UNIVERSITY OF TARTU
Institute of Computer Science
Software engineering Curriculum

Einar Linde

Implementation of JIT (Just in Time) Visualization of Changes in Source Code

Master's Thesis (30 ECTS)

Supervisor(s): Kristiina Rahkema

Tartu 2022

Implementation of JIT (Just in Time) Visualization of Changes in Source Code

Abstract:

The need for software developers increases at a high pace. But the programming is a difficult cognitive skill to learn and the supply of good programmers does not meet the need. Especially hard it is for beginner programmers and therefore there are high drop-out rates in universities. Tools that help beginner programmers to understand the code exist. But none provides real-time visualization of source code evolution. The goal of this thesis is to develop a tool that visualizes source code changes in real-time. The thesis describes the development process and architecture of the source code analysis tool and gives usage scenarios on how the tool could help new developers understand object-oriented code. The main objective is to analyze Java code. The real-time analysis is achieved by using a language server protocol that provides real-time data from the user's editor. To analyze the code, GraphifyEvolution is used. Even though the main analyzed language is Java, the tool's architecture supports the addition of a new language. The developed tool is called JitEvolution.

Keywords:

source code evolution, source code analysis, visualization, object-oriented programming, language server

CERCS: P170 - Computer science, numerical analysis, systems, control

Reaalajas koodimuudatuste visualiseerimine

Lühikokkuvõte:

Vajadus tarkvara arendajate järele kasavab jõudsalt. Kuid programmeerimine on raske kognitiivne oskus ja heade programmeerijate saadavus ei ole vastavuses nõudlusega. Eriti raske on programmeerimine algajatele ja seetõttu on ülikoolides kõrge informaatika eriala väljalangevus. On olemas tööriistu mis aitavad uusi programmeerijad, kuid ükski neist ei analüüsi koodimuudatusi reaalajas.

Selle magistritöö eesmärk on arendada tööriist, mis visualiseerib koodimuudatusi reaalajas. See magistritöö kirjeldab loodava tööriista arendusprotsessi ja arhitektuuri. Lisakas veel annab kasutamisstenaariume, mis kirjeldavad kuidas tööriista kasutada ning kuidas need aitavad alustavaid programmeerijaid. Tööriista peamine ülesanne on analüüsida Java lähtekoodi. Reaalajas analüüsimine on saavutatud keeleserveri abil, mis saadab reaalajas kasutaja koodiredaktorist muudatusi analüüsimiseks. Lähtekoodi analüüsimiseks kasutatakse GraphifyEvolution koodi analüsaatorit. Kuigi peamine analüseeritav keel on Java on tööriista arhitektuur selline, et uuele keelele toe lisamine on lihtne. Tööriista nimi on JitEvolution.

Võtmesõnad:

lähtekoodi evolutsioon, lähtekoodi analüüsimine, visualiseerimine, objektorienteeritud programmeerimine, keeleserver

CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Table of Contents

1 Introduction

Software development is an evolving industry and educational institutions are teaching more and more students [1]. Bergin and Reilly [2] said that "It is well known in computer Science Education community that students have difficulty with programming courses and this can result in high drop-out failure rates". Bosse and Gerosa surveyed students and instructors to find out what are those difficulties [1]. They found out that students have difficulties in parts of the programming. For example, students struggle with the scope of the variables, loops, and syntax errors. Also, one of the difficult topics for students is object-oriented programming [3]. Students have difficulty understanding classes, class variables, overloaded constructors, and methods [3].

Therefore, to mitigate the drop-out failure rates, there must be good methods and tools to help new students better understand object-oriented programming and algorithms. One of the methods could be analyzing source code evolution in real-time and visualizing it for the students. The goal of this thesis is to create a tool that uses a GraphifyEvolution source code analyzer to give real-time feedback to the students by visualizing source code evolution. The developed tool is called JIT Evolution.

1.1 Problem statement

This master's thesis captures the topic of visualization of source code evolution. The scope of this research is getting source code changes in real-time, analyzing them, and then presenting different aspects of the source code evolution to the user in real-time and visualizing them. The developed tool should be modular so that it would be easy to change some of the parts. Also, the system should be easily extensible in case the source code analyzer that the system uses adds new features. All of this allows future developers to change one of the parts of the system without the need to write a whole new system. The developed tool should be easy to install for the end-user. A complicated installation process could cause more problems than benefits that the tool could offer. Therefore, the developed tool should require as few dependencies that need to be installed as possible.

Given these requirements, the following research questions are posed:

RQ1. Is the developed tool easy to use?

RQ2. Is the developed tool beneficial in learning software development?

Research question RQ1 answers the questions: how easy it is to install it, is the tool easily understandable. The analysis of the source code can be very slow and with research question RQ1 it is evaluated if the developed tool can be considered as a real-time application. Additional information for the answer to the research question RQ1 is also received by measuring the performance of the tool. Additionally, usage scenarios are created to show how easy it is to use the developed tool. Research question RQ2 determines if the tool accomplishes its purpose. Research question RQ2 is answered by creating usage scenarios that show that the tool could be beneficial in learning software development.

1.2 Structure of the thesis

The thesis is separated into six chapters: Introduction, Background, Method, Results, Discussion, and Conclusion. Background chapter describes the existing applications that are relevant to the developed tool, and the source code analyzer used. Method chapter describes the process of how the tool will be developed, how the functional requirements are found, key architectural and technological options, and the testing process. Finally, it covers how the tool is evaluated. Results chapter will cover the elicited functional requirements, the

development of the tool, different modules of the developed tool, and the evaluation. It also explains the architectural and technical choices made. Discussion chapter focuses on encountered problems and restrictions while developing the tool, and discussion of achieved results. Also, it describes future possibilities of the tool and learned lessons that the author learned during developing the tool. The last chapter concludes the achieved results and summarizes the work done.

2 Background

This chapter gives an overview of previous and related work. In the year 2021, two students defended a thesis about visualizing source code quality and evolution. Section 2.1 describes those two and how the tool developed in this thesis is different from the previous tools. It also describes one other source code evolution visualization that the author found interesting. Section 2.2 gives the details about the GraphifyEvolution tool. GraphifyEvolution is a source code analyzer that is developed by this thesis supervisor Kristiina Rahkema. There are some benefits of using the GraphifyEvolution analyzer. Understanding the analyzer capabilities is crucial for the success of the real-time application. The thesis supervisor can provide an in-depth understanding of its capabilities. Also, adding features to GraphifyEvolution can be done quickly by discussing possible modifications to GraphifyEvolution with this thesis supervisor. In addition, this analyzer can analyze source code by its commits which might be useful to visualize source code history [4]. The previously mentioned points are the main reason why GraphifyEvolution was chosen for this tool.

2.1 Existing work on source code visualization

Programming is a difficult cognitive skill to learn [4]. Mastering the basis of a programming language is a huge problem for many students [4]. Code visualization can help new programming students to teach basics of the programming [4]. There are such visualization applications but every application visualizes source code in different ways.

2.1.1 Exploration of Techniques to Visualise Code Quality

In the year 2021, Miron Storožev defended his master thesis about developing a tool to visualize the code quality of an entire project [5]. The tool used GraphifyEvolution static code analyzer. Miron Storožev focused his work on code smell analysis. He developed an application that allowed a user to upload source code and then analyze it. The application consisted of two parts: the source code analyzer module and the web application. The visualization part was done with the Angular framework1 and d3 library2. An example of the visualization can be seen in Figure 1.

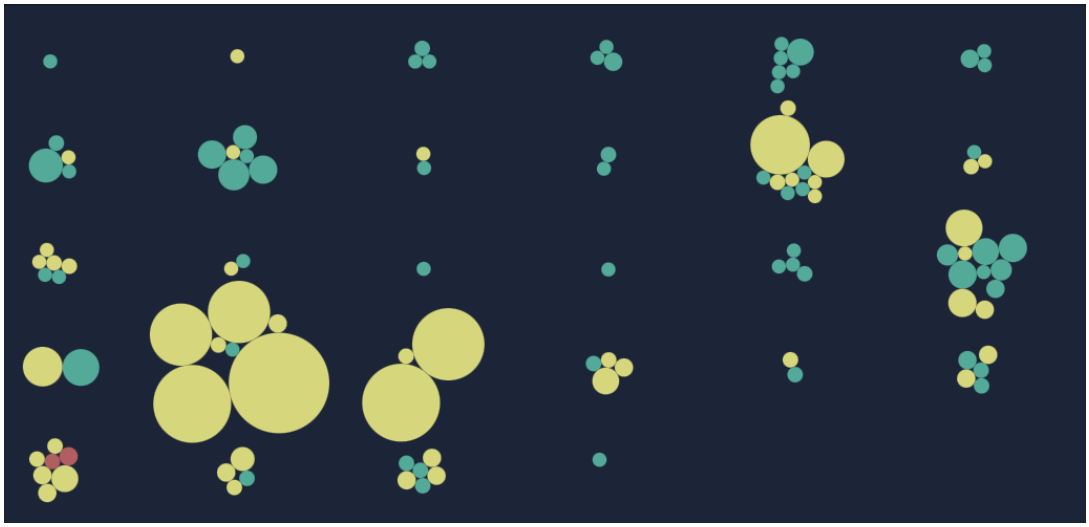

Figure 1: Screenshot of code-smell-visualizer-api code smell analysis [5]

[1] https://angular.io/
[2] https://d3js.org/

Circles represent classes in the project. Green circles mean there are no code smells in that class, yellow circles mean that there are some code smells, and red circles there are a lot of circles. Thresholds for each color are configurable. The circle size shows how many usages that class has. It is a little bit counter-intuitive that the circle size doesn't represent the class size but the author explained that the reason is that the class size is used in the code smell analysis and it is visualized using color notations.

The advantages of that tool are that it analyses the whole project, and is easy to use. It has an understandable GUI to show code smell occurrences. This thesis tries to visualize source code evolution and various object-oriented programming concepts like a class variable, instance variables, variables scopes, etc using class diagrams. This is one of the main differences between this thesis and M. Storožev's. In addition, the target group of M. Storožev's tool is programmers in software development companies but this thesis targets beginner programmers. Also, M. Storožev's visualization does not work in real-time. It requires user interaction to update the visualization. This thesis tries to analyze the source code in real-time and visualize it for the user.

Even though M. Storožev's visualization is understandable, it concentrated mainly on code smell analysis, but this thesis concentrates on object-oriented programming concepts. This is the reason why it does not suit as a base application for this thesis.

2.1.2 Software Analytics: Visualization of Source Code Evolution

In the year 2021, another thesis was made. Turkhan Badalov defended his master thesis about visualizing source code evolution [6]. Badalov used different techniques to produce 4 interactive diagrams. These are the general view, commit range view, and class volume view. The tool used also previously mentioned tool GraphifyEvolution. Figure 2 shows the produced class overview diagram.

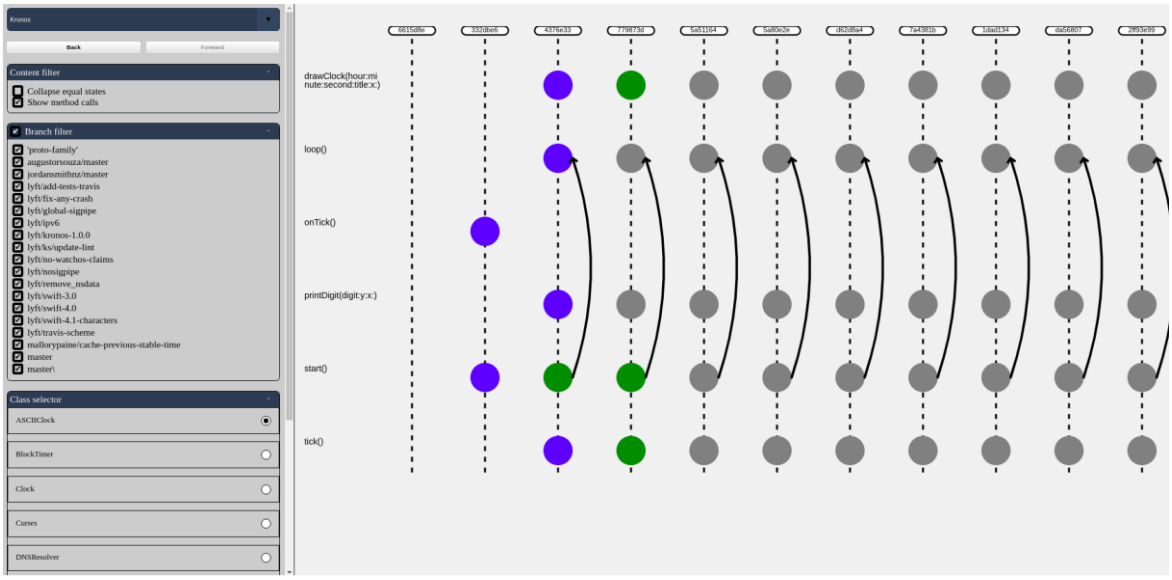

Figure 2: Class overview diagram [6]

Even though it had some interesting diagrams. It did not visualize one of the main requirements that are the scope of the variables. Also, it did not work in real-time and targeted programmers in software development companies but not beginner programmers.

2.1.3 CodeMetropolis

CodeMetropolis is a command-line tool written in C# that creates a Minecraft3 world from source code [7]. The authors Gergo Balogh and Arpad Beszedes describe in the paper „CodeMetropolis – code visualization in Minecraft“ how their application creates the Minecraft world from source code [7]. The application uses architectural metaphors to visualize the source code. For example, methods are floors of the building where its width and height represent the size of the method, classes are represented as buildings, and buildings are grouped into districts to visualize namespaces [7]. The example can be seen in Figure 3.

Figure 3: Junit project visualized by CodeMetrpolis [7]

In the paper „CodeMetropolis: Eclipse over the City of Source Code“ [8] authors Gergo Balogh, Attila Szabolics, and Arpad Beszedes created an IDE plugin that can invoke visualization from the Eclipse IDE. The user can interact with Minecraft through IDE toolbars. It has a feature follow that automatically navigates the user to a building that represents the opened file in IDE. The user has to initiate the building of the world manually because the building could take a long time and is not reasonable to do in real-time. In Figure 4 the comparison of the two methods can be seen.

[3] https://www.minecraft.net/en-us/about-minecraft

Figure 4: Comparison of two methods in the Metropolis [8]

Metropolis has a unique way to visualize the source code for the user. It could be used to help students to understand object-oriented programming. The advantages of this application are that it can be used within IDE and its interesting concept to use the existing computer game for visualization. There are a couple of drawbacks: it is specific to only one IDE, it requires an external program Minecraft, and it is not working in real-time to allow fast feedback to the user.

2.2 GraphifyEvolution – the source code analysis tool

GraphifyEvolution4 is a command-line application developed by Kristiina Rahkema [9]. The application author describes it in the documentation5 of the application as follows. The main objective of the app is to analyze iOS applications in bulk but it also supports other languages such as Java, Python, and C++ [9]. Meaning it could analyze Java code that object-oriented programming course students are using. GraphifyEvolution stores classes, methods, variables, relationships between classes, modules, history of the application, and much more into the Neo4j6 database where data can be later queried using Cypher query language. Cypher's syntax provides a visual and logical way to match patterns in graphs using ASCII-Art syntax7. GraphifyEvolution has support for analyses of git repositories [9]. That allows analyzing existing projects to visualize their evolution [9]. Rahkema provides multiple example queries that can be used to visualize different metrics of the source code. The queries have customizable parameters and can be customized according to the user's needs8. The example of the query can be seen in Figure 5. It answers the question did the methods become too long over time.

[4] https://github.com/kristiinara/GraphifyEvolution
[5] https://github.com/kristiinara/GraphifyEvolution/tree/master/documentation
[6] https://neo4j.com/
[7] https://neo4j.com/developer/cyhttps://neo4j.com/developer/cypher/pher/
[8] https://github.com/kristiinara/GraphifyEvolution/blob/master/documentation/example_queries.md

```
MATCH (m:Method)
WHERE
    m.is_long_method = true
OPTIONAL MATCH
    p=(:Method)-[:CHANGED_TO*]->(m)
OPTIONAL MATCH
    (m2)-[:CHANGED_TO]->(m)
WHERE
    m2.is_long_method = true
WITH
    m, count(relationships(p)) as changes, m2
WHERE
    m2 is null
RETURN count(*), changes
```

Figure 5: Query for question did methods become too long overtime taken from GraphifyEvolution documentation

2.3 Class diagrams

The initial vision of the visualization is to be similar to UML class diagrams. The class diagram is a static view of the application [10]. They mainly consist of classes and their relationships. A class may have attributes and operations associated with them. Classes are rectangles and relationships a line between them. The example of a class diagram is seen in Figure 6.

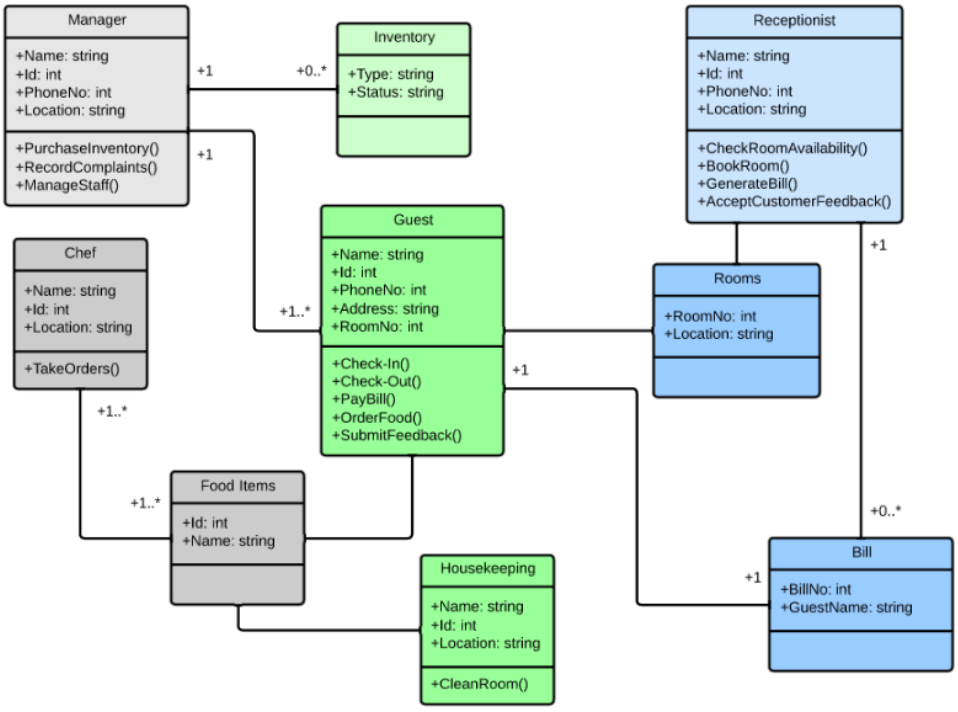

Figure 6: Example of class diagram9

The advantages of the class diagram are the following:

- illustrate data models for information systems,

[9] https://www.lucidchart.com/pages/uml-class-diagram

- understand the general overview of schematics better,
- visually express specific needs of a system,
- implementation-independent description of types used in a system

The class diagrams are generally static and do not have interactions. The class diagram has some similarities with the developed visualization. Thus it can be taken as a basis of the visualization.

3 Method

The main goal of this thesis is to build a tool that visualizes source code changes in real-time that will help students to understand the code better. The tool will take in code changes from the user's editor in real-time and show the analysis results in the web application. This chapter focuses on the methodology used to achieve the goal of this thesis. The methodology is divided into four subsections: requirements, mockup, choices of technologies and architectures, visualization, and evaluation. The requirements section discusses the process of finding functional and non-functional requirements and prioritizing the requirements. The section mockup shows the author's vision of the visualization before the development cycles. The section choices of technologies and architectures describe the aspects that helped to decide on technologies and architectures. Finally, the evaluation process is described.

The development of the software follows some of the agile software development principles10. One of the most notable principles is that the development consists of small sprints. Before the development, the initial set of requirements is elicited for the development process. Sprints will be small and determined by the frequency of the meetings with the supervisor. The usual sprint length is one week long. The sprint process can be seen in Figure 7. At the beginning of the sprints, requirements are gathered and prioritized. After that the software is designed, followed by implementation of the requirements, testing, deployment, and sprint review. During the requirements phase, existing requirements are revised and new ones added. After that design is thought through and implemented. Testing will be done manually by the developer to find bugs. Deployment to the test server will be automated for easy validation if the tool works also on other than the developer computer and for the presentation during the last phase of the sprint – sprint review. During the sprint review, the developer will present the done work to this thesis supervisor and discuss the potential improvement to the tool.

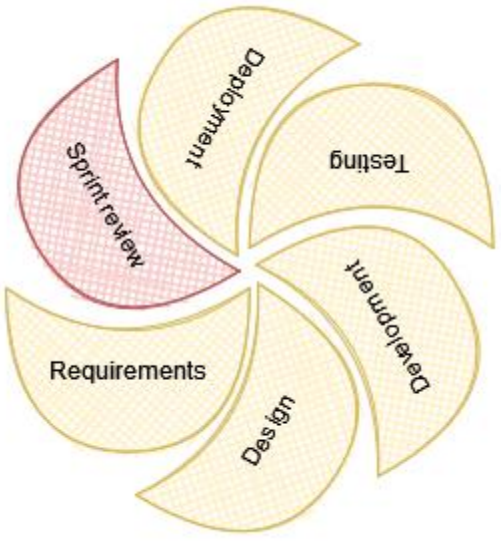

Figure 7: The sprint process

[10] https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/

3.1 Requirements

Requirements of the developed tool are elicited by discussing the problems with the thesis supervisor Kristiina Rahkema. Additionally to better understand the problems of the beginner programmers' research papers databases are queried. This helps to understand more deeply the problems of beginner programmers and produce more useful visualization. It is expected to have about 30 requirements. To know which requirements should be implemented first, the prioritization of requirements is important. Prioritization shows what requirements are considered more important than others and should be implemented first.

3.1.1 Requirements gathering

Initial requirements are gathered before the first sprint. Those requirements are elicited by discussing the problems of beginner programmers with thesis supervisor Kristiina Rahkema. The discussion is conducted in free form. Questions that are answered are the following:

- What concepts of object-oriented programming do students find difficult to understand?
- What do you expect from a program that is going to visualize source code evolution?

It is crucial to understand the problems of students. Therefore additionally the research papers' databases are queried. For the database, Google Scholar is used. Google Scholar provides a large set of published and yet not published papers. It is chosen based on the author's personal preference and ease of use. The results from the database only support the discussion with the supervisor and course instructors and are not the source of requirements.

The development process will contain multiple sprints. The previously mentioned method for finding requirements is used before the first sprint. At the beginning of the sprint, all the requirements are revisited and potentially enhanced based on the discussion with the thesis supervisor during the sprint review. Due to that, the development is more agile to the changes and guides the development to better results. After the last sprint, the requirements should be polished enough for evaluation if the tool could be used to support learning object-oriented programming.

Requirements will have ID to identify uniquely the requirement and description. Written language is often very imprecise [11]. A customer and a developer might interpret a statement differently [11]. Thus the requirements should be written as precisely as possible. Due to that descriptions are written as user stories to have a better understanding of who is the user, what they want to do, and why is this important.

From the topic of this thesis and research questions, some of the non-functional requirements were elicited to help to elicit more requirements. Those requirements can be seen in Table 1.

Table 1: Non-functional requirements

ID	Description
JIT-NFR-1	As a user, I want to analyze the project in under 60 seconds, so that I do not have to wait for results for a long time.
JIT-NFR-2	As a user, I want to start the analysis of the project in under 5 seconds, so that I don't have to start the analysis manually.
JIT-NFR-3	As a user, I want to the tool be as lightweight as possible, so that I can run it within IDE and the development environment.

3.1.2 Requirements prioritization

Based on the author's personal experience the developed tool is expected to have about 30 requirements. To know which requirements should be implemented first, requirements are prioritized. Two techniques were considered for requirements prioritization. These are the analytic hierarchy process (AHP) and numeral assignment techniques.

Mikko Vestola compared different prioritization techniques [12]. He said that the numerical assignment technique is said to be the most common technique mainly because of its simplicity and ease of use. The idea of the numerical assignment technique is to create prioritization groups and then requirements are classified into those groups. One variation of the numerical prioritization technique is the MoSCoW technique. In the MoSCoW technique there are 4 priority groups: M – must have, S – should have, C – could have, and W – won't have. Those priority groups are self-explanatory. Must have requirements - must be implemented for a minimum viable product (MVP). Should have and could have requirements - are nice to have but aren't required in the MVP. Won't have requirements - are out of the scope of the current iteration but might be implemented in future iterations. Compared to the AHP, the MoSCoW technique is fast and easy.

The second considered technique was AHP. The AHP generates a large number of comparisons if there are over 20 requirements and because it is expected to have over 30 requirements then it was not used.

As the developed tool is expected to have over 30 requirements, the MoSCoW technique is chosen for its self-explanatory categories and prioritization speed for the expected requirements amount.

3.2 Mockup

The mockup of the visualization is constructed to better understand the requirements of the tool and forethink the technologies and architectural design before the development iterations. The mockup is used only to plan the initial look of the visualization and demonstrate the basic functionality of the tool.

The mockup is made using a Miro board11. The tool choice is based on the author's previous experience.

[11] https://miro.com/

3.3 Choices of technologies and architectures

The quality of the software and the time of implementation can be improved by forethinking technological and architectural designs before writing the code. This section describes different criteria that are considered when making choices of architecture, programming languages, tools, and technologies that will be used to develop the real-time source code evolution visualization tool.

In the big picture, there are a couple of aspects that should be forethought before the first sprint. These are the following:

- database,
- communication between the source code analyzer and an editor (IDE),
- source code analyzer,
- visualization.

Forethinking those key points reduces the chances that the tool should be heavily rewritten because of the misunderstanding of the requirements or bad architectural design.

3.3.1 Database

In section 2, it was stated that the developed tool will use GraphifyEvolution as a source code analyzer. GraphifyEvolution uses the Neo4J database. Therefore one of the required databases is the Neo4j database. The Neo4J is a graph database and it is excellent when data is highly-connected but the tool will also have other types of data like login information. For storing login information a relational database is a better choice. Due to that and to make the tool not dependent on the source code analyzer, an additional database is needed. The secondary database is chosen by the author's personal preference. It is a Postgres database.

3.3.2 Communication between the source code analyzer and IDE

The primary language that the tool is used is Java because the future objective is to use the tool in an object-oriented programming course at the University of Tartu. The mentioned course uses Java to teach object-oriented concepts. Running the tool successfully on the user's computer requires that the user have a correct runtime environment. As the users will analyze Java projects, they will have a Java runtime environment installed. Due to that, the communication between the source code analyzer and the editor should be written in Java. So that the users don’t need to install additional software on their computer.

The analysis of the source code can be quite resource-intensive and slow down the users' computers. Therefore, another important keyword is that the tool should be as lightweight as possible. To combat that the communication should be encapsulated from the rest of the tool. Proper encapsulation allows installing only the communication module on the user’s computer and the rest of the tool, such as the source code analyzer and visualization, could be deployed to the centralized location that the user’s editor uses. It also allows easier change of communication modules in the future when there are better ways to send the information from the editor to the source code analyzer.

There are a lot of different IDEs for users to use. So not restricting the users to only one editor, the communication method should be chosen in a way it is easy to implement

for the new editor. One of such technologies is the language server protocol (LSP)12. LSP is discussed in detail in section 4.4.

3.3.3 Source code analyzer

The source code analyzer will be GraphifyEvolution. As previously mentioned GraphifyEvolution is a command-line tool developed in Swift. Swift is used for developing apps for Mac OS. Mac OS holds second place in the global market share of desktop PC operating systems13. Even though it is in the second, its market share is small - about 15% compared to Windows 73%. The source code analyzer strongly restricts the system the tool can be used. GraphifyEvolution requires a Mac OS [9]. Because of that modularity and encapsulation is also important. It gives the ability to easily change the source code analyzer in the future and deploy it separately from the rest of the tool.

GraphifyEvolution uses the Neo4j database. To allow changing the source code analyzer the rest of the system should not depend on that database.

3.3.4 Visualization

Most of the information that the user gets from the tool is from visualization. Thus, choosing a good architecture and technology stack for the visualization is important. There are a couple of key points that should be considered when choosing the architecture for the visualization.

Firstly, the visualization is expected to be interactive. Thus, the selected technology should provide updating the view smoothly in real-time. One of the possibilities is to use server-side rendering. Server-side rendering converts the HTML files on the server to useable information for the browser14. Due to that when the view should be updated the browser requests the whole page from the server again. This kind of rendering is not ideal for dynamically changing views because this reloads the page to show updated information. Opposite of the server-side rendering is client-side rendering. With client-side rendering, the browser gets a bare-bones HTML document and Javascript file that is responsible for rendering the actual content. This way the browser does not need to request the whole page every time and instead the Javascript script will update affected parts. Client-side renderings achieved with Javascript frameworks. According to Monocubed15, the most popular front-end frameworks are Svelte, React.js, Vue.js, and Angular.js. This thesis author is familiar with Vue.js.

Secondly, according to the non-functional requirement JIT-NFR-3, the tool should be as lightweight as possible. Therefore visualization that is embedded into the communication module is not ideal as it adds additional dependencies to the users' system. Other possible solutions are to use server-side renderings like razor pages where each update requires reloading the page, hybrid solutions like razor pages with javascript components, and single page application that uses some Javascript frameworks.

Thirdly, separation and encapsulation are important. It should be easy to change the visualization. Due to that visualization that is embedded into some other module is not ideal. The visualization could be embedded into the communication module, embedded into the

[12] https://microsoft.github.io/language-server-protocol/
[13] https://www.statista.com/statistics/218089/global-market-share-of-windows-7/
[14] https://www.freecodecamp.org/news/what-exactly-is-client-side-rendering-and-hows-it-different-from-server-side-rendering-bd5c786b340d/
[15] https://www.monocubed.com/blog/best-front-end-frameworks/

API, or completely separate from other parts and use API endpoints to get the required information.

The fourth key point is the ease of future developments. Based on the author's experience as a full-stack developer the worst thing is unreadable code. Unreadable code slows the developing speed and the developer's motivation to continue working on the tool. Therefore the developed visualization technology should be developer-friendly. There are 3 potential rendering method possibilities. Using preexisting components library like Bootstrap-Vue, use SVG rendering library d3.js, or write the plain SVGs. All of those three methods are going to be described in detail in section 4.7.

The author has previously used c#, .net core framework, and Vue.js to develop web applications and thus the frontend framework is chosen based on the author‘s personal experience.

3.3.5 Development

The development follows some agile software development principles. There will be multiple small sprints. Based on the initial requirements the development will begin. At the beginning of the sprint, the requirements are enhanced based on the last sprint review. At the end of the sprint, there are going to be sprint reviews where this thesis author will showcase the results to the supervisor, and a discussion of what can be improved is taken place. The discussion results will be the basis for the next sprint requirements enhancement.

The developed tool code will be in four repositories. Repositories will be hosted on GitHub16.

3.3.6 Principles

Based on the author's experience the developed tool uses some principles to make the code more readable and maintainable in the future.

One of the most important principles is YAGNI. YAGNI is an abbreviation for You aren’t gonna need that. It is important because based on the author’s experience the unused code is a hassle for future developers. Therefore often it just confuses the future developer.

Secondly, the SOLID principles are used where appropriate. In SOLID the letters stand for the following [13]:

- S – single-responsibility principle,
- O – open-closed principle,
- L – Liskov substitution principle,
- I – interface segregation principle,
- D – dependency inversion principle.

The single-responsibility principle states that every class should have only one responsibility. Open-closed principle describes that the classes should be written such that they are open for extensions but do not need modifications. Doing this will decrease the chance that the developers adds new bugs to existing code. Liskov's substitution principle states that every subclass class should be substitutable for its parent class. The interface segregation principle states that you should not be forced to implement an interface you don’t use. The dependency inversion principle states that the entities must depend on abstractions, not concretions. This makes it easier to change the implementation of the interface.

[16] https://github.com/

3.4 Finding the visualization

The most important part of the tool is visualization as through that the end-user gets feedback for their code. The visualization of the tool is worked out with the help of art professor Kärt Summatavet and this thesis supervisor Kristiina Rahkema. In the first session, the individual brainstorming by author and supervisor Rahkema will take place. That produces two independent sketches that will be combined into one by taking the best parts from both sketches. K. Summatavet explained that one of the important parts of visualization is symbols. The symbols will be designed by the author to be memorable and interesting. This makes it easier for a user to interact with the visualization. Based on the sessions with the art professor and requirements the mockup of the visualization will be created.

3.5 Evaluation

The evaluation will concentrate on the completeness of the requirements and the speed of the program because this is the most critical part of the tool. The completeness of the requirements is evaluated continuously throughout the development and at the end of the development. The tool should start the analysis of the project as soon as possible. The tool can't be called a real-time application if the user has to start the analysis manually. According to the non-functional requirements JIT-NFR-1 and JIT-NFR-2, the analysis time of the project should be in 60 seconds, and the analysis of source code changes should start in a maximum of five seconds. The tool will be tested on various Java projects to measure its speed. With that, the question of can the tool considered a real-time source code analyzer will be answered. This is also a crucial part of how the tool can be presented to object-oriented programming course students in the future. If the tool is not fast enough it should be introduced as a tool that will help to learn programming and not as a real-time application.

4 Results

This section will describe the choices and decisions made during the development of the tool. The first section is focused on the requirements that were found during the development of the tool. Requirements were also prioritized. The second section showcases the mockup that was made to make it easier to understand the developed tool. The third section describes the architecture and technological choices of the tool. It mentioned also other possible architectural design choices that were considered. The fourth to seventh section describes in detail every module of the tool. The eighth section describes the deployment and installation of the tool. The ninth section showcases the achieved functionalities. And the last section describes the evaluation of the tool.

4.1 Requirements

The system is seen to be used by two types of actors. A person who will benefit the most from the tool is called a user. A user uses the tool to visualize the source code and interacts with it through IDE or web application. A user who will be responsible for setting the whole system up is called an advanced user. All actors can be seen in Table 2.

Table 2: Actors

Actor	Description
User	The regular user of the system. E.g., students,
Advanced user	The user who knows how to set up the whole tool for the end-users. Eg. instructors

4.1.1 Functional requirements

As described in section 3.1.1 the functional requirements were elicited based on the discussion with the supervisor, reading the papers on the topic “Difficulties of learning software development” that were read, and during sessions with the art professor K. Summatavet. A set of elicited functional requirements can be seen in Table 3. A full list of requirements can be seen in the appendix in Table 9. Each requirement has an ID to identify it and a description. Descriptions are written as user stories to have a better understanding of who is the user, what they want to do, and why is this important.

Table 3: Functional requirements

ID	Description
JIT-FR-1	As a user, I want to log in to the visualization, so that I can see only my projects.
JIT-FR-2	As a user, I want to add an access key to the JIT evolution plugin, so that the plugin can send data to the visualization.
JIT-FR-3	As a user, I want to use IntelliJ IDEA, so that I can use my favorite editor.

JIT-FR-4	As a user, I want to see the whole project visualization, so that I can get an overview of the project.
JIT-FR-9	As a user, I want to see a class detailed view, so that I result of the analysis.
JIT-FR-10	As a user, I want to see a method detailed view, so that I result of the analysis.
JIT-FR-14	As a user, I want to see which classes the method uses so that I can understand the code better.
JIT-FR-15	As a user, I want to see which class uses which classes so that I can understand the code better.
JIT-FR-16	As a user, I want to see variable access modifiers (private, public, protected), so that I understand the code better.
JIT-FR-17	As a user, I want to see methods' access modifiers, so that I understand the code better.
JIT-FR-20	As a user, I want to select the project, so that I can see the analysis of that project.

4.1.2 Non-functional requirements

Non-functional requirements were elicited based on the discussion with the thesis supervisor. A full list of elicited non-functional requirements can be seen in Table 4. This table also includes non-functional requirements listed in .

Table 4: Non-functional requirements

ID	Description
JIT-NFR-1	As a user, I want to analyze the project in under 60 seconds, so that I do not have to wait for results for a long time.
JIT-NFR-2	As a user, I want to start the analysis of the project in under 5 seconds, so that I don't have to start the analysis manually.
JIT-NFR-3	As a user, I want to the tool be as lightweight as possible, so that I can run it within IDE and the development environment.
JIT-NFR-4	As a user, I want to deploy visualization to the central position, so that end-users have fewer dependencies to install.

4.1.3 Requirements prioritization

Prioritization of the requirements is done using the MoSCoW technique. Every requirement is assigned one of the following priority groups: M – must have, S – should have, C – could have, or W – won't have. Both functional and non-functional requirements

are prioritized. Requirements that are key to the success of the tool are assigned to the "Must have" group. Those requirements are implemented in the first place. Should-have and could-have requirements are implemented after the must-have requirements have been implemented. Won't-have requirements are out of the scope of this thesis and are not implemented but might be implemented in the future. A set of prioritized requirements can be seen in Table 5. A full list of prioritized requirements can be seen in the appendix in Table 10.

Table 5: Requirements prioritized based on the MoSCoW method

Requirement ID	Priority
JIT-FR-1	Must have
JIT-FR-2	Must have
JIT-FR-3	Must have
JIT-FR-4	Must have
JIT-FR-9	Must have
JIT-FR-10	Must have
JIT-FR-14	Must have
JIT-FR-15	Must have
JIT-FR-16	Must have
JIT-FR-17	Must have
JIT-FR-20	Must have

4.2 Mockup

A mockup was made based on the functional requirements described in section 4.1.1. The mockup can be seen in Figure 8 and Figure 9. The goal of the mockup was to get a general idea of the visualization. The mockup visualizes the most important functionalities. The most important functionalities are chosen based on requirements prioritization. The mockup does not show how the tool looks in the end as not all requirements are used in the mockup. In Figure 8, the classes of the project are shown. Each class can contain methods and variables. Relationships between different classes and methods are visualized by arrow lines. Arrow lines represent methods and class uses. In Figure 9, the hovering functionality is displayed. A user has positioned the cursor on the instance variable and a popup of that class is shown.

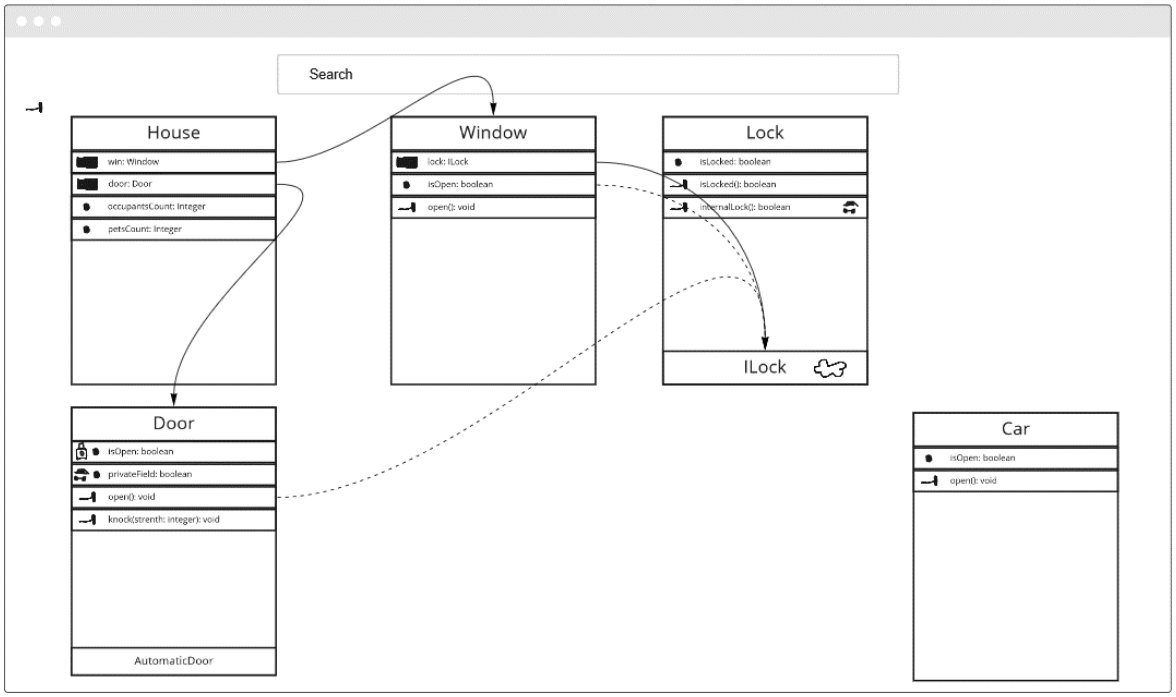

Figure 8: Screenshot of classes, methods, and variables

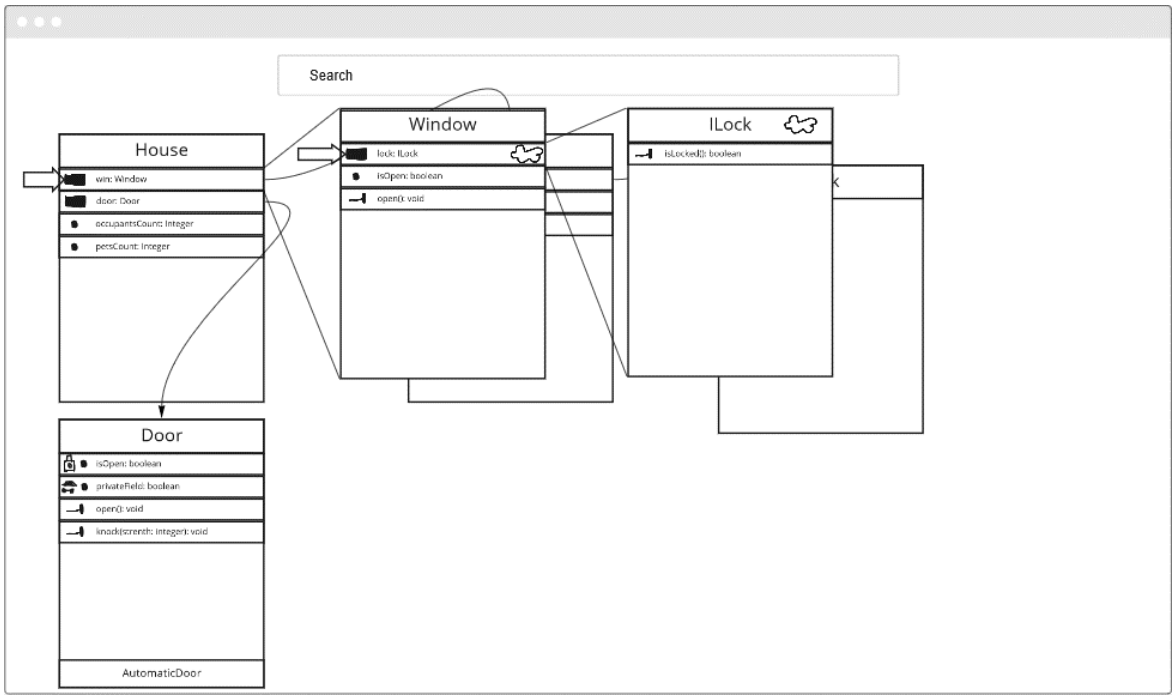

Figure 9: Screenshot of hovering instance variable

4.3 Architecture and technology

As mentioned in section 3.3, there are a couple of key points that should be forethought before the development of the tool. These are

- database,
- communication between the source code analyzer and an editor (IDE),
- source code analyzer,

- visualization.

After taking into account those key points possible architectural designs were drawn up. One of the designs is shown in Figure 10.

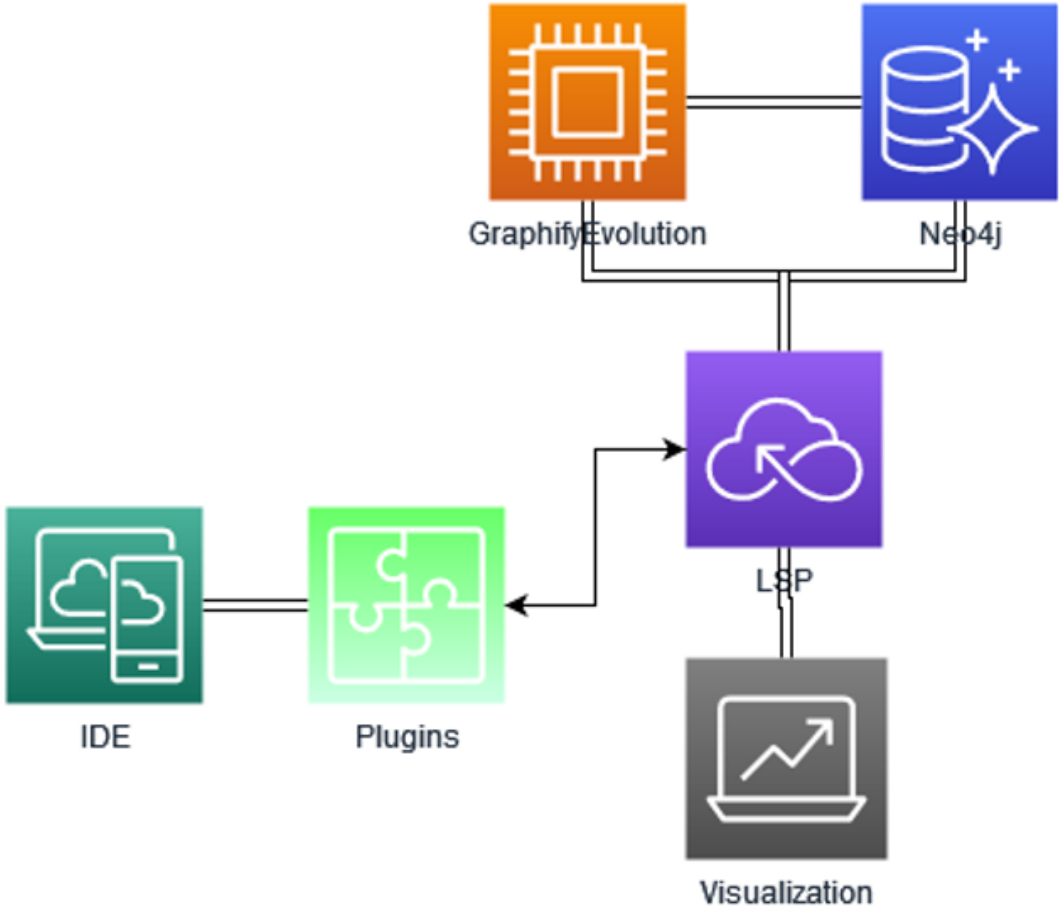

Figure 10: Single module architecture

In this architecture, the whole tool is one big module and the user should run all of it on their system. Since the tool users are expected to be students, the ease of installation of the tool is important. Also, the developed tool should be as lightweight as possible for the end-user. Shown design requires that the user have a Neo4J database running a runtime environment for Swift and Java as GraphifyEvolution needs those to analyze the code, and possibly more requirements depending on the implementation of the language server and visualization. Thus, this architecture design is not well suited because it will use various technologies that the user should install to their system.

As previously mentioned, GraphifyEvolution is implemented in swift programming language and uses some MacOS-specific libraries. Therefore it adds complexity to the installation of the tool. To reduce the complexity of the installation, some parts can be separated from the rest of the tool. Visualization, API, GraphifyEvolution, and Neo4j database are moved to a central position. On the user system then will be only a plugin for the editor and the language server. The language server will communicate with a central API. The chosen architecture can be seen in Figure 11

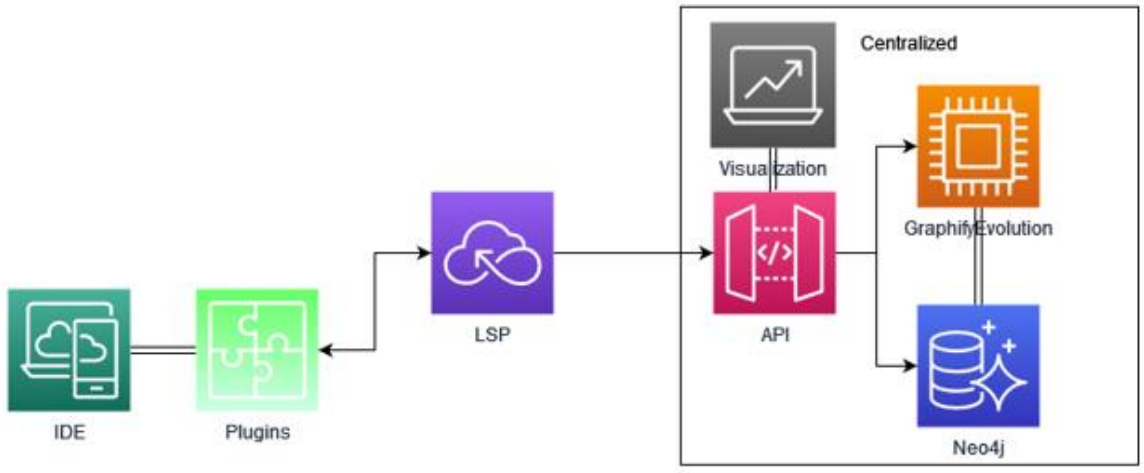

Figure 11: Chosen architecture

Based on Figure 11 the tool can be divided into 4 parts. Language server with plugins for editors, API, source code analyzer, and visualization. The next sections will describe those parts in detail.

4.4 Language server

The most important part is getting the source code changes from the user's editor to the centralized API and source code analyzer. The relevant part of the chosen architecture is shown in Figure 12.

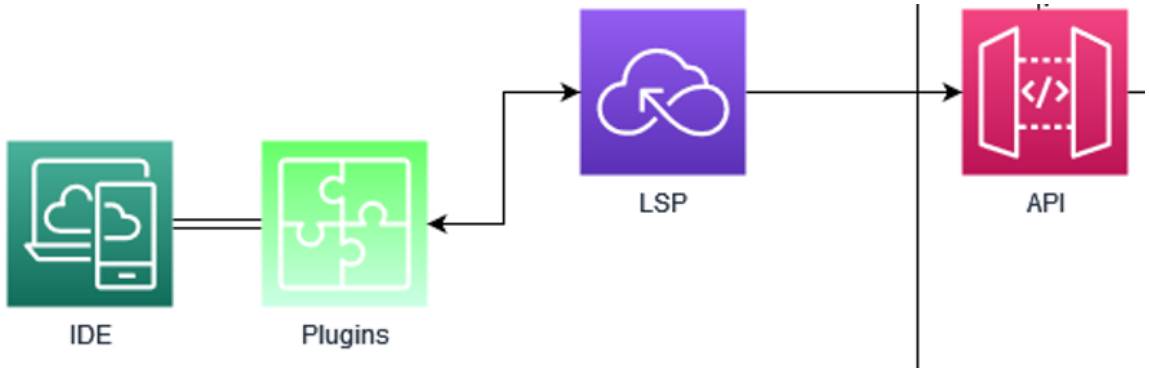

Figure 12: Communication between editor and centralized API

The communication between the source code analyzer and an IDE is achieved by creating a language server. The language server uses language server protocol (LSP)17. LSP was initially developed for Microsoft Visual Studio Code [14]. Microsoft collaborated with Red Hat and Codenvy and made the protocol public to standardize the protocol [14]. The protocol defines communication between a user's editor and the language server. The protocol can be used between client tools and language servers to integrate such features as autocomplete, find all, code lens, colors, and many more [14]. The flow of LSP is shown in Figure 13. Using LSP makes supporting an editor for a language easier as all the business logic is concentrated in one place - the language server. For each editor, that wants to support a new language, there is needed to create a small plugin that connects to the language server. There are a lot of IDEs that support the LSP. Based on the LSP GitHub page, there are over

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

30 editors that support the language server protocol18. According to the requirement JIT-FR-3, the developed tool should be used with the Intellij IDEA editor but the supported editors' list does not mention and officially the Intellij IDEA editor does not support LSP. Even though it is not officially supported the protocol is implemented by third-party developers as an extension of the Intellij IDEA19.

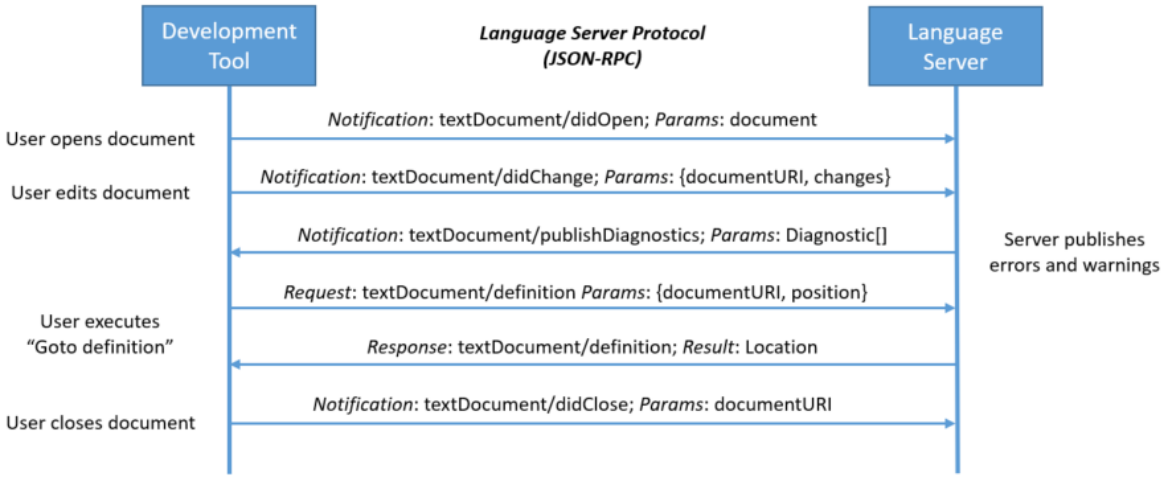

Figure 13: Example of the language server protocol20

The editor provides real-time information to the language server about the users' actions like changing code, opening files, requesting autocomplete, and many more. The information about where which files are open and what is changed is essential to the developed tool. As the visualization should show the changes in real-time.

That is the reason why the communication between the editor and the source code analyzer is achieved using the language server.

As a result, an executable of the language server was developed. Also, a plugin for the editor was made. As previously mentioned the language server can be written in various languages but according to the requirement JIT-NFR-3 the tool should be as lightweight as possible. The language server is usually developed in the same language as the language the server is designed for. As previously mentioned in chapter 1, the developed tool is developed with the idea to help students learn object-oriented concepts in Java. The object-oriented course uses Java as its programming language. Thus, the language server will be implemented in the Java language to minimize the dependencies of the tool as users already have Java installed.

The proper implementation of the language server protocol can be quite time-consuming. To fasten the development of the tool and not reinvent the wheel, the existing LSP implementation library was used. The chosen library was Eclipse's Lsp4J21. It was chosen because it is written in Java as it was decided to use Java for the language server, it allows customizing the functionality as needed, and there were tutorials for how to use it.

The language server and its plugins formed a git repository. The repository is hosted on GitHub22.

[18] https://github.com/Microsoft/language-server-protocol/blob/gh-pages/_implementors/tools.md
[19] https://plugins.jetbrains.com/plugin/10209-lsp-support
[20] https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/
[21] https://github.com/eclipse/lsp4j
[22] https://github.com/L1nde/JitEvolutionLSP

In Figure 14, the structure of the repository can be seen. Folder Plugins holds created plugins for editors. Two plugins were made. One for Microsoft's Visual Code that was used during the development of the tool, and one for IntelliJ IDEA that was required by functional requirement JIT-FR-3. The folder Server contains two subfolders: launcher and jitevolution. The launcher contains a class that is responsible for starting the language server. The folder jitevolution is divided into two subfolders. The core folder holds the core business logic for the language server like domain models, interfaces, etc for the tool, and the folder services holds the implementation for the interfaces. This gives a clear separation between abstractions and implementations. The core components are definitions of the objects. This allows easy swap-out implementations of the interfaces as other code does not need to change because the definition for the class stays the same.

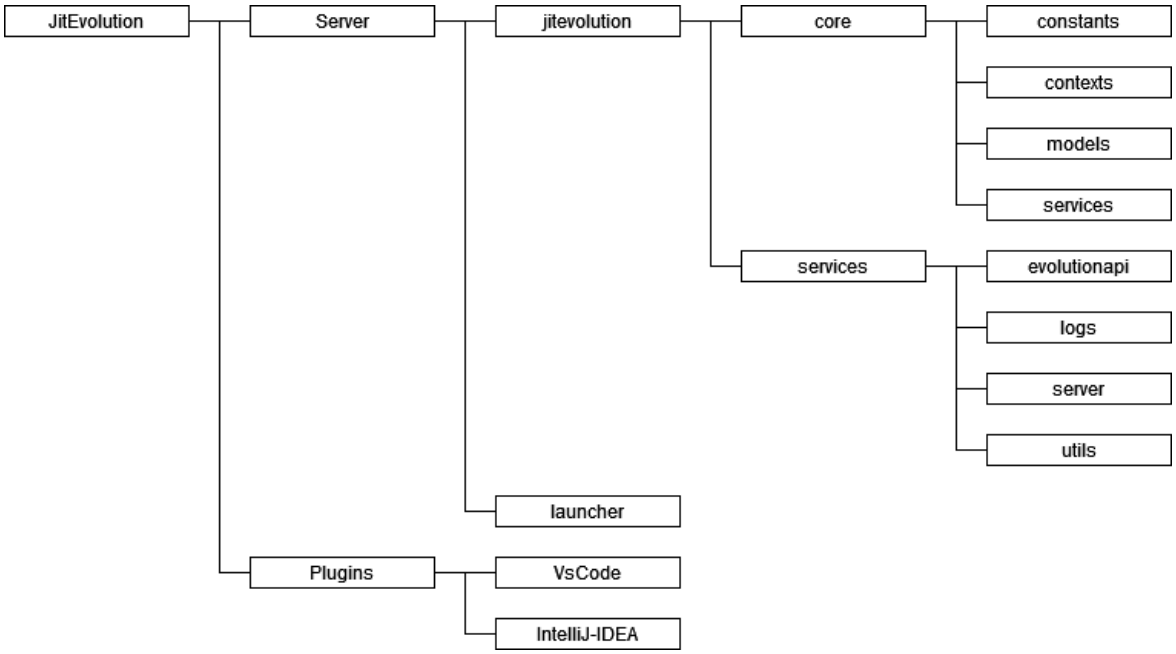

Figure 14: Structure of language server project

The Lsp4J library provides interfaces that have definitions for available language server protocol methods that can be implemented. The most important method is didSave(DidSaveTextDocumentParams didSaveTextDocumentParams) that can be seen in Figure 15. During the discussion with the supervisor, it was decided that a new version of the project should be made when a user saves a file. That way the language server would not analyze typos and incomplete changes so much because the user usually saves after they have finished changing that part of the code.

```
@Override
public void didSave(DidSaveTextDocumentParams didSaveTextDocumentParams) {
   try {
      var projectId = ProjectConfigs.ensureProjectConfigs();
      var zip = TextDocumentExtensions.zipProject(context.getConfigura-
tion().getFileExtension());
      context.getEvolutionApi().notifyFileSaved(didSaveTextDocumentParams.get-
TextDocument().getUri(), zip);
      zip.delete();
   } catch (Exception e) {
      context.getLogger().logError("Unable to zip project");
   }
}
```

Figure 15: didSave(…) method in the language server

This method uses the utility method ProjectConfigs.ensureProjectConfigs() that ensures that the project has a unique id. The unique id is needed to tie a project with a certain user and afterward show correct projects to that user. The JitEvolution configurations are saved to the file .jit-evolution/config.json. The content is in JSON format to allow easy change of project id and adding new configuration values. An example of a configuration file can be seen in Figure 16.

```
{
   "projectId": "Carcassonne-ce0f0e7e-b585-4038-aaf4-f7e3bf754f3f1"
}
```

Figure 16: config.json file

In addition to ensuring that the project has a unique id, the method didSave(…) adds all the files to the archive which extension equals the extension given during launching the server with argument. The archive is sent to the JitEvolution API that is responsible for the analysis and visualization of the project. The decision was made that the language server will always send the whole project because of the encountered limitation of the source code analyzer. The encountered limitations will be described in detail in section 5.1.

Requirement JIT-FR-23 says that the visualization should show the user a class that is being changed. To implement that another Lsp4J set of methods was implemented. These were didOpen(DidOpenTextDocumentParams didOpenTextDocumentParams) and documentColor(DocumentColorParams params). The need for implementation of the first method didOpen(…) is self-explanatory. When a user opens the file then the language server will send the notification to the API that this user has opened the following file. Implementation of documentColor(…) acts as a fail-safe for the previous method. During the testing, it was found out that in the Visual Code editor, the didOpen(…) method is invoked only if this file was not previously opened. The documentColor(…) is invoked every time the file is shown to the user because the intended purpose of that method is to color a language syntax with it.

Only the language server is on the user's system. Therefore a method to communicate with the rest of the tool was needed. The communication is done using hypertext transfer protocol (HTTP) with JSON body or form data. This was chosen based on the author's personal experience and the ease of implementing it. In Figure 17 the interface for the class that is responsible for communication with the rest of the tool is shown.

```
public interface JitEvolutionApi {
    void notifyFileOpened(TextDocumentIdentifier textDocumentIdentifier);
    void notifyFileOpened(TextDocumentItem item);
    void createProject(String projectId, File project);
    void notifyFileChanged(TextDocumentIdentifier textDocumentIdentifier);
    void notifyFileSaved(String fileUri, File project);
}
```

Figure 17: JitEvolutionApi interface

According to the requirement JIT-FR-3, the main editor for the tool is IntelliJ IDEA but the development of the tool was made using the Microsoft's Visual Studio Code editor because the LSP is not officially supported by the makers of IntelliJ IDEA and there are more tutorials on how to create a language server for Visual Code. In addition, the testing during the development of the tool would have been overly complicated as it was not possible to properly debug the language server with IntelliJ IDEA. Afterward, when requirements were implemented it was made sure that the tool also works with IntelliJ IDEA.

The language server was designed in such a way it supports using SOLID principles. The single-responsibility principle states that every class should have only one responsibility. The example of the single-responsibility principle can be seen in Figure 18. The LanguageClientLogger's only job is to send a message to the language client i.e to the user's editor.

```
public class LanguageClientLogger implements Logger {
    private final LanguageClient languageClient;

    public LanguageClientLogger(LanguageClient languageClient) {
        this.languageClient = languageClient;
    }

    @Override
    public void log(String message) {
        languageClient.logMessage(new MessageParams(MessageType.Info, message));
    }

    @Override
    public void logWarning(String message) {
        languageClient.logMessage(new MessageParams(MessageType.Warning, mes-
sage));
    }

    @Override
    public void logError(String message) {
        languageClient.logMessage(new MessageParams(MessageType.Error, mes-
sage));
    }
}
```

Figure 18: Example of single-responsibility principle

The dependency inversion principle states that the entities must depend on abstractions, not concretions. This makes it easier to change the implementation of the interface. An example of usage can be seen in Figure 18 as the LanguageClientLogger's constructor takes one parameter which is an interface. Therefore the developer can change the implementation of the LanguageClient without changing the LanguageClientLogger class. This also helps to apply the open-closed principle.

4.5 Source code analyzer

The developed tool will use the GraphifyEvolution application as the source code analyzer. The relevant part of the chosen architecture is shown in Figure 19.

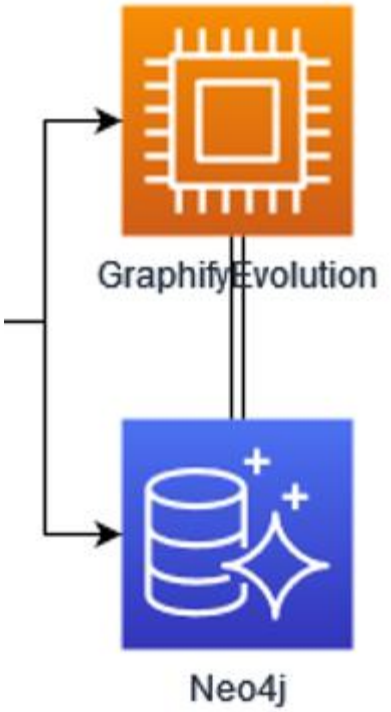

Figure 19: Source code analyzer in the architectural diagram

As previously mentioned the source code analyzer is written in Swift language and needed macOS to be able to run. This complicates the development of the tool because developers need to have a system with macOS. Also according to the Amazon EC2 pricing23, running a macOS machine costs over two times more than running a Linux machine. So limiting the tool to only macOS machines reduces the chances of successful uses of the tool and a future development possibility. To combat that support for the Linux system was added. The GraphifyEvolution depended on heavily on the Swift's Foundation library. At the start of the GraphifyEvolution creation, there was no available Foundation library on Linux. Thus it was not possible to run GraphifyEvolution on Linux. But now there is a library for Linux. On macOS, the Foundation is one big library but on Linux, it is a couple of small libraries. By updating the versions of the dependencies of the GraphifyEvolution and conditionally importing required Foundation sublibraries, support for the Linux systems was achieved.

As mentioned in chapter 2.2, GraphifyEvolution uses the Neo4j database to store the results of the analysis. This adds complexity to the installation of the tool. As previously decided the source code analyzer and Neo4J database will be deployed to the centralized location. This removes the installation complexity from users as only advanced users need to set up the source code analyzer and Neo4J database. To reduce the complexity for future developers and advanced users, the source code analyzer is put inside a container. One of the most popular container software is Docker2425. Based on the author's personal experience and the popularity of the Docker software, the source code analyzer will be deployed to the Docker container. Another advantage of using a container system is that the developers can preconfigure everything making it easier for the advanced user to deploy it to the centralized server.

[23] https://aws.amazon.com/ec2/pricing/on-demand/
[24] https://www.softwaretestinghelp.com/container-software/
[25] https://www.docker.com/

During the development of the tool, some difficulties regarding the chosen source code analyzer were found. Firstly, GraphifyEvolution is good at analyzing the whole project but not great at analyzing it incrementally. The whole project analysis suits well for the first analysis of the project where according to requirement JIT-NFR-1 there is more time to analyze it. But the main point of the developed tool is that it tries to analyze the changes in real-time. Thus, the analysis should be incremental. That means the source code analyzer should analyze the project and relate it to the previous version of the project and ideally take as an input only changes of code and based on that produce an analysis result. GraphifyEvolution did not do either of those things. GraphifyEvolution analyzed the source code once and was done with it. There was no possibility to continue analyzing the previously analyzed project. To resolve that the fork of the GraphifyEvolution was made that consisted JitEvolution specific changes26. Without the modification, the flow that was possible is shown in Figure 20. GraphifyEvolution was closely related to the Neo4J database. First, it created an object in the Neo4J database and then started filling its fields with information. After everything was analyzed the analyzer finished and by that the API knew that the analysis was successful.

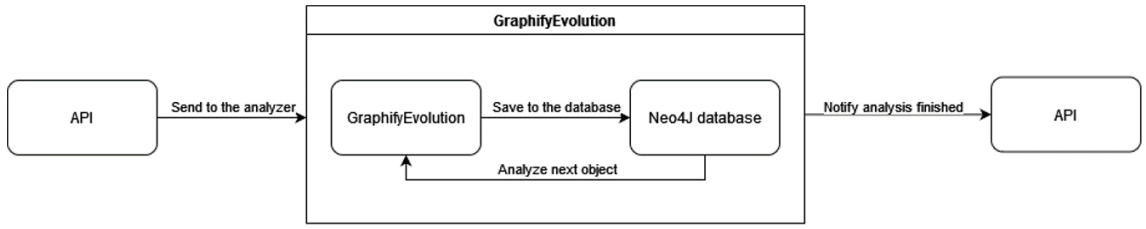

Figure 20: GraphifyEvolution before modifications

This kind of flow is not suitable for the JitEvolution tool because as previously mentioned GraphifyEvolution only analyses the whole project and does not allow continuously analyzing the existing project. Thus, the changes to GraphifyEvolution were made. Initially, GraphifyEvolution used a concrete class Neo4JClient that was responsible for communication with the Neo4J database. From that class, a new interface was derived. It was DatabaseClient. It defined methods that were needed by the rest of the application. In Figure 21, the DatabaseClient can be seen.

```
protocol DatabaseClient {
    func createAndReturnNodeSync(node: Node) -> Node?
    func requestNodeSync(label: String, properties: [String:String]) -> Int?
    func mergeNodeSync(node:Node) -> Node?
    func updateNodeSync(node: Node)
    func relateInParallel(node: Node, to: [Node], type: String)
    func relateInParallel(node: Node, to: [Node], type: String, batchSize: Int)
    func relateSync(node: Node, to: Node, relationship: Neo4jRelationship)
    func runQuery(transaction: String)
    func mergeDuplicates()
}
```

Figure 21: DatabaseClient interface

Deriving the interface allowed easy change of implementation of DatabaseClient. A new implementation was added – ApiClient. The ApiClient communicated over HTTP with a specified API endpoint. This modification allowed the use of the shown flow in Figure 22. Advantages of this flow are the following:

1. The API can choose how it will save the information to the database.

[26] https://github.com/L1nde/GraphifyEvolution

2. The API can decide itself if it needs to save the received information to the database.
3. A developer that is not experienced with Swift language can use other programming languages as the API's language is not restricted to anything.
4. Add additional processing of the information before saving it to the database.
5. Support for incremental analysis of the project.

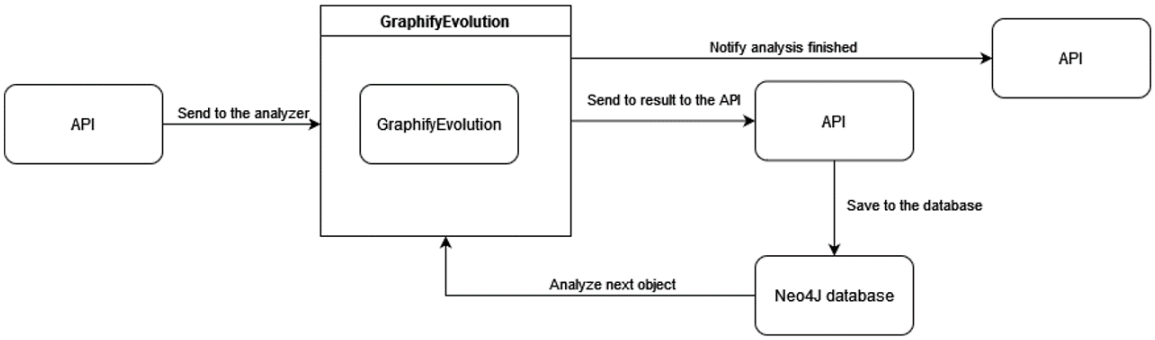

Figure 22: GraphifyEvolution after modifications

Using this flow, it was experimented with analyzing only a single file that the user changed. This caused a couple of different problems. GraphifyEvolution did not have support to analyze only one file of the project. Also, relationships between classes and methods were broken because proper analysis of the relationships required analysis of the whole project. It was tried to implement that but the complexity of the implementation increased constantly, so it was decided to use a different method to analyze the project.

The chosen method was that each time the whole project is analyzed. This ensured that the relationships between objects were correct. This also made it easier to relate the new analysis result to the existing project. As GraphifyEvolution itself does not relate the new analysis result to the existing analysis, it was done on the API side. After the analysis is done by the source code analyzer, the source code analyzer sends one final request to the API to notify that it can relate now to the existing project.

Relating to the existing project uses similar logic as it would have been when it was analyzed solely with GraphigyEvolution. A relationship CHANGED_TO was added between the changed object and the existing object. The following rules describe the logic of how the CHANGED_TO relationship was added:

1. Every analysis will create a new version of the app i.e a new app object is created that is related to the previous version by the CHANGED_TO relationship.
2. A class is changed when any of its properties are changed.
3. A method is changed when any of its properties are changed or when any of its incoming relationships are changed.
4. A variable is changed when its properties are changed
5. A parameter change is out of scope.
6. An external object change is out of scope.

Parameters and external object change are out of scope because based on the requirements of the developed tool those do not give anything extra. An example of an analyzed project can be seen in Figure 23. The project with two classes is analyzed twice. Pink circles are versions of the project, orange circles are classes, red circles are methods, and blue circles are variables.

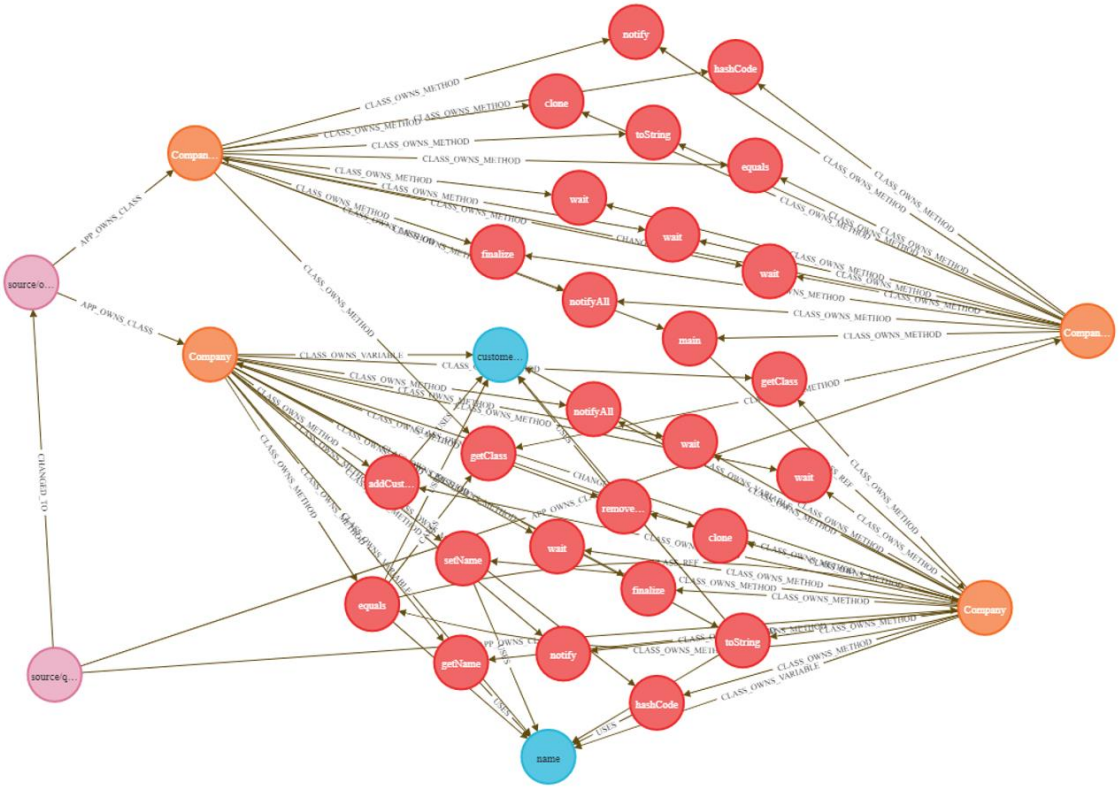

Figure 23: Example of an analyzed project in the Neo4J database

4.6 API

One of the most important parts of the developed tool is the API. The API binds everything together. It provides endpoints for the language server, starts the analysis of the changed code, handles authentication, and endpoints for the visualization. Because the API is a separate module, the programming language is not constrained by anything. Based on the author's previous experience in developing websites using the .NET Core framework27, the visualization module will use the C# language and the .NET Core framework. .NET Core provides excellent tools for developing web APIs.

Like the other parts of the developed tool, separation and encapsulation were important. Also, the SOLID principles and YAGNI were kept in mind when developing the API.

The API is designed as a RESTful API. REST stands for Representational State Transfer [15]. It defines six architectural constraints which should be followed. These constraints are

- uniform interface,
- client-server,
- stateless,
- cacheable,
- layered system,
- code on demand.

As the name of the uniform interface constraint states, the resources inside the system should be exposed using the same interface. Client-server states that the API consumer and server must be able to evolve separately. Stateless - all requests are stateless i.e. the

[27] https://dotnet.microsoft.com/apps/aspnet

API does not store anything. Everything that is needed like authentication and authorization is included in the request. Cacheable – resources should be cacheable to make the request faster. Layered system – It is possible to deploy different parts of the API to different machines. Eg. Database is in Server A and API itself is in Server B. Code on demand – some of the endpoints may return executable code.

The file structure of the API is shown in Figure 24. In C# the root of the project is called a solution. A solution contains one or many modules (in C# they are called projects).

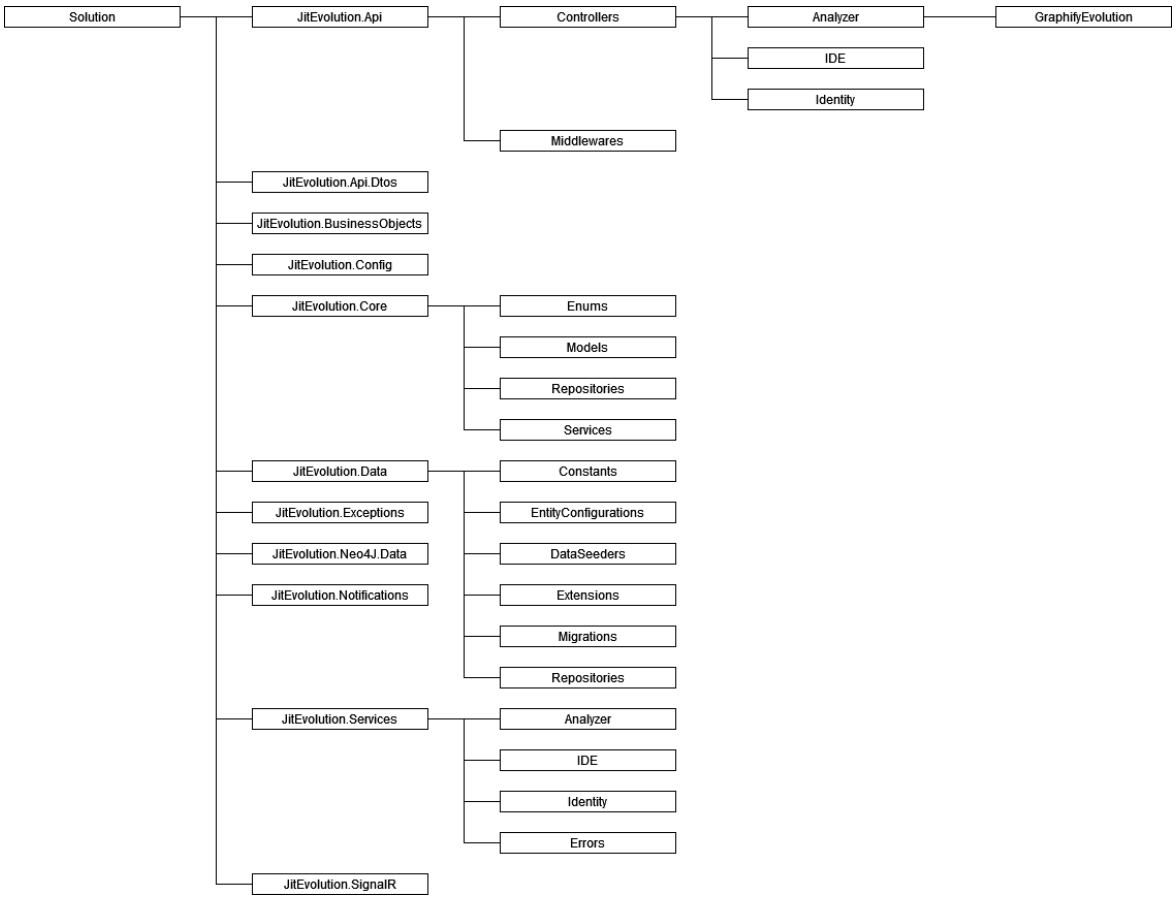

Figure 24: API's file structure

The module JitEvolution.Api binds all the modules together. The folder Controllers holds classes that provide endpoints for the visualization and language server. There are three folders: Analyzer, IDE, and Identity. Folder Identity contains endpoints that are related to users i.e. authentication, user creation, etc. Folder IDE contains endpoints for the language server. E.g FileChangedController, FileOpenedController, FileSavedController, and ProjectController. An example of the controller method can be seen in Figure 25.

```
[HttpPost]
public async Task<IActionResult> FileSaved([FromForm] FileSavedDto dto)
{
    ValidateZipFile(dto.ProjectZip);

    await _projectService.CreateOrUpdateAsync(dto.ProjectId, dto.ProjectZip);

    await _mediator.Publish(new ProjectUpdated());

    return Ok();
}
```

Figure 25: Example of a controller method

Module JitEvolution.Api.Dtos contains data transfer objects that are exposed to the API consumers. Module JitEvolution.BusinessObjects contains data transfer objects that are used internally and should not be exposed to the API consumer. Module JitEvoltion.Core holds the core business logic for the API like domain models, interfaces, etc. Module JitEvolution.Data contains implementation for the repositories and connection to the Postgres database. The API uses a code-first workflow[28] to evolve the database schema. It means that the developer writes the database tables as classes and uses a tool to create a database migration. The tool that is used is Entity Framework Core Migrations[29]. That way the developer only needs basic knowledge about SQL to verify that the automatically generated code is correct. Generated migrations are stored in the Migrations folder. An example of the generated code can be seen in Figure 26.

```
public partial class ProjectEntity : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "Project",
            schema: "JitEvolution",
            columns: table => new
            {
                Id = table.Column<Guid>(type: "uuid", nullable: false),
                UserId = table.Column<Guid>(type: "uuid", nullable: false),
                ProjectId = table.Column<string>(type: "text", nullable: false),
                CreatedAt = table.Column<DateTime>(type: "timestamp with time
zone", nullable: false),
                CreatedById = table.Column<Guid>(type: "uuid", nullable: false),
                ChangedAt = table.Column<DateTime>(type: "timestamp with time
zone", nullable: false),
                ChangedById = table.Column<Guid>(type: "uuid", nullable: false),
                DeletedAt = table.Column<DateTime>(type: "timestamp with time
zone", nullable: true),
                DeletedById = table.Column<Guid>(type: "uuid", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Project", x => x.Id);
                table.ForeignKey(
                    name: "FK_Project_AspNetUsers_ChangedById",
                    column: x => x.ChangedById,
                    principalSchema: "JitEvolution",
                    principalTable: "AspNetUsers",
                    principalColumn: "Id",
                    onDelete: ReferentialAction.Cascade);

. . .
```

[28] https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/migrations/
[29] https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli

```
            });

      . . .

        migrationBuilder.CreateIndex(
            name: "IX_Project_UserId_ProjectId",
            schema: "JitEvolution",
            table: "Project",
            columns: new[] { "UserId", "ProjectId" });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Project",
            schema: "JitEvolution");
    }
}
```

Figure 26: Example of a database migration

Module JitEvolution.Neo4J.Data contains the implementation for the Neo4J database connection. Module JitEvolution.Notification contains event messages that are used throughout the API to make different parts of the API encapsulated. E.g. JitEvolution.Services module should not know anything about how to communicate with visualization over WebSocket. The module JitEvolution.SignalR is responsible for the WebSocket connection.

The API has connections to two databases. First, the source code analyzer database is Neo4J, and another relational database is Postgres for authentication and project management. Accessing the database is done through a repository pattern [16]. The repository pattern is used to encapsulate the logic required to access the database. That way all the data access logic is in one place. In Figure 27

```
internal class ClassRepository : NodeRepository<Class>, IClassRepository
{
    public ClassRepository(IGraphClient graphClient) : base(graphClient)
    {
    }

    public override NodeLabelEnum Label => NodeLabelEnum.Class;

    public async Task<IEnumerable<Result<Class>>> GetAllAsync(long appId,
string? filter = null)
    {
        var query = (await ClientAsync()).Cypher
            .Match("(app:App)-[:APP_OWNS_CLASS]->(class1:Class)")
            .Where($"not(app)-[:CHANGED_TO]->(:App)")
            .AndWhere($"ID(app) = {appId}");

        if (!string.IsNullOrWhiteSpace(filter))
        {
            query.AndWhere(filter);
        }

        var model = await query
            .Return(class1 => new { Id = class1.Id(), Data = class1.As<Class>()
})
            .ResultsAsync;

        return model?.Select(x => new Result<Class>(x.Id, x.Data)) ?? Enume-
rable.Empty<Result<Class>>();
    }

    public async Task<Result<Class>?> GetByUsrAsync(string projectId, string
usr)
    {
```

```
        var model = await (await ClientAsync()).Cypher
            .Match("(app:App)-[:APP_OWNS_CLASS]->(class1:Class)")
            .Where($"not(app)-[:CHANGED_TO]->(:App)")
            .AndWhere($"not(class1)-[:CHANGED_TO]->(:Class)")
            .AndWhere($"app.appKey = '{projectId}'")
            .AndWhere($"class1.usr = '{usr}'")
            .Return(class1 => new { Id = class1.Id(), Data = class1.As<Class>()
})
            .ResultsAsync;

        var item = model.FirstOrDefault();

        return item != null ? new Result<Class>(item.Id, item.Data) : null;
    }

    public async Task<Result<Class>?> GetAsync(long id)
    {
        var model = await (await ClientAsync()).Cypher
            .Match("(class1:Class)")
            .Where($"ID(class1) = {id}")
            .Return(class1 => new { Id = class1.Id(), Data = class1.As<Class>()
})
            .ResultsAsync;

        var item = model.FirstOrDefault();

        return item != null ? new Result<Class>(item.Id, item.Data) : null;
    }
}
```

Figure 27: Example of a repository pattern

All the API endpoints have minimal documentation in the form of a swagger30. The swagger is an interface description language. It describes the endpoints, the request parameters, and the body. The API documentation can be seen in Figure 28. The swagger documentation is also used by the visualization module to generate API data models in the Vue.js app.

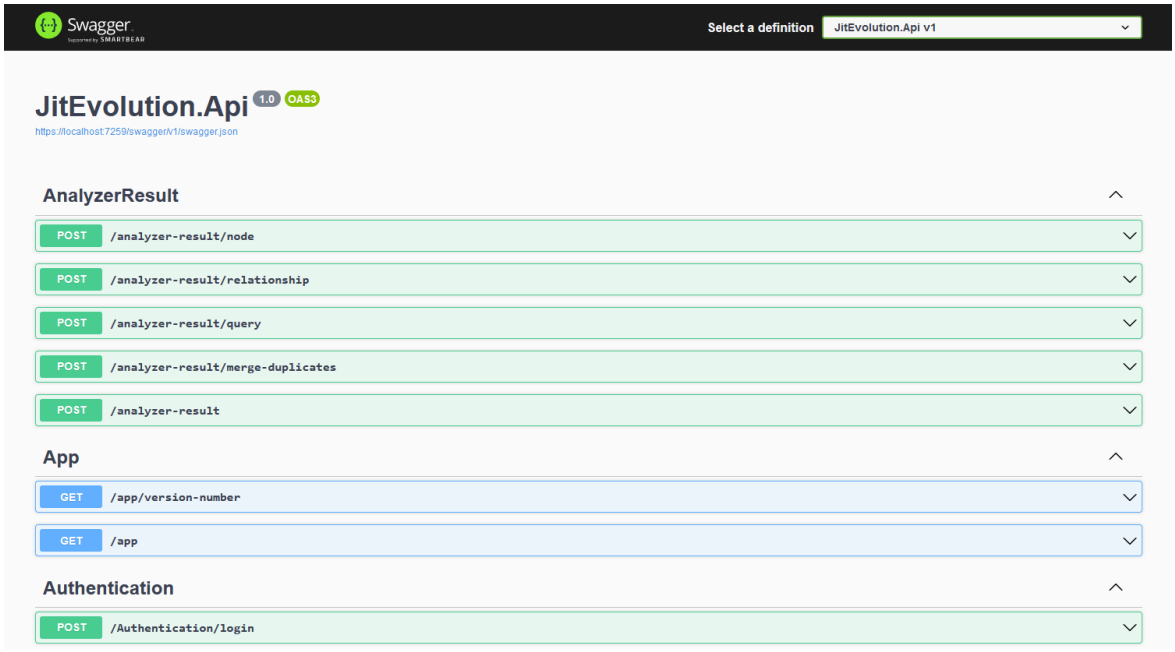

Figure 28: Swagger documentation

[30] https://swagger.io/

The API is a key module of the tool because every other module of the developed tool depends on it. The analysis starts when an editor notifies the language server that the user saved a file. The language server does not care if anything was changed. Doing it this way gave an interesting side effect that the visualization can show how frequently users save the project. The language server zips together the project files and notifies the API that a file was saved. Then the API sends the project to GraphifyEvolution for analysis. When analysis ends then API relates the analysis result to the existing project version and notifies the visualization over WebSocket31 that there is a new version of the project. The WebSocket provides 2-way communication between API and a visualization. Traditional HTTP protocol allows only one-way communication. I.e. Only a visualization can send a request to the API but the server can't notify the visualization that something changed and it should request data again from the API. The WebSocket enables that. After the API notifies visualization that the project was analyzed, it ends a request from the language server with a successful status code that the analysis ended. The flow can be seen in Figure 29.

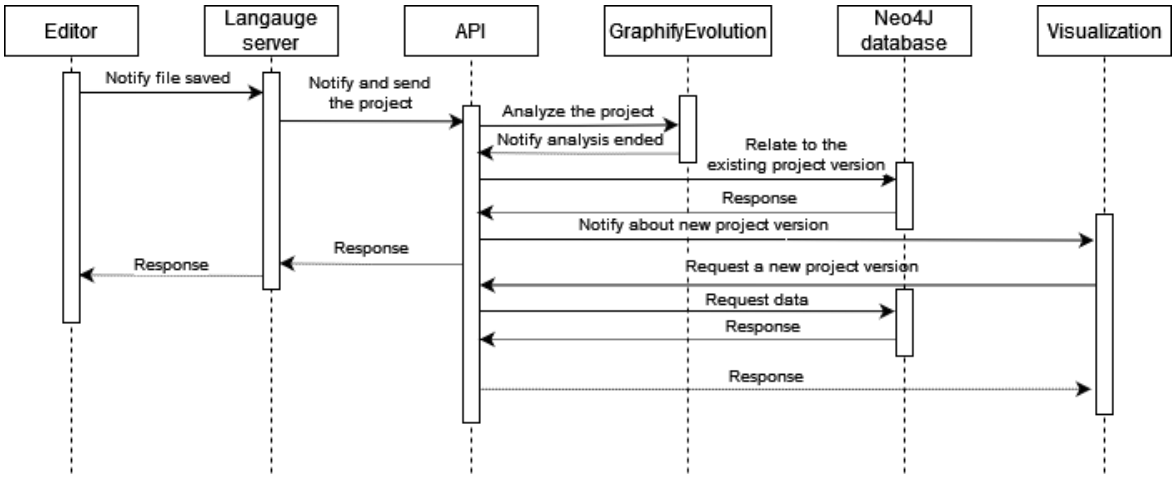

Figure 29: The flow of the change

During testing, it was discovered that the editor and language server set some limits for this flow. The language server was able to handle only one request at a time. That means the language server stopped processing other changes until the API finished a request. Because the language server waited for the API to finish the request and it caused a large amount of lag. Initial testing showed that project analysis took about 30 seconds. So the changes could have been analyzed once every 30 seconds. In Microsoft's Visual Code new file save notification during the analysis of the previous version was put on hold and sent after the API finished the previous request. Meanwhile, the user was able to continue changing the code. But with IntelliJ IDEA the editor froze and was not responsive until the API ended the request. This kind of behavior was not suitable for real-time application and the flow was redesigned to use a queue of analysis requests. Overall the flow is the same as in Figure 29 but the API does not analyze the project right away. Instead, it inserts into the queue and processes the queue asynchronously. Each project has its own ordered queue because the changes have to be analyzed on a first-in-first-out basis. Multiple projects can be analyzed at the same time because projects do not depend on each other. This way the API responds fast enough so that JitEvolution does not interrupt the developer and analysis is done in the background. The updated flow can be seen in Figure 30.

[31] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

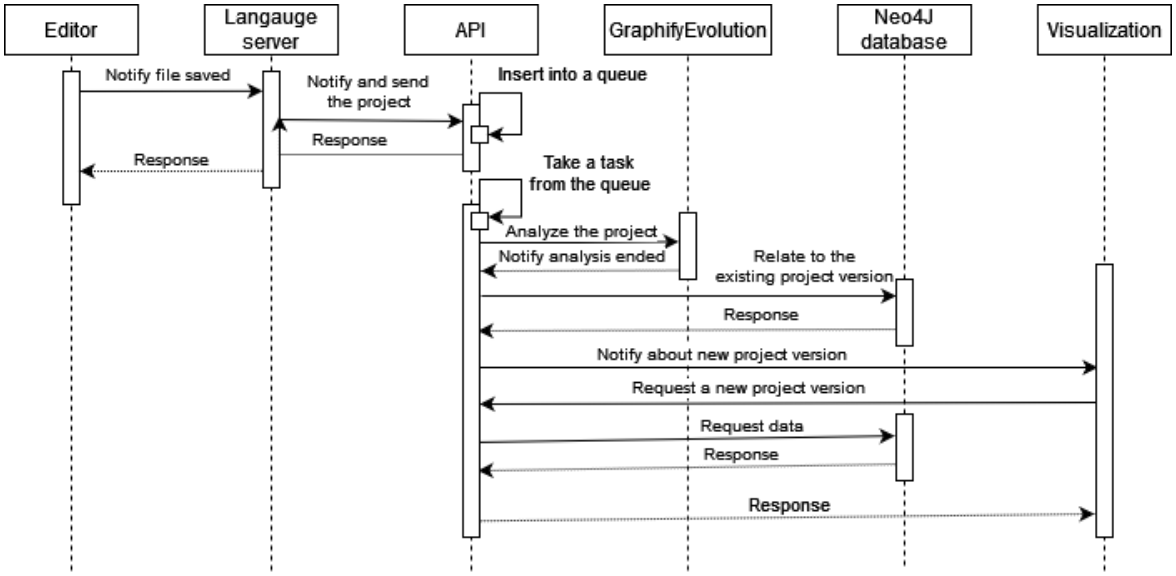

Figure 30: The modified flow of change

4.7 Visualization

The visualization module is separated from the language server and the source code analyzer. The relevant part of the chosen architecture design is shown in Figure 31.

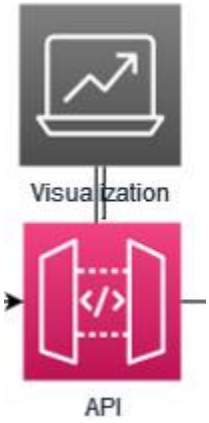

Figure 31: Visualization in the architectural design diagram

Because the visualization is a separate module, the programming language isn't constrained by anything. Based on the author's previous experience in developing websites, the chosen programming language was Javascript and Vue.js framework. The Vue.js framework was chosen because the visualization should be interactive and choosing the framework that allowed that was crucial. Javascript itself is known to be a weakly typed programming language. That means that Javascript itself will make assumptions about the types of the variables. Weak typing makes early detection of typos and errors harder. Also, in the author's experience, it decreases the chance of future development as weakly typed language code will be more likely unreadable. Due to that the Typescript library was used. Typescript adds strong typing support to the Javascript.

The file structure of the visualization repository is shown in Figure 32. The root of the project consists of two main folders: src and utils. The utils folder contains a script that generates types for Typescript based on the API's documentation. The src folder contains created Vue files that are divided into subfolders. The folder api contains functions that are

responsible for communication with created API. Folder assets holds static assets like images, SCSS files, etc. Folder components contains different components that are used in the visualization. Folder views on the other hand holds the basis of the pages that use different components. One such view is a login page.

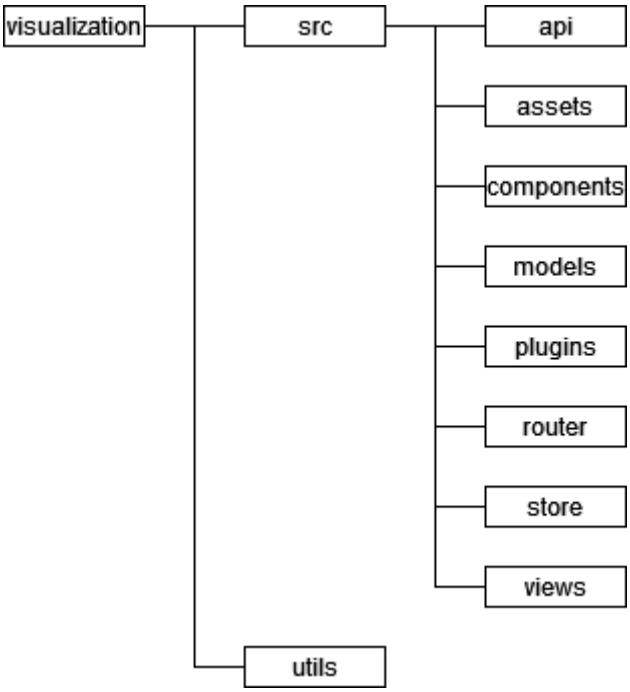

Figure 32: Visualization project file structure

For easier communication between created Vue.js components, Vuex was used. Vuex is a state management pattern and a library [17]. It provides a single source of truth32 that can be accessed throughout the application. Vuex was chosen because at the start of the development the Vuex was recommended33 library for state management and the author had a previous experience with it. At the end of the development, the recommended state management library changed to Pinia34 which would have been a strong candidate.

The general view of the visualization, like the navigation bar, login page, etc, uses the Vuetify components library35. It was selected based on the author's personal preference and other frameworks were not considered. BootstrapVue was used to simplify the development and for a better user interface. No other component libraries were considered.

The main view was achieved by using SVG tags in the Vue template. It was chosen by experimenting with three different methods. The very first method was using the previously mentioned BootstrapVue component library. The advantages of this method were that it provided components with smooth animation that was easy to use, and that had good documentation. The problems with this method came out during the visualization of the relationships. The relationships between method and variables were tried to visualize using SVG line tags. BootstrapVue provides responsive components and using it with SVG tags turned out to be complicated. The main reason for that was that the relationships had to be rendered after other components like classes, and methods to get actual coordinates for the SVG tags. In the author of this thesis's opinion, the produced code was not readable.

[32] https://www.mulesoft.com/resources/esb/what-is-single-source-of-truth-ssot
[33] https://vuex.vuejs.org/
[34] https://pinia.vuejs.org/
[35] https://vuetifyjs.com/en/

Also, the visual of this kind of composition method was not very pretty as it looked out of place. An example of this visualization can be seen in Figure 33 Thus, alternative methods were looked for.

Figure 33: Example of the visualization using Vuetify

The second method in the visualization experiments was based on a master thesis chapter by Miron Storožev explaining the exploration techniques to visualize source quality. [5]. It used the D3.js36 library for visualization. D3.js has good documentation for Javascript but not so good documentation for Vue.js and typescript. D3.js solves the problem that classes and relationships are drawn differently and looked out of place because everything is being drawn the same way. Also, getting coordinates for the relationships is solved with this method because the drawn objects are not responsive and all the coordinates and lengths of the objects are calculated by the developer code. In the big picture, d3.js is a set of functions to generate SVG tags programmatically. That means the d3.js in the Vue.js does not fully take advantage of the features of the Vue.js. Vue.js has templates i.e HTML code, scripts, and CSS parts. The Vue.js detects if it needs to rerender the template in case something has changed. Due to the d3.js using mainly scripts part of the Vue.js template it did not utilize it fully. Also in the author's opinion, the code that is written with the d3.js becomes unreadable and hard to maintain because encapsulation is complicated with d3.js and Vue.js.

The third and final method was to exclude d3.js and write SVG tags. In addition to the problems that d3.js solved compared to the first method, the third method also fully utilizes the Vue.js, and encapsulation is achieved easily. A full comparison of the three methods can be seen in Table 6.

[36] https://d3js.org/

Table 6: Comparison of experimented methods

	BootsrapVue	D3.js	SVG
Documentation	Excellent	Good	Good. Must have some knowledge of how SVG works.
Ease of use	Good. There are a lot of examples	A little bit harder. There are a lot of examples but does not utilize all the Vue.js features easily.	Easy if have basic knowledge about SVG.
Vue.js support	Excellent. It is a Vue.js component library	Poor. Documentation on the Vue.js part is poor and there are few examples.	Excellent. Uses basic HTML tags that the developer can use as they need them.
Animations	The components have animations and adding additional animations is easy.	Many animations and implementation are easy.	Complicated. The developer must create all the animations by themselves.
Community	Large	Quite large	Large.
Browser support	All the modern browsers. Does not support older versions of IE	All the modern browsers. Does not support older versions of IE	All the modern browsers. Does not support older versions of IE
Modernity	Created in 2016, last updated April 2022	Created in 2016, last updated April 2022	Created in 1998, last updated June 2021

Based on experiments and the result of the comparison, the third method was chosen.

4.8 Deployment

As mentioned the developed tool will have a couple of different modules. Some of them should be deployed to the users' computer, others to the central server. The exact deployment scheme can be seen in Figure 34. A language server should be installed on the users' computer and other parts should be installed on the central server. Databases (Neo4J and Postgre) are less frequently updated and redeployed than the API and visualization. Thus, they are installed straight on the server. The API and visualization are deployed to the Docker container to make the deployment process easier. GraphifyEvlution is also built to the docker image so that the API can start a container from that and analyze the project.

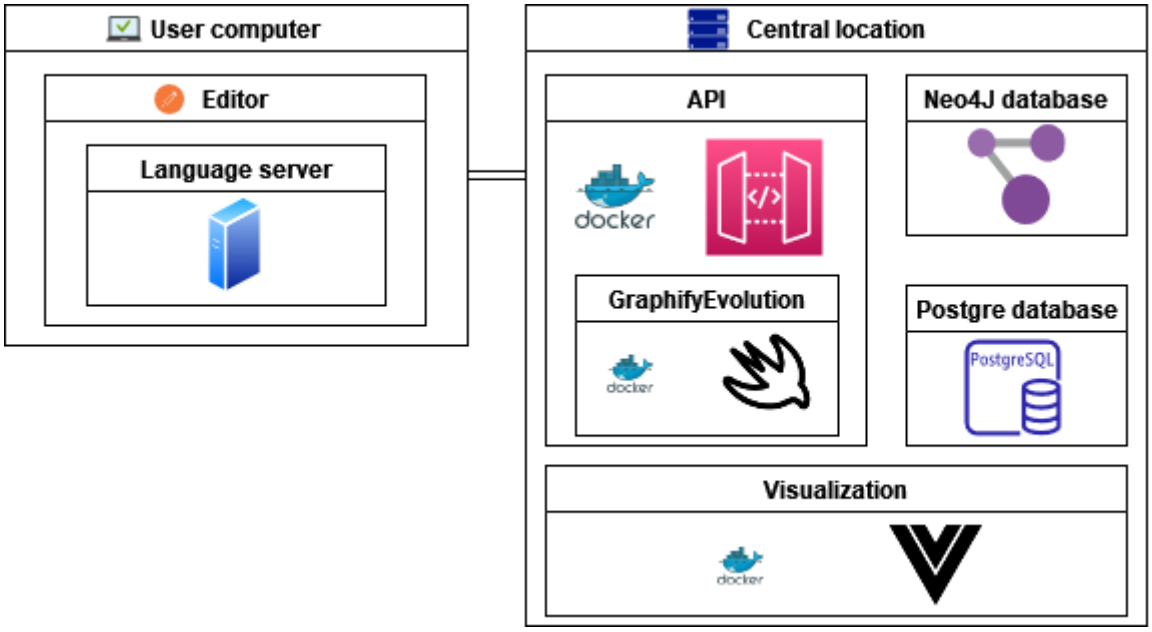

Figure 34: Deployment scheme

4.9 Showcase

The developed tool is called JIT Evolution. The main feature of the tool is visualizing the source code in real-time. The tool will analyze source code on every save. The source code is analyzed using a GraphifyEvolution analyzer. The results of the analyzer are shown to the user in the web application. The main objects in visualization are classes, methods, and variables. All of the implemented functionalities are described in the upcoming sections.

In this section, the term project is used for an application that is analyzed by the JitEvolution tool.

The first step is to configure the tool to use the appropriate JitEvolution server. There are three configurable values. Firstly JitEvolution Api, specifies the server that the editor is going to use to analyze the project. Secondly, JitEvolution Api-key specifies the key that is used to authenticate with the server. The key can be obtained after registering an account at the JitEvolution web application. Lastly, JitEvolution Url specifies the visualization website address that the user can open from the editor. Figure 35 illustrates the configuration values in Microsoft's Visual Code editor.

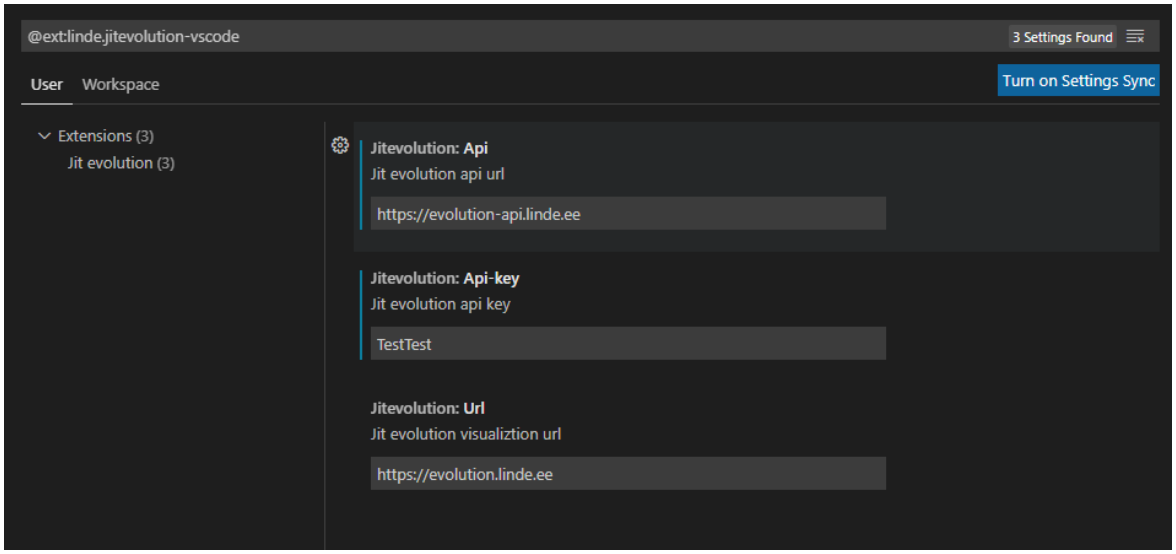

Figure 35: Screenshot of the configuration values in the JitEvolution plugin

Once the plugin is configurated, the user can start using it. The user opens a project and automatically the language server is started. After the user saves the file the project is sent to the API and analyzed. In the web application, the user is greeted by the login view where the user can create a new account or log in to the existing one. Authentication ensures that the user can see only their projects. The first thing the user notices after logging in is a navigation bar. On the navigation bar, there is a project selection dropdown and a project version selection dropdown. The navigation bar can be seen in Figure 36. Also in the figure, the analysis queue is shown. This notifies the user that the analysis is in progress. It also shows the time spent on analyzing the project and the time spent in the queue waiting for the analysis. The rightmost button allows the user to log out and copy the account's access key to the clipboard.

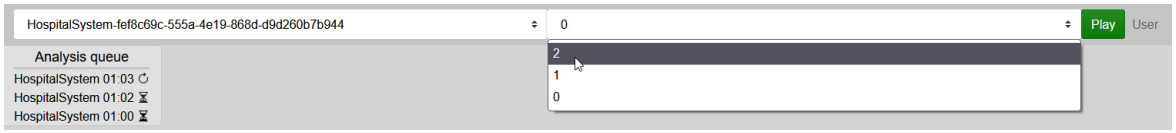

Figure 36: Screenshot of the navigation bar

After the analysis is completed the visualization will automatically reload the view with fresh analysis results but only if the analyzed project was previously selected. The user sees a class diagram-like view. The example of an analyzed project is shown in Figure 37. The user can move the diagram and zoom into it. If the user has a project open in the editor, the visualization will automatically show the file opened in the editor as the user writes code.

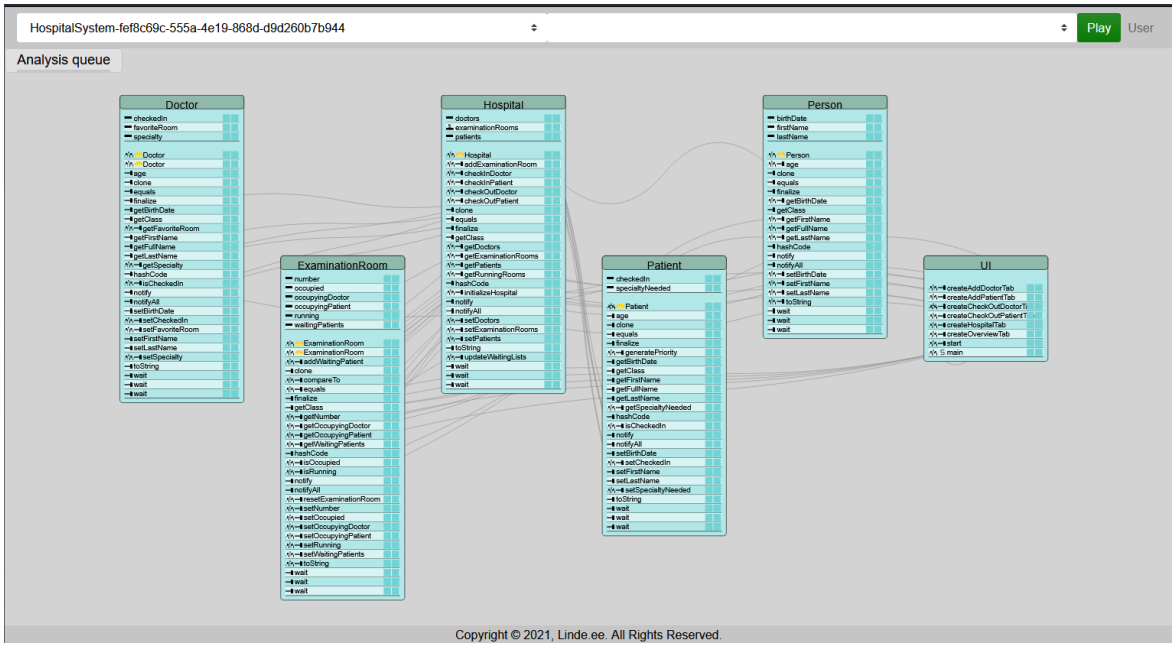

Figure 37: Screenshot of the freshly analyzed project

The largest objects in the visualization are classes. A class is rectangular with the class name in the title and variables and methods as a line in the body. Each line has a symbol to indicate what type of variable/method it is. All the symbols are shown and described in Figure 38.

- – Static variable
- – Instance Variable
- – Constructor
- – Method
- – Static method
- – Private modifier
- – Protected modifier
- – Public modifier

Figure 38: A legend of the symbols in the visualization

If a user hovers over the line, a new rectangular is shown. That rectangular shows additional information about that method/variable. An example of additional information is shown in Figure 39. On the right side of the line that the user hovers there are two buttons. Clicking the rightmost leaves the window with additional information open.

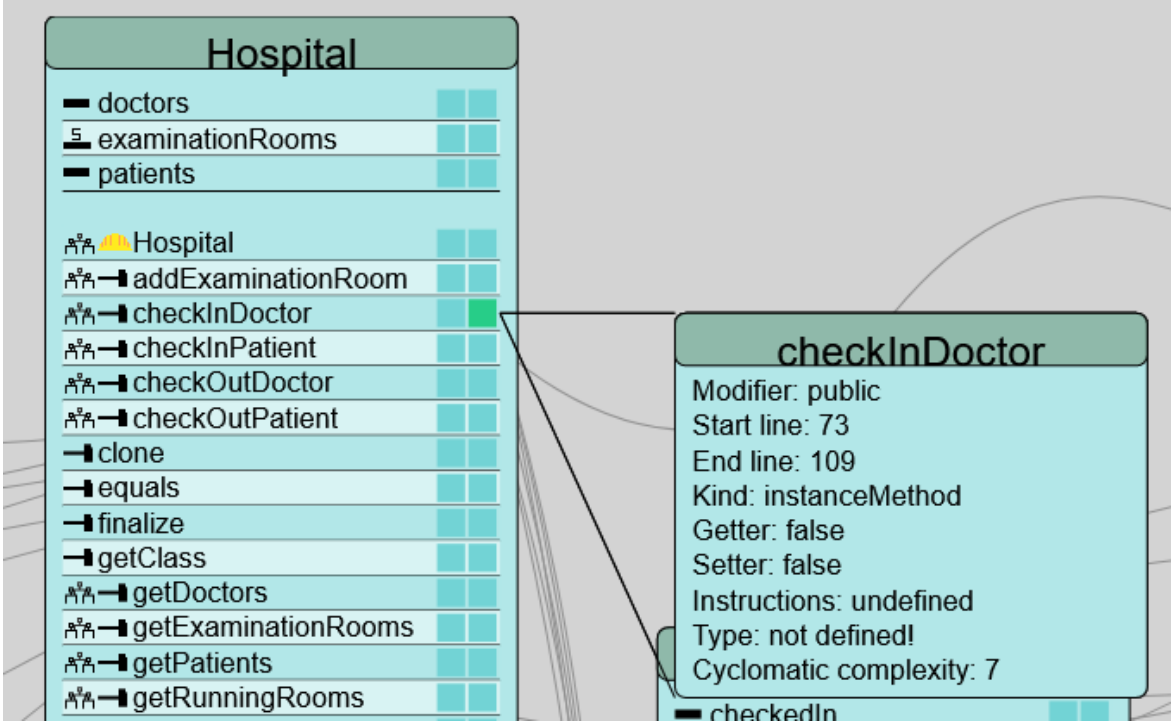

Figure 39: An Example of additional information

Also, the visualization shows relationships between the different methods and variables. Initially, all the relationships are grey and behind the classes. That is because to keep the visualization readable as there might be hundreds or thousands of relationships. In Figure 40, the example of the relationships is shown. When the user hovers over the method or variable, the related relationships are brought up and colored black. The relationship is an arrow. The start of the arrow indicates that the method or variable uses some other object and respectively the end of the arrow shows that it is used by other methods. In Figure 40, the mouse has hovered over the checkInDoctor method. Eight relationships are highlighted. Clicking the left-most button next to that method will keep the relationships highlighted. That allows the user to create a chain of the methods. For example, there is a relationship between the Hospital class checkInDoctor method and the initializeHospital method. The relationship starts from the initializeHospital method. When the user sees that they understand that the method initializeHospital uses the method checkInDoctor.

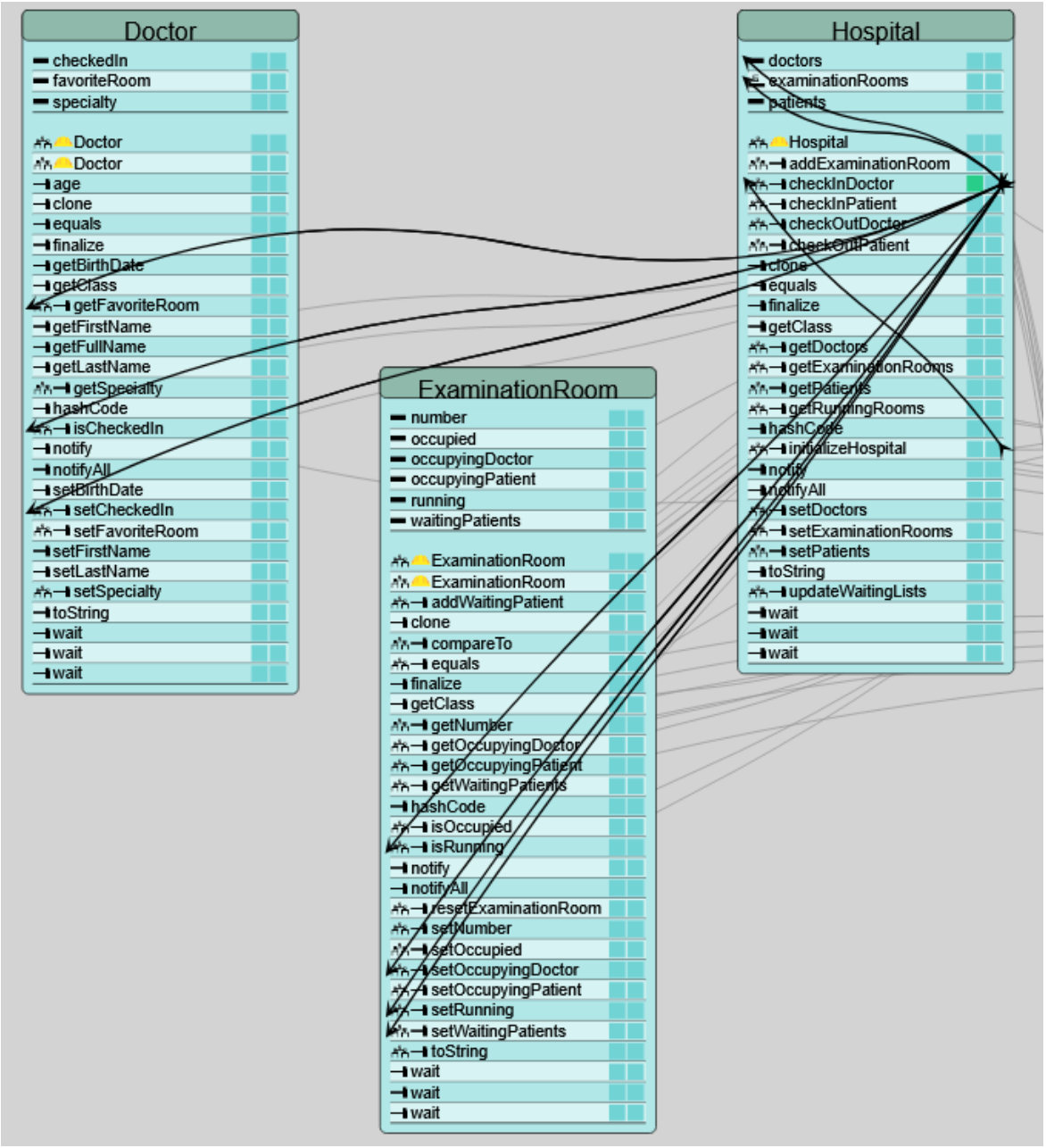

Figure 40: An example of the relationships

The visualization also shows the evolution of the source code. When a new version of the project is analyzed, the visualization will show added objects in light green. The older the change gets the darker green it becomes until it is completely black. In Figure 41, the example of the changed class is shown. The user can see that the method checkInDoctor and initializeHospital have been changed with the most recent changes of the code. The constructor of the class Hospital has been also changed but with previous updates to the code.

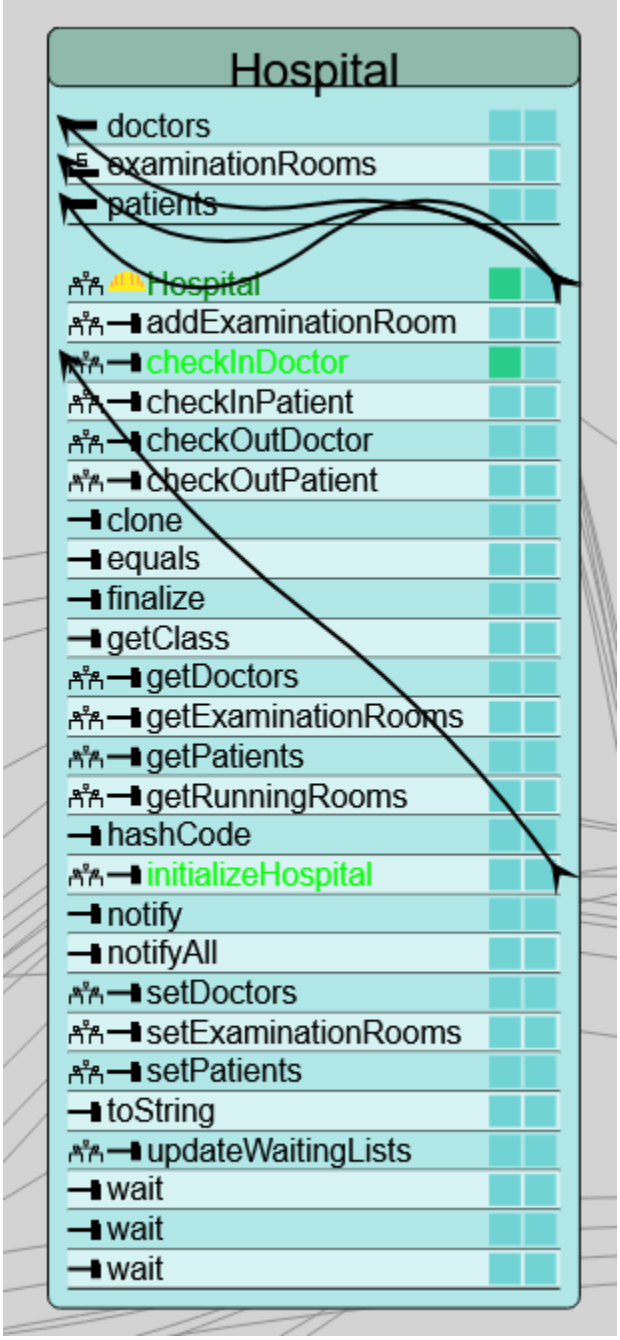

Figure 41: An example of the changed class in visualization

On the navigation bar, there is a button Play, which enables the user to replay the versions of the project to show the user how the code changed over time. Clicking the button the first time, visualization will replay the versions from first to last. Clicking the button again will replay the versions from last to first.

4.10 Evaluation

This section covers the evaluation of the developed tool. The developed tool does not have automatic tests. The reason for that is the developed tool consists of four different modules and each module should have its tests. Also, there should be integration tests. Because all this is time-consuming, the implementation of the requirements was prioritized. The requirements were tested manually by the author of this thesis during the development of the tool. The fulfillment of elicited requirements was ensured throughout the development

process. For each sprint review, the author of this thesis checked the fulfillment. The detailed information about the fulfillment of requirements can be seen in Table 7.

Table 7: Fulfillment of requirements

Requirements IDs	Fulfilled	Explanation
JIT-FR-1, JIT-FR-2, JIT-FR-3, JIT-FR-4, JIT-FR-5, JIT-FR-6, JIT-FR-7, JIT-FR-8, JIT-FR-9, JIT-FR-10, JIT-FR-14, JIT-FR-15, JIT-FR-16, JIT-FR-17, JIT-FR-20, JIT-FR-21, JIT-FR-22, JIT-FR-23	YES	
JIT-FR-11	NO	The requirement is not required. Marked as Should have
JIT-FR-12	NO	The requirement is not required. Marked as Should have
JIT-FR-13	NO	The requirement is not required. Marked as Should have
JIT-FR-18	NO	The requirement is not required. Marked as Could have
JIT-FR-19	NO	The requirement is not required. Marked as Should have
JIT-NFR-1	CONDITIONAL YES	See the results of performance testing below.
JIT-NFR-2, JIT-NFR-3, JIT-NFR-4	YES	

According to the previously seen table, all the Must-have requirements are implemented.

The performance of the tool was tested using a couple of different projects. The tool's centralized modules were deployed on a Raspberry pi 4 model B37 machine. It had the following specification:

- CPU - Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
- RAM – 8GB LPDDR4-3200 SDRAM
- OS – Raspbian 64bit

As the analysis of the project is done in a centralized location the user system specifications do not have an important role. The size of the analyzed project is taken based on the author's personal experience in object-oriented programming courses. In the author's experience, the

[37] https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

projects are small – one to six classes per project. The tool was tested with two projects. The first project had two classes and the other had six classes.

Table 8: Performace tests of the developed tool

Number of classes	Run 1	Run 2	Run 3	Run 4	Run 5
2	31 seconds	30 seconds	30 seconds	30 seconds	31 seconds
6	118 seconds	117 seconds	118 seconds	132 seconds	123 seconds

In Table 8 the results of the performance tests can be seen. As seen the analysis time of the project raises rapidly when the number of classes increases. With a relatively small number of classes, the tool passes the non-functional requirement JIT-NFR-1 but as the class counts increase the analysis time of the project increases rapidly and thus no longer satisfies that requirement.

The answers to the formed research questions were received by creating usage scenarios. The following section describes the usage scenarios visioned by the author of this thesis. These scenarios only give guidelines on how the tool should be used but not how the tool must be used.

4.10.1 RQ1

Research question RQ1 was “Is the developed tool is to use?” The following scenario shows that the tool is easy to use.

As a prerequisite, an advanced user, like the instructor, has already set up centralized modules. The student creates an account on the JitEvolution visualization website to get an account and an access key for the editor's plugin. After that, the student downloads a plugin for their preferred editor. Currently, Microsoft’s Visual Code and JetBrains’ IntelliJ IDEA are supported. Because the student uses the plugin with Java language, they already have the required runtime installed on their systems. After that, the student configures the plugin and the setup of the tool is done. Configuration of the tool done, the student can start using the tool. The student writes code as usual. With each save the plugin sends the changes to the API to be analyzed. Periodically the student can check the JitEvolution web application to see how the project has been changed. Also, the student can click the play button to see the timelapse of the project. This way the student can see visually the parts that got more complicated and might need refactoring.

As seen in the usage scenario the installation and the usage of the tool are quite simple. Therefore the research question RQ1 is answered with a positive answer.

4.10.2 RQ2

Research question RQ1 was “Is the developed tool beneficial in learning software development?” The following scenarios show that the tool could be beneficial to learning software development.

The instructor gave a student an existing project that they have to improve. The project has a couple of different classes. The student feels overwhelmed and decides to use JitEvolution to understand the code better. The student installs and configures the tool the same as in the previous section. After the student configures the plugin, they open the existing project and initiate analysis of the project by saving one of the files. The API will analyze the

project and the web application will show the result. The student views the visualization. The student can see in the visualization that the project has five classes. Also, they see that class A’s method AM calls class B's method BM. Furthermore, the method BM calls class C’s method CM. Therefore the students understand that the method AM implicitly uses the class C method CM. The student deletes a method CM and sees that the relationship between BM and CM is gone and AM does not implicitly use a method CM anymore.

Another potential use case is that an instructor shows the students how to code in the classroom. The instructor shows on the wall the editor and next to it the JitEvolution web application where students can see how the code changes. As the instructor opens a new file in the editor, the visualization automatically changes the view to show the currently opened file to the students. The instructor adds a new method that uses the parent class method. It confused the students but after seeing it in the visualization it is more clear what class method the instructor used as the relationship between the two methods is created.

As seen in the usage scenarios the tool could be beneficial in learning software development. Therefore the research question RQ2 is answered with a positive answer.

5 Discussion

This section covers the usage scenario of the tool, restrictions, future work of the application, and lessons learned during the thesis. This chapter explains the issues and limitations of the tool and future developments of the JitEvolution. Also, the author describes the lessons learned during the development of the JitEvolution.

5.1 Issues and limitations of the tool

The main issues of the tool came from the chosen source code analyzer GraphifyEvolution. Since GraphifyEvolution's current intended purpose is to analyze the whole project, it was challenging to use it as a real-time tool. Analyzing every time the whole project is not optimal because the whole project analysis is very time-consuming. Another limitation of GraohifyEvolution is that it cannot reanalyze the existing project and add the new changes to that project. Thus it is challenging to incrementally analyze the project. Both of those were challenging issues but not impossible to overcome. Solutions for those issues were described in section 4.5. Because the solutions aren't ideal, the tool is not able to run in there is room for optimization and improvements.

Another issue comes from the language server protocol. The language server protocol is implemented differently among different editors. As mentioned in section 4.6, a language server can only process one request at a time. Due to that saving the file multiple times in the editor, the JitEvolution tool in Microsoft's Visual Code was not able to properly analyze each change as the next code change request was processed after the previous was done. Therefore losing changes done during the analysis of previous changes. The same situation in IntelliJ IDEA caused the editor to freeze and wait for the analysis to be done. This issue was solved by implementing a queue in the API and processing the code changes asynchronously.

One serious limitation was provided by editors and language server protocol. As the language server protocol's intended purpose is to provide language support for the editor such as code coloring, auto suggestions, autocomplete, etc, there can be only one language server at a time active. In Microsoft's Visual Code languages support is provided by language servers. Thus activating the JitEvolution removed the ability to use the Java language server. I.e Java-specific autocompletes, coloring, etc. Even though this is a serious limitation, the tool could be still used and it could be solved in the future.

5.2 Future work

The tool is written in a modular and well-structured way that supports easier future developments. The author chose code quality over speed so that there is a higher chance for future work on the tool. Even though in the author's opinion the tool could be used in its current state it needs future development to make the tool faster and provide more information to the user. Some of the future possible future work was described in section 5.1. Some of the ideas are listed below:

1. Currently, the tool takes over 30 seconds to analyze a quite simple project. The analysis could be optimized by analyzing only the changes of the application and not the whole application.
2. The problem of losing Java-specific language features could be solved. One potential solution is to include an existing language server in the tool that provides those features.
3. Currently, *JitEvolution* only supports the Java language. The tool is designed in a way that adding support for additional languages is possible.

4. As the tool is not evaluated in the real world, one of the future works is to evaluate it in a real-world scenario.
5. Additional features and information about object-oriented programming could be added.
6. Settings panel where users can select which features they want to show. For example, an option to highlight all the relationships.
7. The application encounters some performance issues when the code base of the application is very large. Therefore some optimization could be done to improve that.
8. As the tool creates a new version of the project each time the file is saved, it gathers a lot of data from users. This data could be used to analyze how students solve their exercises.

5.3 Lessons learned

There were many lessons learned during the development of the tool. Sharing these lessons helps future developers not make the same mistakes.

The most important lesson is that in-depth research on used 3rd party applications is important. The selection of a suitable 3rd party application can make the development process easier as the developer doesn't need to worry about it. For example, GraphifyEvolution at the time of the writing of this thesis wasn't an ideal source code analyzer because initially, it couldn't run on Linux machines. Also, GraphifyEvolution is in development and has to be optimized and support incremental project analysis to be ideal for JitEvolution. In addition, the author experimented with three different methods to visualize the analysis result – BootstrapVue, D3.js, and SVGs. With planning, research, and thought through, the experimenting could have been omitted. Resulting in a time win that could have been spent on something else.

Another lesson learned was that the agile software development approach was the right decision. The agile approach allowed the author to get feedback from this thesis supervisor in the development phase and modify the requirements accordingly. Another, more traditional approach would have been the Waterfall methodology38. In waterfall methodology phases are the following: the requirements are gathered, a prototype created, development done, tested, and evaluated. With this method, the product owner in the end verifies if this is what they wanted. As this thesis used more of an agile approach, the product owner was able to guide the result to a better outcome.

[38] https://www.workfront.com/project-management/methodologies/waterfall

6 Conclusion

The thesis focused on visualizing source code changes in real-time. The thesis produces a tool that can get changes to the source code from the editor in real-time. Firstly, the background for this thesis and some of the related works already done are discussed. Then the requirements were gathered by discussing them with this thesis supervisor. The tool is implemented based on the requirements that were gathered. The developed tool is called JitEvolution. The tool consists of two big parts: the part that should be installed on the users' computer and another that should be installed in the central location. A more granular way of dividing the tool is the following: the language server, the API, the source code analyzer, and the visualization. The tool uses language server protocol to receive the code changes from the users' editor in real-time. The language server sends the changed code to the API every time the user saves a file. The code changes are put in a queue where the API will analyze the changes asynchronously. The analysis result will be shown to the users in the web application. The whole tool is designed to be modular and easily maintainable to increase the possibility the further development of the tool. The tool uses GraphifyEvolution that is developed by this thesis supervisor as a source code analyzer. The tool was tested and evaluated by the author of this thesis. In conclusion, it can be said that the main goal of the thesis is achieved. The tool is capable to analyze the source code in real-time and the user can see the analysis result in the web application.

The JitEvolution tool has a lot of potential and there are a lot of possibilities for future works. The author of the thesis will continue to work on the improvements of the tool.

7 References

[1] Y. Bosse and M. A. Gerosa, “Why is Programming so Difficult to Learn? Patterns of Difficulties Related to Programming Learning Mid-Stage,” *ACM SIGSOFT Software Engineering Notes,* vol. 41, no. 6, pp. 1-6, 2017.

[2] S. Bergin and R. G. Reilly, “The influence of motivation and comfort-level on learning to program,” *PPIG 17,* pp. 293-304, 2005.

[3] S. Mathew Biju, “Difficulties in Understanding Object Oriented Programming Concepts,” *Lecture Notes in Electrical Engineering,* vol. 152, pp. 319-326, 2013.

[4] S. Maravić Čisa, R. Pinter and D. Radosav, “Effectiveness of Program Visualization in Learning Java: a Case Study with Jeliot 3,” *INTERNATIONAL JOURNAL OF COMPUTERS COMMUNICATIONS & CONTROL,* vol. 6, no. 4, pp. 668-680, 2011.

[5] M. Storožev, “Exploration of Techniques to Visualise Code Quality,” Tartu, 2021.

[6] T. Badalov, “Software Analytics: Visualization of Source Code Evolution,” Tartu, 2021.

[7] G. Balogh and Á. Beszédes, “CodeMetropolis - code visualisation in MineCraft,” *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM),* pp. 136-141, 2013.

[8] G. Balogh, S. Attila and B. Árpád, “CodeMetropolis: Eclipse over the city of source code,” *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM),* pp. 271-276, 2015.

[9] K. Rahkema and D. Pfahl, “GraphifyEvolution - A Modular Approach to Analysing Source Code Histories,,” *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft),* pp. 24-27, 2021.

[10] “UML Class Diagram Tutorial,” [Online]. Available: https://www.lucidchart.com/pages/uml-class-diagram. [Accessed 10 May 2022].

[11] M. Cohn, “Advantages of User Stories for Requirements,” 7 October 2004. [Online]. Available: https://www.mountaingoatsoftware.com/articles/advantages-of-user-stories-for-requirements. [Accessed 10 May 2022].

[12] M. Vestola, “A Comparison of Nine Basic Techniques for Requirements Prioritization,” Helsinki University of Technology, 2010.

[13] S. Oloruntoba, “SOLID: The First 5 Principles of Object Oriented Design,” 21 September 2020. [Online]. Available: https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design. [Accessed 10 May 2022].

[14] P. Krill, “Microsoft-backed Language Server Protocol strives for language, tools interoperability,” 27 June 2017. [Online]. Available: https://www.infoworld.com/article/3088698/microsoft-backed-langauge-server-protocol-strives-for-language-tools-interoperability.html. [Accessed 10 May 2022].

[15] L. Gupta, “https://restfulapi.net/rest-architectural-constraints/,” 9 March 2022. [Online]. Available: https://restfulapi.net/rest-architectural-constraints/. [Accessed 10 May 2022].

[16] “Design the infrastructure persistence layer,” Microsoft, 13 April 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design. [Accessed 10 May 2022].

[17] “Vuex,” 7 April 2022. [Online]. Available: https://vuex.vuejs.org/. [Accessed 10 May 2022].

Appendix

I. Materials

Table 9: Functional Requirements

ID	Description
JIT-FR-1	As a user, I want to log in to the visualization, so that I can see only my projects.
JIT-FR-2	As a user, I want to add an access key to the JIT evolution plugin, so that the plugin can send data to the visualization.
JIT-FR-3	As a user, I want to use IntelliJ IDEA, so that I can use my favorite editor.
JIT-FR-4	As a user, I want to see the whole project visualization, so that I can get an overview of the project.
JIT-FR-5	As a user, I want to see class variables that are available in the class, so that I understand the difference between class and instance variables better.
JIT-FR-6	As a user, I want to see instance variables that are available in the class, so that I understand the difference between class and instance variables better.
JIT-FR-7	As a user, I want to see instance variables that are available in the method, so that I understand the difference between class and instance variables better.
JIT-FR-8	As a user, I want to class variables that are available in the method, so that I understand the difference between class and instance variables better.
JIT-FR-9	As a user, I want to see a class detailed view, so that I result of the analysis.
JIT-FR-10	As a user, I want to see a method detailed view, so that I result of the analysis.
JIT-FR-11	As a user, I want to see the class implement an interface so that I can see that it implements an interface.
JIT-FR-12	As a user, I want to see if the class is abstract so that I can see it can’t be initiated.
JIT-FR-13	As a user, I want to see if the class has a parent so that I can see which its base class is.

JIT-FR-14	As a user, I want to see which classes the method uses so that I can understand the code better.
JIT-FR-15	As a user, I want to see which class uses which classes so that I can understand the code better.
JIT-FR-16	As a user, I want to see variable access modifiers (private, public, protected), so that I understand the code better.
JIT-FR-17	As a user, I want to see methods' access modifiers, so that I understand the code better.
JIT-FR-18	As a user, I want to search for classes from the project and highlight them, so that I can see the result of the analysis of that class.
JIT-FR-19	As a user, I want to see the class accordion view where is shown class, parent classes, and interfaces so that I can easily view parents and interfaces.
JIT-FR-20	As a user, I want to select the project, so that I can see the analysis of that project.
JIT-FR-21	As a user, I want to select the version of the project, so that I can see its evolution of the project.
JIT-FR-22	As a user, I want to automatically see the project version in the order of creation, so that I can see the evolution of the project.
JIT-FR-23	As a user, I want to see in the visualization the class, that I have open in the editor so that I can quickly see the changes.

Table 10: Prioritized requirements

Requirement ID	Priority
JIT-FR-1	Must have
JIT-FR-2	Must have
JIT-FR-3	Must have
JIT-FR-4	Must have
JIT-FR-5	Must have
JIT-FR-6	Must have
JIT-FR-7	Must have

JIT-FR-8	Must have
JIT-FR-9	Must have
JIT-FR-10	Must have
JIT-FR-11	Should have
JIT-FR-12	Should have
JIT-FR-13	Should have
JIT-FR-14	Must have
JIT-FR-15	Must have
JIT-FR-16	Must have
JIT-FR-17	Must have
JIT-FR-18	Could have
JIT-FR-19	Should have
JIT-FR-20	Must have
JIT-FR-21	Must have
JIT-FR-22	Should have
JIT-FR-23	Must have
JIT-NFR-1	Should have
JIT-NFR-2	Must have
JIT-NFR-3	Should have
JIT-NFR-4	Should have

II. Repository links

- API: https://github.com/L1nde/JitEvolution
- Language server and plugins: https://github.com/L1nde/JitEvolutionLSP
- Visualization: https://github.com/L1nde/JitEvolutionVisualization
- GraphifyEvolution fork: https://github.com/L1nde/GraphifyEvolution

III. License

Non-exclusive license to reproduce thesis and make thesis public

I, Einar Linde

1. herewith grant the University of Tartu a free permit (non-exclusive license) to reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,
   Implementation of JIT (Just in Time) Visualization of Changes in Source Code,
   supervised by Kristiina Rahkema.
2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons license CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.
3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.
4. I certify that granting the non-exclusive license does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Einar Linde

17/05/2022