FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Spatialized Software Inspection and Review: Challenges, Opportunities, and Prototypes

Ana Rita de Oliveira Carneiro

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Ademar Manuel Teixeira de Aguiar

July 31, 2025

Spatialized Software Inspection and Review: Challenges, Opportunities, and Prototypes

Ana Rita de Oliveira Carneiro

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. João Pascoal Faria
Examiner: Prof. Vasco Amaral
Supervisor: Prof. Ademar Aguiar

July 31, 2025

Resumo

Com o aumento da complexidade e escala dos sistemas de software, os programadores enfrentam diversos desafios na sua compreensão através dos métodos tradicionais de inspeção de software. Estes incluem o tempo prolongado necessário para compreender o sistema, a sobrecarga cognitiva causada por representações textuais e uma deteção de erros menos eficiente. As ferramentas de inspeção atuais não oferecem visualizações em tempo real, espaciais ou em camadas. Assim, a implementação de técnicas visuais pode ajudar a melhorar a compreensão e a sustentação do sistema ao longo do tempo.

Para testar esta hipótese, foi desenvolvida uma extensão para o VSCode que gera e visualiza diagramas UML de forma imediata e interativa. O InSight UML utiliza inteligência artificial para gerar diagramas em Mermaid.js e oferece uma interface interativa e personalizada com zoom em camadas e controlos de filtragem. A funcionalidade de filtragem em camadas da ferramenta inspira-se na lógica de aplicações de localização geográfica, como o Google Earth, que permitem visualizar diferentes níveis de abstração de forma seletiva.

Com base numa análise refletiva, conclui-se que as técnicas de visualização espacial implementadas no InSight UML parecem apoiar uma compreensão mais rápida e eficaz do código-fonte de sistemas de software.

Como trabalho futuro, pretende-se expandir a ferramenta para suportar outros tipos de diagramas e linguagens de programação, integrar funcionalidades de realidade aumentada e realizar uma validação completa com dois grupos de utilizadores: um utilizando ferramentas tradicionais e outro utilizando o InSight UML.

Abstract

With the increase in complexity and scale of software systems, developers face many challenges in understanding them using traditional software inspection methods. These include extended time needed for comprehension, cognitive overload from textual representations, and less efficient defect detection. Current inspection tools lack real-time, spatial, and layered visualizations. Therefore, implementing visual techniques can help improve software comprehension and maintainability.

To test this hypothesis, a VSCode extension for dynamic UML diagram generation and visualization was developed. InSight UML uses artificial intelligence to generate Mermaid.js diagrams and provides a personalized and interactive webview with layered zoom and filtering toggles. The tool's layered filtering feature is inspired by the logic of geographical information applications such as Google Earth, which present different levels of abstraction that can be selectively displayed.

Based on a reflective analysis, it is concluded that the spatialized visualization techniques in InSight UML appear to support faster and more effective software comprehension.

Future work includes expanding the tool to support additional diagram types and programming languages, integrating augmented reality features, and conducting a full-scale validation with two user groups: one using traditional tools and another using InSight UML.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to achieve a better and more sustainable future for all. It includes 17 goals to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

Despite this dissertation being technical, it addresses the following Sustainable Development Goals indirectly:

SDG 4 Quality Education - Ensure inclusive and equitable quality education and promote lifelong learning opportunities for all

SDG 8 Decent Work and Economic Growth

SDG 9 Industry, Innovation and Infrastructure

SGD	Target	Contribution	Performance Indicators and Metrics
4	4.4	Contributes to improving technical skills and education by implementing a tool that supports better comprehension of code and systems using visualization techniques, which can be used in learning environments	Reported improvement in comprehension and engagement by students in academic settings ...
8	8.2	Contributes to increasing developer productivity and easier system maintainability using innovative technologies, leading to more efficient work practices in software engineering	Decrease in task completion time during inspections and increase in code review efficiency ...
9	9.5	Contributes to promoting innovation in tool development by extending spatial visualization and layered filtering capabilities within IDEs, promoting smarter infrastructures for software teams	Tool adoption rate ...

Acknowledgements

I would like to thank my supervisor, Prof. Ademar Aguiar, who was always available to answer my questions, no matter the time. Without his support, I wouldn't have been able to write this dissertation and learn so much. His advice, suggestions, and ideas continually brought me one step closer to the right path. He helped make this experience a little less scary.

I also want to thank all the professors who guided me throughout my Bachelor's and Master's degrees, and who taught me everything I know over the past five years.

On a personal note, I would like to thank my family for always being there for me, especially my mother, who supported me throughout this entire journey. Without her, none of this would have been possible.

I would also like to thank Hugo and Xanxo, for being the best people I could have met during this journey. They accompanied me through everything and supported me throughout these years. Their friendship and presence were a constant source of motivation, especially to actually go to class.

Lastly, I want to thank all my friends from Fafe, who listened to me complain about this dissertation for months, but always offered their support, encouragement, and much-needed distractions when I needed them most. A special thanks to Tânia and Mendes, our Zoom study sessions were what kept me focused and productive throughout this entire process.

Ana Rita

"Our greatest glory is not in never falling, but in rising every time we fall"

Confucius

Contents

List of Figures

List of Tables

List of Acronyms

UML	Unified Modeling Language
AI	Artificial Intelligence
AR	Augmented Reality
SDLC	Software Development Lifecycle
DSL	Domain Specific Language
SVG	Scalable Vector Graphics
CSS	Cascading Style Sheets
IDE	Integrated Development Environment
LLM	Large Language Models
OAL	Object Action Language
NLP	Natural Language Processing
QA	Quality Assurance

Chapter 1

Introduction

In the evolving landscape of software development, ensuring code quality, maintainability, and architectural consistency has become increasingly challenging. As systems grow in complexity, so does the need for more intuitive and efficient tools to support software inspection and review. This chapter introduces the context, motivation, and core challenges addressed by this research, which explores the use of spatial and interactive visualization techniques to enhance the software comprehension process.

1.1 Context

Software inspection and revision are fundamental processes in system development. Their main goals are to identify defects, refactor code, analyze software architectures, and ensure everything follows the right patterns and the correct specifications.

The use of spatial and visual representations, mainly using graphical representations and diagrams, can enrich overall software analysis and comprehension, for example of code and architecture designs, allowing developers to see that information in a more intuitive and interactive manner. These non-textual representations that enable the user to view the system information as a scheme allow for easy identification of problems, helps with the refactoring of code structures, and for a deeper comprehension of the software architecture and flow. Other works have explored embedded visualizations in code editors, using dynamic analysis of the behavior of a system while it's being executed to improve the understanding of complex software systems in real time (Krause-Glau et al., 2023).

Program comprehension is essential for software developers, who usually spend a long time trying to understand source code and architectures (Siegmund, 2016). This research explores the use of spatialization techniques with responsive interaction to optimize the software inspection and revision process. It will be focused on the logical architecture and coding phases and activities of the Software Development Life Cycle (SDLC) with the main goal of increasing the maintainability of systems, allowing faster and more efficient problem solving for software developers and quality assurance (QA) testers.

1.2 Motivation

The motivation underlying this research comes from the need to rethink traditional software inspection and revision perspectives. These are often based on textual documentation and linear processes, for example, which are methods that can lead to misunderstandings between colleagues, lack of clarity and understanding and waste of time, demanding high cognitive effort, impacting team efficiency and consequently the quality of the final product.

By putting into practice some spatial visualization techniques by representing the information in diagrams, we aim to create a more dynamic and interactive work environment. This approach allows for a better comprehension of the systems, enabling developers and Quality Assurance (QA) testers to view not only the code, but also the way the system interacts in the general context of the project. In addition to that, such visual techniques can offer better visualization and analysis of the software architecture. Software comprehension is essential for programming activities, yet it is an underexplored field (Boehm-Davis, 1988). This change in software inspection, architecture analysis, and coding can make the process of improving the code base and the architecture design a lot easier, which is essential for system maintainability.

1.3 Problem

Traditional processes of software inspection and review are usually quite slow and strongly dependent in static textual representations that sometimes aren't really able to convey a complete vision of complex systems (Oliveira, 2024), causing various inefficiencies, for example:

- Ineffective communication and understanding between team members, mainly in bigger teams, where developers may have different levels of understanding of the system overall architecture and areas with defects.
- Difficulties in fast identification of architecture flaws and code defects, since architectural failures, logic consistencies or problematic interactions between components may not be detected very easily with conventional techniques.
- Lack of real time interactive visualization, which makes understanding the interactions between all the different parts of the system harder and prevents easy and fast defect detection in multiple layers, like code, architecture and their logic.

With the increase in the complexity of modern software systems, traditional inspection and review methods have become less adequate for large scale and high complexity projects, raising the need for more interactive, intuitive and comprehensive tools that support the understanding and analysis of code bases. The failure to easily visualize code structure and component interactions with immediate and responsive updates can negatively affect the ability to understand how different modules relate, making it harder to identify potential design flaws or areas for refactoring.

1.4 Objectives

The goal is to develop a practical tool to help developers and QA testers in software inspection and review processes, enabling faster and more effective understanding of large complex systems.

By implementing a tool for spatial diagram generation and visualization, with interactive and layered visualization and selection techniques, this research aims to reduce the overall time and effort required for software comprehension and maintenance tasks, while improving accuracy and completeness.

Although the title refers to both "inspection" and "review", the main focus of this work is on supporting software inspection activities, primarily those performed during code comprehension and quality analysis. In this context, the term “review” is used in a broader sense, rather than referring strictly to formal reviews by external stakeholders. The prototype is designed to assist in developer-driven exploratory inspection tasks.

1.5 Structure

This dissertation contains four chapters in total. It begins with this Introduction chapter, followed by Chapter 2, which explains some background terms that might be useful for understanding concepts mentioned later and presents a Literature Review and a State of the Art analysis of research made in the field of graphical visualizations and diagram generation techniques.

Chapter 3 explains in detail the thesis problem, goals, and the foundation of the tool developed, including its architecture, features, and a reflection on its usefulness.

Finally, Chapter 4 includes information regarding future work and the conclusions drawn from this research.

Chapter 2

Literature Review

2.1 Introduction

This chapter analyzes, compares, and discusses different approaches and studies found in the literature that aim to improve software visualizations for software comprehension and inspection, mainly regarding UML diagram creation and visualization.

As observed, understanding the behavior and structure of large systems is still a big challenge for developers. As a result, the demand for efficient visualization tools that enable them to perform software comprehension tasks faster and with less effort is increasing, alongside the growing complexity and size of systems. These tools and techniques show relevant ways to face these comprehension challenges with new ways to visualize different artifacts of the SDLC, including spatial visualizations.

This literature review was conducted by selected articles from major digital libraries, such as ACM Digital Library, IEEE Explore, Science Direct, using search terms such as "software visualization", "UML generation and visualization", "diagrams for software comprehension", and focusing on the relevant works published in the last decade.

The following sections will summarize the main contributions, solutions, and results of the ongoing research on this theme.

2.2 Background

Firstly, some concepts that are relevant for the theme of this research and that serve as a basis to understand some of the studies mentioned later will be briefly explained.

Throughout this document, the terms diagram and model are used with awareness of their distinction. A model represents the abstract structure and behavior of a certain system, while a diagram is a specific visual representation of that model. UML is centered around diagrammatic notation, but essentially conveys underlying models. In this paper, both terms may be used depending on the context.

2.2.1 Software Comprehension

Software Comprehension is the ability to understand what a program does and how it does it (Boehm-Davis, 1988). It is the main activity of software developers and where they spend the most time (Siegmund, 2016).

With software systems getting bigger and more complex every day, due to advances in technology, developers face more difficulties and spend more time on software comprehension tasks. Looking directly at raw source code and trying to mentally reconstruct its logic, structure, and interactions can be cognitively demanding, so visualization techniques that can ease this comprehension process are needed. Researches believe that better software visualization enables better software comprehension (Nur et al., 2010).

Good software comprehension is necessary to successfully solve software maintenance problems (Austin et al., 2005) since maintainers must first understand the existing system before they can safely modify or extend it.

2.2.2 Reverse Engineering

Reverse Engineering is a software engineering discipline that inverts the traditional software life cycle. More specifically, it is the process of analyzing a system, understanding its components and how they interact, and creating representations for that system in another form or with a higher level of abstraction (Buss et al., 2010).

In the maintenance process, when most of the important information is contained in the source code with no added context, reverse engineering can be very helpful by creating high-level representations for an existing software system. Reverse Engineering tools help extract information from source code, execution traces, or historical data, query the extracted info, and build high-level views that are more readable for humans, aiding with software comprehension (Canfora et al., 2011).

A common use of Reverse Engineering is the creation of UML diagrams based on source code (Vinita et al., 2008). By bringing code content into visual UML models, programmers or software engineers are able to easily review an implementation, identify potential bugs or deficiencies and look for possible improvements.

2.2.3 The C4 Model

The C4 model is a framework for visualizing software architecture at multiple levels of abstraction, from system context down to individual components (Figure 2.1).

It was created to help software development teams describe and communicate software architecture using a set of abstraction layers at different levels of detail, which users can zoom in and out of (Brown, 2024).

These abstractions are organized into a full software system, composed of containers (applications and data stores), each containing one or more components, which are ultimately implemented

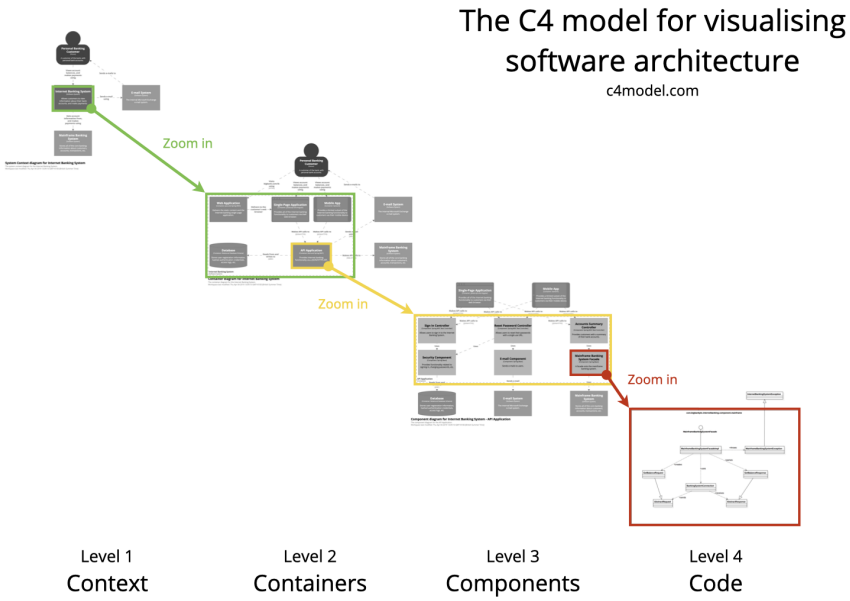


Figure 2.1: The C4 model for visualizing software architecture from Brown, 2024. This figure is included to illustrate layout and concept; fine detail is not necessary.

by code elements, as shown in Figure 2.2. This layered organization allows developers to understand the system in manageable sections, rather than from a single, overwhelming global view.

2.2.4 Model-Based Software Engineering

Model-Based Software Engineering (MBSE) is a software development approach that promotes the use of abstract models to support and partially automate software development. Models play a central role in this approach (Jolak; Ho-Quang, et al., 2018).

While this dissertation does not follow a fully model-driven development process, it draws inspiration from MBSE in its use of diagrams to enhance software comprehension and support analysis tasks.

2.3 Automated UML Diagram Generation

This section explores automated approaches to generating diagrams using UML (Unified Modeling Language), a widely adopted modeling standard (Object Management Group, 2017), from natural language requirements or source code using various approaches. It covers tools and techniques mainly for class and sequence diagram creation, including the use of Large Language Models (LLMs), Natural Language Processing (NLP), rule-based tools, and reverse engineering methods. These approaches were selected to represent the main current strategies for transforming requirements or code into models, each containing its own unique advantages and challenges. Although not the core subject of this dissertation, particular attention is given to LLMs due to their

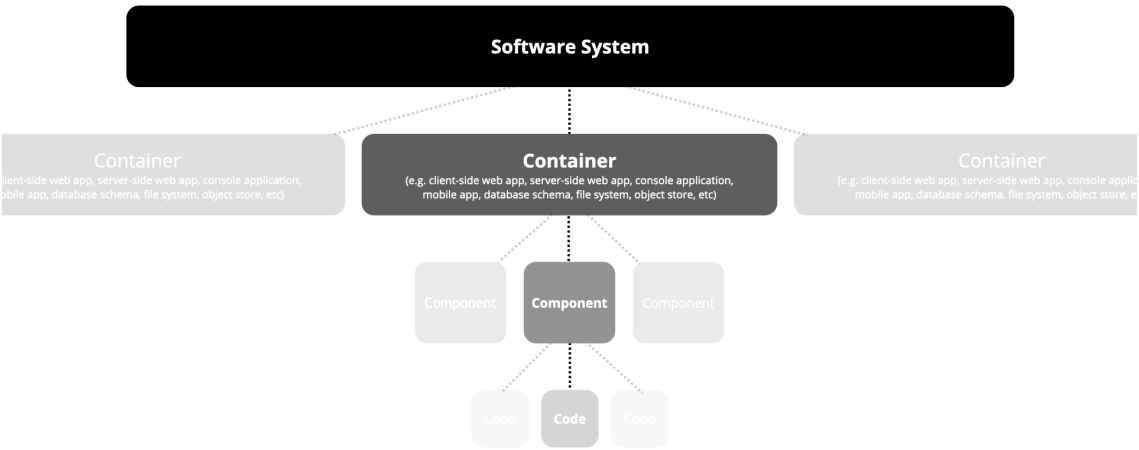


Figure 2.2: The C4 model abstractions from Brown, 2024. This figure is included to illustrate layout and concept; fine detail is not necessary.

growing relevance in software engineering. Understanding how LLMs are being applied to UML diagram generation provides valuable context for later chapters.

2.3.1 Large Language Models

Large Language Models (LLMs) have been growing exponentially in the last few years, one of the reasons being their ability to support and automate software engineering activities. While they have been widely explored in the code generation area, their use in software modeling, including UML diagram generation, has been largely under-explored (Shehata et al., 2024). Lately, more research has been made with LLMs creating diagrams from requirements specifications written in natural language or directly from the source code.

Research shows that even if LLMs produce overall less accurate class diagrams, they are still correct enough to support good comprehension of the system when compared to human-generated diagrams, while being much faster. With slightly more detailed instructions and requirements, LLMs are expected to perform significantly better, suggesting that they can serve as a useful aid in software-related tasks (De Bari et al., 2024).

A related example is UML Miner (Ardimento et al., 2024) which is a Visual Paradigm tool that uses a Retrieval-Augmented Generation (RAG) LLM to analyze and provide natural language feedback on class diagrams (Figure 2.3).

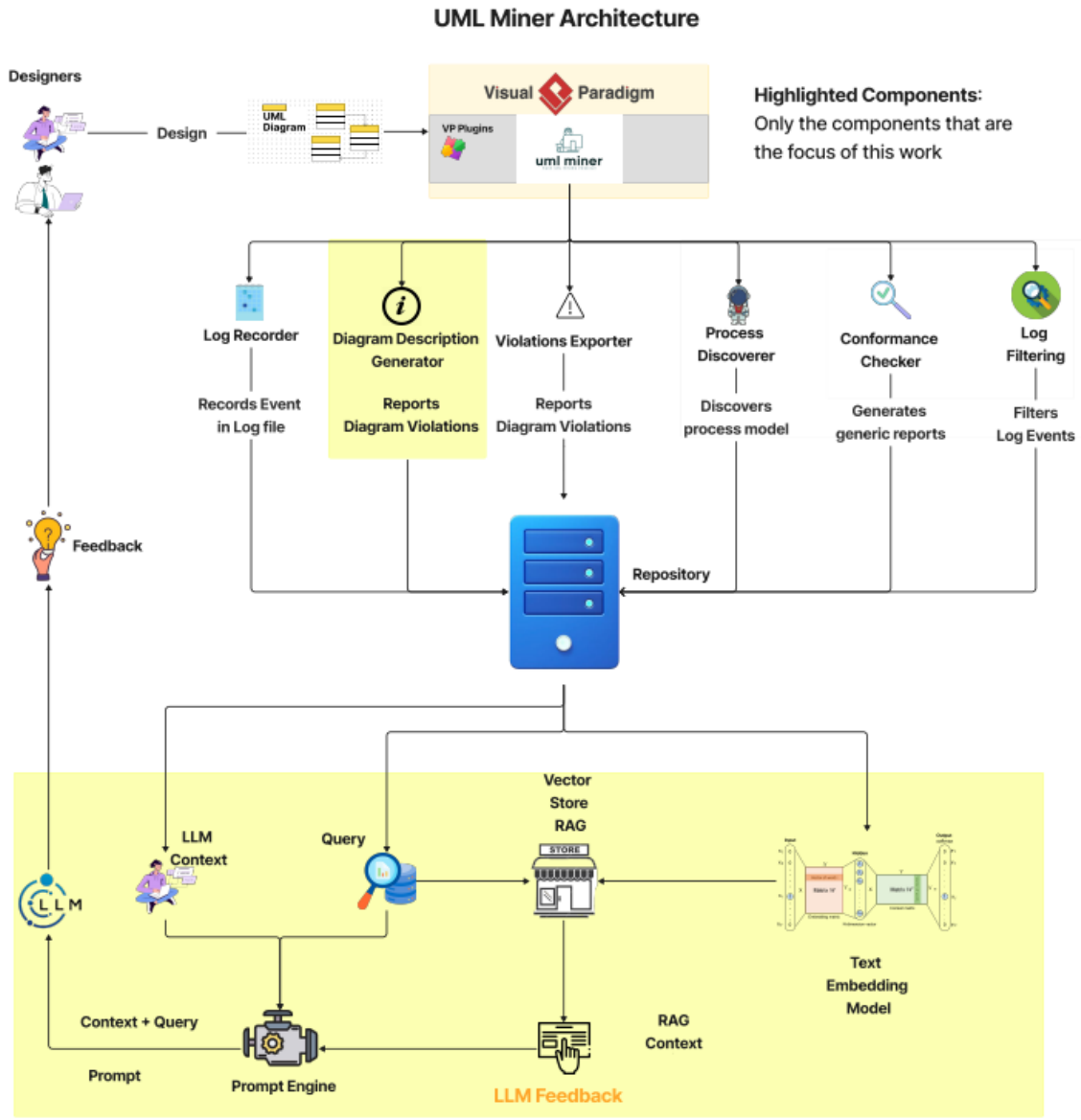


Figure 2.3: UML Miner Architecture from Ardimento et al., 2024

It keeps track of modeling behavior and uses reference diagrams for comparison, generating detailed structural and semantic insights. This tool demonstrates how LLMs can improve UML diagram inspection and refinement with dynamic feedback based on context.

Regarding sequence diagrams, studies show that LLM-generated outputs are highly relevant and accurate, and tend to be more detailed and complete when prompted with more complex requirements (Jahan; Hassan, et al., 2024).

2.3.2 Generating Sequence Diagrams from Natural Language Requirements

The system developed in the research by Jahan; Abad, et al., 2021 also contributes to the field of automatic diagram generation. It uses Natural Language Processing (NLP) along with some rule based decision approaches in order to generate sequence diagrams from input requirements in English (see Figure 2.4).

Generating these diagrams manually from long textual descriptions can be both exhausting and time consuming. However, this approach demonstrates that automating this task can help overcome these challenges, granting promising results in terms of correctness and completeness.

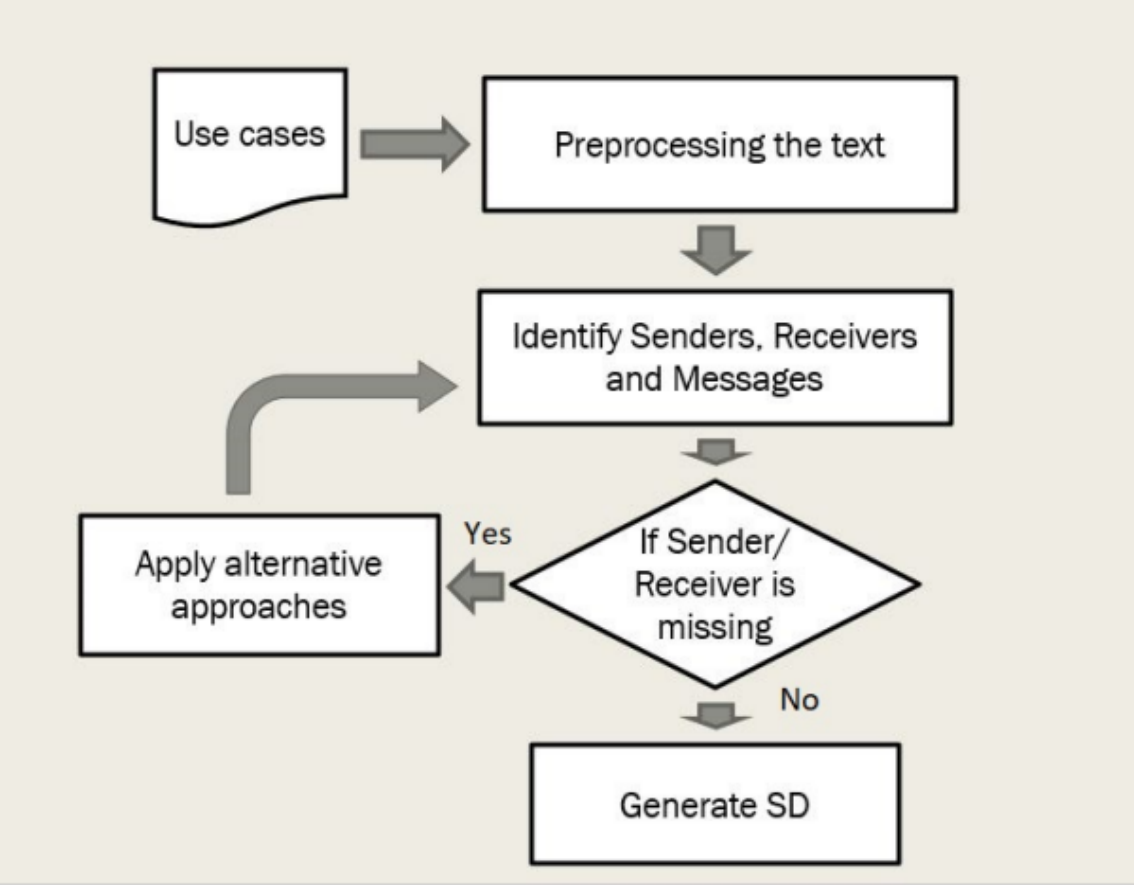


Figure 2.4: Workflow of the proposed tool from Jahan; Abad, et al., 2021

2.3.3 Prose to Prototype

Prose to Prototype is another tool designed to create UML diagrams, both structural and dynamic, from unstructured natural language requirements (Ramackers et al., 2021). The tool uses NLP techniques combined with a "human-in-the-loop" approach to reduce any ambiguity and incompleteness often present in requirements documents (Figure 2.5).

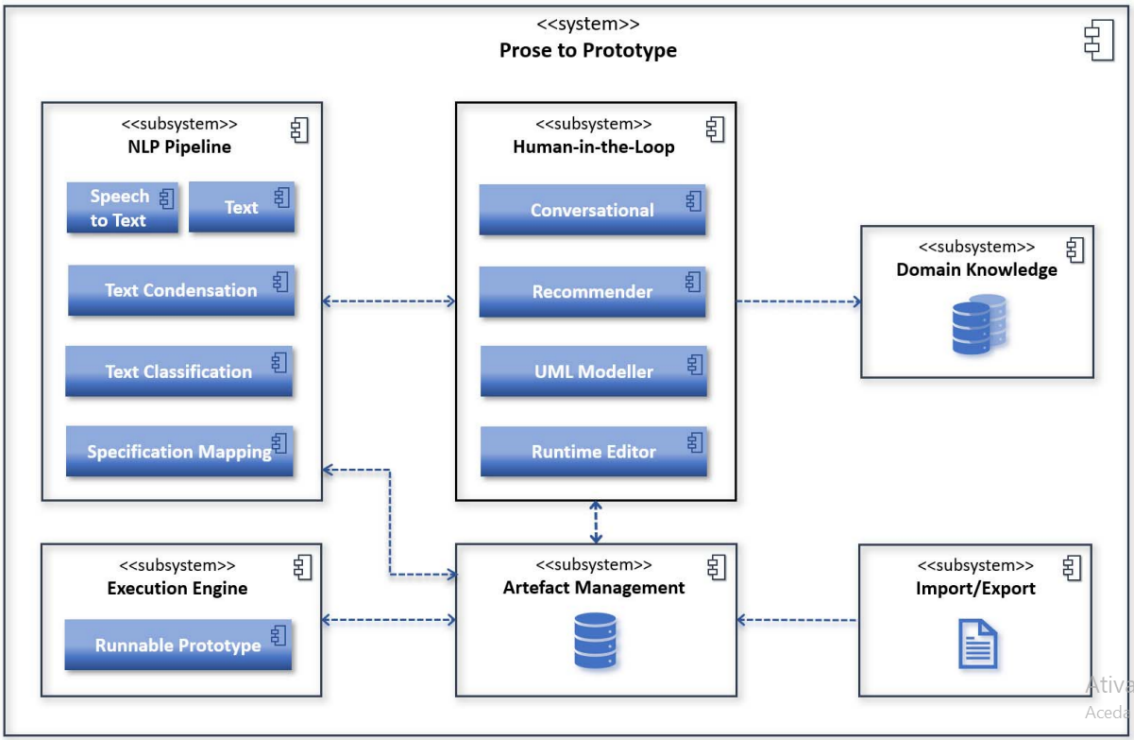


Figure 2.5: Prose to Prototype Architecture from Ramackers et al., 2021

This can be a valuable addition to demonstrate real-world applications of NLP in the automation of the process of generating UML diagrams.

2.3.4 AutoER

AutoER (Figure 2.6) is an educational tool for automatic generation and real-time visualization of UML database design diagrams based on student input (Foss et al., 2022).

As students construct diagrams from structured form-based answers, the system provides immediate graphical feedback on the diagrams that are being created, helping them realize and correct mistakes in real time, which enhances comprehension and engagement. In addition to that, the tool includes automatic grading capabilities by comparing student solutions with predefined reference models.

This dual function of generation and visualization aligns with some of the key goals of this research, mainly in the way it uses automation to support understanding and assessment of software structures and diagrams. In general, AutoER reinforces the effectiveness of dynamic UML rendering to improve usability and comprehension.

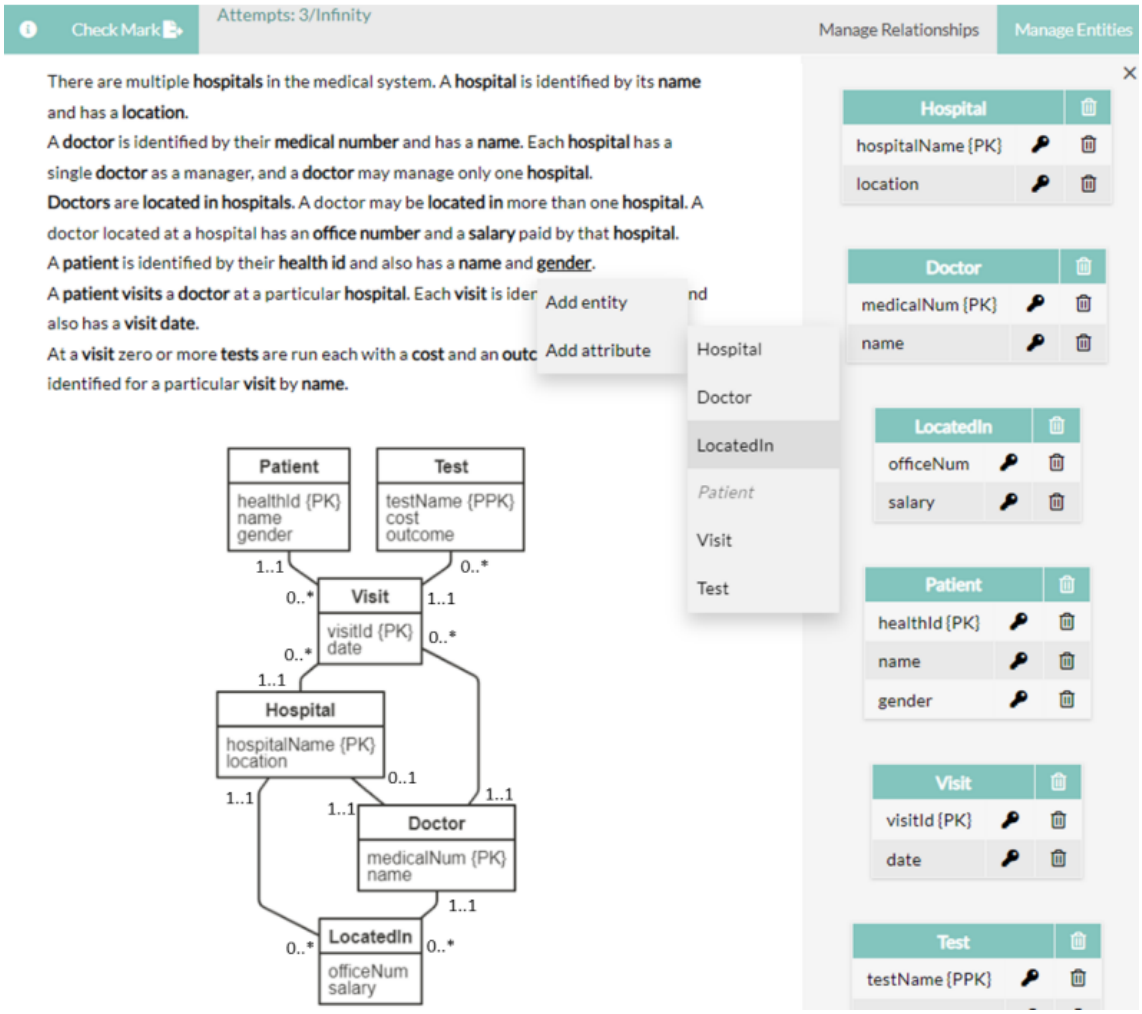


Figure 2.6: AutoER User Interface from Foss et al., 2022

2.3.5 Dynamic Reverse Engineering from Source Code

In Balde et al., 2024, a reverse engineering tool that automatically generates UML class and sequence diagrams from Java source code, using PlantUML, is proposed. Besides diagram generation, it also includes features such as dynamic visualization of diagrams, detailed representation of code structures, error detection in the code, and customization of the generated diagrams. The tool provides both static and dynamic views of the system by generating class diagrams that display the structure, and sequence diagrams that illustrate runtime interactions.

This tool can facilitate software inspection activities by automating the process of diagram generation, saving developers significant time when analyzing systems, and facilitating comprehension of system behaviors and interactions.

2.4 Modeling Activities

This section focuses on tools and practices in Model-Based Software Engineering (MBSE) for creating, editing, and visualizing UML diagrams. It discusses approaches to improve the usability and effectiveness of modeling activities in software development.

2.4.1 Modeling Practices and Visualization

Modeling activities are crucial in Model-Based Software Engineering. Tools that support the creating and editing of UML diagrams should reflect common and expected practices related to graphical modeling, regarding the type of elements used, their frequency, the formatting of their names, their placement, and use of color. Adhering to these informal conventions, increases tool usability and enhances user familiarity and trust.

Empirical studies made in this area are a necessity to identify not only limitations in current notations and tools, but also to surface positive trends that reflect how users naturally structure diagrams. These insights can serve as a foundation for building more intuitive and effective tools. For example, the findings of Savary-Leblanc; Pallec, et al., 2022 show patterns in class diagram usage, such as consistent class shapes, positioning habits, and element symmetry, all of which are relevant for designing tools that aim to support human-centered diagramming.

These findings are particularly relevant in the context of this dissertation, which addresses diagram visualization techniques, where supporting natural modeling tendencies can improve the usability and clarity of the generated diagrams.

2.4.2 BIGUML

BIGUML is an open-source UML modeling tool implemented as a Visual Studio Code extension based on the Graphical Language Server Protocol (GLSP) offering a modeling experience focused on enhanced usability (Metin et al., 2023). It supports manual diagram construction, editing, and visualization through a flexible, modular architecture. It serves as a good example on how extensible modeling platforms can deliver smooth, usable, and intuitive UML editing experiences in browser-based and IDE-based environments, while also enhancing diagram usability through interface features and customization.

2.4.3 AMADEOS

AMADEOS is an online system that automatically generates and visualizes UML class diagrams derived from business process models (Dujlovic et al., 2019). It provides an interactive interface where users can further edit and save the diagrams (see Figure 2.7).

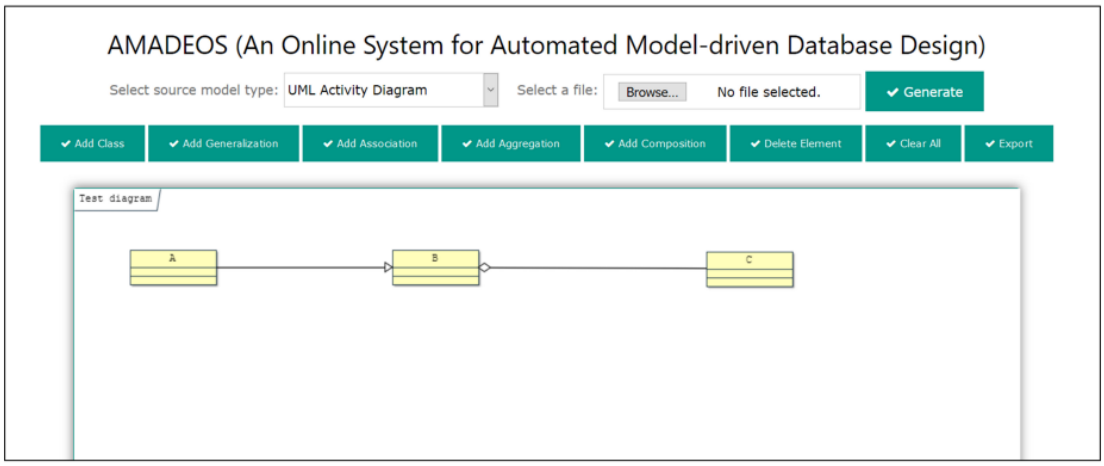


Figure 2.7: AMADEOS Webview from Dujlovic et al., 2019

The system uses refined automatic layout computation techniques to enhance readability and usability, with the overall goal of improving user comprehension during model inspection and analysis.

2.4.4 AnimArch

AnimArch is a tool for software visualization and animation based on the fusion of static and dynamic models. It uses class diagrams to represent the static model and executable UML (xUML) in the form of Object Action Language (OAL) to represent the dynamic model. This multi-layered environment allows the execution of source code to be illustrated visually, animating both the object-oriented structure and the procedural logic of methods (Radosky et al., 2024).

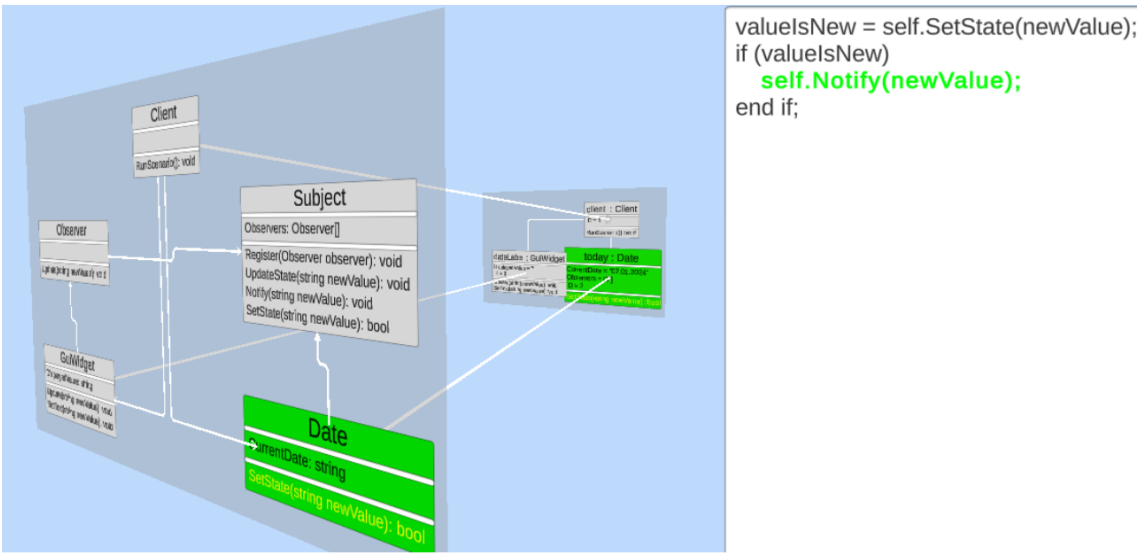


Figure 2.8: AnimArch layered visualization from Radosky et al., 2024

The tool is implemented as a monolithic desktop application using Unity engine for rendering, and its architecture is inspired by the C4 model at a different abstraction level. It's goal is to simplify the transaction from understanding software structure to understanding its implementation, by enabling users to define a class diagram and corresponding source code methods. During execution, the tool visually represents actions such as method calls, object instantiation, and attribute assignment. This layered visualization aims to enhance the software inspection and review process by increasing comprehension of the code through interactive, animated diagrams (see Figure 2.8).

2.4.5 Layered Visualization for Microservices Systems

A visualization concept for microservices-based systems using the Augmented Reality (AR) technique, which merges static and dynamic behaviors into a single centric view, is introduced in Abdelfattah, 2022. This system presents 3 different levels of views (System Level, Service Level, Function Level - Figure 2.9), two view filtration techniques (Node Filter, Path Filter), among other features. Its goal is to reduce search space and improve system navigation. This aligns with the core theme of this research by using layered visualization, spatialization, and filtering techniques to reduce user cognitive load when inspecting and understanding complex systems.

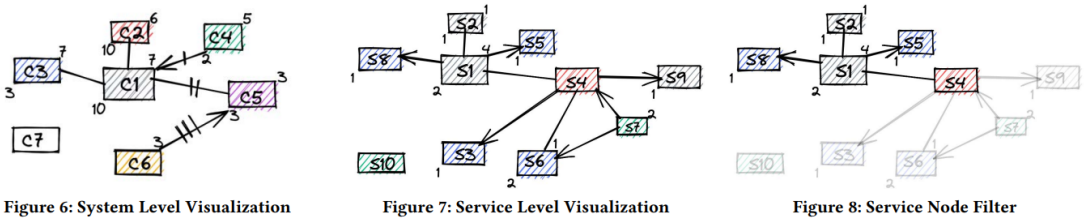


Figure 6: System Level Visualization Figure 7: Service Level Visualization Figure 8: Service Node Filter

Figure 2.9: Multi-Level Visualization for Microservices Systems from Abdelfattah, 2022

2.4.6 Arvisan

Arvisan (Kakkenberg et al., 2024) is an interactive tool for visualizing and analyzing low-code OutSystems architecture landscapes using graph-based representations and user-driven exploration (Figure 2.10). It supports different levels of abstraction through a multi-layered view, allowing users to interactively navigate the system's components. The tool also provides multiple filtering options the user can select in order to tailor the visualizations to better fit his needs.

It relates to the theme of this research by aiming to enhance the visualization of low-code architecture assessment tools and the management of large amounts of information.

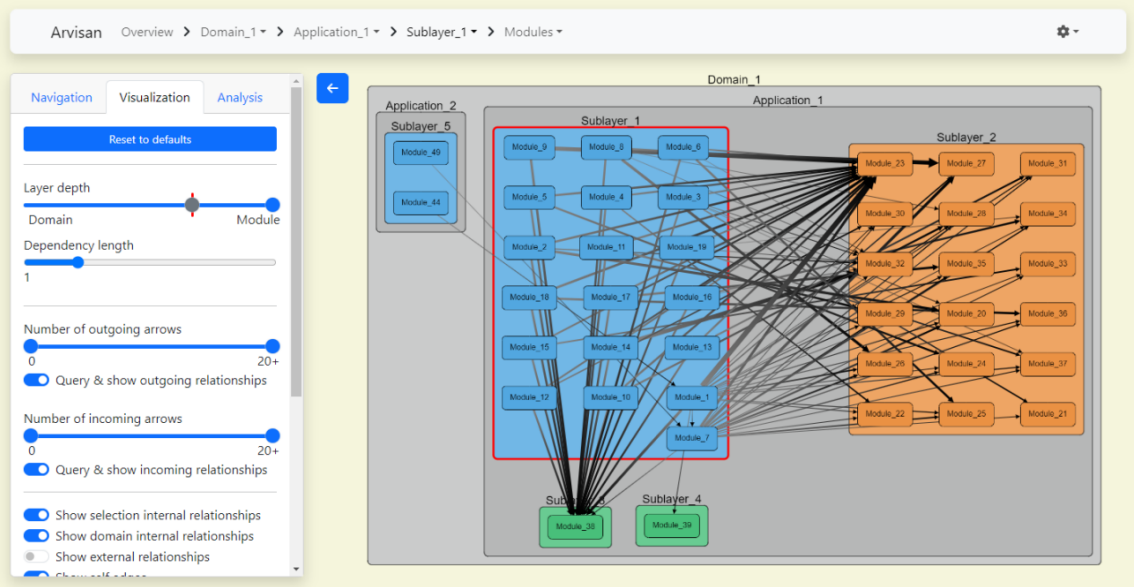


Figure 2.10: Arvisan visualisation of all dependencies of an OutSystems application's end-user sublayer from Kakkenberg et al., 2024

2.4.7 Hunter

Hunter is a tool for visualizing JavaScript applications, enabling users to analyze the dependencies and structure of large projects, without the need to spend too much time navigating the source code (Dias et al., 2020). This visualization is provided through a node-link diagram containing the dependencies between the many components of a system, and a treemap that helps guide programmers when exploring its structure (Figure 2.11).

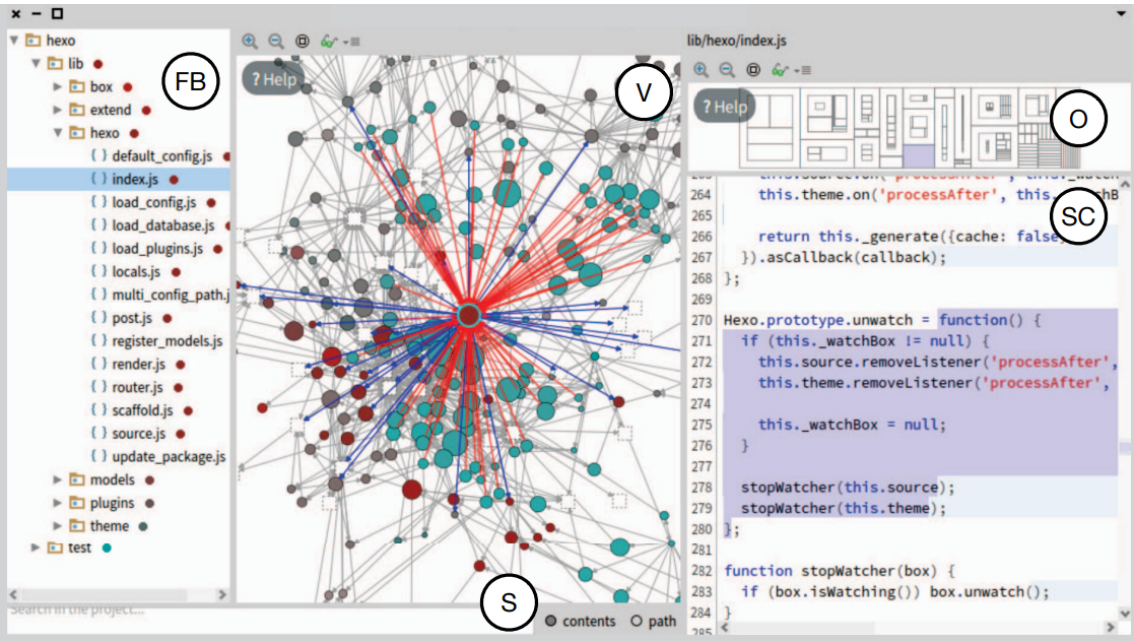


Figure 2.11: Overview of Hunter from Dias et al., 2020

User experiments reveal that this tool helps developers solve software comprehension tasks faster and with higher accuracy, increasing user performance. The results also show great user satisfaction, particularly with the node-link diagram panel, aligning with the broader theme that multi-view visualizations with intuitive and complete interfaces help enhance software comprehension.

2.4.8 Dert

Dert (Diagram Enhanced Review Tool) is a Github plugin that augments traditional code review by introducing UML-like relationship diagrams and artifact maps (Balcı et al., 2021). The tool provides visual information about changes made in pull-requests, especially structural changes such as the introduction of new features, removal of deprecated ones, or alterations in dependencies between services.

These diagrams improve the understanding of the reviewer, allowing him to make more informed decisions about the changes to accept, particularly those that affect several components of a medium to large project. This aligns with the ongoing theme that spatial visualizations increase accuracy, enhance productivity and reduce effort in software inspection and review processes.

2.4.9 OctoBubbles

OctoBubbles is a multiview interactive environment that synchronizes software models and code, allowing users to visualize and interact with both concurrently (Jolak; Le, et al., 2018). It achieves this through a scaling-based approach, that enables interactive, bidirectional, and smooth navigation between models and source code.

This technique helps avoid overcrowded workspaces and visual clutter, reducing cognitive load on users by making navigation between model elements and code more intuitive. It reinforces how visual synchronization and spatial structuring are able to improve comprehension in complex systems.

2.4.10 Interactive Highlighting for UML Class Diagrams

The study Savary-Leblanc; Pallec, 2022, made on facilitating the work of system developers by integrating features that automatically assist navigation within diagrams into existing modeling tools, focuses on the proposal of an automatic highlighting feature for UML class diagrams.

This addition to UML editors has been shown to help users navigate more efficiently, improving their ability to understand what they observe while reducing the required effort. In practice, users respond faster and make fewer errors when answering basic questions, and they report higher satisfaction. This relates to the present dissertation by reinforcing the value of layered and dynamic visual assistance in diagram comprehension.

2.4.11 Interactive Exploration of Sequence Diagrams

The work in Nair et al., 2020 proposes an interactive technique to explore and visualize compact sequence diagrams. This technique uses an enhanced call-tree implementation in Jive to allow users to expand the compacted parts of a diagram on demand through SVG-based hyperlinks (Figure 2.12).

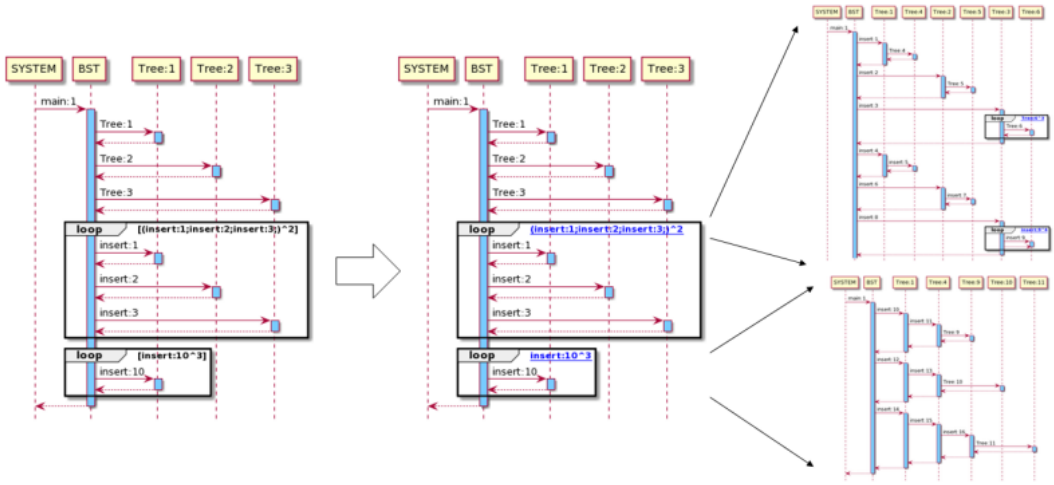


Figure 2.12: Interactive visualization of compact sequence diagrams through hyperlinks from Nair et al., 2020

This exploration is semi-automated, enabling users to choose the depth to which they want to expand a sequence diagram in order to better analyze and understand object interactions. This research highlights how comprehension is greatly improved when diagrams are layered, rather than presented in full detail all at once.

2.4.12 Design Patterns in UML diagrams

Detecting software design patterns is an important aspect of software reverse engineering, as it facilitates maintenance activities by recognizing familiar structures and solutions. To overcome the limitations of traditional static analysis and manual rule-based methods, Zhang et al., 2022 proposes a deep-learning approach to detect design patterns directly from UML class diagram images. This is done by training a deep-learning based classifier to do the image classification task, so then users can input the class diagram they wish to analyze into the model and receive the predicted design pattern. Features like these in existing UML modeling tools can greatly support diagram inspection and improve overall system understanding, aligning with the main goals of this dissertation.

2.5 Summary

The chapter references major advances made in the field of automatic and interactive software visualizations to aid with software comprehension, inspection, and review, mainly those related to the generation and exploration of UML diagrams. These studies introduce revolutionary tools that use 2D spatial representations that have the potential to change diagramming practices and the way we study and understand software, especially when systems become too large and complex to be analyzed based solely on the source code.

Many of the articles are notable for their ability to bring new interactive and innovative solutions to recurring issues related to software inspection and review. Some of the observed trends include the increasing use of dynamic models and abstraction layers to simplify navigation and reduce cognitive load, ultimately supporting program comprehension. These alternative ways to visualize different aspects of a large software system enable developers to quickly and easily identify defects and patterns in the system. Tables 2.1 and 2.2 provide a summary of the tools discussed throughout this chapter.

Table 2.1: Summary of tools and approaches for UML diagram generation

Tool / Work	**Technique / Approach**	**Diagram or View Type**	**Features / Innovations**	**Relevance to This Work**
UML Miner (Ardimento et al., 2024)	RAG-based LLM feedback	UML Class	Natural language critique from LLM using reference diagrams	Demonstrates how LLMs support learning and inspection of UML models
Sequence Diagram Generator (Jahan; Abad, et al., 2021)	NLP + rule-based logic	Sequence	Extracts object interactions from NL input, builds sequence diagrams	Shows how structured sequence models can be derived from English requirements
Prose to Prototype (Ramackers et al., 2021)	NLP + Human-in-the-loop	Class / Activity (planned)	Interactive generation from NL requirements	Combines automation with user input to improve completeness
AutoER (Foss et al., 2022)	Form-based input + grading	UML Class	Real-time feedback, auto-generation, interactive inputs	Supports learning and understanding through dynamic diagrams

Continued on next page

Tool / Work	Technique / Approach	Diagram or View Type	Features / Innovations	Relevance to This Work
Reverse Engineering from source code (Balde et al., 2024)	Reverse engineering with PlantUML	Class / Sequence	Static + dynamic views, customization, code analysis	Highlights reverse engineering from source code

Table 2.2: Summary of tools and approaches for UML diagram visualization

Tool / Work	Technique / Approach	Diagram or View Type	Features / Innovations	Relevance to This Work
BIGUML (Metin et al., 2023)	VS Code plugin + GLSP	Class	Manual modeling, browser and IDE integration	Emphasizes usability in extensible environments
AMADEOS (Dujlovic et al., 2019)	Business process to UML	Class	Auto-layout, editable diagrams, web interface	Improves readability in model inspection
AnimArch (Radosky et al., 2024)	Executable xUML + animation	Class + Behav-ioral	Visualizes structure and runtime via Unity-based layered animation	Enhances comprehension through interactive execution and multi-layered diagrams
Layered Visualization for Microservices (Abdelfattah, 2022)	AR + Multi-level filtering	System / Services / Func-tions	3-layered centric views with node/path filtering	Uses layered spatial visualization to reduce cognitive load in system inspection
Arvisan (Kakkenberg et al., 2024)	Graph-based low-code explorer	Architecture / Compo-nents	Multi-layer abstraction, interactive filtering, OutSystems support	Improves navigation and comprehension of large low-code systems

Continued on next page

Tool / Work	**Technique / Approach**	**Diagram or View Type**	**Features / Innovations**	**Relevance to This Work**
Hunter (Dias et al., 2020)	Node-link + treemap visualization	JavaScript structure	Multi-view layout for dependency analysis and exploration	Boosts comprehension and task performance via intuitive visual decomposition
Dert (Balcı et al., 2021)	UML-like diagrams for Git reviews	Structural / Class-like	Visualizes structural changes in PRs with artifact maps	Enhances review accuracy and understanding of code evolution
OctoBubbles (Jolak; Le, et al., 2018)	Multiview sync + scaling	Models + Code	Bidirectional navigation, reduces clutter	Highlights visual synchronization for comprehension
Auto Highlighting in UML (Savary-Leblanc; Pallec, 2022)	Auto-navigation + guided highlighting	UML Class	Faster responses, fewer errors, improved satisfaction	Reinforces layered visual guidance for better diagram navigation
Interactive Sequence Diagrams (Nair et al., 2020)	Expandable call-trees	Sequence	SVG hyperlinks, detail-on-demand	Layered exploration for selective focus
Design Pattern Detection in UML (Zhang et al., 2022)	Deep learning image classifier	UML Class	Predicts design patterns from UML diagram images	Supports automated inspection and recognition of structural patterns

That being said, there remains considerable margin for improvement and under-explored aspects that could serve as a baseline for future research. Limitations related to scalability, integration with popular IDEs, and broader programming language support are frequently mentioned. In addition to that, many of the available experiments and tool validation processes are based on very simple scenarios, or small user groups, which may limit the generalization of results.

To conclude, this analysis reinforces the need for more versatile and collaborative spatial visualization tools integrated with popular IDEs, such as Visual Studio Code, with personalized interfaces and support for automatic diagram generation from requirements or source code. Features like filtering, layered navigation, and synchronized visualization seem to be very useful to create useful, intuitive, effective, user-centered solutions, particularly in large scale projects.

Chapter 3

InSight UML

With the new context provided by the literature review, it is concluded that the demand for software visualization tools that go beyond traditional, static, and textual representations, and instead use interactive, layered, and spatial visualizations, is increasing. Research highlights the significant value of filtering mechanisms combined with different abstraction layers and interactive views in enhancing the ability to easily comprehend large and complex systems, while also reducing mental load.

As mentioned previously, the main problem addressed by this research is how to improve software inspection and review processes through responsive and interactive spatial visualization techniques and diagram representations, in order to enhance the maintainability of software code during evolution and review.

Main Research Question

> In what ways can responsive, interactive spatial visualizations and diagramming techniques improve software inspection and review processes, particularly the analysis and understanding of source code, to increase maintainability?

In addition to this central question, there are other important ones to investigate:

- What specific aspects do we want to visualize, and how can we visualize them?
- How can these graphical representations actually help us improve maintainability?

Hypothesis

> The adoption of spatialized visualization techniques, including diagram representations, in software inspection and review processes may improve the understanding of the overall structure and flow of the system code, and consequently increase maintainability.

Essentially, the goal of this research is to investigate in what ways software can be visualized in order to improve understandability and comprehension, thus ensuring system maintainability

— particularly in complex systems and large teams. To accomplish this, validate the hypothesis, and answer the research questions, a visualization tool was developed and analyzed.

This chapter will describe the process of creating the prototype, from the initial vision and requirements planning to tool development, designed to address some of the gaps identified in the literature review regarding UML diagram creation, inspection, and editing.

3.1 Vision

InSight UML is a VSCode extension that extends the features of an existing tool called vscode-mermAId. It aids in the visualization of class and sequence UML diagrams, by generating diagrams directly from the Java code using AI (see Figure 3.1). By doing this, it allows for a better understanding and view of the overall project, especially in very complex systems.

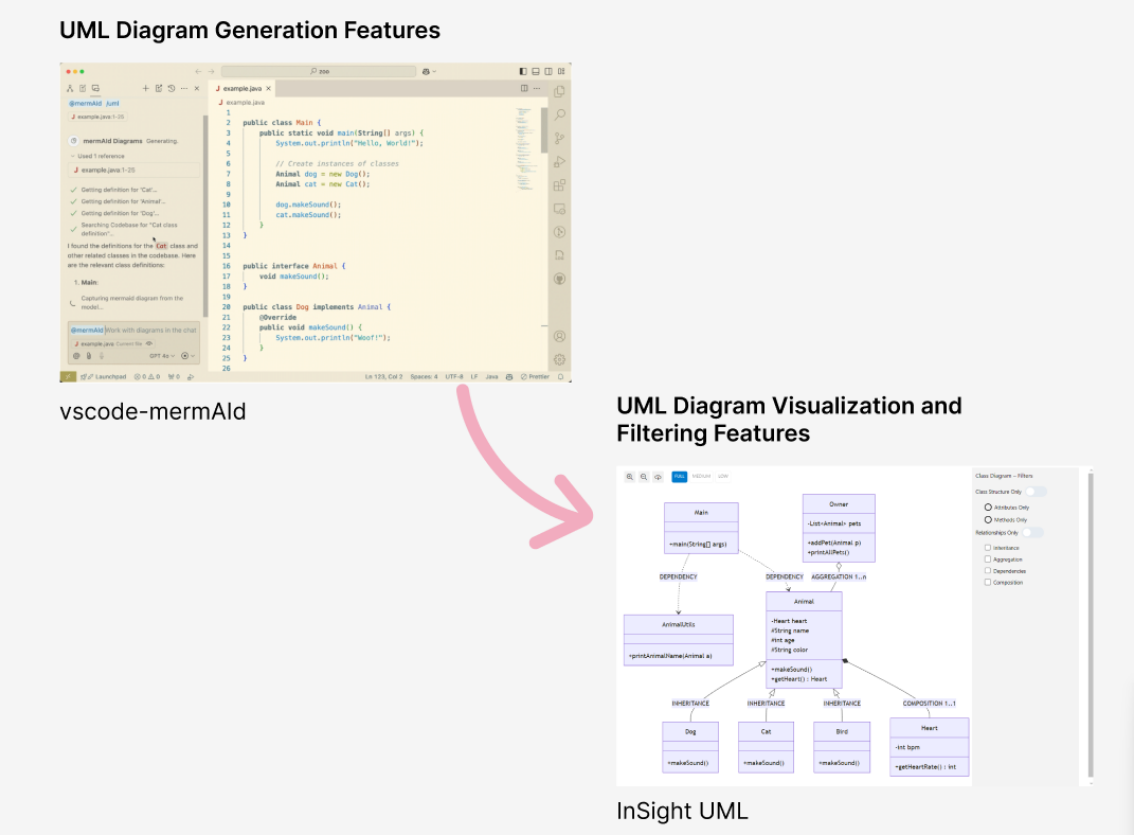


Figure 3.1: vscode-mermAId to InsightUML

The widespread and standardized use of UML in software engineering guides the choice to make it the foundation of the tool. The decision to focus specifically on UML class and sequence diagrams was based on their relevance and frequency in software inspection and comprehension tasks.

Class diagrams offer a static view of the architecture of the system, showing the structure of components, attributes, and their relationships with each other, which is essential to identify

design flaws and understand modular boundaries. On the other hand, sequence diagrams provide a dynamic perspective by illustrating object interactions over time, making them very valuable for tracing execution flows and debugging behavior.

These two types are some of the most widely used and understood in both academic and professional contexts, ensuring that the developed tool addresses real, practical needs without overwhelming users with less commonly used diagram types such as state or activity diagrams, which were not prioritized at the prototype stage.

3.2 Requirements

The first step was to make a list of requirements that the tool needed to meet to be able to fulfill this vision and to effectively be useful in providing an improved visualization technique for UML diagrams.

The base idea for the extension required it to be able to generate both class and sequence UML diagrams solely from the class code, while also displaying them in a webview positioned side-by-side with the source code and allowing the user to select different options as to how and what in the diagrams is exhibited. This spatial representation of the code is meant to help in the software inspection and review process.

3.2.1 Generate UML Diagrams From Java Code

The first main activity includes the generation of UML diagrams from the code. Figure 3.2 shows the user story map for this activity, including the user journey steps, the sequential steps a user will encounter to complete the activity, and the detailed features contained in each step. The blue features represent the minimum viable product features and were also the ones implemented in this prototype. The purple boxes indicate the planned "MVP 2.0" enhancements, which have not yet been implemented. The pink boxes describe aspirational “dream” features and nice-to-have ideas that remain on the wishlist for future iterations.

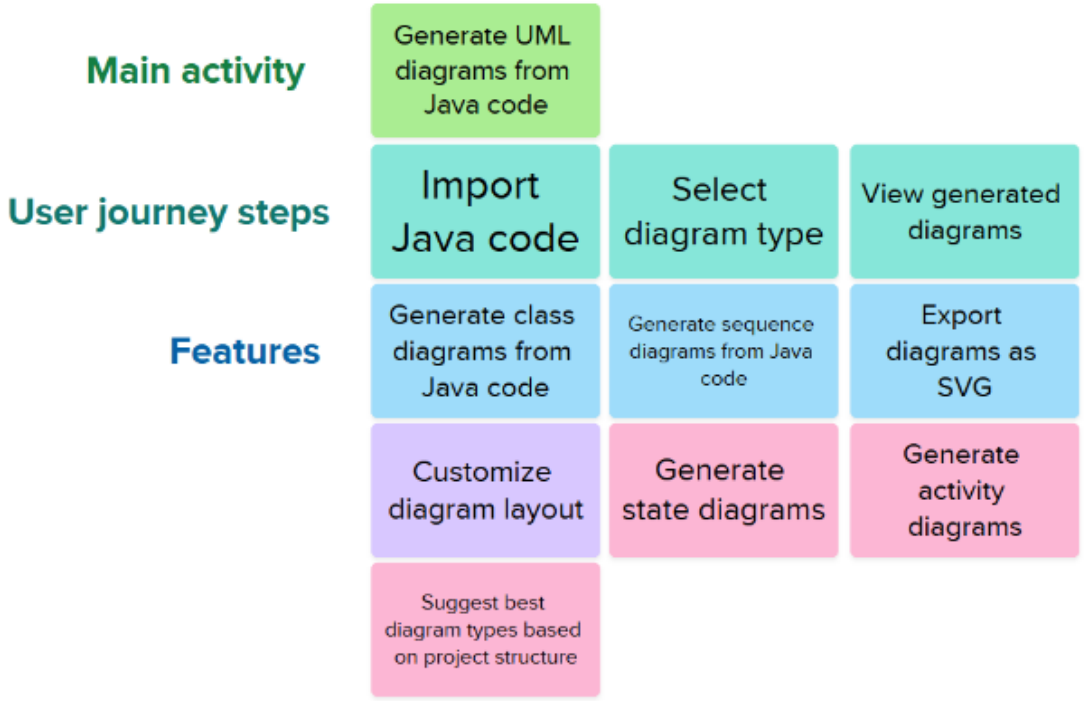


Figure 3.2: User Story Map - UML Generation

3.2.2 Navigate And Interact With UML Diagrams

The second main activity relates to the navigation and interaction the user is able to have with the generated diagrams in order to use them to improve their insight into the project flow and properties. Figure 3.3 shows the user story map for this activity, with the same structure as previously.

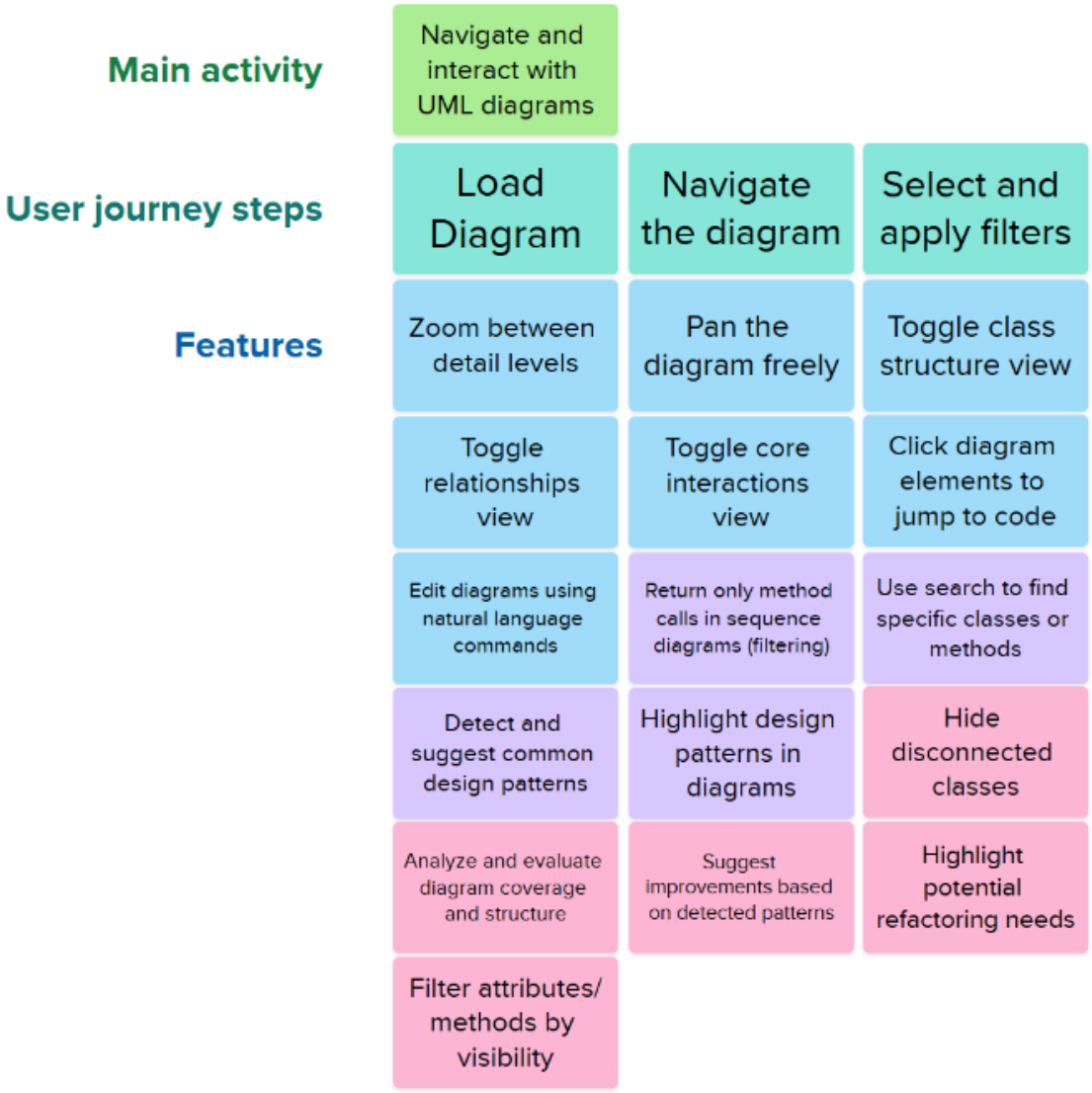


Figure 3.3: User Story Map - UML Navigation and Interaction

3.2.3 Augmented Reality Visualization And Real-Time Collaboration

As for future work, there is a goal of integrating the tool with AR and real-time collaboration features, in order to have more immersive and beneficial visualizations and to enable interaction with team members aligning everyone's thinking and improving shared understanding. Figure 3.4 represents the steps and features that would be included in this activity.

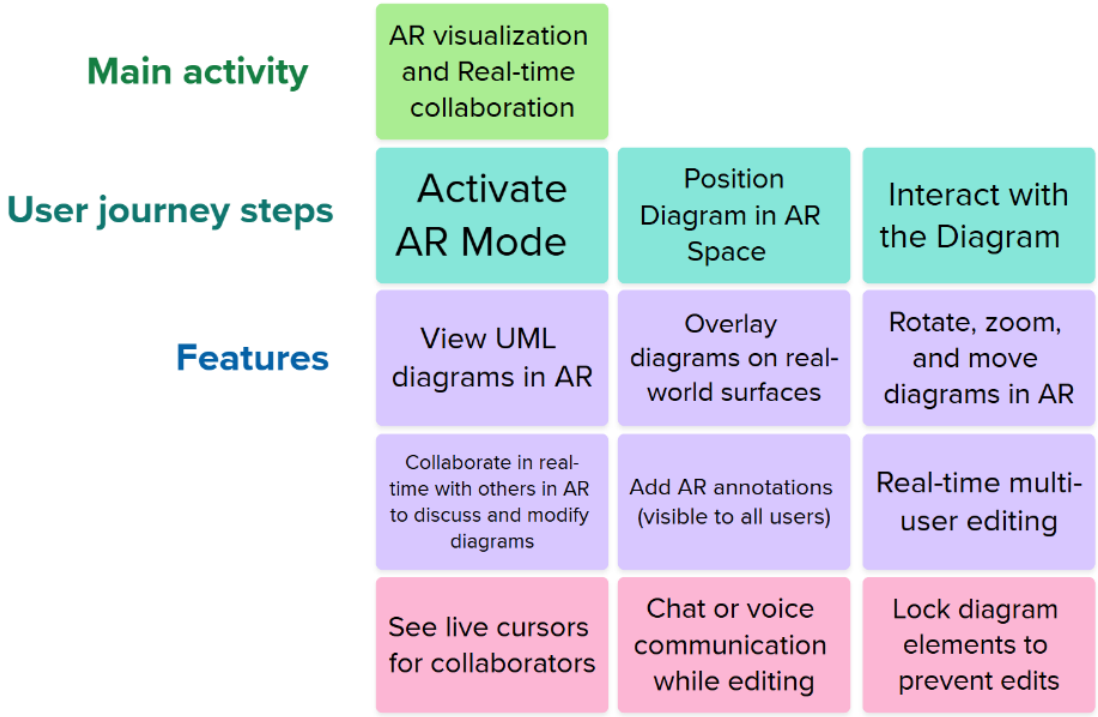


Figure 3.4: User Story Map - AR and Collaboration

3.3 Logical and Technological Architecture

This section presents both the logical and technological architecture of the InSight UML tool, detailing its high-level structure and implementation layers.

Eclipse-based tools were considered in the initial stage of development due to their integration with Java and UML. However, they ended up being discarded for many reasons. Some of them include the fact that their architecture made customization and extension difficult, and Eclipse is not used as much for code development as editors like VSCode. There were also issues with the available Eclipse plugins lacking active maintenance, which was another reason to discard them and choose a VSCode plugin instead.

As a result, InSight UML is made up of three main high-level components: the VSCode Extension Host, the Mermaid Syntax Generator (using Copilot), and the Webview Visualization Engine. As mentioned previously, the extension listens for slash commands and forwards them along with the entire active Java file to Github Copilot. Then, it receives back the Mermaid.js code and renders it in a webview with interactive filtering controls (see Figure 3.5).

3.3.1 Command and Data Flow

Figure 3.6 contains a sequence diagram that describes the flow of the extension since the moment the user enters a slash command or switches on a filter. The extension captures the respective user action, extracts the active Java file text, and uses the Copilot API to translate the code into

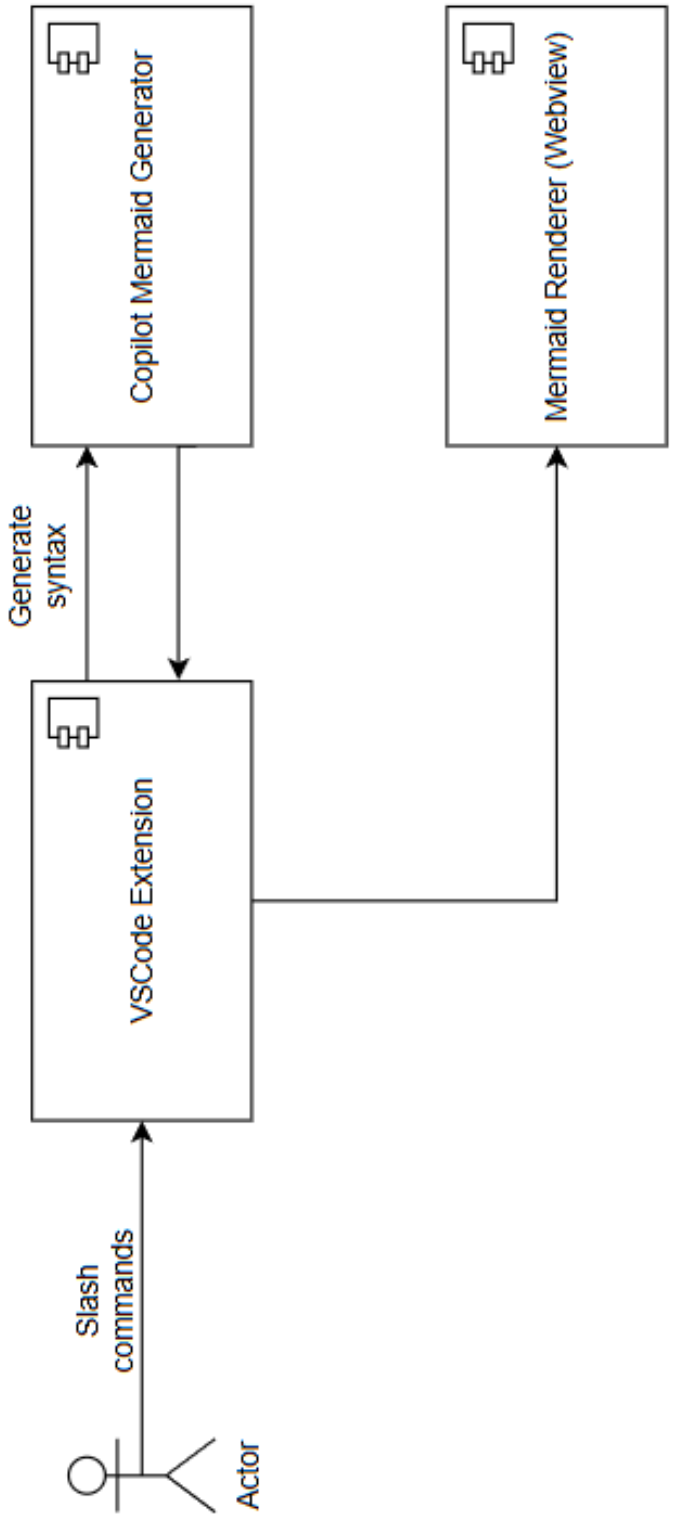


Figure 3.5: InSight UML Architecture Diagram (Logical View)

Mermaid DSL. The renderer module then takes the generated Mermaid syntax and renders the correspondent SVG which is dynamically updated based on the filter commands present in the webview.

3.3.2 Filtering and Visualization Layer

As noted above, the "layers" concept was borrowed from geographic information applications, such as the Google Earth model (Figure 3.8). In this case, the user is able to navigate the different layers to reveal just the information he needs either with zoom buttons or with specific toggles that work according to the following client-side processing pipeline:

- When the webview first loads, the tool captures the unmodified original Mermaid text and immediately generates a small map of pre-filtered strings for each filtering/"layer" case, stored in `window.diagramVariants`. These are obtained by applying a set of regular expression transforms in the original syntax to remove different things based on the level selected.
- The tool achieves a clear separation of concerns by building every filtered variant just once, up front, in the browser, while rendering is handled offline by Mermaid itself. In this way, when a user switches layers, the tool simply swaps the respective DSL string and calls `mermaid.contentLoaded()` again.
- Because all filtering is just a few string replace calls, there is no extra network or extension-host roundtrip. It is basically just fast, in-memory regex on the already-loaded DSL and an SVG swap. The result is near-instant visual feedback even on very large diagrams.

3.3.3 Technological Architecture: Extension Module Breakdown

Essentially, InSight UML is a VSCode entension plus a small in-browser JavaScript "app" that expands the features of vscode-mermAId. While mermAId focuses on diagram generation features, InSight UML integrates visualization and filtering features with those. It can be broken down into these sections (Figure 3.7):

- The VSCode Extension in TypeScript that contains more essentially `extension.ts`, the entry point for the slash commands that when receiving a command invokes `DiagramEditorPanel`, and `DiagramEditorPanel.ts`, that manages a WebviewPanel instance, listening for messages back from the webview, and builds two variants of HTML based on the type of diagram selected, so different filtering options appear (connecting to local scripts like `main.js` and `main2.js`, CSS style files, and to the raw Mermaid DSL string, `window.raw` and the pre-computed `diagramVariants`). It also includes some TypeScript helpers like `commands.ts`, that wraps all user-facing commands in one place, and `mermaidHelpers.ts` for parsing/templating Mermaid blocks.

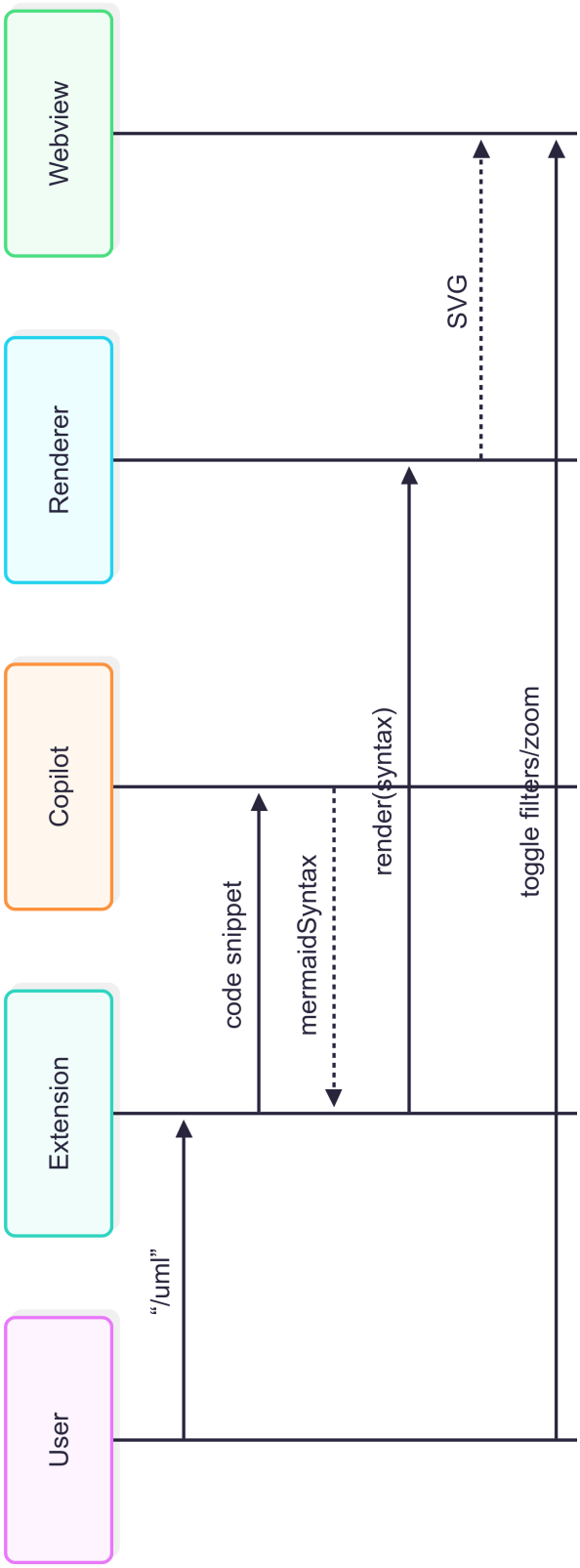


Figure 3.6: InSight UML Sequence Diagram (Runtime Interaction)

- The Webview "App" made up mainly of `main.js` (relative to the class diagram view) and `main2.js` (for the sequence diagram view) in JavaScript. These scripts handle the pan-around and zoom features, with detail-level cycling, toggle filters, and the export button. They access `window.diagramVariants`, swap in the DSL that corresponds to the user request, and reload the mermaid content. The HTML of the application is built by `diagramEditorPanel.getHtmlForWebview`, which defines the toolbar buttons and filter toggles.

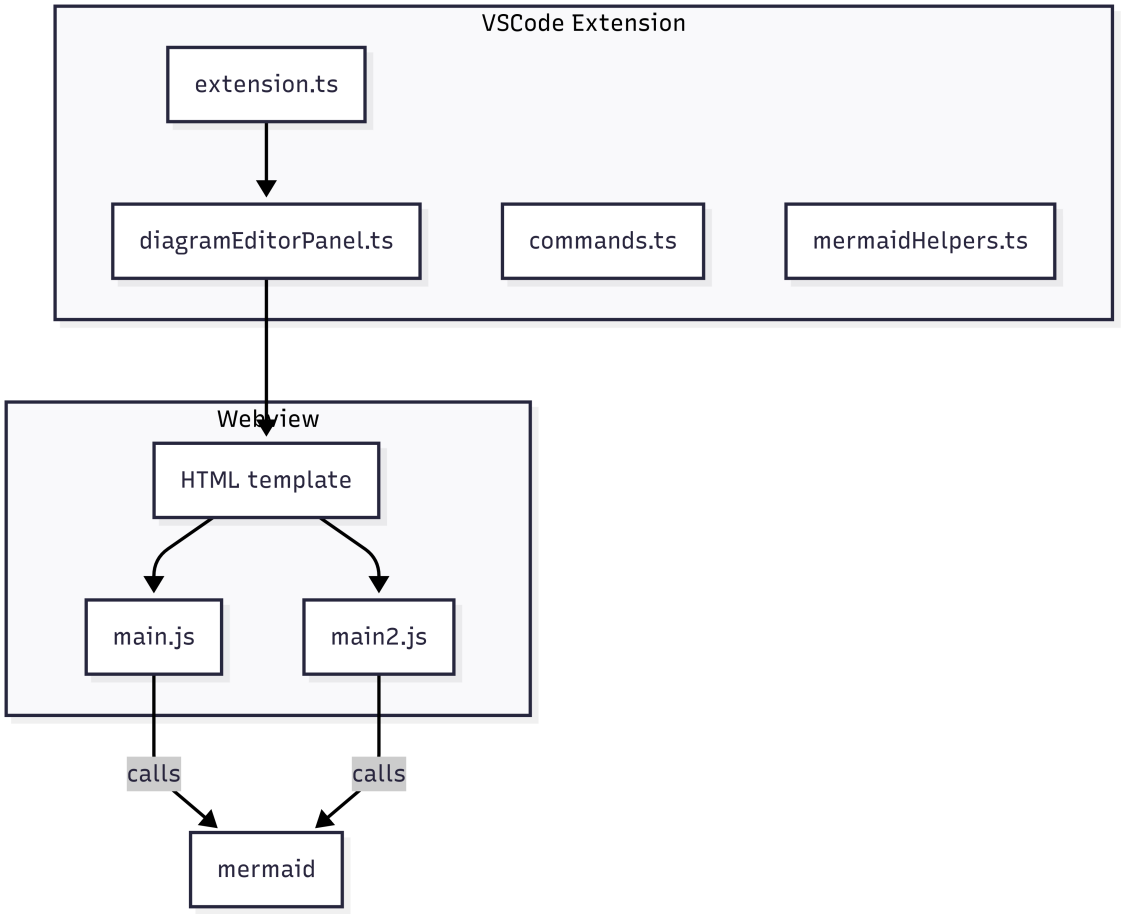


Figure 3.7: VSCode Extension Module Overview and Webview Interactions

Basically, the Copilot integration and diagram generation, and Mermaid syntax rendering were already handled by the vscode-mermAId extension, which opened a webview displaying the generated diagram. InSight UML builds on top of this by adding an interactive layer, splitting the existing webview into two views, one for class diagrams and another for sequence diagram, and creates dynamic filtering controls, layered zoom with detail cycling, and visual toggles. This is done through variant management (`diagramVariants`) and a customized webview interface.

3.4 Diagram Creation and Visualization

This section will describe the approaches, tools, and techniques used for the diagram generation and visualization features, while also describing the interface and the filtering options present in the webview.

3.4.1 Preliminary Approach: Visual Paradigm

In the initial stages of this research, while exploring tools capable of automatically generating UML class and sequence diagrams from Java source code, Visual Paradigm was heavily considered.

Visual Paradigm is a web-based diagramming and modeling tool designed for collaborative creation of a wide range of diagrams and models (Visual Paradigm, [n.d.]). It includes many features to support project management and improve productivity, and it offers robust UML support. Specifically, it can generate UML diagrams from Java source code through its reverse engineering functionality.

The goal was to create a VSCode extension/plugin that would call Visual Paradigm's `CodeSyncManager` API, responsible for round-trip engineering, to generate diagrams within VSCode and display them in a webview with multiple visualization and filtering options. However, due to licensing constraints and limitations in accessing Visual Paradigm's API, this approach was abandoned, and the search for alternative automated UML generation tools continued.

3.4.2 Mermaid UML

As the search for alternative solutions continued, an open-source extension more compatible with the project's objectives and technical constraints was identified.

The diagram creation part of the tool originated in another VSCode extension, mermAId (Microsoft, 2025). This tool uses Github Copilot along with a "@mermaid" chat participant to generate a diagram based on the current open editor. Table 3.1 describes the four predefined chat commands it includes: "/help", "/uml", "/sequence", and "/iterate".

Command	Description
/help	Tutorial on how to use the tool
/uml	Create a UML class diagram
/sequence	Create a UML sequence diagram
/iterate	Make changes on an existing diagram using Natural Language

Table 3.1: vscode-mermAId commands

When the user requests a UML diagram typing either "/uml" or "/sequence", GitHub Copilot generates the Mermaid.js syntax that corresponds to the type of diagram selected and the file active in the editor (*Mermaid – JavaScript-based Diagramming and Charting Tool*, 2020). The extension

then injects that syntax into a Markdown code block, invokes a Mermaid renderer, and displays the resulting diagram in a VSCode webview.

To create InSightUML, some visualization options were added to the existing webview provided by the vscode-mermAId tool.

3.4.3 Google Earth

One example that inspired the layered visualization in InSight UML is Google Earth (Google, 2019), among others.

Google Earth is a virtual globe that allows users to explore geographic data in 3D, providing different layers that the user can toggle on and off based on what information he wishes to be displayed. These layers add context to the base satellite imagery, allowing users to see additional details such as roads, buildings, and weather patterns. The way Google Earth is set up served as a guide for how the InSight UML tool could allow users to see the different components of the diagrams in different levels, filtering what they want to focus on so there is less visual clutter, especially in very big and complex systems. Figure 3.8 portrays a view of the Google Earth interface.

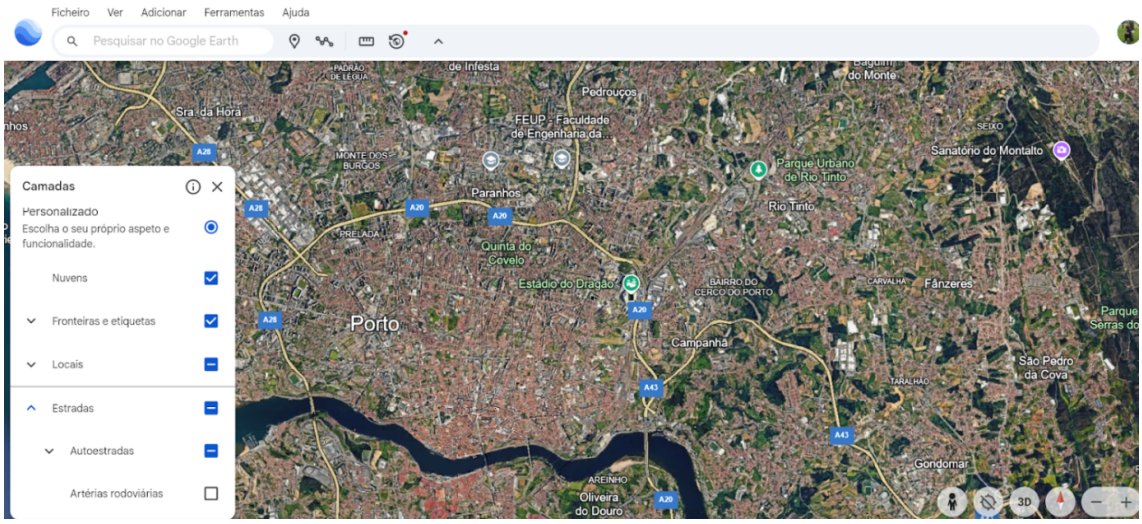


Figure 3.8: Google Earth

Table 3.2 presents some of the options for the layers that a user can toggle on and off in Google Earth, in order to visualize many different levels.

Layer	Description
Borders and Labels	Shows country and city boundaries with place names.
Roads	Displays highways, streets, and pathways.
3D Buildings and Terrain	Adds elevation and 3D models of buildings and landscapes.
Weather and Climate	Shows cloud cover, storms, and some temperature data.
Historical Imagery	Lets you view past satellite images to see changes over time.
Street View	Provides a ground-level view with panoramic images.
Places of Interest	Highlights landmarks, businesses, parks, and more.
User Content	Includes photos, reviews, and custom data added by users.

Table 3.2: Main Layers in Google Earth

Inside each main layer, there can be separate sub-options. For example, the "Roads" layer may show only highways or include smaller streets, and the "Weather" layer might have options for clouds or temperature.

This structure acted as a base for the filtering feature in the developed visualization tool with different filtering options tailored separately for class diagrams and sequence diagrams, ensuring a clear and structured view based on user needs.

3.4.4 Layered Visualization

With different layers showing different details, InSight UML presents three main detail levels ("FULL", "MEDIUM" and "LOW"), that hide different parts of a class and sequence diagram, and a filter toolbox specific for each diagram type where the user can toggle on/off what he desires to be presented in a diagram. This layering technique can be really useful in order to reduce visual clutter in really complex systems and to enable the user to look exactly at what he needs and cares about.

Figure 3.9 shows an example view of the entirety of the InSight UML extension inside the VSCode IDE.

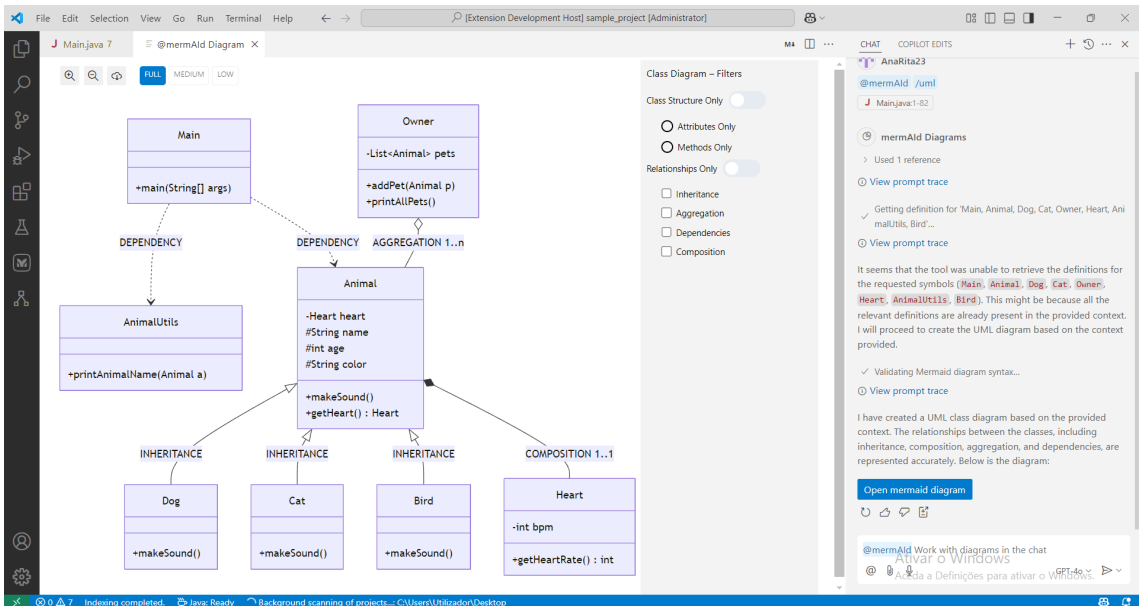


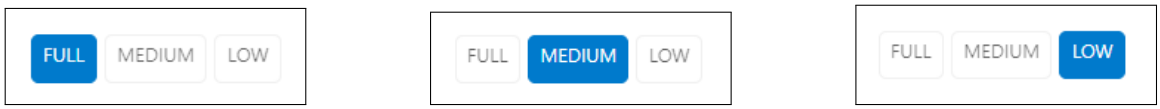
Figure 3.9: InSight UML

3.4.5 Toolbar and Controls

For both types of diagram, there is a similar structure regarding controls.

Figure 3.10: InSight UML Webview Controls

On the top left of the webview (Figure 3.10), there is a section with zoom in/out buttons, that allow the user to zoom out until the lowest detail level ("LOW") or to zoom in until the highest ("FULL"). When at any of these limits, if the user keeps zooming in or out the tool performs normal zoom, making the diagram appear closer or further away, respectively. There are also some boxes that tell the user the current detail level of the diagram, highlighting the correspondent one. In Figure 3.11 there are the possible details levels highlighted.



(a) FULL Highlighted (b) MEDIUM Highlighted (c) LOW Highlighted

Figure 3.11: Detail Box Highlight States

In this section, there is also an Export button the user can click to export the current open diagram into SVG and select where he wants to save it (see Figure 3.12 and 3.13. This way, users are able to export any level of the original diagram if they want to.

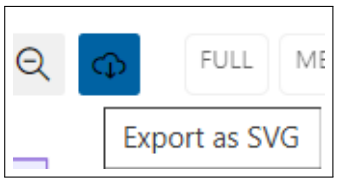


Figure 3.12: Export Button

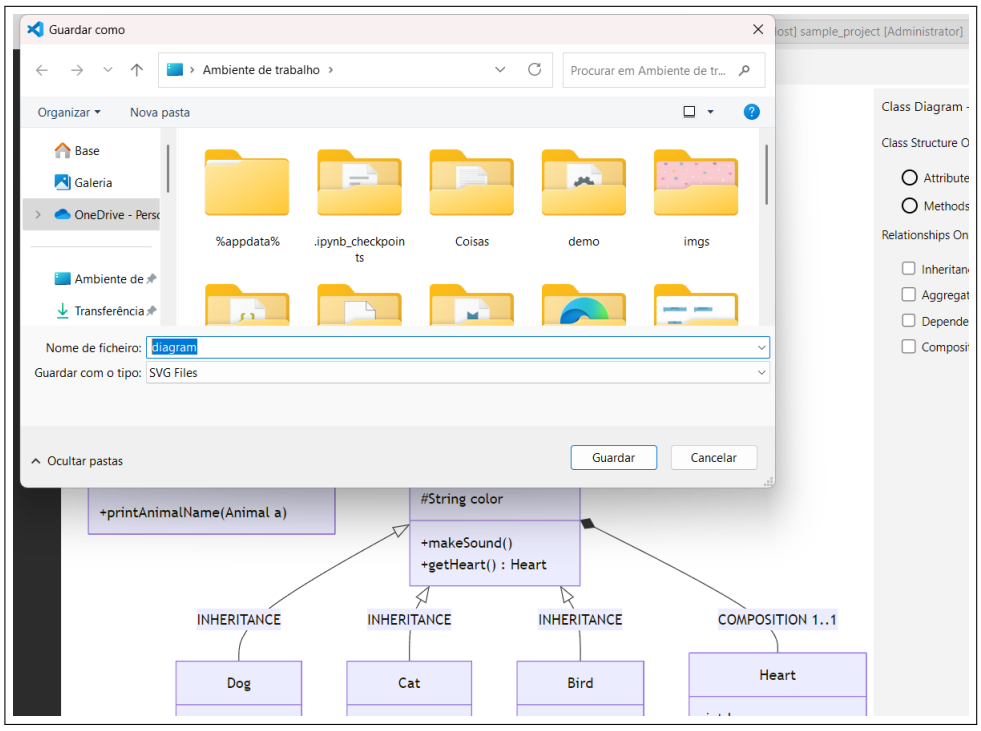
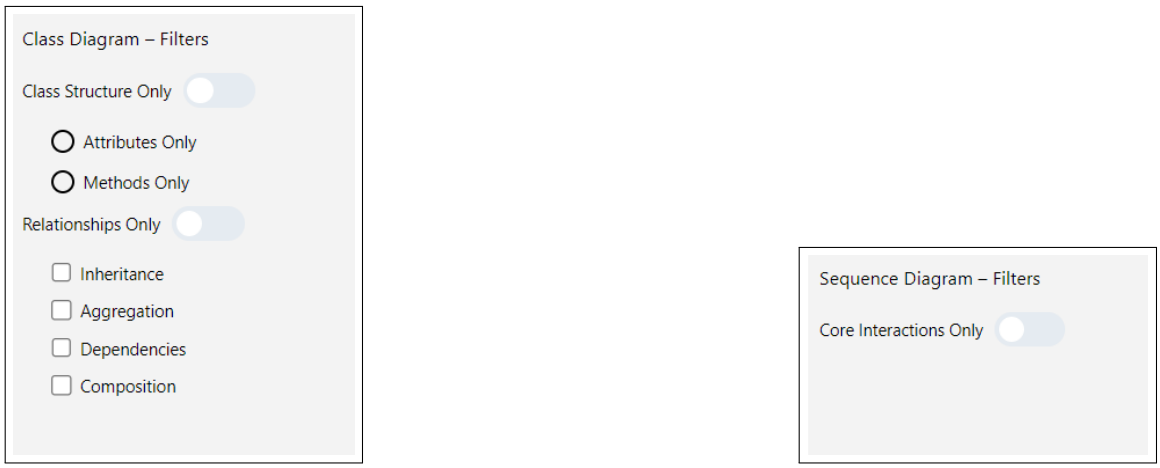


Figure 3.13: Export Diagram Options

Besides this controls section, on the right panel of the webview there is a tool box with different toggle/layer options for class and sequence diagrams (Figure 3.14) . How each layer works will be explained in the next sections.



(a) Class Diagram Toolbox

(b) Sequence Diagram Toolbox

Figure 3.14: Filtering Toolboxes

Finally, users can click and drag the diagram to reposition it as needed. Underlined class names, attributes, or methods are clickable, and clicking one opens the corresponding Java file at the line of the class where that element is defined. This is possible since class nodes in the rendered diagram are wired up to VSCode callback via Mermaid's built-in `click` directives. At the end of the Mermaid DSL generated, click lines are appended in the form `click ClassName call linkCallback("path/to/File.java#L<lineNumber>")`, as it can be seen in Figure 3.15.

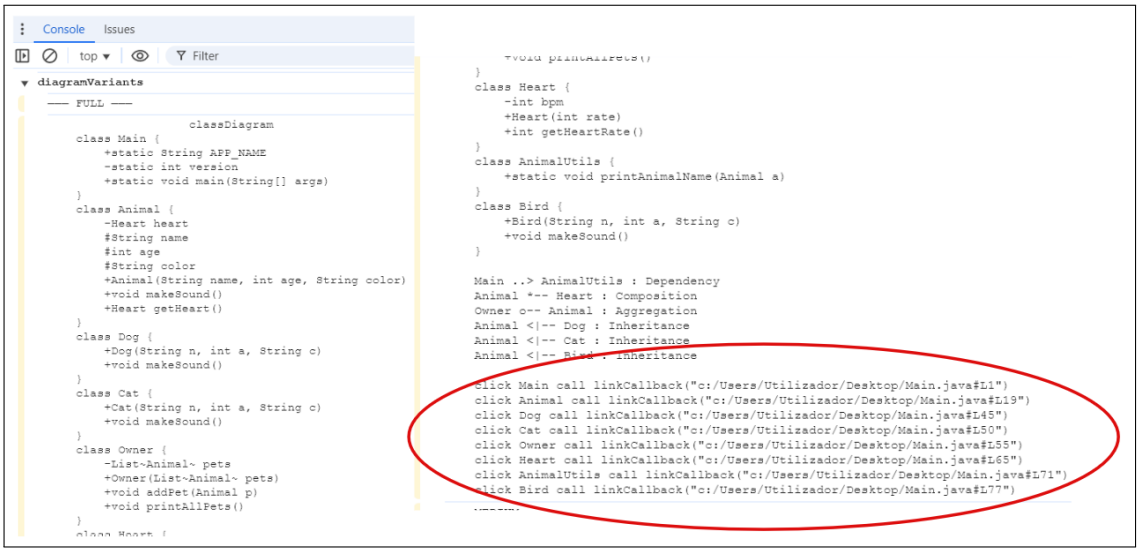


Figure 3.15: Raw Mermaid DSL including 'click' directives for source-code navigation

3.4.6 Detail Levels

The diagram detail levels are controlled using the zoom buttons, as explained previously. The diagram is first loaded at "FULL" level with everything visible, Then, for each click in the zoom

out button it goes from "FULL" to "MEDIUM" and then to "LOW", and vice versa. For each detail level, there is a different DSL generated, according to the specifications in the following tables, Table 3.3 and Table 3.4, with separate definitions of each level for each diagram type.

Level	Removed Content
FULL	Nothing
MEDIUM	All class attributes and methods
LOW	All class attributes and methods Relationship lines

Table 3.3: Class Diagram Detail Levels and Removed Content

Level	Removed Content
FULL	Nothing
MEDIUM	All note/comment lines
LOW	All note/comment lines Loop/alt/par sections

Table 3.4: Sequence Diagram Detail Levels and Removed Content

In Figures 3.16, 3.17 and 3.18, it is portrayed a sequence of the three main detail levels for UML class diagrams.

Following the same logic, in Figures 3.19, 3.20 and 3.21, an example of the UML sequence diagrams detail levels is depicted.

3.4.7 Class Diagram Toggles

The toolbox specific to Class Diagrams, shown in Figure 3.14 - (a), has two main toggles. When selecting the "Class Structure Only" toggle, all relationships are removed from the diagram, so the user can focus only on class definitions like attributes and methods (see Figure 3.22).

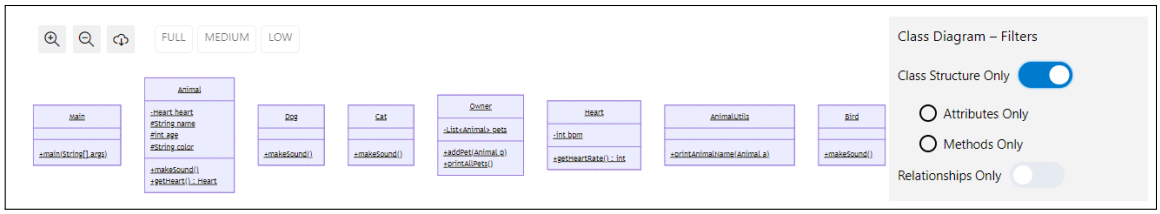


Figure 3.22: UML Class Diagram - Class Structure Only Toggle

Inside this toggle, there are also two sub-options where the user can choose to display either only class names and attributes (see Figure 3.23), or class names and methods (Figure 3.24).

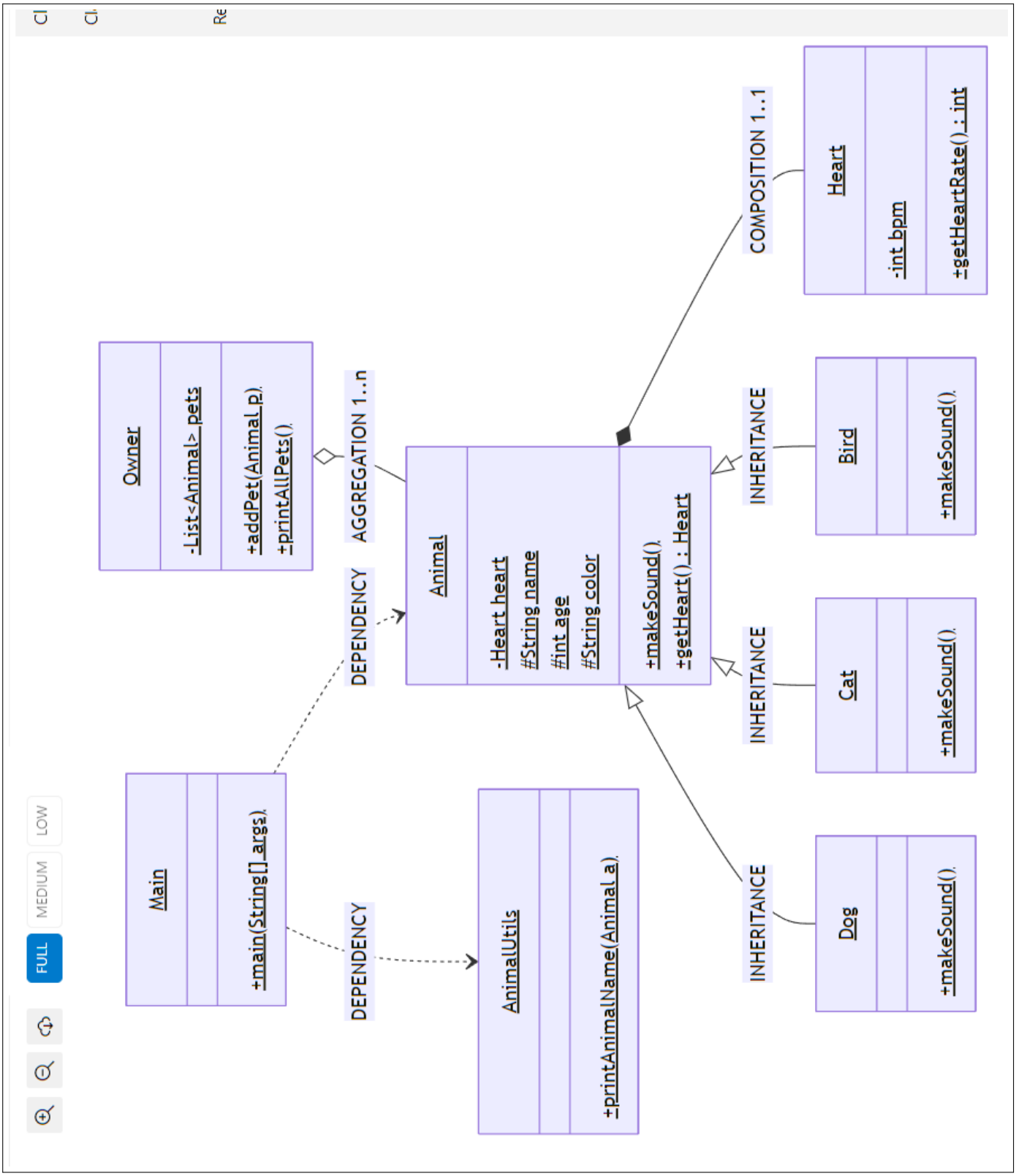


Figure 3.16: UML Class Diagram - Detail Level: "FULL"

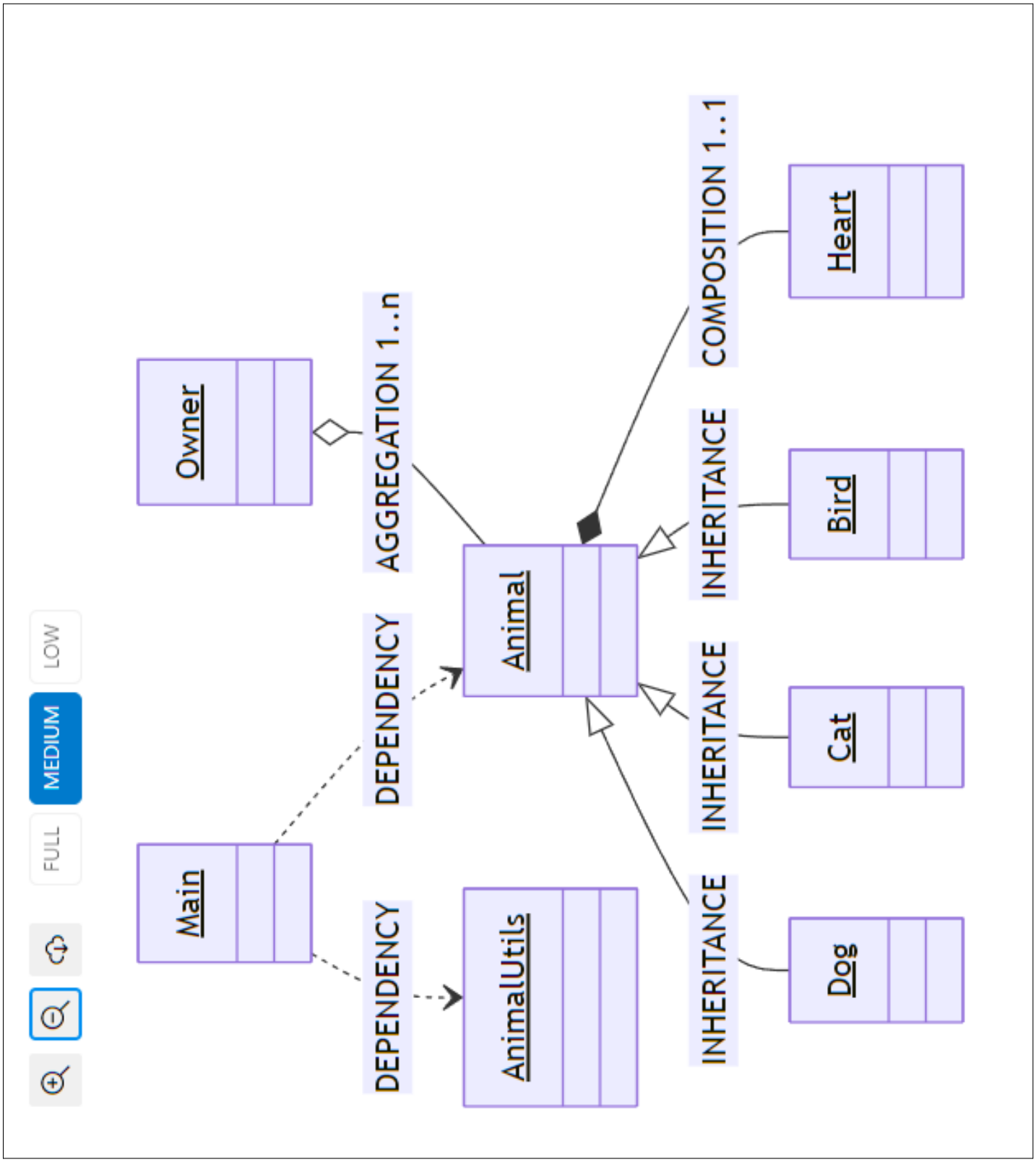


Figure 3.17: UML Class Diagram - Detail Level: "MEDIUM"

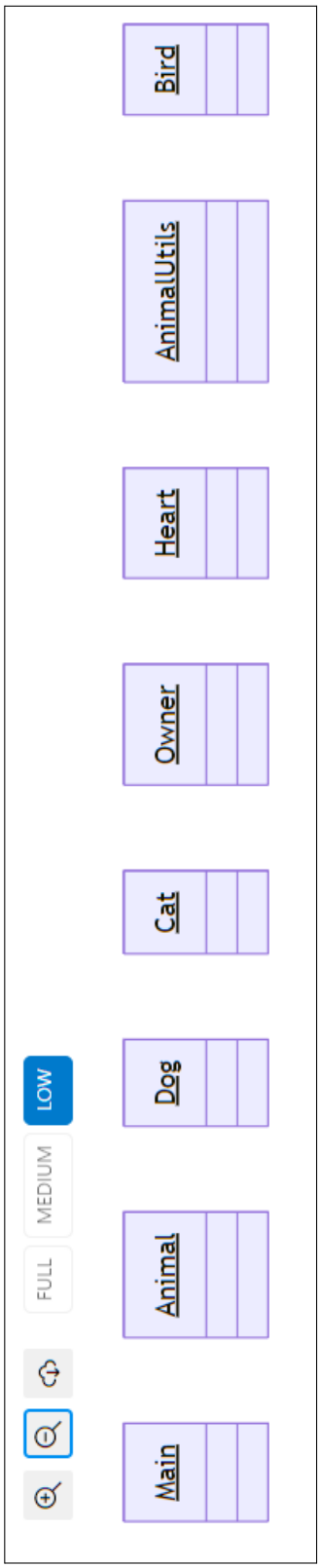


Figure 3.18: UML Class Diagram - Detail Level: "LOW"

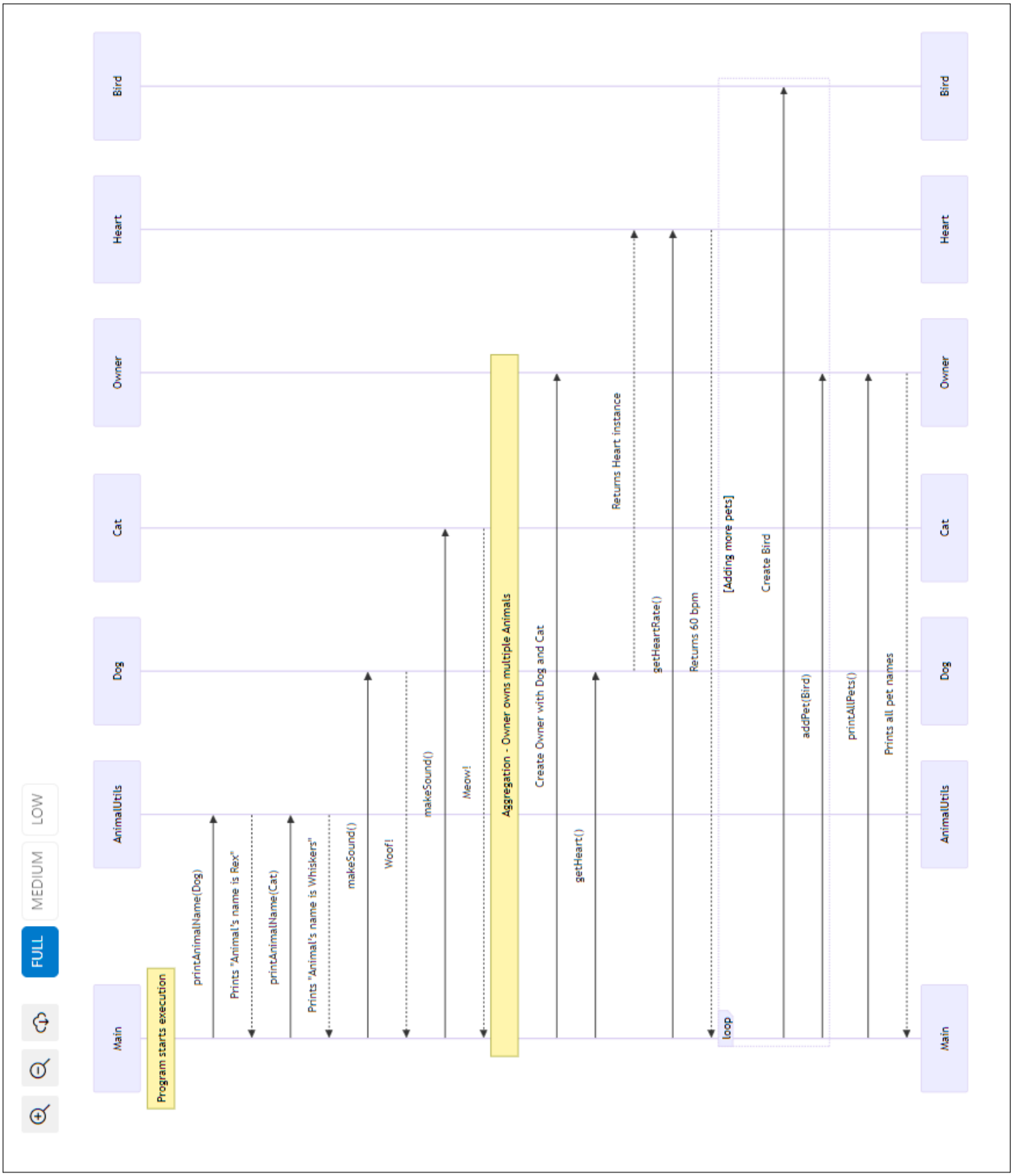


Figure 3.19: UML Sequence Diagram - Detail Level: "FULL"

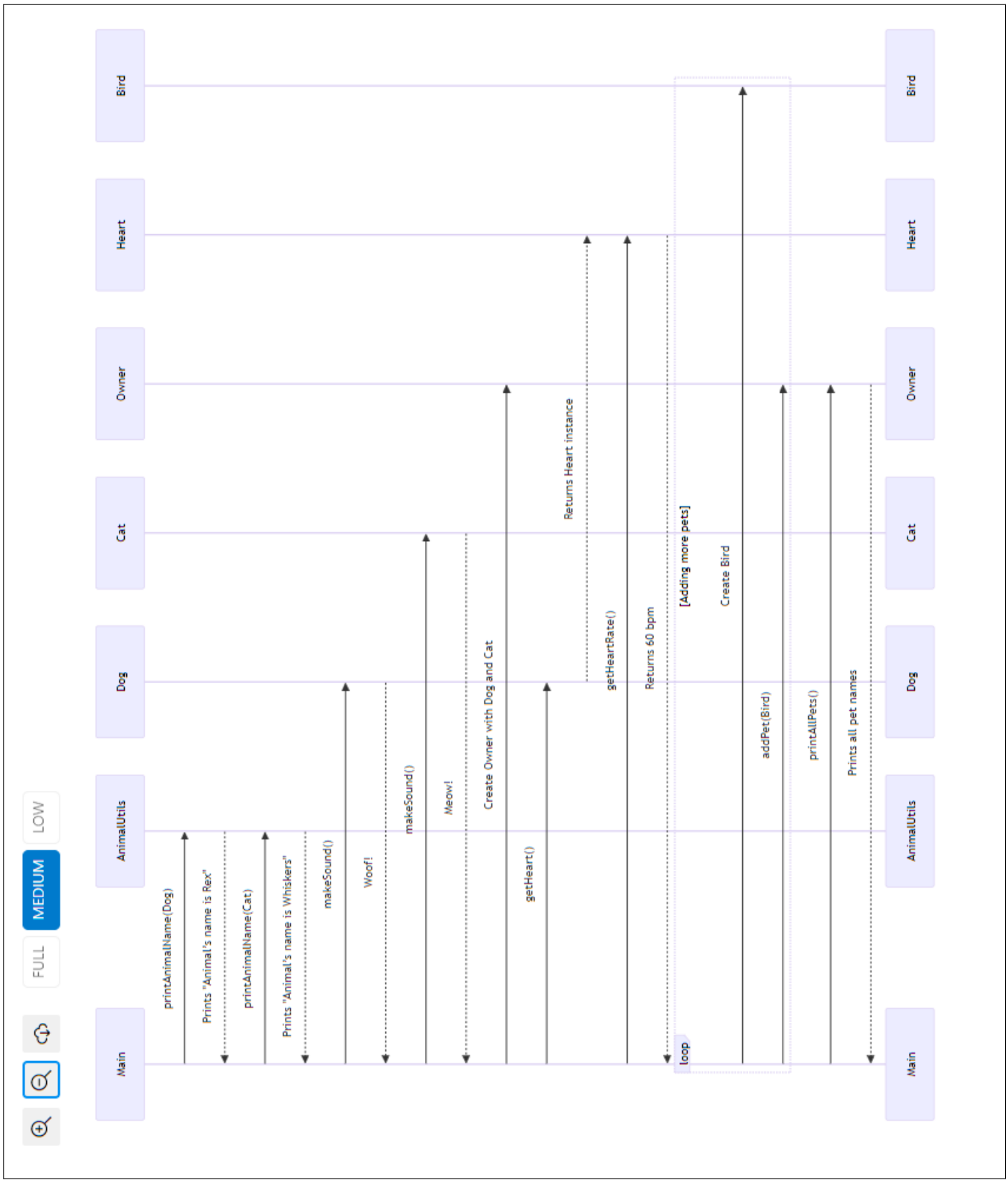


Figure 3.20: UML Sequence Diagram - Detail Level: "MEDIUM"

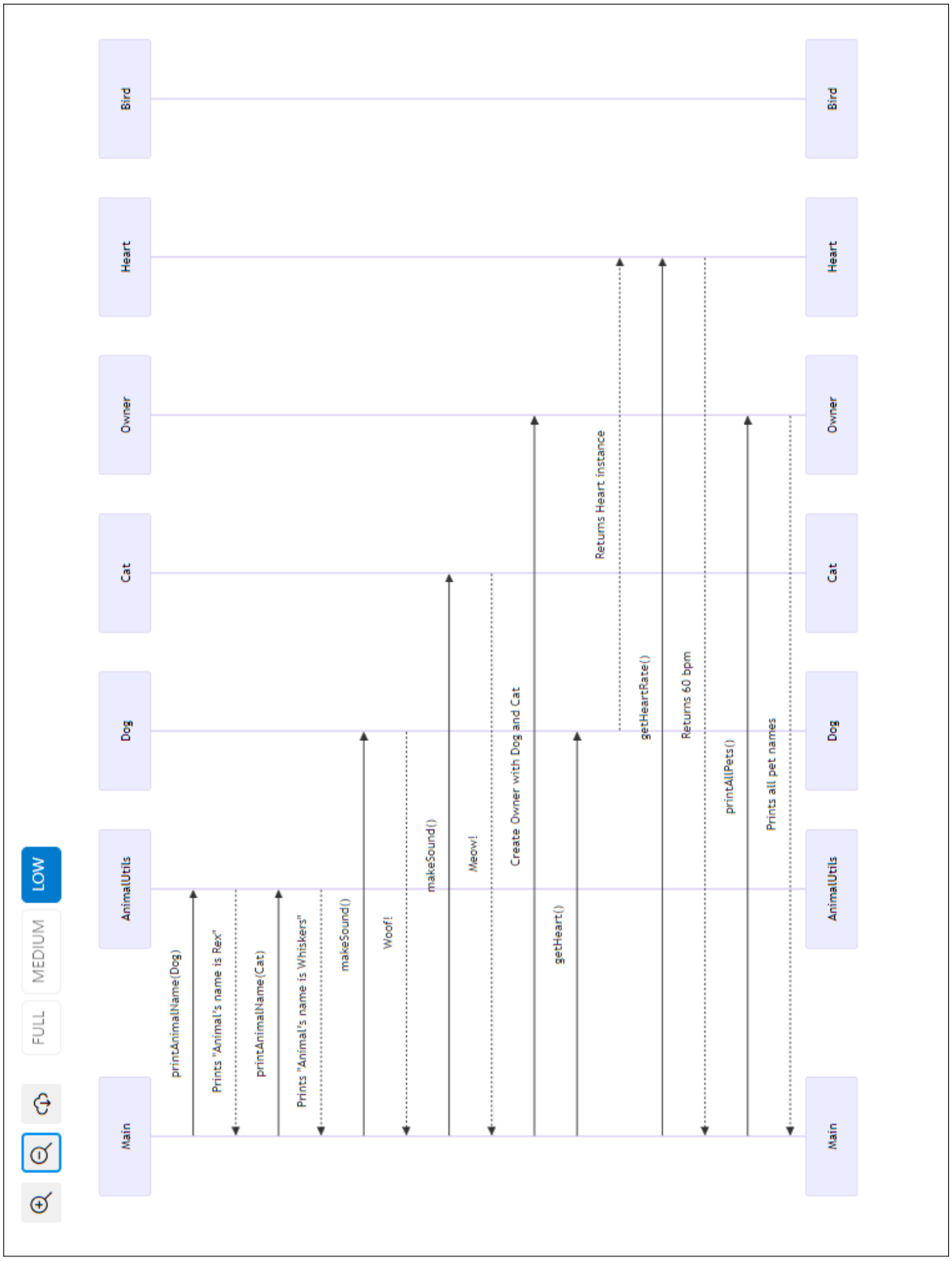


Figure 3.21: UML Sequence Diagram - Detail Level: "LOW"

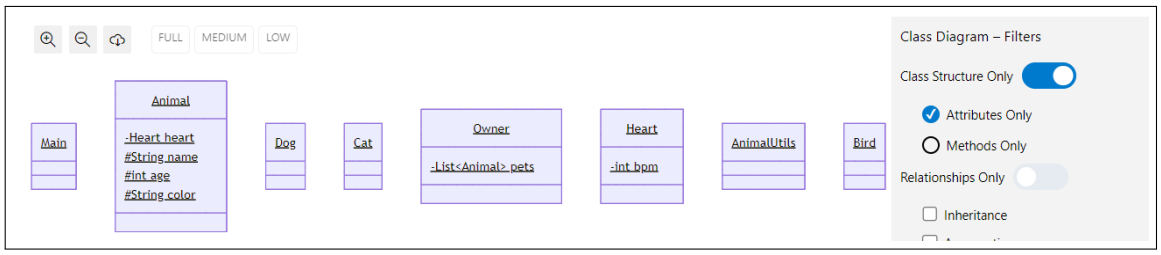


Figure 3.23: Class Structure Only Toggle - Attributes Only

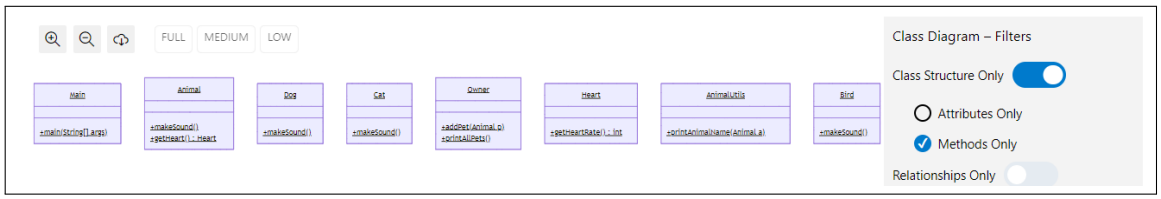


Figure 3.24: Class Structure Only Toggle - Methods Only

As for the "Relationships Only" toggle, when turned on, it removes all class attributes and methods, focusing only on the relationships present (Figure 3.25). It automatically checks all its sub-options, allowing the user to uncheck and check again the relationship types he wishes to be displayed.

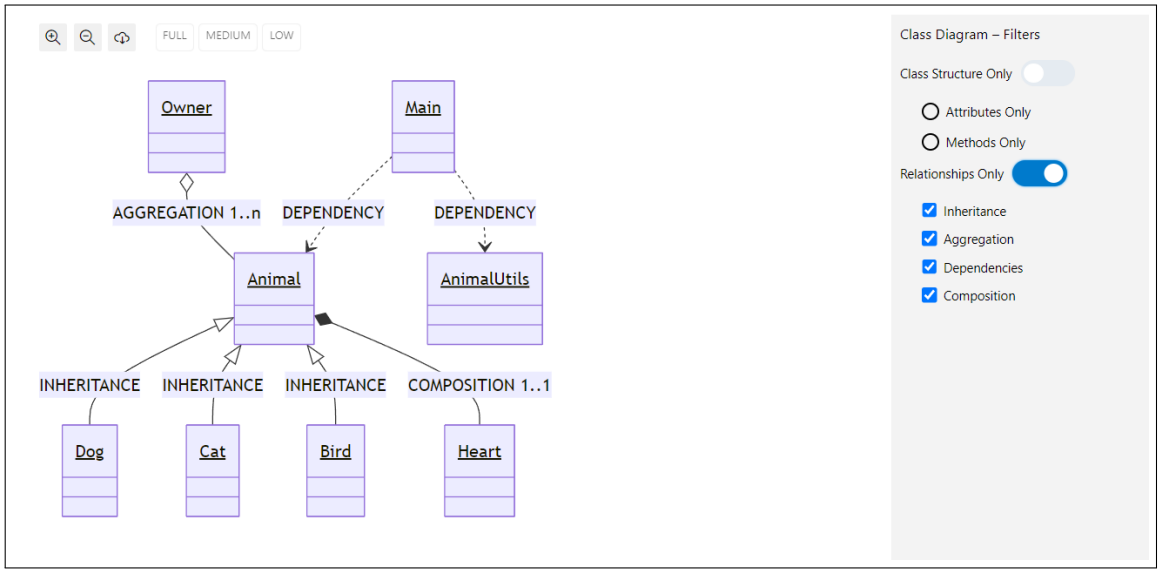


Figure 3.25: UML Class Diagram - Relationships Only Toggle

For example, in Figure 3.26, only Inheritance relationships and Dependencies are depicted in the diagram, as selected in the check boxes.

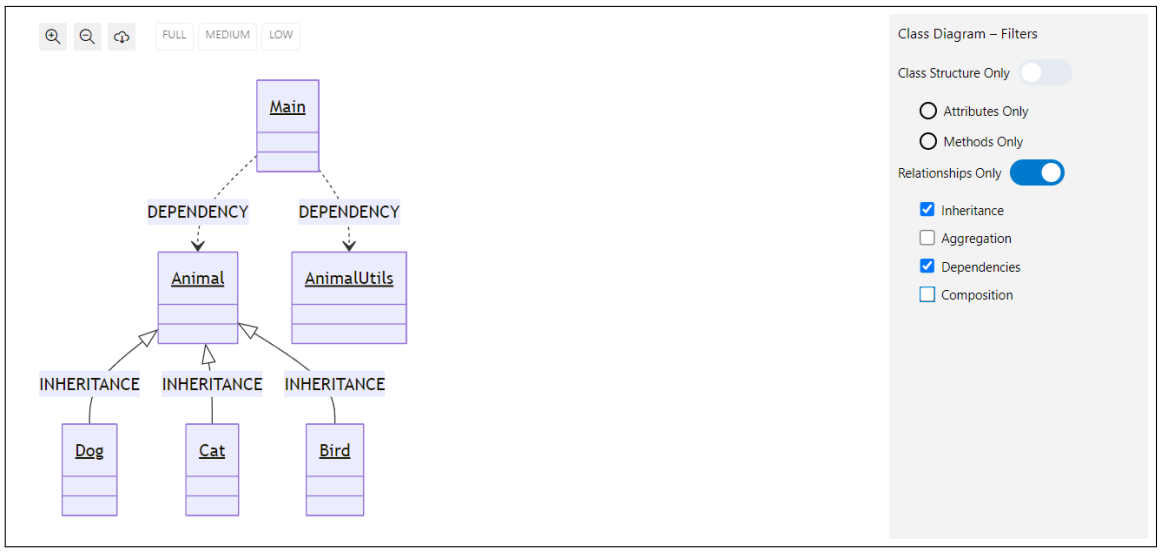


Figure 3.26: Relationships Only Toggle - Inheritance and Dependencies

Both toggles cannot be "On" at the same time.

3.4.8 Sequence Diagram Toggles

On the other hand, the toolbox specific to sequence diagrams (Figure 3.14 - (b) has only one available option for now, the "Core Interactions Only" toggle. This toggle, when turned on, applies the same changes as sequence diagram detail level "LOW" (see Table 3.4), removing notes, comments, any loops, alternatives, and parallel fragment blocks, as seen in Figure 3.27.

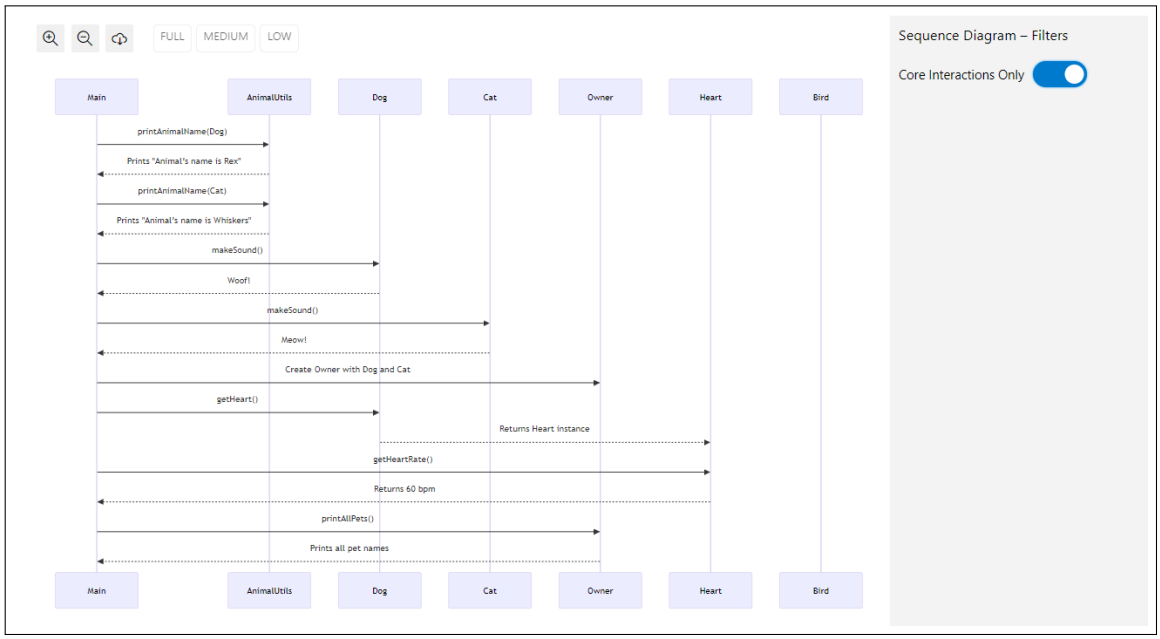


Figure 3.27: UML Sequence Diagram - Core Interactions Only Toggle

3.5 Reflection

Even with the lack of a full formal validation study, during the development and testing of InSight UML it was possible to reflect on its usefulness and effectiveness.

It became clear how the flexibility added by the layered filtering and the interactive controls implemented in this prototype could simplify software comprehension processes and reduce cognitive overload by choosing the detail in which the diagrams are displayed.

Despite the absence of a validation, this initial hands-on experience reinforces the idea that spatialized and layered visualizations are relevant in the software inspection process, encouraging future testing in large scale.

3.6 Summary

Combining and adding all these new interactive features to the original vscode-mermAId tool generated InSight UML, a tool for visualizing UML class and sequence diagrams, created automatically from the Java code files using Mermaid syntax and Copilot. Following an approach similar to the Google Earth layers logic to create a filtering system for the diagrams, it allows users to toggle context on and off and focus on specific parts of the diagram, in order to better comprehend it. A full view of the tool can be seen in Figure 3.9, with the diagram webview on the left and the Github Copilot chat on the right.

This tool offers maximum flexibility by letting users switch diagram types (class vs. sequence) and toggle exactly which elements are shown, so they can focus only on what matters. Because all filtering happens client-side, updates happen almost instantly, keeping the experience fast and responsive.

Chapter 4

Conclusions And Future Work

4.1 Conclusions

This research set out to explore how software inspection and review processes can be improved by enhancing code comprehension through spatial visualization techniques and layered diagram representations.

To fulfill this goal, a tool was developed that automatically generates UML class and sequence diagrams directly from source code, using easily available artificial intelligence tools, and that enables users to visualize different parts of these diagrams by interacting with filtering options and personalized toggles.

Based on the reflection carried out, the initial hypothesis, that spatialized visualization techniques can provide a better understanding of software systems maintenance tasks, appears to be supported.

The studies reviewed in the field of abstraction layers in graphical visualizations to better comprehend systems served as a foundation to understand current expectations in diagram-based tools and to identify which features remain unexplored. These insights directly informed the design and implementation of the proposed solution.

4.2 Key Contributions

The main contribution of this research is Insight UML, a VSCode extension that extends the features of vscode-mermAId. It introduces a dual webview for automatically generated class and sequence diagrams, integrates specific filtering options for each view, and includes layered zoom capabilities. The diagrams are generated in Mermaid.js using Github Copilot.

The tool focuses on improved filtering and navigation features to support software inspection tasks, allowing the customization of the diagram view dynamically based on what the user wants to focus on in each moment.

In addition to that, this dissertation provides more information on the developed tool's implementation and offers a broad survey of related tools in the field of software visualization for comprehension.

4.3 Major Challenges

During the course of this research, many challenges were faced.

The first major difficulty was finding a base tool capable of generating both UML class and sequence diagrams from source code, to extend with visualization and filtering features. Many tools were considered, but few were actively maintained or supported multiple types of diagrams. Visual Paradigm initially seemed promising, yet after extensive experimentation and contact with their support team, licensing and API limitations made the approach unfeasible. This led to a change in direction and a full restart of the development and tool architecture.

Regarding the vscode-mermAId extension, there were many challenges, particularly in the initial stages of development. The extension lacked clear documentation and contextual guidance, so figuring out how to modify and re-render Mermaid diagrams dynamically was a major complication.

Finally, balancing the Mermaid.js limitations with the user experience goals proved to be a difficult task, mainly regarding the layout of the diagrams changing when applying filtering options. By changing the underlying Mermaid syntax, the diagram layout often shifted, with class elements repositioned in ways that hindered spatial consistency, which is a problem for a tool meant to support software comprehension. The generated diagrams are also not 100% correct, with less accuracy in larger systems. These are aspects to be mitigated in the future.

4.4 Future Work

As for future work, among the features mentioned in Chapter 3 (Figures 3.2, 3.3, 3.4), there is a goal to add support for more diagram types, improve Copilot prompts in order to increase diagram accuracy, enable collaborative features, integrate a dynamic layout engine to enhance spatial organization, and add AR immersive and interactive visualization techniques with spatial representation of diagram elements.

In addition, there is a need to improve the tool to automatically browse through projects with many files and compile all the information in one single diagram at once, since currently Copilot takes into consideration only the file active in the editor, so the user must manually open each file and use the `/iterate` command so that file is added to the diagram.

Porting the extension to other IDEs and adding support for more programming languages would be beneficial.

There is also a need for complete validation at a large scale, comparing and collecting the data from several participants with half completing software comprehension tasks using traditional methods and the other half completing the same tasks using the developed tool. The metrics

collected will include the time taken to complete tasks, accuracy and user satisfaction scores (using surveys and interviews).

References

ABDELFATTAH, Amr S., 2022. Microservices-based systems visualization: student research abstract. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. Virtual Event: Association for Computing Machinery, pp. 1460–1464. SAC '22. ISBN 9781450387132. Available from DOI: 10.1145/3477314.3506963.

ARDIMENTO, Pasquale; BERNARDI, Mario Luca; CIMITILE, Marta; SCALERA, Michele, 2024. A RAG-based Feedback Tool to Augment UML Class Diagram Learning. In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz, Austria: Association for Computing Machinery, pp. 26–30. MODELS Companion '24. ISBN 9798400706226. Available from DOI: 10.1145/3652620.3687784.

AUSTIN, M.A.; SAMADZADEH, M.H., 2005. Software comprehension/maintenance: an introductory course. In: *18th International Conference on Systems Engineering (ICSEng'05)*, pp. 414–419. Available from DOI: 10.1109/ICSENG.2005.77.

BALCI, Faruk; HALILOĞLU, Dilruba Sultan; ŞAHIN, Onur; TILKI, Cankat; YURTSEVER, Mehmet Ata; TÜZÜN, Eray, 2021. Augmenting Code Review Experience Through Visualization. In: *2021 Working Conference on Software Visualization (VISSOFT)*, pp. 110–114. Available from DOI: 10.1109/VISSOFT52517.2021.00021.

BALDE, Ketki; PRASAD, Lalji; YADAV, Rashmi, 2024. Visualizing Complexity: A Dynamic Approach to Reverse Engineering. In: *2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET)*, pp. 1–7. Available from DOI: 10.1109/ACROSET62108.2024.10743279.

BOEHM-DAVIS, Deborah A., 1988. Software Comprehension. In: Elsevier. Available from DOI: 10.1016/B978-0-444-70536-5.50010-5.

BROWN, Simon, 2024. *The C4 Model for Visualising Software Architecture*. Available also from: https://c4model.com/.

BUSS, Erich; HENSHAW, John, 2010. A software reverse engineering experience. In: *CASCON First Decade High Impact Papers*. Toronto, Ontario, Canada: IBM Corp., pp. 42–60. CASCON '10. Available from DOI: 10.1145/1925805.1925808.

CANFORA, Gerardo; DI PENTA, Massimiliano; CERULO, Luigi, 2011. Achievements and challenges in software reverse engineering. *Commun. ACM*. Vol. 54, no. 4, pp. 142–151. ISSN 0001-0782. Available from DOI: 10.1145/1924421.1924451.

DE BARI, Daniele; GARACCIONE, Giacomo; COPPOLA, Riccardo; TORCHIANO, Marco; ARDITO, Luca, 2024. Evaluating Large Language Models in Exercises of UML Class Diagram Modeling. In: *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. Barcelona, Spain: Association for Computing

Machinery, pp. 393–399. ESEM '24. ISBN 9798400710476. Available from DOI: 10.1145/3674805.3690741.

DIAS, Martin; ORELLANA, Diego; VIDAL, Santiago; MERINO, Leonel; BERGEL, Alexandre, 2020. Evaluating a Visual Approach for Understanding JavaScript Source Code. In: *Proceedings of the 28th International Conference on Program Comprehension*. Seoul, Republic of Korea: Association for Computing Machinery, pp. 128–138. ICPC '20. ISBN 9781450379588. Available from DOI: 10.1145/3387904.3389275.

DUJLOVIC, Igor; OBRADOVIC, Nikola; KELEC, Aleksandar; BRDJANIN, Drazen; BANJAC, Goran; BANJAC, Danijela, 2019. An Approach to Web-based Visualization of Automatically Generated Data Models. In: *IEEE EUROCON 2019 -18th International Conference on Smart Technologies*, pp. 1–6. Available from DOI: 10.1109/EUROCON.2019.8861729.

FOSS, Sarah; URAZOVA, Tatiana; LAWRENCE, Ramon, 2022. Automatic Generation and Marking of UML Database Design Diagrams. In: *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education - Volume 1*. Providence, RI, USA: Association for Computing Machinery, pp. 626–632. SIGCSE 2022. ISBN 9781450390705. Available from DOI: 10.1145/3478431.3499376.

GOOGLE, 2019. *Google Earth* [https://earth.google.com]. Accessed 2025-06-21.

JAHAN, Munima; ABAD, Zahra Shakeri Hossein; FAR, Behrouz, 2021. Generating Sequence Diagram from Natural Language Requirements. In: *2021 IEEE 29th International Requirements Engineering Conference Workshops (REW)*, pp. 39–48. Available from DOI: 10.1109/REW53955.2021.00012.

JAHAN, Munima; HASSAN, Mohammad Mahdi; GOLPAYEGANI, Reza; RANJBARAN, Golshid; ROY, Chanchal; ROY, Banani; SCHNEIDER, Kevin, 2024. Automated Derivation of UML Sequence Diagrams from User Stories: Unleashing the Power of Generative AI vs. a Rule-Based Approach. In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz, Austria: Association for Computing Machinery, pp. 138–148. MODELS '24. ISBN 9798400705045. Available from DOI: 10.1145/3640310.3674081.

JOLAK, Rodi; LE, Khan-Duy; SENER, Kaan Burak; CHAUDRON, Michel R.V., 2018. OctoBubbles: A Multi-view interactive environment for concurrent visualization and synchronization of UML models and code. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 482–486. Available from DOI: 10.1109/SANER.2018.8330244.

JOLAK, Rodi; HO-QUANG, Truong; CHAUDRON, Michel R.V.; SCHIFFELERS, Ramon R.H., 2018. Model-Based Software Engineering: A Multiple-Case Study on Challenges and Development Efforts. In: *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*. Copenhagen, Denmark: Association for Computing Machinery, pp. 213–223. MODELS '18. ISBN 9781450349499. Available from DOI: 10.1145/3239372.3239404.

KAKKENBERG, Roy; RUKMONO, Satrio Adi; CHAUDRON, Michel; GERHOLT, Wim; PINTO, Miguel; OLIVEIRA, Claudio Ribeiro de, 2024. Arvisan: an Interactive Tool for Visualisation and Analysis of Low-Code Architecture Landscapes. In: *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*. Linz, Austria: Association for Computing Machinery, pp. 848–855. MODELS Companion '24. ISBN 9798400706226. Available from DOI: 10.1145/3652620.3688331.

KRAUSE-GLAU, A.; HASSELBRING, W., 2023. Collaborative, Code-Proximal Dynamic Software Visualization within Code Editors. Available from DOI: 10.1109/VISSOFT60811.2023.00016.

Mermaid – JavaScript-based Diagramming and Charting Tool, 2020 [https://mermaid.js.org]. Accessed 2025-06-21.

METIN, Haydar; BORK, Dominik, 2023. Introducing BIGUML: A Flexible Open-Source GLSP-Based Web Modeling Tool for UML. In: *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 40–44. Available from DOI: 10.1109/MODELS-C59198.2023.00016.

MICROSOFT, 2025. *vscode-mermAId* [https://github.com/microsoft/vscode-mermAId]. Release v0.0.3, commit 6cb5d8b.

NAIR, Namitha S; MOHAN, Aswathy; JAYARAMAN, Swaminathan, 2020. Interactive Exploration of Compact Sequence Diagrams - JIVE based approaches. In: *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pp. 907–912. Available from DOI: 10.1109/ICSSIT48917.2020.9214261.

NUR, Kamruddin; SARWAR, Hasan, 2010. Software Visualization Tools for Software Comprehension. In.

OBJECT MANAGEMENT GROUP, 2017. *Unified Modeling Language (UML) Specification, Version 2.5.1*. Accessed: 2025-06-29.

OLIVEIRA, Margarida Rocha Raposo de, 2024. *Spatialized Live Refactoring*. MA thesis. Faculdade de Engenharia da Universidade do Porto.

RADOSKY, Lukas; POLASEK, Ivan, 2024. Executable Multi-Layered Software Models. In: *Proceedings of the 1st International Workshop on Designing Software*. Lisbon, Portugal: Association for Computing Machinery, pp. 46–51. Designing '24. ISBN 9798400705632. Available from DOI: 10.1145/3643660.3643938.

RAMACKERS, Guus J.; GRIFFIOEN, Pepijn P.; SCHOUTEN, Martijn B.J.; CHAUDRON, Michel R.V., 2021. From Prose to Prototype: Synthesising Executable UML Models from Natural Language. In: *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pp. 380–389. Available from DOI: 10.1109/MODELS-C53483.2021.00061.

SAVARY-LEBLANC, Maxime; PALLEC, Xavier Le, 2022. Interactive highlighting for digital UML class diagrams: a new feature. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 247–256. MODELS '22. ISBN 9781450394673. Available from DOI: 10.1145/3550356.3561557.

SAVARY-LEBLANC, Maxime; PALLEC, Xavier Le; PALANQUE, Philippe; MARTINIE, Célia; BLOUIN, Arnaud; JOUAULT, Frédéric; CLAVREUL, Mickael; RAFFAILLAC, Thibault, 2022. Mining human factors general trends from +100k UML class diagrams. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. Montreal, Quebec, Canada: Association for Computing Machinery, pp. 913–922. MODELS '22. ISBN 9781450394673. Available from DOI: 10.1145/3550356.3559098.

SHEHATA, Mina; LEPORE, Blaire; CUMMINGS, Hailey; PARRA, Esteban, 2024. Creating UML Class Diagrams with General-Purpose LLMs. In: *2024 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 157–158. ISSN 2832-6555. Available from DOI: 10.1109/VISSOFT64034.2024.00031.

SIEGMUND, Janet, 2016. Program Comprehension: Past, Present, and Future. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. Available from DOI: 10.1109/SANER.2016.35.

VINITA; JAIN, Amita; TAYAL, Devendra K., 2008. On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms. *SIGSOFT Softw. Eng. Notes*. Vol. 33, no. 5. ISSN 0163-5948. Available from DOI: 10.1145/1402521.1402527.

VISUAL PARADIGM, [n.d.]. *Visual Paradigm Official Website* [https://www.visual-paradigm.com/].

ZHANG, Xun; WASHIZAKI, Hironori; YOSHIOKA, Nobukazu; FUKAZAWA, Yoshiaki, 2022. Detecting Design Patterns in UML Class Diagram Images using Deep Learning. In: *2022 IEEE/ACIS 23rd International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pp. 27–32. ISSN 2693-8421. Available from DOI: 10.1109/SNPD54884.2022.10051795.