systems. *European Journal of Information Systems*, 22(3):336–359, 2013.
- [obe] Obeo - graphical server protocol. <https://github.com/ObeoNetwork/GraphicalServerProtocol>. Accessed: 23.04.2021.

- [OBEL10] Tom Owen, George Buchanan, Parisa Eslambochilar, and Fernando Loizides. Supporting early document navigation with semantic zooming. In *International Conference on Asian Digital Libraries*, pages 168–178. Springer, 2010.
- [PF93] Ken Perlin and David Fox. Pad: an alternative approach to the computer interface. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 57–64, 1993.
- [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.
- [RKP12] Louis M Rose, Dimitrios S Kolovos, and Richard F Paige. Eugenia live: a flexible graphical modelling tool. In *Proceedings of the 2012 Extreme Modeling Workshop*, pages 15–20, 2012.
- [RM10] Hajo A Reijers and Jan Mendling. A study into the factors that influence the understandability of business process models. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 41(3):449–462, 2010.
- [RMG07] Tobias Reinhard, Silvio Meier, and Martin Glinz. An improved fisheye zoom algorithm for visualizing and editing hierarchical models. In *Second International Workshop on Requirements Engineering Visualization (REV 2007)*. IEEE, 2007.
- [Rob08] Stewart Robinson. Conceptual modelling for simulation part i: definition and requirements. *Journal of the operational research society*, 59(3):278–290, 2008.
- [Rob13] Stewart Robinson. Conceptual modeling for simulation. In *2013 Winter Simulations Conference (WSC)*, pages 377–388. IEEE, 2013.
- [Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.
- [SC00] Sônia Fernandes Silva and Tiziana Catarci. Visualization of linear time-oriented data: a survey. In *Proceedings of the first international conference on web information systems engineering*, volume 1, pages 310–319. IEEE, 2000.
- [Sch95] George Schussel. Client/server past, present, and future. *Formerly Available URL: <http://news.dci.com/geos/dbsejava.htm>*, 1995.
- [SFA⁺11] Michael Stengel, Mathias Frisch, Sven Apel, Janet Feigenspan, Christian Kästner, and Raimund Dachsel. View infinity: a zoomable interface for feature-oriented software development. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1031–1033, 2011.

- [SGJ93] Guttorm Sindre, Björn Gulla, and Håkon G Jokstad. Onion graphs: aesthetics and layout. In *Proceedings 1993 IEEE Symposium on Visual Languages*, pages 287–291. IEEE, 1993.
- [Shn03] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*, pages 364–371. Elsevier, 2003.
- [SKA94] Samudra Sengupta, Takayuki Dan Kimura, and Ajay Apte. An artist’s studio: a metaphor for modularity and abstraction in a graphical diagramming environment. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 128–136, 1994.
- [Spe14] Robert Spence. *Information Visualization: An Introduction*. Springer Publishing Company, Incorporated, 3rd edition, 2014.
- [SZG⁺96] Doug Schaffer, Zhengping Zuo, Saul Greenberg, Lyn Bartram, John Dill, Shelli Dubs, and Mark Roseman. Navigating hierarchically clustered networks through fisheye and full-zoom methods. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 3(2):162–188, 1996.
- [TAVHS06] Christian Tominski, James Abello, Frank Van Ham, and Heidrun Schumann. Fisheye tree views and lenses for graph visualization. In *Tenth International Conference on Information Visualisation (IV’06)*, pages 17–24. IEEE, 2006.
- [TM04] Melanie Tory and Torsten Moller. Rethinking visualization: A high-level taxonomy. In *IEEE symposium on information visualization*, pages 151–158. IEEE, 2004.
- [TRS21] Benjamin Ternes, Kristina Rosenthal, and Stefan Strecker. User interface design research for modeling tools: A literature study. *Enterprise Modelling and Information Systems Architectures (EMISAJ)*, 16, 2021.
- [TSS09] Christian Thum, Michael Schwind, and Martin Schader. Slim - a lightweight environment for synchronous collaborative modeling. In *International Conference on Model Driven Engineering Languages and Systems*, pages 137–151. Springer, 2009.
- [VMP14] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*, pages 233–238. IEEE, 2014.
- [YM15] YoungSeok Yoon and Brad A Myers. Semantic zooming of code change history. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 95–99. IEEE, 2015.
- [ZMG⁺03] Polle Zellweger, Jock Mackinlay, Lance Good, Mark Stefik, and Patrick Baudisch. City lights: Contextual views in minimal space. 05 2003.