UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Software Engineering Curriculum

Miron Storožev

Exploration of Techniques to Visualise Code Quality

Master's Thesis (30 ECTS)

Supervisor(s): Dietmar Alfred Paul Kurt Pfahl, PhD
Kristiina Rahkema, MSc

Tartu 2021

Exploration of Techniques to Visualise Code Quality

Abstract:
As the size and complexity of the software increase, it becomes more challenging to maintain the quality of the code. Various static code analysis tools have been developed to help software engineers to find inefficiencies in the code. However, most static analysis tools focus on specific parts of the development and not the project as a whole, thus failing to provide a general overview of the code quality.

The goal of the thesis is to develop a software tool to visualize the code quality of an entire project. Requirements for the application are elicited based on the interviews conducted with different stakeholders in digital transformation company Nortal. The application is developed on top of GraphifyEvolution static code analysis tool, but the architecture of the application allows the addition of other static analyzers. The application is evaluated by conducting interviews with various stakeholders. Interviews show that the tool can be used in a real production environment by developers and scrum masters.

Keywords:
code smell, code quality, visualization

CERCS: P170 (Computer science, numerical analysis, systems, control)

Koodikvaliteedi Visualiseerimise Tehnikate Uurimine

Lühikokkuvõte: Tarkvara toote mahu ja keerukuse kasvades muutub koodi kvaliteedi säilitamine tüsilikuks. Aitamaks arendajatel tuvastada koodist ebatuhususi on tehtud mitmeid staatilisi koodi analüsaatoreid. Paraku keskendub enamik staatilisi analüsaatoreid arenduse kindlale osale ja mitte projektile tervikuna. Seetõttu puudub täielik ülevaade kogu rakenduse koodi kvaliteedist.

Käesoleva magistritöö eesmärk on arendada rakendus visualiseerimaks koodi kvaliteeti rakenduses tervikuna. Nõuded rakendusele koostati tuginedes Nortali töötajatega tehtud intervjuudele. Tarkvara arendati kasutades GraphifyEvolution staatilise koodi analüsaatorit, kuid rakenduse arhitektuur võimaldab lisada ka teisi staatilisi analüsaatoreid. Tagasisidet rakenduse kohta saadi samuti intervjuude abil. Intervjuude tulemused näitavad, et magistritöös valminud tööriista saab kasutada päris tarkvara toote arenduses. Tööriista potensiaalsed kasutajad on arendajad ja *scrum masterid*.

Võtmesõnad:
lõhnav kood, koodi kvaliteet, visualiseerimine

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

Contents

1 Introduction

With the increasing scale and complexity of software, keeping the code *clean* becomes a challenge for many developers [1]. Many tools have been developed to help software engineers to analyze source code and point out possible deficiencies [2]. For a long time, Software Development companies have inspected their code using static code analysis tools as a complementary technique to regular quality assurance [1]. Despite the vast popularity of static code analysis tools, they still have many disadvantages like finding false-positive code smells or focusing only on specific code smells, and not visualizing the application as a whole [3].

1.1 Problem statement

This master's thesis captures the topic of visualising code smells to improve the understandability and quality of the code as a whole. The scope of the research covers getting code smells from the application, finding additional relevant data about the code to display, and discovering ways to present data to the user.

The main research goal of the thesis is to build an easily extensible software application for visualizing the code quality of the source code. The application uses static code analysis as input and produces a visual image of the source code. Application is built so static code analyser can be easily changed, and addition or modification of features is straightforward. The results produced by the application are evaluated by performing static code analysis on live applications and presenting visualizations to stakeholders of the analyzed projects. To understand the usefulness of the application, the following research questions were raised:

- Q1. Can code smell visualization tools be beneficial for Software Development?
- Q2. Which stakeholders will benefit from the code visualization application?

Research question Q1 focuses on whether the code smell visualization can be used in real production environment at all. Research question Q2 focuses on certain stakeholders. The goal of the question is to understand which roles in software development can benefit from the tool. Even though code smells are usually handled by developers, it is possible that other stakeholders like Product Owners[1] or Scrum Masters[2] can benefit from the application as well. Both Q1 and Q2 research questions are answered based on the results of the conducted interviews.

[1] https://www.scrum.org/resources/what-is-a-product-owner
[2] https://www.scrum.org/resources/what-is-a-scrum-master

1.2 Structure of the thesis

The thesis is separated into six parts: Introduction, Background, Method, Results, Discussion, and Conclusion. In the Background part of the thesis, two projects are introduced. These projects are used for the testing and evaluation of the results. In addition, Background part covers existing code visualization approaches, static code analysis tool that is used in the development of the application, and different existing architectures that were considered when developing the application.

Method part of the thesis briefly describes what will be developed in the application and how requirements of the application are gathered and prioritized. Additionally, the section describes which criteria are used when making architectural and technological selections. Finally, section covers how the application is tested and evaluated.

The Results section of the thesis focuses on the authors contribution to the work. The section describes the results of the conducted interviews, functional and non-functional requirements that were elicited based on the interviews and application developed based on these requirements. Additionally, implementation of front-end and back-end modules is described. In these sections, it is explained which architectural and technological choices were made for the development of the modules. Also high level architecture of the modules is described. Finally, the section covers test and evaluation results of the application.

The discussion part focuses on restrictions and problems that were faced when developing the application. Additionally, chapter covers future works of the application and lessons that author of the thesis learned while developing the application.

The last chapter of the document concludes the results that were achieved in the thesis and summarizes the research topic.

2 Background

This section provides a brief overview of the company for whom the code smell visualization solution is built. Additionally, the section covers existing code smell visualization approaches and applications. The section also describes a code smell recognition software used in the development of the visualization application and multiple architectural styles considered when selecting the top-level architecture type for the application.

2.1 Organisational context

Nortal is an international software company founded in Estonia in 2000 as Webmedia[3]. The company is mostly known for providing large-scale digitization solutions in the public sector of the Estonian government. However, the firm has more than 150 private and public sector projects in more than ten countries, including Finland, Lithuania, Oman, and United States [4]. The thesis is focusing on two projects in Nortal. For legal purposes, the names of the project are not revealed. These two projects were chosen because the author of the thesis was granted access to the projects' source code and project teams agreed to cooperate with the author of the thesis.

Project A. Project A is an internal application meaning the users of the software are employees of Nortal. The scale of the project is rather small, and the team of the project is smaller than ten people. People in the development team frequently rotate and change. Some parts of the project are made public for all developers in Nortal, meaning that anyone in the company can contribute to the project if they desire. As there are many developers who are given access to the project, it is vital that the code follows strict patterns and is clean. Thus, the author of the thesis believes that the code smell visualization tool might benefit the project.

Project B. Project B is a collaboration between Nortal and another company. The scale of the project is quite large and developers follow strict code conventions. The project uses SAFe (Scaled Agile Framework) [5] pattern as a daily development process. In Scaled Agile Framework, an iteration is usually divided into n sprints, where one sprint is specifically reserved for alleviating technical dept. The sprint is mostly done for developers to work on existing code smells and try out new technologies. As in a regular agile framework, length of the sprints depends on personal preference and can be anywhere from one week to four weeks [6]. Before each sprint, there is a planning and each sprint ends with a demo and retrospective. In the current project, sprints are two weeks long. In this project, code smell visualization tool can be used in technical

[3]https://nortal.com/about-us/

dept alleviation sprint demos in order to demonstrate differences between code smells in project before and after the sprint.

It is important to note that the usage of the tool is not restricted to previously mentioned projects, but requirements elicitation and product evaluation will be conducted on the described projects.

2.2 Existing work on code and code smell visualization

Visualization techniques that allow developers to convert software into apprehensive visual output enhance comprehensibility and understandability of the software, thus helping to reduce development and maintainability costs of the application [7]. In recent years, multiple 2d, 3d, and even VR code visualization applications have been made [8, 9, 10, 11].

CodeHouse. In the article "CodeHouse: VR Code Visualization Tool" [11] the authors Akihiro Hori, Masumi Kawakami and Makoto Ichii describe that CodeHouse is an immersive VR tool, which represents code as a cylindrical house, where each module of the software is visualised as a separate floor or room in the house. Classes are represented as cubes inside the room (module) and methods in the class are visualised as circles inside the cube. The authors explain, that user is placed in the middle of the cylinder and interaction with the system is done via headset and remote controls. There a user can move around, zoom in and out of the system and click on objects for additional information about them. Figure 1 shows an abstract representation of modules and classes in CodeHouse. Figure 2 displays user interaction with the application. In addition to navigation users have an opportunity to configure the colors, shapes and sizes of the objects based metrics like complexity, number of references, number of bugs and number of lines in the object. CodeHouse tool uses *Understand* static code analysis tool, which generates functions metrics, relations and finds code smells of the application.

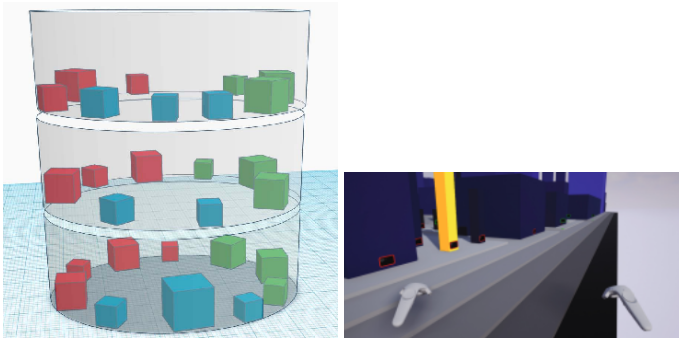

Figure 1. Abstract representataion of modules and classes in CodeHouse [11]

Figure 2. User interaction with CodeHouse application [11]

CodeCity. CodeCity is another application that uses VR to make users interact with the application. In the article "Visualising a Software System as a City Through Virtual Reality" [10] authors Nicola Capece, Ugo Erra, Simone Romano and Giuseppe Scanniello describe how their tool creates a 3d city model from the code. The buildings in the city visualize java classes and the color of the ground visualizes packages. The darkness of the buildings represents the size of the class, the darker the building the more lines of code does the class have, and height of the building represents the number of methods in the class. Similar to the CodeHouse application, the interaction between the user and the software is achieved with the usage of VR headset and remotes. Figure 3 shows the top view of the classes and packages visualized in the software (left image) with the interaction between user and the software (right image). In the software, user can navigate between the city and click on the buildings to get some additional information about the class that the building represents. The authors of the article mentioned that the software uses the Eclipse plugin CodePro Analytix as a static code analysis to get the data about the classes.

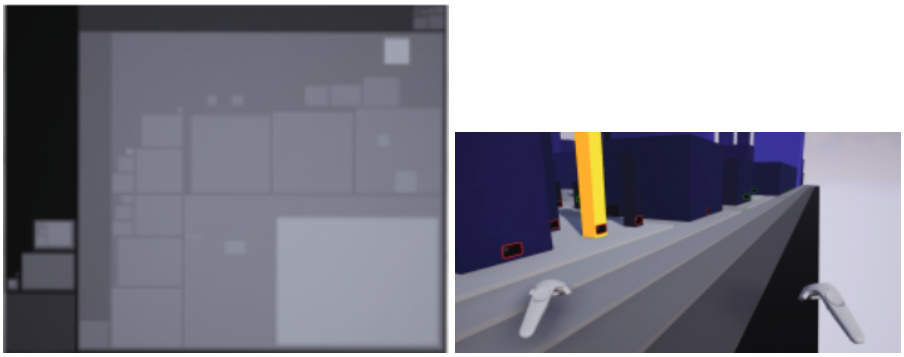

Figure 3. CodeCity top view of classes and packages on the left and user interaction with the software on the right [10]

Code Park Compared to CodeHouse and CodeCity, CodePark is a simpler application as it doesn't use any static analysis to bring out additional information about the software. As the developers of the CodePark, Pooya Khaloo, Mehran Maghoumi, Eugene Taranta, David Bettner, and Joseph Laviola, describe in the article "Code Park: A New 3D Code Visualization Tool" [8], the tool was designed to visualize large software in a simple way and improve developers cognitive load and engagement when learning new codebase by introducing fun and visual way to interact with the source code. The authors explain that in CodePark, all the classes are visualised as blocks with the size of the block representing the size of the class. Classes are grouped together based on packages, meaning classes in the same packages are closer to each other. Figure 4 shows how visualization of the code is done in CodePark.

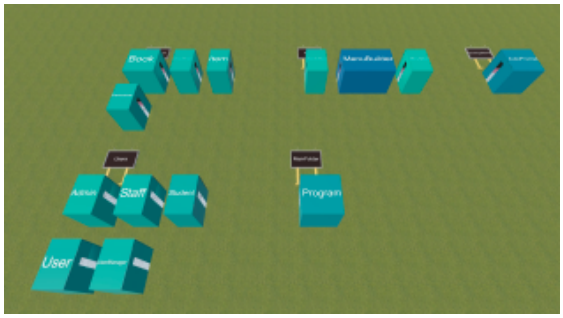

Figure 4. Top view of an example codebase in CodePark application [8]

All of the previously mentioned visualization approaches had a unique way of visualizing software. Applications had multiple positive and negative features, which will be covered in future sections, but already here it can be seen that in order to show more data about software, static analysis tools should be used.

2.3 GraphifyEvolution - code smell recognition tool

GraphifyEvolution[4] is an application developed by Kristiina Rahkema [12]. As the author of the application describes in the documentation[5] of the app, the main purpose of the tool is to analyze applications source code and its history. The author describes that currently the tool supports Swift, Java and C++ programming languages, meaning applications written in those languages can be analyzed. As it is described in the documentation, GraphifyEvolution stores classes, methods, variables, relationships between classes, modules, history of the application, and much more into Neo4j[6] database where the data

[4]https://github.com/kristiinara/GraphifyEvolution
[5]https://github.com/kristiinara/GraphifyEvolution/tree/master/documentation
[6]https://neo4j.com/

can be later queried. In addition to the analysis, Rahkema provides multiple queries for determining code smells in the application[7]. The queries have adjustable parameters and thus can be customized according to the needs of the user. Figure 5 shows an example query of Long method[8] code smell detection.

```
MATCH (c:Class)-[r:CLASS_OWNS_METHOD]->(m:Method)
WHERE m.number_of_instructions > veryHighNumberOfInstructions
RETURN
    distinct(m.app_key) as app_key,
    count(distinct m) as number_of_smells
```

Figure 5. Example of Long method code smell detection query. Example taken from GraphifyEvolution.

2.4 Different types of software architecture

This section describes three different software architecture types that are considered when selecting architecture for the development of the visualization application. The selection of the architecture is done in the Method and Results chapters of the document.

2.4.1 N-tier architecture

Multi-layer or n-tier architecture is one of the most common software development architecture patterns[9]. The main concept behind the architecture pattern is to separate an application into n layers [13]. The most common n-tier architecture is three-tier architecture, where code is separated into the presentation layer, business layer, and data layer. The main advantage of n-tier architecture over other architecture types is its ease of use. It is one of the most widely used architecture types, so it is easy to use it in the production environment, as most of the developers are well informed about this architecture[9]. The main disadvantage of the architecture is that the architecture doesn't provide strict separation between different tiers, which may cause different layers of architecture to overlap [13]. This may cause problems with the maintainability of the project.

2.4.2 Hexagonal architecture

Hexagonal architecture, also known as Ports and Adapters, is an example of layered software development design, where main emphasis is on separating business logic from

[7]https://github.com/kristiinara/GraphifySwift
[8]https://refactoring.guru/smells/long-method
[9]https://dzone.com/articles/5-major-software-architecture-patterns

external devices that are responsible for inputs and outputs of the software, such as HTTP context or database. [14, 15]. Generally, hexagonal architecture defines three fundamental blocks: user interface, business logic and infrastructure [16]. Figure 6 shows that in the middle of hexagonal architecture is business logic, which consists of Use Cases and Entities. Use cases communicate with ports. In Figure 6 there are 4 ports total: two input and two output ports. As the inventor of the Ports and Adapters architecture, Alistair Cockburn, explains, ports in Hexagonal architecture serve the purpose of connecting external devices with business logic [17]. Cockburn gives the analogy of the ports in operating systems, where any suitable device that meets required electrical and mechanical criteria can be plugged into the port. In the same article, the author of the architecture describes that for each external device that is plugged into a port, there is an adapter, the main purpose of which is to convert signals coming from the device into data that is suitable for the business logic and vice versa. Figure 6 visualizes 4 adapters: Web Adapter, Persistence Adapter, and two External System Adapters. Adapters that connect into Input Ports (Web Adapter and External System Adapter) are responsible for user interface and the ones connected to the Output Ports (Persistence Adapter and External System Adapter) serve as infrastructure of the software. The main advantage of hexagonal architecture over n-tier architecture is that hexagonal architecture separates domain logic from the adapters and thus allowing domain logic to be tested independently [17]. One of the disadvantages however is that the architecture is not as widely used as n-tier architecture, so developers can be unfamiliar with its concepts[10].

2.4.3 Onion architecture

As the proposer of the Onion architecture, Jeffrey Palermo said, both Onion and Hexagonal architecture share a similar ideology: the domain model is separated from the rest of the application by externalizing infrastructure and providing adapters that the business part of the application communicates with [19]. In the same article, Palermo explains, that the main difference between Onion and Hexagonal architecture is that Onion architecture separates application code into layers, where the code from the outer layer can depend on code from the inner layer, but not the other way around. Figure 7 visualizes onion architecture. In the middle of the image is a Domain Model, which is surrounded by Domain Service and Application Service layer. The most outer layers are User Interface, Tests and Infrastructure layers. In the architecture represented by the Figure 7, User Interface layer can depend and most of the time depends on Application Services layer, but Domain Services layer is not allowed to depend on Application Services layer. As in Hexagonal architecture, Domain Model is in the middle of the code and cannot depend on anything other than itself [19]. Since the onion architecture is

[10]https://dzone.com/articles/5-major-software-architecture-patterns

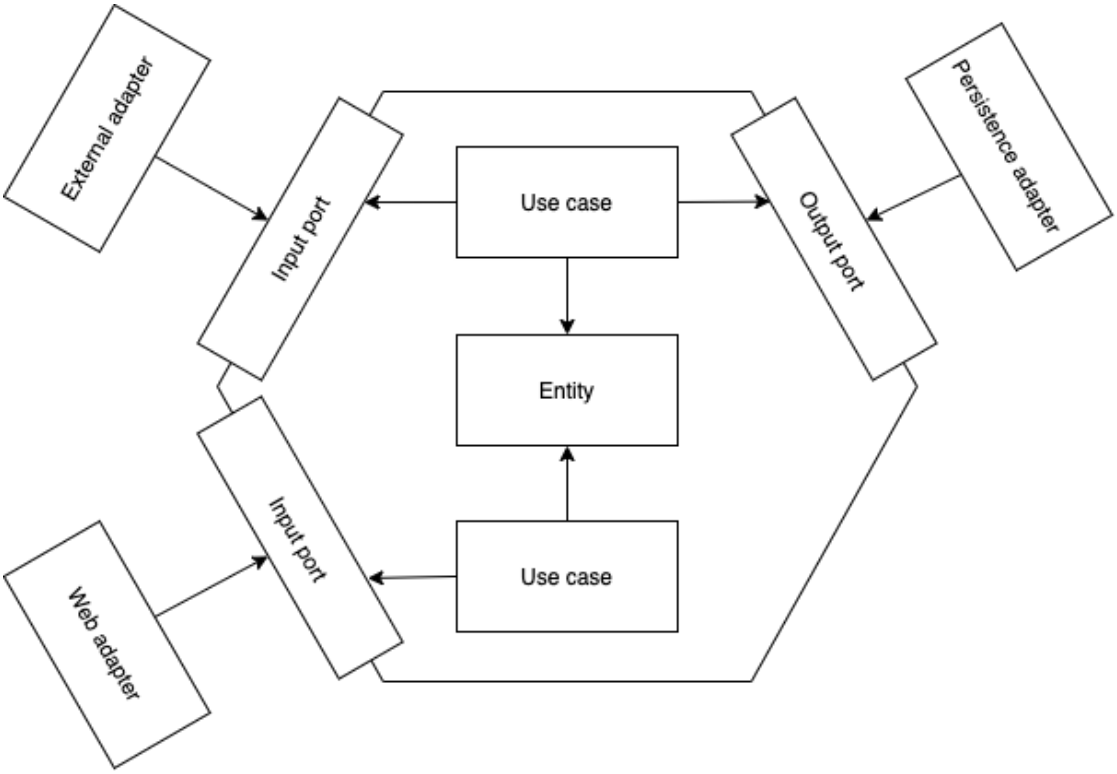

Figure 6. Hexagonal architecture [18].

very similar to the ports and adapters architecture, their advantages and disadvantages compared to the n-tier architecture are the same [11].

[11] https://dzone.com/articles/5-major-software-architecture-patterns

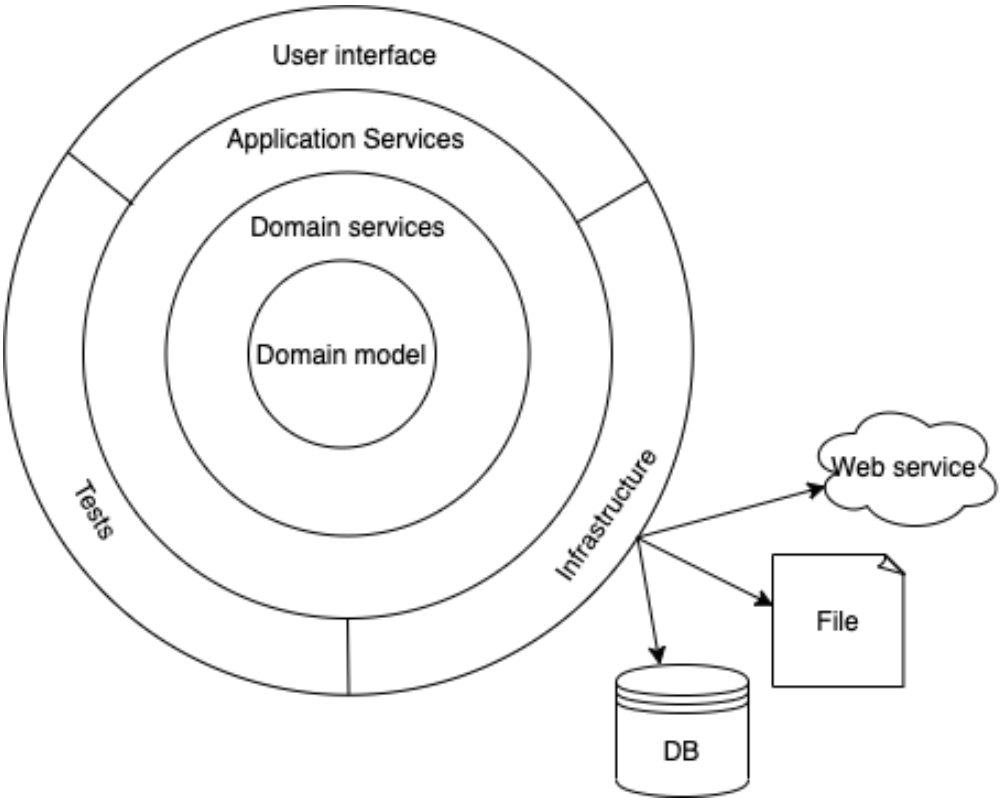

Figure 7. Onion architecture [19]

3 Method

The main goal of this thesis project is to build an application to visualize code smells. The application needs to take a static analysis of some source code and present it visually. For each class of the source code, the application needs to visualize the severity of code smells in the class. Some additional features for viewing growth of the source code and evolution of code smells should be added. The tool uses static code analysis to gather all the data about the application. Further features of the application are found based on the requirements.

This section focuses on the methodology used to achieve the goal of the thesis project. The section explains how requirements will be gathered for the application. In addition, the chapter answers the questions of what technologies are going to be used to develop the software and how the selection of the technologies is made. Furthermore, the section explains how the developed software will be evaluated.

The development of the software, as well as the structure of Method and Results chapter of the thesis follow the standard waterfall method[12]. As it can be seen from figure 8 the development process begins with gathering requirements, followed by designing the software, implementing the software and evaluating it. There main reason for selecting waterfall is that the method works well for small projects if the requirements are well-defined [20]. Since the scale of the development is small and the author of the thesis is the only developer working on the project there is no need to refine requirements or evaluate their complexity.

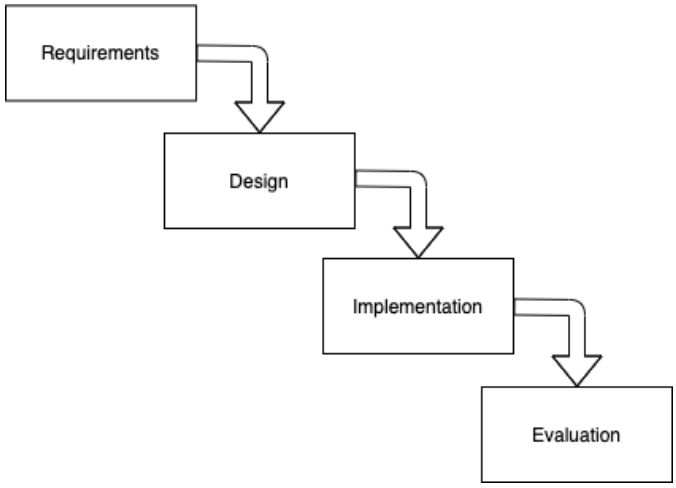

Figure 8. Illustration of waterfall method used in the development [19]

[12]https://economictimes.indiatimes.com/definition/waterfall-model

3.1 Requirements

In order to gather high-level requirements of the software, multiple interviews are conducted with stakeholders from projects A and B. In addition to gathering requirements, the purpose of the interview is to understand whether the software is needed at all and who could be the users of the software.

3.1.1 Interviewees

It is important to note that at the current state, it is unknown which stakeholders in the company could be potential users of the application. The gathering of the requirements is done after the users of the application are determined. Table 1 shows which stakeholders in the author's opinion need to be interviewed in order to understand whether the application can be used in production. Developers are chosen as they face code smells on a daily basis and might benefit from the visualization of the code smells. Quality Assurance and Product Owners need to be interviewed because usually they decide whether some task, use case, or user story is accepted or not. The tool might be an additional layer for validation of the software. Scrum masters are interviewed as they might find the application useful for visualization purposes in client demos or on daily scrum meetings. Managers don't need to participate in the interviews because generally they don't work with the development of the software. Clients are also excluded from the interviews as code smells are not in their field of interest and expertise. The visual representation of the code might be used during client demos, but clients are not the ones who use the tool. In these scenarios, tool is still operated by the development team.

3.1.2 Interview

In the interview, it is critical to understand whether the stakeholder will benefit from the tool or not and to collect some ideas about usages for the tool. With that in mind, some of the questions are asked to understand stakeholders' previous needs for the application, and other questions are open-ended discussions with brainstorming [13]. Figure 52 in the Materials shows what questions are in the interviews. All the interviews are informal, meaning some of the questions might be left out, and some questions are not the same as presented on Figure 52. The interviews are informal because all interviewees are the coworkers of the thesis author and making interviews formal might affect the flow of brainstorming. During the interviews, an interviewer is taking notes, but discussions are not be recorded. The reason for not recording interviews is also to avoid placing interviewees into uncomfortable situation that might affect brain storming and idea sharing.

[13] https://www.mindtools.com/brainstm.html

Table 1. Some stakeholders in the company and whether they will be interviewed

Stakeholder	Is interviewed	Reason for interviewing / not interviewing
Developer	Yes	Most of the developers use static code analysis and work with code smells on a daily basis.
QA	Yes	Could be used as an additional layer of validation in tests. Could find visualisation useful for maintainability purposes.
Product Owner	Yes	Might find the application useful when accepting user stories or check the maintainability of the product,
Scrum master	Yes	Might use the tool in demos or for some daily scrum activities
Manager	No	Does not work with the code or application in general.
Client	No	Works with application, not the development. Client should not validate quality of the code.

3.1.3 Requirements prioritization

Prioritization of requirements is done using the MoSCoW prioritization technique. In this technique, each requirement is marked with one of the following letters: M - must have, S - should have, C - could have, W - will not have [21]. Requirements that are marked as must have, must be implemented before software releases, should have and could have requirements are good to have but not necessary and will not have requirements are the requirements which will not be done during current iteration [22]. Note marking *will not have* does not mean that requirements can be ignored. It just means that the implementation of these requirements is postponed until future iterations.

The reason for prioritizing requirements using the MoSCoW technique is its simplicity. Four categories of classification are straightforward and self-explanatory. Other considered prioritization technique was Simple ranking, meaning each requirement is assigned a number from 1 to n based on its priority in the application [22]. The choice between two techniques was made based on the authors personal preference.

3.1.4 Requirements traceability

The book *Perspectives on Software Requirements* written by Francisco A. C. Pinheiro describes that very often, single requirements can derive into many others, and multiple requirements can collapse into one requirement [23]. The author of the book emphasizes that in such cases, traceability of the requirements is important for the success of the software product. Pinheiro mentions that for traceability matrix is one of the well-known

techniques for requirements traceability. The book describes the traceability matrix as a table that captures requirements relationships towards each other and towards the system's functionality.

Requirements traceability matrix is used for tracing requirements in the project. Selection of the technique was made based on the authors personal preference, and the ease of use [23]. Other considered traceability techniques were traceability graph and traceability hyperlinks [24].

3.2 Prototype

The prototype of the application is constructed based on the elicited functional requirements. The prototype is simply for illustrative purposes and is not going to be used in demos. The primary purpose of the prototype is to plan the visualization of the code smells and sketch initial placement of some components. It is important to note that the prototype and final product can look different.

The prototype is made using Figma[14]. There are no specific requirements and restrictions for the prototype, so the selection of the tool was made based on the author's personal preference.

3.3 Choices of technologies and architectures

The implementation of the software can be facilitated by forethinking the technologies and architecture styles before writing the code. This section covers different criteria that are considered when making choices in technologies, programming languages, tools and architectures, that will be used for developing the application.

3.3.1 Technological and architectural choices of database

As mentioned in chapter 2.2, for better code analysis, a static code analyzer should be used. Before the development of the application, it was already decided to use GraphifyEvolution app as a static code analyzer. The main advantage of GraphifyEvolution over other static code analyzers is that it allows users to analyze the entire history of the application by commits [12]. Analyzing the history of the application by commits is important, because based on the author's experience, in some projects, all developers commit to the same branch. This means that analyzing history by branches would not give much result as there is only one branch. SonarQube[15] and FindBugs[16] were also considered as static analyzer options. SonarQube allows users to view code analysis for

[14]https://www.figma.com/
[15]https://www.sonarqube.org/
[16]http://findbugs.sourceforge.net/

each branch[17], but the analysis for each commit is not yet implemented. Findbugs has a functionality for computing bug history, but the feature analyzes all appearances of the bug and doesn't show code smells per class. For these reasons, GraphifyEvolution is preferred over SonarQube and Findbugs. In chapter 2.3 it is described that GraphifyEvolution stores code analysis in the Neo4j database, and code smells can be acquired by using queries provided by the author of the tool. This means that the application that is going to be developed also needs to support the Neo4j database, so no additional database selection is required.

Since Neo4j is not a very widely used database language[18][19] it is vital that the database is easy to install and can be accessed without additional setup. This will alleviate the discomfort of the new development environment setup for future developers. To do that, the database can be encapsulated and containerized inside the application that will be developed. This way, the database can be set up by running a script that comes with the repository meaning that installation of Neo4j will not be required. The selection of the container software is made based on the availability of examples and the authors personal preference. The availability of examples is important because the Neo4j database is not very widely used, and the container setup can be more challenging than setting up a container for a more popular database.

3.3.2 Technological and architectural choices of back-end application

Architecture. There are many aspects that are taken into consideration when selecting an architecture for the application. Since the application is going to use the GraphifyEvolution application as a static analysis tool, it is important that the communication with GraphifyEvolution is encapsulated and separated from the domain logic. The reason for encapsulation and separation is the maintainability and future developments of the application. The development process should take into consideration that in the future, there might be a need to change the static analysis tool. In addition, there might be a possibility that the database of the application will be changed. As it was mentioned in chapter 3.3.1, the application is using Neo4j database however, the application should allow to easily change the database to any other relational or non-relational database. In addition to encapsulation and separation of the domain logic, communication with the database and communication with GraphifyEvolution, the application should be testable. For maintainability and future development purposes, the application should be covered with unit and integration tests. This means that domain logic, database connection, connection with GraphifyEvolution, and other parts of the application should be testable independently. In conclusion, the selection of the backend architecture is based on how well does the architecture support testability and encapsulation of different components.

[17] https://docs.sonarqube.org/latest/branches/overview/

[18] https://www.explore-group.com/blog/the-most-popular-databases-2019/bp46/

[19] https://www.geeksforgeeks.org/top-10-open-source-nosql-databases-in-2020/

Technologies. The programming language and the framework are chosen based on the selected architecture, Neo4j database support, and author's personal preference. The language and framework should support graph database meaning there should be some libraries that provide communication with the database and construct graph entities based on the application objects. In addition, the language and framework should support the architecture. For example, if the hexagonal architecture is selected, the language should support separating domain logic, ports and adapters from each other.

3.3.3 Technological and architectural choices of front-end application

architecture. Similarly to the back-end part of the application, in the front-end application encapsulation and separation of the components is important. The architecture should support the idea of keeping different components separately. However, unlike in the back-end part of the application, most of the modern front-end development frameworks support grouping components into libraries and keeping libraries separated from each other. That means that unlike in the back-end part, the front-end architecture choice is based on the framework that is chosen for development.

technologies. Technological choices like programming language and framework are selected based on the author's personal preference. Here it can be mentioned that the variety of the front-end programming languages is not as wide as for the back-end languages, so most probably JavaScript or TypeScript with HTML and CSS will be used as they are by far the most used front-end programming languages [20]. The state management toolkit is selected based on the compatibility with the main programming language and the architecture of choice. For better design of the components, an additional library needs to be used. The selection of the library is made based on the author's personal preference of the design and the availability of required components in the library. When it comes to visualization of the code smells, the drawing framework selection is decided based on the quality of the documentation, ease of use, support of the programming language, quality of animations, community size, browser support, number of existing components and modernity of the framework. Since there are numerous open-source visualization tools, it is easy to select an unsuitable frameworks and that can cause complications in the development process. Thus the number of criteria that frameworks are judged against is also quite high.

[20] https://insights.stackoverflow.com/survey/2020

3.4 Evaluation

The evaluation of the software is done with testing and interviews.

Manual and automated testing. Testing of the application is done using three testing techniques: unit testing, integration testing and system testing. Unit tests ensure that domain logic and other different parts of the application work well independently. Integration tests verify that software components work together as a unit. System tests are used to make sure that software satisfies elicited requirements for the current application. For unit and integration tests, test coverage is measured to check whether all use cases of the application are tested.

Interviews. In order to get feedback from the users multiple interviews are conducted. Similarly to requirements gathering described in section 3.1, interviews are conducted with different stakeholders from two projects. If in the requirements gathering interview analysis it is found out that the stakeholder will not benefit from the application, then the representatives of these stakeholders are not going to participate in the evaluation interview. Interviews are conducted based on the technology acceptance model (TAM) [25]. TAM presumes that systems potential usage can be determined based on two variables: *perceived usefulness* and *perceived ease of use* [26]. *Perceived usefulness* showcases how useful users think the new technology is and *perceived ease of use* shows how easy it is to use the software [25]. In the interviews, the author of the thesis will ask interviewees whether they will use the application, what they like or dislike about the application, how well they understand what the application does, how will they rate the user interface and user experience, etc. Similar to requirements gathering interviews, all evaluation interviews are informal for the same reasons mentioned in chapter 3.1.2. The interviews are not recorded so the interviewees feel more comfortable. Interview plan with questions that are asked will be covered in the Results part of the thesis.

4 Results

This section focuses on the work that was done by the author of the thesis. The section covers the analysis of the interviews that were conducted before the development, requirements of the software that will be developed, description of the software from an architectural and technological standpoint, and evaluation of the application that was made.

4.1 Requirements

As mentioned in section 3.1, requirements are written based on the interview results and additional brainstorming. The section covers requirements gathering interview results and functional and non-functional requirements that were elicited based on the interview analysis.

4.1.1 Interview results

The interviews were conducted with five Developers, two Quality Assurance specialists, two Product Owners, and two Scrum Masters.

Discussions with Quality Assurance and Product Owners. From the interviews, it was conducted that neither quality assurance nor product owners felt that their work would benefit from the tool since they don't work with the code smells directly. One product owner mentioned that the tool could be used by product owners to create technical tasks based on the visualizations, but it was also mentioned that developers could also do that, and since product owners have much less experience with the technical side of the project, the creation of technical tasks should be done by developers. Quality Assurance representatives said that they are expecting code quality to be already checked before the task is sent to them, meaning once again that it is the developers' responsibility to keep the code clean.

Discussions with developers. Developers mentioned that an average developer would not need to visualize code smells in the entire application, and simple static analysis would be enough to cover their needs. However, it was noted that senior and lead developers or architects might benefit from the tool as their job is to keep an eye on the entire code base and such a tool might help them in doing that. It was noted that the tool, however, needs to have some additional metric that static code analysis cannot present, for example, to visualize code smell in correlation with the static or dynamic class usage. By static class usage, it was meant that how many other classes use methods or variables from the observed class. By dynamic analysis, it was meant how much do

users actually use the class. Usage of the classes can be found, for example, by reading logs, but dynamic code analysis is out of the scope for this thesis.

Discussions with scrum masters. When discussing the code smell visualization with scrum masters, the idea was proposed to use the application in client demos. The scrum master, who is also a developer, mentioned that in SAFe (Scaled Agile Framework), there is a sprint designed for developers to fulfill their needs in trying out new technologies or solutions and to relieve technical dept that has accumulated during previous sprints. The code smell visualization tool can be used in client demos in order to show visualizations of code before and after the sprint. This way, the client will be knowledgeable about the work done during the sprints if some refactoring was done. For that reason, the tool needs to support viewing the history of the code and comparing branches in case not all the code is merged into one branch. In conclusion, the application seems to be beneficial for developers who want to have a simple overview of the entire application or the person who is presenting sprint work to the customer and needs to be able to show differences between source code at different times. Requirements that were conducted based on the interview are described in upcoming sections.

4.1.2 Functional requirements

List of elicited functional requirements with requirement ID and description can be seen in table 3. Requirements actors are descried in table 2 in the Materials. Full list of requirements is described in table 15. Based on the functional requirements, use cases were constructed. Use cases can be seen in the use case diagram displayed in figure 9.

Table 2. Functional requirements actors and their description

Actor	Description
User	Regular user of the software: developer, scrum master or other
Advanced user	Person who knows what code smells are, how they are calculated and what does each parameter of the code smell does

Table 3. Functional requirements

ID	Description
CSV-FR1	User should be able to view all existing code smells according to their type
CSV-FR2	User should be able to see impacts of individual code smells on the application
CSV-FR3	Advanced user should be able to adjust code smell parameters
CSV-FR4	User should be able to view code analysis
CSV-FR5	User should be able to distinguish between severity of code smell in different classes of the application
CSV-FR6	User should be able to distinguish between often used and rarely used classes
CSV-FR7	User should be able to see classes in the same packages
CSV-FR8	User should be able to adjust coloring of the code analysis
CSV-FR9	User should be able to navigate in the application
CSV-FR10	User should be able to view branches in the application
CSV-FR11	User should be able to select branches in the application
CSV-FR12	User should be able to view commits under the branch
CSV-FR13	User should be able to select commits
CSV-FR14	User should be able upload application
CSV-FR15	User should be able to select the project
CSV-FR16	User should be able to see methods that belong to the class
CSV-FR17	User should be able to see code smells severity in the methods

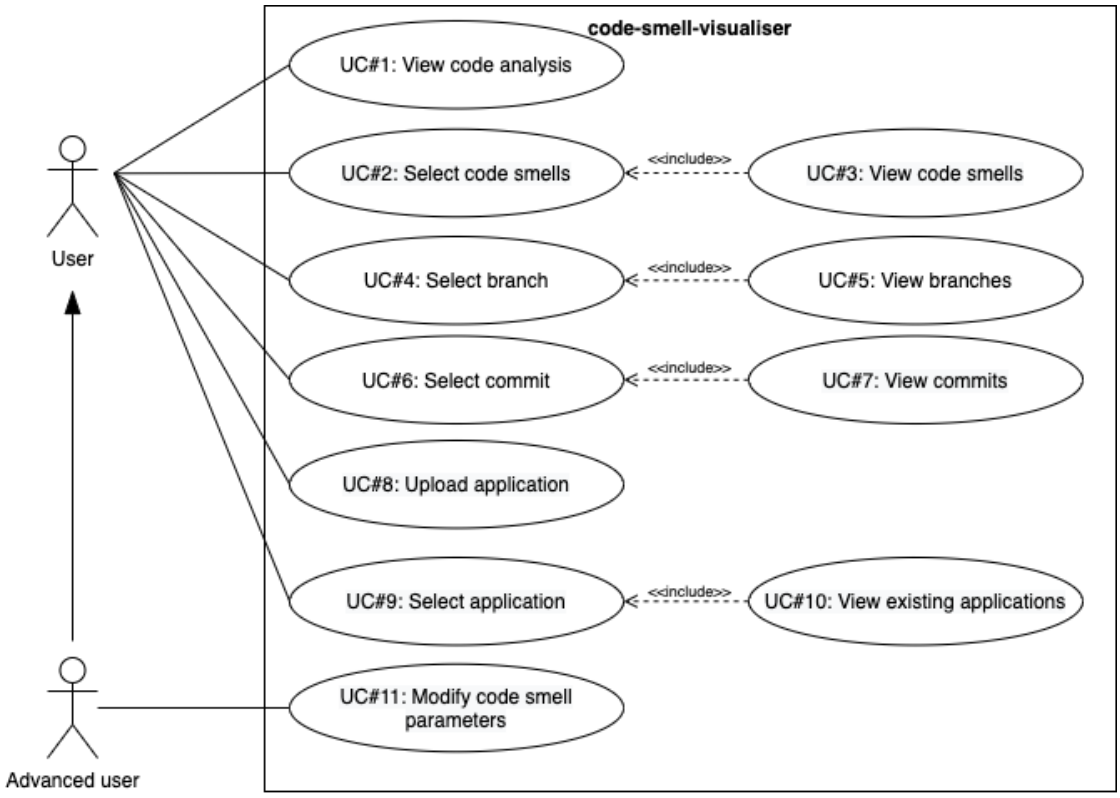

Figure 9. Use case diagram

4.1.3 Non-functional requirements

List of elicited non-functional requirements with requirement ID and description can be seen in table 4. Full list of requirements with success criteria is described in table 32.

Table 4. Non-functional requirements

ID	Description
CSV-NFR1	User should be able to see code smells analysis in Google Chrome
CSV-NFR2	Application is covered with tests
CSV-NFR3	Domain logic, web layer and database layer should be separated from each other

4.1.4 Requirements prioritization

Prioritization of the requirements is done using the MoSCoW prioritization technique. Each requirement is assigned one of the following priorities: M - must have, S - should have, C - could have, and W - will not have. Prioritization results of functional requirements are shown in the table 5. Results of prioritizing non-functional requirements is shown in table 6. Requirements marked as must have requirements are the first priority and are implemented first. After that, additional features are added based on requirements that are marked as should have and could have. Requirements marked as 'will not have' are not in the scope of the thesis and are implemented. This however, doesn't mean that requirements will not be done in the future.

4.1.5 Requirements traceability

Table 7 captures traceability of functional and non-functional requirements specified in sections 4.1.2 and 4.1.3. In the table abbreviation P stands for precondition and C for constraint. For example requirement CSV-FR1 is precondition for requirement CSV-FR2 and requirement CSV-NFR1 constrains requirement CSV-FR1. In the table it can be seen that all non-functional requirements constrain all functional requirements of the system.

Table 5. Functional Requirements prioritized based on MoSCoW model

Requirement ID	**Priority**
CSV-FR1	M - must have
CSV-FR2	C - could have
CSV-FR3	S - should have
CSV-FR4	M - must have
CSV-FR5	M - must have
CSV-FR6	M - must have
CSV-FR7	M - must have
CSV-FR8	C - could have
CSV-FR9	M - must have
CSV-FR10	S - should have
CSV-FR11	S - should have
CSV-FR12	S - should have
CSV-FR13	S - should have
CSV-FR14	M - must have
CSV-FR15	M - must have
CSV-FR16	W - will not have
CSV-FR17	W - will not have

Table 6. Non-functional requirements prioritized based on MoSCoW model

Requirement ID	**Priority**
CSV-NFR1	M - must have
CSV-NFR2	M - must have
CSV-NFR3	M - must have

Table 7. Traceability matrix of functional and non-functional requirements

	CSV-FR1	CSV-FR2	CSV-FR3	CSV-FR4	CSV-FR5	CSV-FR6	CSV-FR7	CSV-FR8	CSV-FR9	CSV-FR10	CSV-FR11	CSV-FR12	CSV-FR13	CSV-FR14	CSV-FR15	CSV-FR16	CSV-FR17
CSV-FR1		P															
CSV-FR2																	
CSV-FR3																	
CSV-FR4					P	P	P	P	P							P	P
CSV-FR5																	
CSV-FR6																	
CSV-FR7																	
CSV-FR8																	
CSV-FR9																	
CSV-FR10											P	P	P				
CSV-FR11																	
CSV-FR12													P				
CSV-FR13																	
CSV-FR14																	
CSV-FR15																	
CSV-FR16																	P
CSV-FR17																	
CSV-NFR1	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C
CSV-NFR2	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C
CSV-NFR3	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C	C

4.2 Prototype

Based on the functional requirements listed in section 4.1.2 the prototype was made. The prototype can be seen in figures 10 and 11. As described in section 3.2 the goal of the prototype is to construct an approximate image of the code smells visualization. The prototype was constructed based on all listed functional requirements. This means that the final product is different from the prototype as not all requirements are implemented during the current iteration (see table 6 in requirements prioritisation chapter).

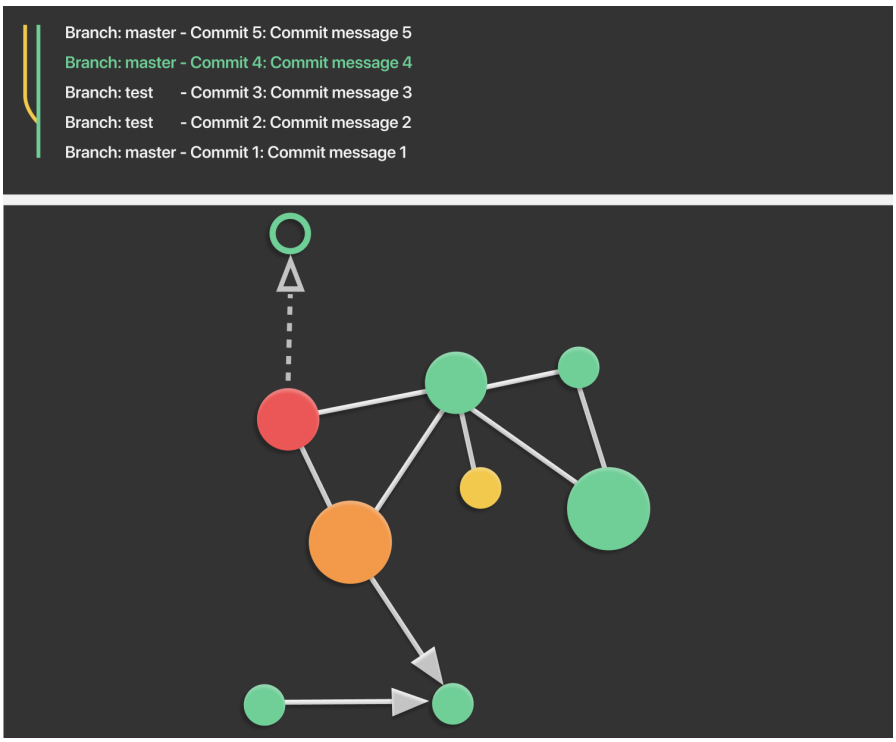

Figure 10. Screenshot from prototype.

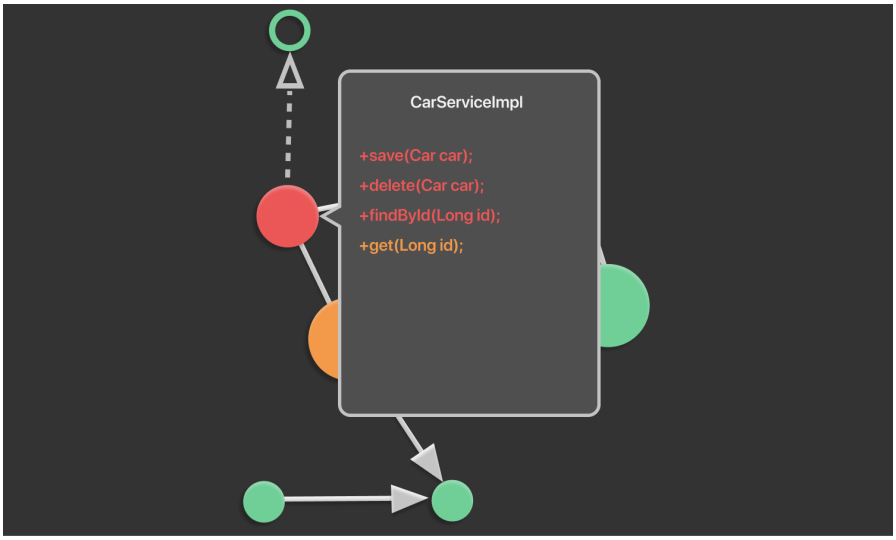

Figure 11. Screenshot from prototype.

4.3 Overview of the implemented system

The developed application is called code-smell-visualizer. Its main purpose is to visualize the impact of different code smells on the classes of the application. The main parts of the application are uploading source code into the application, visualization of the uploaded application, and additional functionalities for adjusting the visualization such as selection of the code smells, adjustment of code smells parameters, etc. All the functionalities are described in upcoming sections.

In this section, the term application under analysis is used. The application under analysis is an application that is analyzed by the code-smell-visualizer.

4.3.1 Uploading the application

The first step in interacting with code-smell-visualizer is uploading source code to the system. For that user can insert the path of the application under analysis into the input field provided by code-smell-visualizer. Figure 12 illustrates source code upload component. In the figure, it can be seen that in addition to uploading the code, user can select already existing applications. Once an application is selected or uploaded, user is redirected to the main page of code-smell-visualizer.

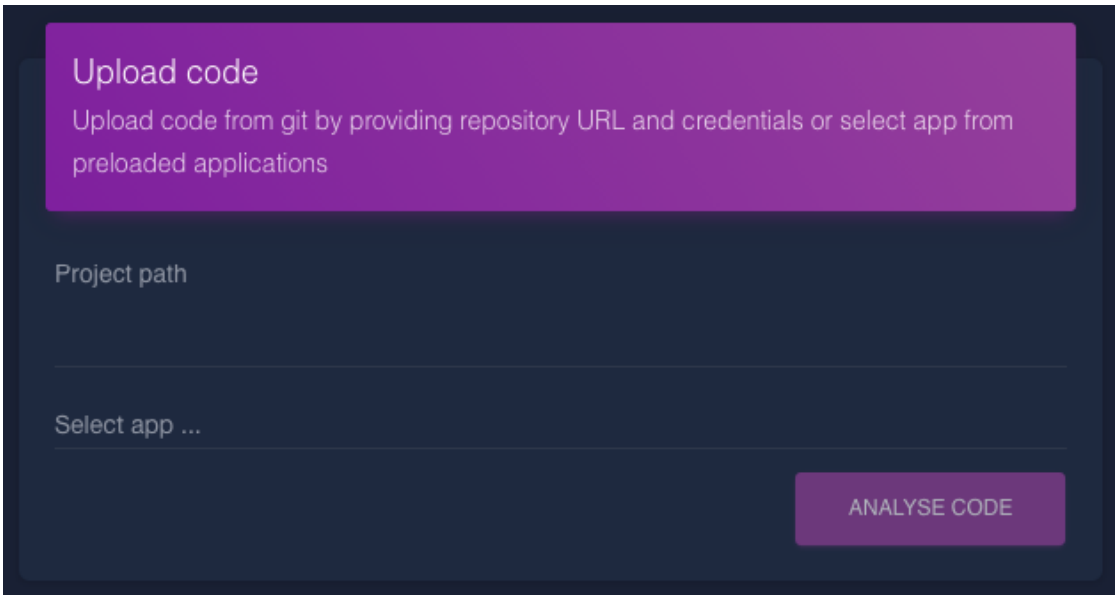

Figure 12. Screenshot of the source code uploading component in code-smell-visualizer.

4.3.2 Code smells selection

Once navigated to the main page, the first thing that user sees is the selection of code smells. The application presents a list of code smells that are grouped based on whether the code smell is class-based or a method-based code smell. Class-based smells are code smells that occur in the class, and method-based smells are the code smells that occur in methods. For example, long method code smell is classified as method-based code smell as it occurs on methods. Class-based code smells can occur in class only once and method-based code smells can occur in every method of the class. In addition to grouped code smells, the impact of each code smell is presented. The impact of the code smell is calculated based on the number of appearances of the code smell in the application under analysis. The impact is presented using circles that are colored based on the following notation:

- Green: Code smell does not appear in the application under analysis at all.
- Yellow: Code smell appears in some classes of the application under analysis.
- Red: Code smell appears in many classes of the application under analysis.

Here it is important to note that number behind *some* classes and *many* classes is configurable in the code-smell-visualizer using the slider described in section 4.3.5. Figure 13 illustrates code smell selection component. In the visual it can be seen that user can select code smells that will be included in the visualization image of the application under analysis (section 4.3.4).

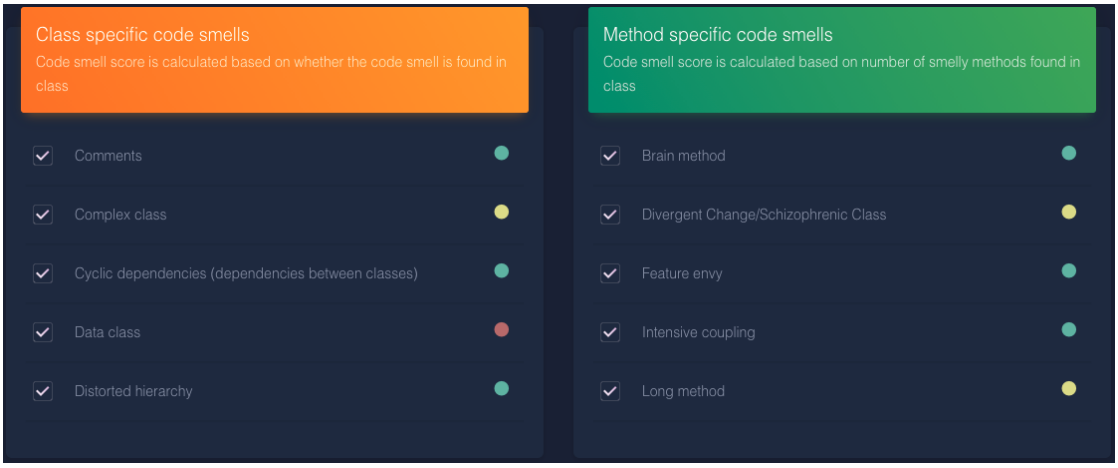

Figure 13. Screenshot of the code smells selection component in code-smell-visualizer.

4.3.3 Code smell parameter modification

Most of the code smells have parameters that can be modified by an advanced user. For that, advanced user is redirected to code smell parameter adjustment page (figure 14). There a user can adjust all the parameters of the code smells. Adjusted parameters apply only to the application under analysis.

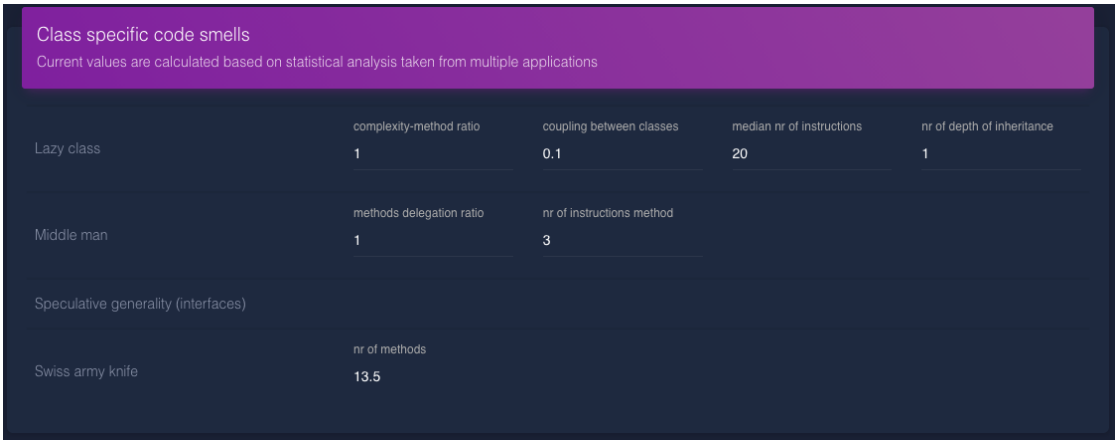

Figure 14. Screenshot of the code smells parameter adjustment component in code-smell-visualizer.

4.3.4 Visualization of the application under analysis

The most important part of the code-smell-visualizer is the visualization of the application under analysis. Figure 15 demonstrates an example visualization of the application under analysis. In the figure, it can be seen that each class of the source code is represented as a circle. The color of the circle represents *code smell score* of the class. *Code smell score* is a numerical value proposed by the author of the thesis to count number of code smells in the class. The *code smell score* is used because class-based code smell can appear in class only once, and method-based code smell can appear in every method of the class, meaning that method-based code smells can appear a lot more in the application. *Code smell score* is counted using the following equation $score = CB + MB/N$, where CB is a number of class-based code smells in the application, MB is a number of method-based code smells in the application, and N is a number of methods in the class. That way, each code smell can affect the *code smell score* value by at most 1. The selection of the color is similar to the color selection of code smell impacts described in section 4.3.2:

- Green: Class has no code smells
- Yellow: Class has some code smells

- Red: Class has a lot of code smells

Similarly to the code smell impact numbers, the actual number behind *some* and *a lot of* code smells can be modified by using the slider described in section 4.3.5.

The size of the circle represents the number of usages that the class has. It might seem counter-intuitive why the size of the circle does not represent the size of the class. The reason is that the size of the class is accounted for in the code smells analysis, meaning that if the class is too big, it is considered a code smell and is visualized using color notation. However, visualizing the size of the circle as the number of usages that the class has, adds an additional metric that code smells analysis does not take into consideration. The idea behind using colored circles with different sizes is that when the user looks at the visual of the application under analysis, they can easily see what classes need refactoring. For example, by looking at figure 15, a large red circle in the bottom left of the visualization pops out. This should indicate the user that the class has many code smells and that the class is used in multiple places, so refactoring of the class should be a high priority.

Circles in the figure 15 are grouped together based on their path, meaning that classes that are in the same folder are grouped together, and classes with similar paths are closer to each other. The grouping of the classes is achieved by splitting the screen into n pieces, where n is the number of unique paths in the application under the analysis. The formula for calculating the best grid size is $(x, y) = (\lceil \sqrt{m} \rceil, \lceil \sqrt{m} \rceil)$, where m is the number of unique paths in the application under analysis. After the grid is calculated, the center of each grid square is found based on the width and height of the window, and each package is given unique center coordinates of the grid.

4.3.5 Adjustment of the colors with slider

As mentioned in previous chapters, the coloring of the circles is adjustable by the user. By using slider, users can select the range of green, yellow and red colors. Figure 16 shows two visualizations of the application under analysis with two different slider adjustments. Notice that on the bottom image slider is adjusted, so that range for color green is extensive, meaning that *code smell score* of the class has to very high in order for a class to become yellow.

User needs to be able to adjust the range of *code smell score* because in different projects there is a different level of code smells allowed. Some legacy projects contain more code smells and that is considered a norm, and other projects, typically newer ones, have more strict policies regarding code smells.

4.3.6 Branch and commit selection of the application under analysis

The visualization of the application under analysis is performed on the latest commit of the master branch, however user can select any branch and any commit of the application

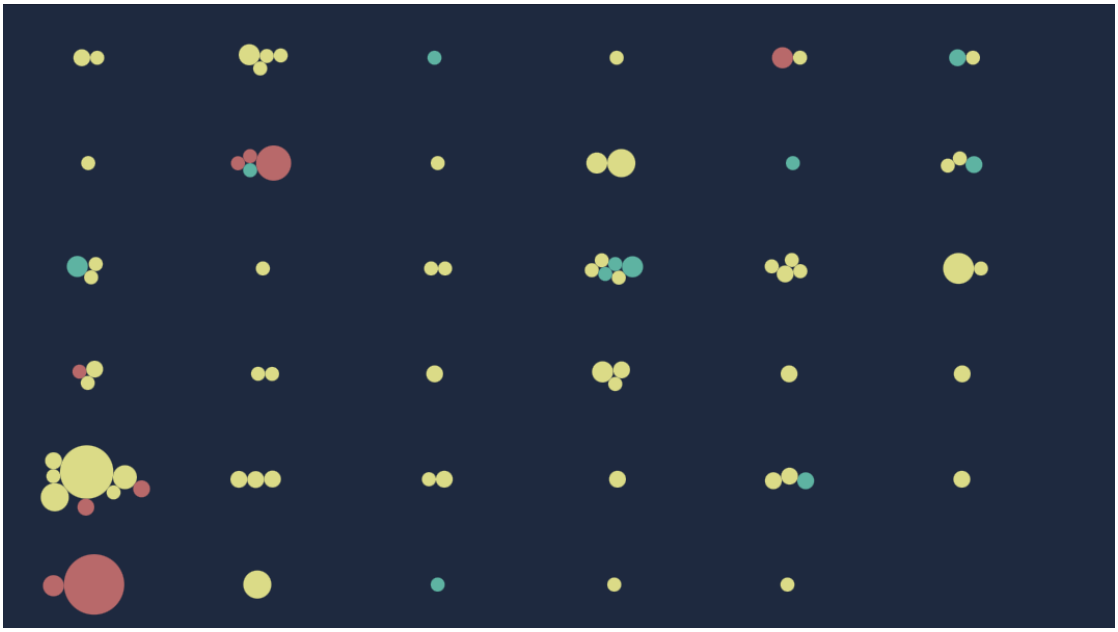

Figure 15. Screenshot of the visualization of application under analysis.

under analysis. This functionality is needed not only in case the master branch doesn't exist in the application but also to provide the user ability to see the evolution of the application and its code smells. Once user has selected the branch, additional component visualised in figure 17 becomes visible. Here user can select any commit that has been done under the selected branch, and the application visualization will be adjusted accordingly.

Figure 16. Screenshot of the visualization with different slider adjustments

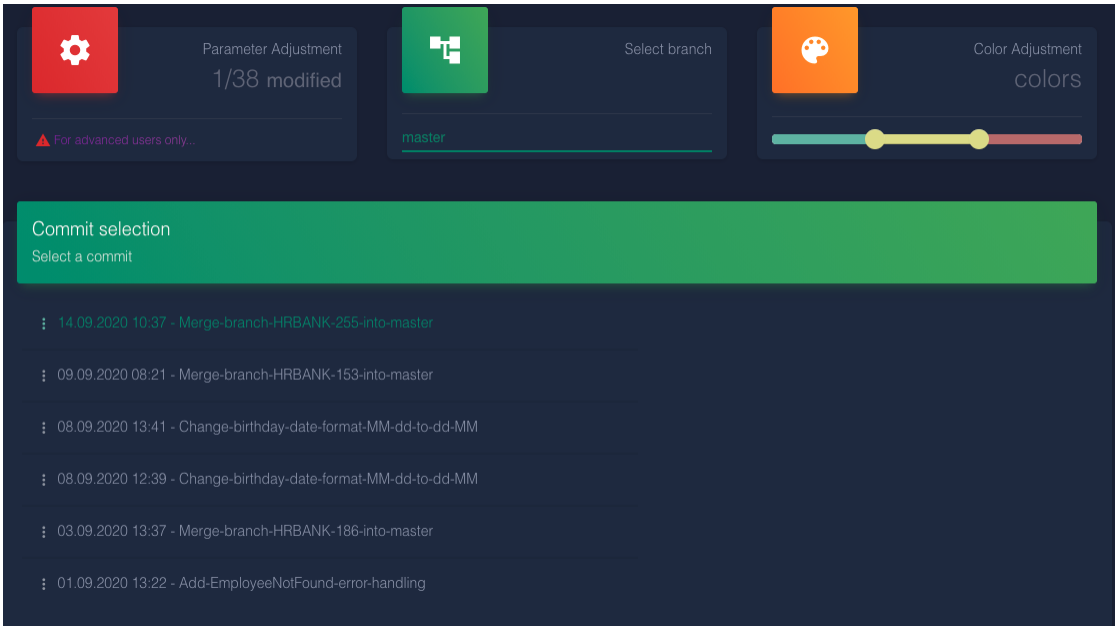

Figure 17. Screenshot of the branch and commit selection in code-smell-visualizer.

4.4 Top-level architecture of code-smell-visualizer

The code-smell-visualizer application is divided into three separate parts: code-smell-visualizer-api, which is the backend part of the application, code-smell-visualizer-web, which is the frontend part of the application, and Neo4j repository, which is the database of the project. Neo4j repository is stored inside Docker container. In addition, code-smell-visualizer contains two submodules: GraphifyEvolution and dev-scripts. All parts of the application will be described in upcoming chapters. As it can be seen on figure 18, code-smell-visualizer-web communicates only with code-smell-visualizer-api and code-smell-visualizer-api gets the data from Neo4j repository. The software is designed in a way that code-smell-visualizer-api knows nothing about the frontend, and Neo4j repository doesn't know about anything else in the application.

4.5 Dev scripts

Figure 18 shows that the database of the application is inside the docker container. As it was mentioned in chapter 3.3.1, since the static analysis tool (GraphifyEvolution) that is used for the development of code-smell-visualizer uses the Neo4j database, there is no need for making any choices related to the database selection. However, as mentioned in chapter 3.3.1, the database needs to be containerized to simplify the setup process of the application for future developments. For containerization, Docker was used. In addition

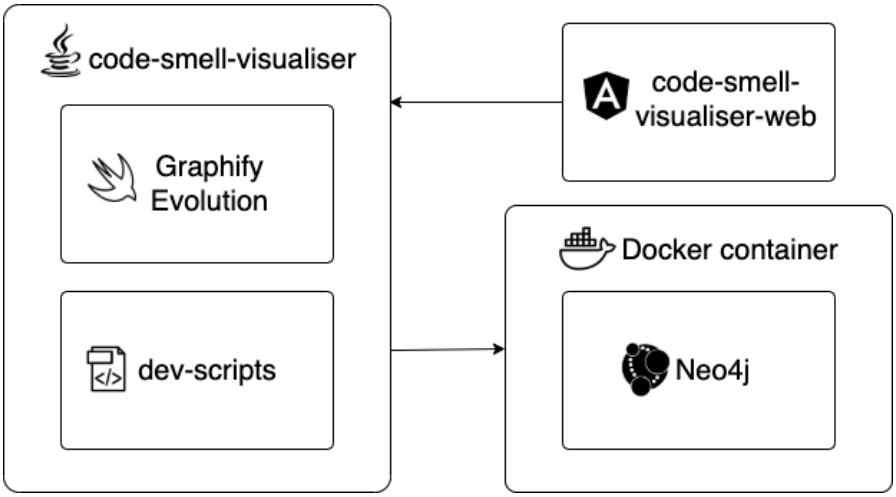

Figure 18. Architecture of code-smell-visualizer.

to authors personal preference, Docker is selected as it has large community [21] and good Neo4j support [22] [23] [24] [25], meaning there are many examples explaining how to set up Neo4j database in docker container. Since the selection of the container software is based on the authors personal preference, no other container software was considered. The scripts that are used to compose the docker container are inside dev-scripts module, which is a sub-module of the code-smell-visualizer-api. The script that is used to compose the docker container can be seen in figure 53 in the Materials.

4.6 code-smell-visualizer-api

As visualised in Figure 18, code-smell-visualizer-api contains of three parts: main logic, dev-scripts sub-module and GraphifyEvolution sub-module. This subsection focuses only on main logic. Description of sub-modules is done in separate chapters.

4.6.1 Architecture

Three architecture styles that are described in section 2.4 are compared in order to select the most suitable type of architecture for developing code-smell-visualizer-api.

[21] https://www.datadoghq.com/docker-adoption/
[22] https://hub.docker.com/_/neo4j
[23] https://neo4j.com/developer/docker-run-neo4j/
[24] https://neo4j.com/labs/kafka/4.0/docker/
[25] https://thibaut-deveraux.medium.com/how-to-install-neo4j-with-docker-compose-36e3ba939af0

N-tier architecture The reason why this architecture type was not chosen for code-smell-visualizer-api is the lack of separation between the data layer and the business layer. As mentioned in chapter 3.3.2 one of the key aspects of developing code-smell-visualizer-api was to keep it as independent from GraphifyEvolution as possible. Additionally testing domain logic separately from web and persistence components can be more complicated than in Onion and Hexagonal architecture.

Onion vs. Hexagonal architecture Since Onion architecture is very similar to hexagonal architecture, the choice between two architecture types was made based on the author's preference. Since the application is relatively small compared to real production applications, there is no need to separate business logic into multiple layers as Onion architecture suggests. Considering that and the author's previous experience with Hexagonal architecture, the Ports and Adapters architecture style was selected for the development of the code-smell-visualizer-api. Both architecture types were preferred over n-tier architecture as they separate domain logic from the rest of the application and testing it separately is easier than in n-tier architecture.

4.6.2 Technologies

The main technologies used in code-smell-visualizer backend application are Java 11 with Spring Boot 2.3.2s. The database queries are written with Neo4j-OGM and Cypher.

Java and Spring Boot The main reason for choosing Java as a primary programming language in addition to the author's personal preference, is good Neo4j repository support. Spring framework provides *spring-data-neo4j* library, which allows developers to easily write Neo4j queries as Java interfaces using string annotation provided by Neo4j-OGM.

Neo4j-OGM. Neo4j-OGM (Object Graph Mapper)[26] is an extension of a widely used Object Graph Mapper framework Hibernate-OGM[27]. Hibernate-OGM is specifically designed to target the mapping of NoSQL data stores. Neo4j-OGM is even more definitive as it targets only Neo4j data store entity mappings[23]. Both frameworks simplify the usage of CRUD operations by providing mappers that convert existing POJOs (Plain Old Java Object)[28] into database entities, thus alleviating the problem of mismatching field names. Conversion between Neo4j entities and POJOs is done by annotating Java entities with labels that specify the name of Neo4j node name, name of the field, or relationship between the Neo4j nodes. Figure 19 shows the usage of Neo4j-OGM annotations *@NodeEntity*, *@Property* and *@Relationship*.

[26] https://neo4j.com/developer/neo4j-ogm/
[27] https://hibernate.org/ogm/
[28] https://martinfowler.com/bliki/POJO.html

```java
@NodeEntity(label = "CodeSmell")
public class CodeSmellNode {

    @Id
    @GeneratedValue
    private Long id;

    @Property("app_id")
    private Long appId;

    @Property("name")
    private String name;

    @Property("type")
    private String type;

    @Property("description")
    private String description;

    @Relationship(type = "HAS_PARAM")
    private List<ParameterNode> parameters;
}
```

Figure 19. example of Neo4j-OGM annotations Java object.

Annotation *@NodeEntity* identifies Neo4j node. The parameter *CodeSmell* is the name of the node. *@Property* annotation describes the field names of *CodeSmell* node. *@Relationship* annotation describes the relationship between different nodes. The name of relationship is given as parameter. In Figure 19, the name of relationship is *HAS_PARAM* and it describes relationship between *CodeSmellNode* and *ParameterNode*. By marking *CodeSmellNode* class with mentioned annotations the entity can now be used in writing queries. Figure 20 shows an example of how *CodeSmellNode* and its field *appId* were used to formulate a query.

Cypher. In some cases, simple CRUD operations are not enough, which means using Neo4j-OGM is not sufficient. When requesting code smells analysis, some code smell recognition queries are too complex to describe them as a single string. In those situations Cypher[29] queries are used. Cypher is an open-source native query language for Neo4j's data store, which allows developers to write complex yet logical and understandable queries for matching nodes and relationships in the graph [27].

[29]https://neo4j.com/developer/cypher/

```java
import org.springframework.data.neo4j.repository.Neo4jRepository;
import org.springframework.data.repository.query.Param;

import java.util.List;

interface CodeSmells extends Neo4jRepository<CodeSmellNode, Long> {

    List<CodeSmellNode> findAllByAppId(@Param("app_id") Long appId);
}
```

Figure 20. Example of Neo4j repository and OGM

4.6.3 code-smell-visualizer-api modules

The code-smell-visualizer-api application is separated into 4 parts: Adapters, Application, Common and Configuration. Each of the four parts is implemented as a separate module. Additionally, the Adapters module is separated into three modules: graphify, persistence and web module.

Common. Common is the module where all reusable components or parts of the application are. Currently, common module contains only annotations, which are used to add readability to the code. As it can be seen from the example annotation code in Figure 21, all annotations are marked as Spring Boot components (@*component* annotation), which essentially allows dependency injection of those components. For example, all use cases are marked with @*Usecase* annotation. This allows other modules to use core business logic by autowiring [30] use cases.

```java
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface UseCase {

    @AliasFor(annotation = Component.class)
    String value() default "";

}
```

Figure 21. Code snippet from definition of custom component annotation

[30] https://dzone.com/articles/autowiring-in-spring

Application. The Application module is the center of the code-smell-visualizer-api as it contains core business logic. The Application module has no external dependencies other than common module dependency, meaning it can operate entirely on its own. As it can be seen from Figure 23, core business logic contains domain model and use cases. The domain model of the application can be seen in figure 22. All use cases in the application module are linked to use cases described in figure 9.

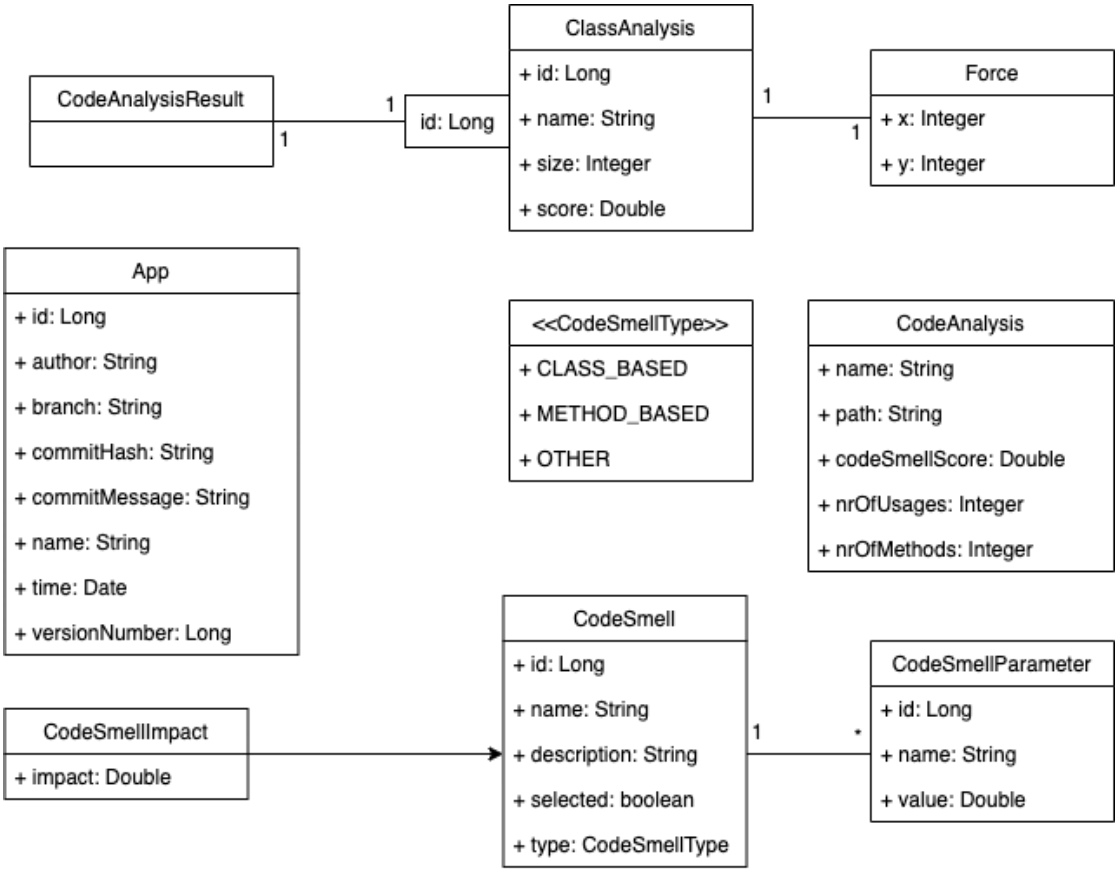

Figure 22. Domain model of code-smell-visualizer-api.

In addition to use cases and the domain model, the Application module provides multiple gateways for communication with adapters. Gateways are developed as interfaces. In essence, the connection between the adapter and the Application module is established by the adapter implementing the gateway.

Web adapter. The main purpose of the Web adapter is to serve as connection layer between code-smell-visualizer-web and code-smell-visualizer-api. The Adapter depends

on Common and Application modules. Web adapter receives an input signal from code-smell-visualizer-web and translates it into data that is suitable for the corresponding use case. Note that in figure 23 Web adapter is connected directly to the core business logic and no gateway is used. Since the dependencies between Web adapter module and Application module is inward (Web adapter knows about Application module, but Application module does not know about Web adapter), core business logic remains isolated and rules of hexagonal architecture are not broken. Web gateway is not used in order to preserve better understand-ability of the code and preserve the natural flow of use case (Described in section 4.6.4).

Persistence adapter. Persistence adapter serves as connection between code-smell-visualizer-api and the database. Similar to Web adapter, the Persistence adapter also depends on Common and Application modules. As it can be seen from figure 23, the Persistence adapter is connected to the application using Persistence gateway. As mentioned in the Application paragraph, the Use case communicates with the database via the Persistence gateway, which in essence, is an interface implemented by the Persistence adapter.

Graphify adapter. The final adapter of the application is the Graphify adapter. The main purpose of the adapter is to communicate with GraphifyEvolution application. As Web and Persistence adapters, Graphify adapter depends on Common and Application modules. The Application module communicates with the Graphify adapter module by providing the Graphify gateway interface, which the Graphify adapter implements. Integration with GraphifyEvolution is covered in chapter 4.6.5.

Configuration. Configuration module can be described as a glue that holds an entire application together. The module depends on all the previously mentioned modules. The module has only one class with one method: main. The purpose of the module is to start the Spring application. In addition module contains all the integration tests of code-smell-visualizer-api, but testing is covered in future sections.

All dependencies between modules that were mentioned above can be seen in Figure 24. More in-depth analysis of application flow is described in future chapters.

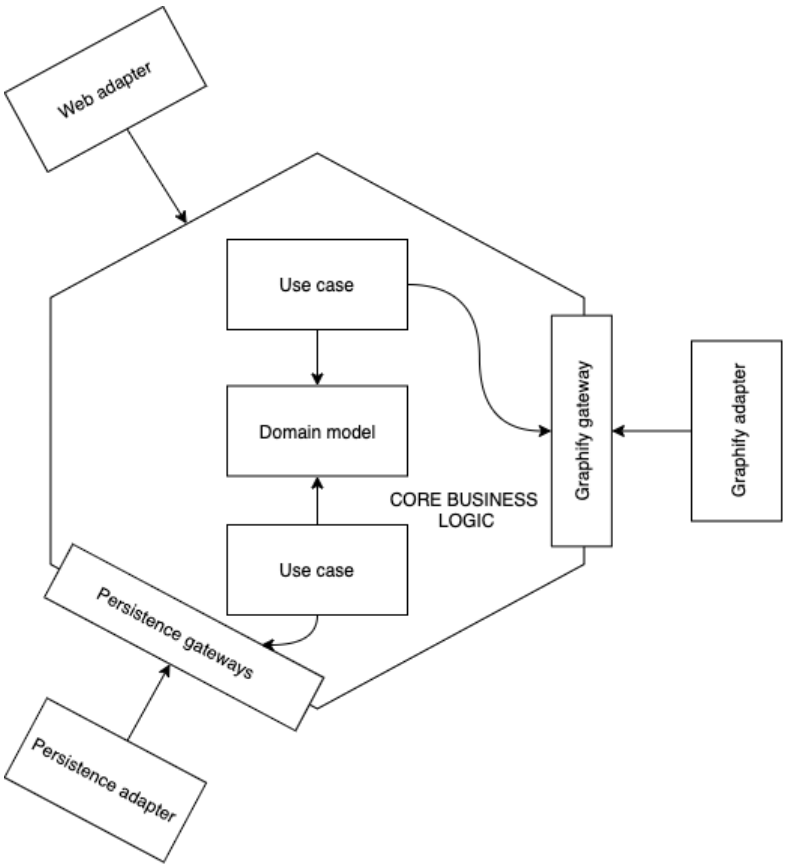

Figure 23. Abstract representation of hexagonal architecture in code-smell-visualizer-api.

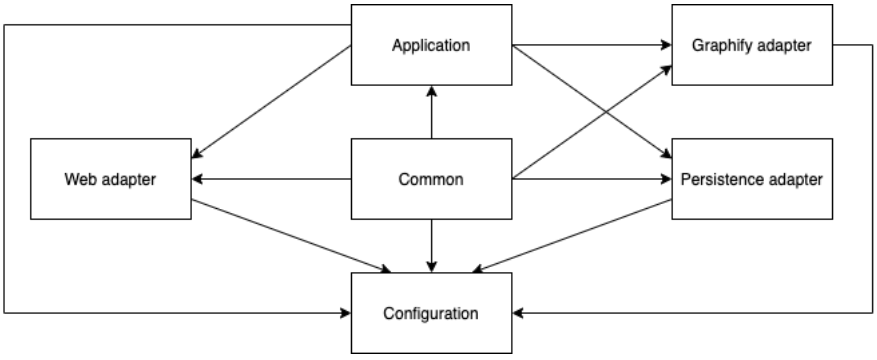

Figure 24. Dependencies between code-smell-visualizer-api modules.

4.6.4 Example flow

In this section example flow of communication between different modules is shown. Example taken from the code is based on use case UC#7: View commits (see figure 9). The reason this use case is chosen as an example flow is that it is one of the simpler use cases, meaning it doesn't have any complex business logic nor technological components, and it covers necessary parts of the application (Input for Web adapter, request and response for the use case class and communication with the database). Sequence diagram of the entire flow is visualised in Figure 25. Note that in the code, the name of the use case is *GetCommits*.

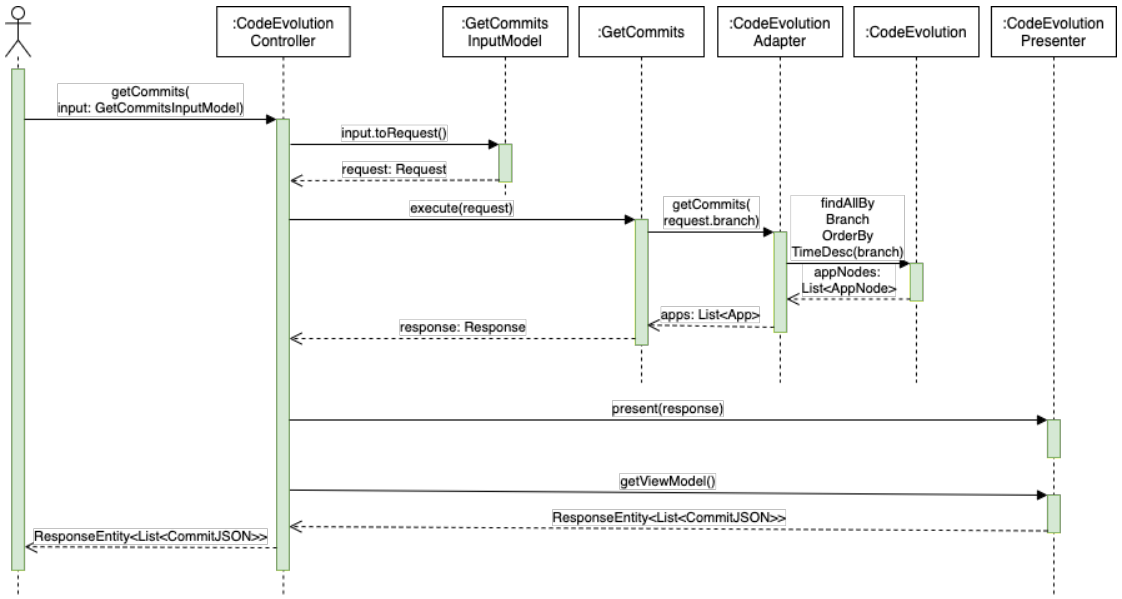

Figure 25. Sequence diagram of use case UC#7.

Web adapter. As mentioned in section 4.6.3 when the application receives the signal through REST API, the Web adapter is the module that handles the received signal and transforms it into the data for the use case. All available REST API URL-s are displayed in figure 54 in the Materials part of the thesis. Web adapter contains four main parts: controller, input model, presenter, and output model. When a request from another application is received, the controller is invoked. If the request has any data (request body), the controller uses an input model to convert that data into the request for the use case. After the data is transformed into the request, the use case is called. As it can be seen in Figure 26 (line 10), *CodeEvolutionController* converts input from the Request body into the request for *GetCommits* use case, and then use cases execute method is

called. The use case can be called from the controller since it is described as a spring component and autowired into the controller.

```
1  @RestController
2  @RequestMapping("api/code-evolution")
3  @RequiredArgsConstructor
4  public class CodeEvolutionController {
5
6      private final GetCommits getCommits;
7
8      @PostMapping("/commits")
9      public ResponseEntity<List<CommitJSON>> getCommits(@RequestBody
           GetCommitsInputModel input) {
10         var response = getCommits.execute(input.toRequest());
11         var presenter = new CodeEvolutionPresenter();
12
13         presenter.present(response);
14         return presenter.getViewModel();
15     }
```

Figure 26. Example of Web adapter controller

Application. All the use cases are written as separate components and marked with *@Usecase* annotation. All use cases follow the strict convention: they must have an execute method that takes Request as input (if any input is needed) and returns Response (if any response is returned). Both Request and Response are static classes defined inside the use case (see Figure 27 for code example). Input validation is performed inside the Request class with annotations. For example, figure 27 has two *@NotNull* annotations (line 15). Execute method usually calls some method of the gateway (Either Persistence gateway or Graphify gateway), which is autowired by the use case and implemented by the adapter. In Figure 27 on line 6, it can be seen that *CodeEvolutionGateway* (One of the Persistence gateways) is autowired, and its method *getCommits* is called (line 9). The example code of gateway can be seen in Figure 28.

Persistence adapter. As mentioned in the previous paragraph, the Persistence adapter implements the gateway. Its main goal is to communicate with the database. For that, a repository interface that extends *Neo4jReposetory* is called. Figure 29 shows how *CodeEvolutionAdapter* implements *CodeEvolutionGateway* (lines 3 and 11) and accesses Neo4j repository called *CodeEvolution* (lines 5 and 13). As mentioned in chapter 4.6.2, *Neo4jRepository* provides an opportunity to access data in the database without writing an SQL query. Figure 30 illustrates source code of *CodeEvolution* repository. After the response from the database is received, it is mapped into Domain Entity and sent back to

```
1  @UseCase
2  @RequiredArgsConstructor
3  @Transactional
4  public class GetCommits {
5
6      private final CodeEvolutionGateway codeEvolutionGateway;
7
8      public Response execute(Request request) {
9          return Response.of(codeEvolutionGateway.getCommits(request.
               branch));
10     }
11
12     @Value(staticConstructor = "of")
13     @EqualsAndHashCode(callSuper = false)
14     public static class Request extends SelfValidating<Request> {
15         @NotNull
16         public String branch;
17     }
18
19     @Value(staticConstructor = "of")
20     public static class Response {
21         public List<App> commits;
22     }
23 }
```

Figure 27. Example code of use case

```
1  public interface CodeEvolutionGateway {
2      List<App> getCommits(String branch);
3  }
```

Figure 28. Example code gateway

the use case. In figure 28 it can be seen, that repository returns *List<AppNode>* entity, but Figure 29 shows, that return type of *getCommits* method is *List<App>*. Mapping from entity to Domain value can also be seen in Figure 29 (line 13). In order to keep example code as short as possible, the implementation of *toAppList* method was excluded from the code snippet.

Back to Web adapter. Once the response from the Persistence adapter is received and additional business logic is done, the response is wrapped into the *Response* class and passed back to the Web adapter controller. Web adapter then creates a new *Presenter* entity and presents *viewModel* as *ResponseEntity*. Note that all presenters have a method called *present*, which wraps the use case *Response* class into the output model. This

```
@PersistenceAdapter
@Transactional
public class CodeEvolutionAdapter implements CodeEvolutionGateway {

    private final CodeEvolution codeEvolution;

    public CodeEvolutionAdapter(CodeEvolution codeEvolution) {
        this.codeEvolution = codeEvolution;
    }

    @Override
    public List<App> getCommits(String branch) {
        return toAppList(codeEvolution.findAllByBranchOrderByTimeDesc
            (branch));
    }

    ...
```

Figure 29. Example code of adapter.

```
public interface CodeEvolution extends Neo4jRepository<AppNode, Long>
    {

    List<AppNode> findAllByBranchOrderByTimeDesc(@Param("branch")
        String branch);

}
```

Figure 30. Example code of repository.

additional mapping from domain model entity to output model is crucial for not exposing core business logic to other applications. The process of presenting response from the use case is illustrated in Figure 26 on lines 11-14.

4.6.5 Integration with GraphifyEvolution

As it was mentioned in section 2.3, GraphifyEvolution is a command-line tool, so integration with the tool cannot be done using REST API. For that reason, GraphifyEvolution is set as a sub-module of the application, and the communication with the tool is done by using command-line tools provided by java. Currently, GraphifyEvolution is used only for uploading the application. GraphifyEvolution is called from Graphify adapter with source code path as a parameter. GraphifyEvolution analyses the code and populates the Neo4j database that is already running in the Docker container. After the application under analysis is analyzed, code smells can be queried. Querying of the code smells,

and other data about the application under analysis is done by Persistence adapter. Code smells are queried using Cypher queries provided by the author of GraphifyEvolution. All queries are slightly modified to access code smells for each class separately.

4.7 code-smell-visualizer-web

As mentioned previously, code-smell-visualizer-web is a front-end application for code-smell-visualizer. Its main functionality is to retrieve data from code-smell-visualizer-api and to convert the data into a graphical interface that can be presented to the user.

4.7.1 Technological choices

Main technologies used in code-smell-visualizer-web are TypeScript (version 4.0.3) with Angular (version 11.0.0). For state management NgRx (version 10.1.2) was chosen. Visualisation of application components was done with Bootstrap (version 11.2.7) and material-dashboard (version 2.1.0). For workspace management Nx was selected.

TypeScript and Angular. The main reason for selecting TypeScript and Angular as the main technologies for the code-smell-visualizer-web application is the author's previous experience with the technologies. React and Vue.js where considered as an alternative TypeScript libraries but angular was selected based on the author's personal preference. No additional comparison was made between libraries.

d3 library. For visualization of the application, it was decided to use the d3 library. Selection of the visualization library is made based on ease of use, compatibility with TypeScript and angular, the modernity of the library, availability of documentation, etc. Libraries like Raphaël and Paper.js were also taken into consideration. Table 8 shows the comparison of the previously mentioned libraries. From the table, it can be seen that the libraries are quite similar in terms of angular support, user interactions, animations, and modernity. The biggest difference and the deciding factor was the ease of use and the documentation. Documentations of Raphaël and Paper.js are much less informative than the documentation of the d3 library. In addition, community comparison[31] shows that the community size of the d3 library is much bigger than communities of Raphaël and Paper.js, meaning there are much fewer examples of Raphaël and Paper.js code snippets.

Nx. For better component and library separation in code-smell-visualizer-web Nx was used. Nx is an open-source toolkit designed for Angular applications to help developers write more consistent and robust code [28]. As the same article mentions, Nx provides encapsulation of the application by allowing developers to separate components into

[31] https://www.npmtrends.com/d3-vs-paper-vs-raphael-vs-nvd3

Table 8. Comparison of d3, Raphaël and Paper.js libraries

	d3	**Raphaël**	**Paper.js**
Documentation quality	Good documentation for javascript, some available tutorials for TypeScript and angular	Confusing documentation, almost no available tutorials for angular and TypeScript	Decent documentation for javascript, very little documentation on angular and TypeScript
Ease of use	Basic things are easy to learn, complex things require much more reading and research	Simple things are quite complicated to do. For example drawing moving circles is already challenging	Basic things are quite easy, but complex animation can be tricky
Angular and TypeScript support	Does support TypeScript. Poor documentation on angular part, but overall works	Does support TypeScript and angular. Easy setup for angular, but very hard to use.	Does support TypeScript and angular. Easy setup for angular, but hard to use
User interaction	Has a lot of user interaction abilities: zooming, moving around, hovering, clicking. Implementation is very easy	Many user interaction abilities: zooming, hovering, clicking. Implementation is quite complicated because of lacking documentation	Many user interaction abilities: zooming, moving, hovering, clicking. Implementation is easier than for Raphaël, but more complicated than in d3.
Animations	Many animations that are usually used in charts. Implementation is easy	Many animations. Implementation is quite complicated	Many animations. Implementation is easier than in Raphaël but more complicated than in d3
Community	Quite large community	Small community	Small community
Browser support	Works with Chrome, does not work with older versions of IE	Works with almost all browsers	Works with Chrome, does not work with older versions of IE
Predefined components	A lot of predefined components	No predefined components, everything needs to be built from scratch	Very few predefined components
Modernity	Created in 2010, last updated April 2021	Created in 2008, last updated March 2021	Created 2011, last updated March 2021

libraries. Libraries are independent of each other and can be used separately. The architecture provided by Nx suits with code-smell-visualizer-web applications as it

fulfills the need of separating d3 library from the rest of the application. Alternatives like Lerna and Bit.dev were considered, but the decision was made based on Nx angular support and the author's previous experience with the toolkit. No further analysis of the toolkits was made.

NgRx. In order to communicate between Nx libraries, NgRx was used. NgRx is a reactive data and state management tool for Angular [29]. It provides a solution for shared *Single source of truth*[32] store that can be accessed by any component or service of the application [30]. NgRx was selected because it was specifically designed for angular, and it has decent Nx support. Alternatives like NgXs and Akita were also considered. Visualization of empty application state can be seen in figure 31.

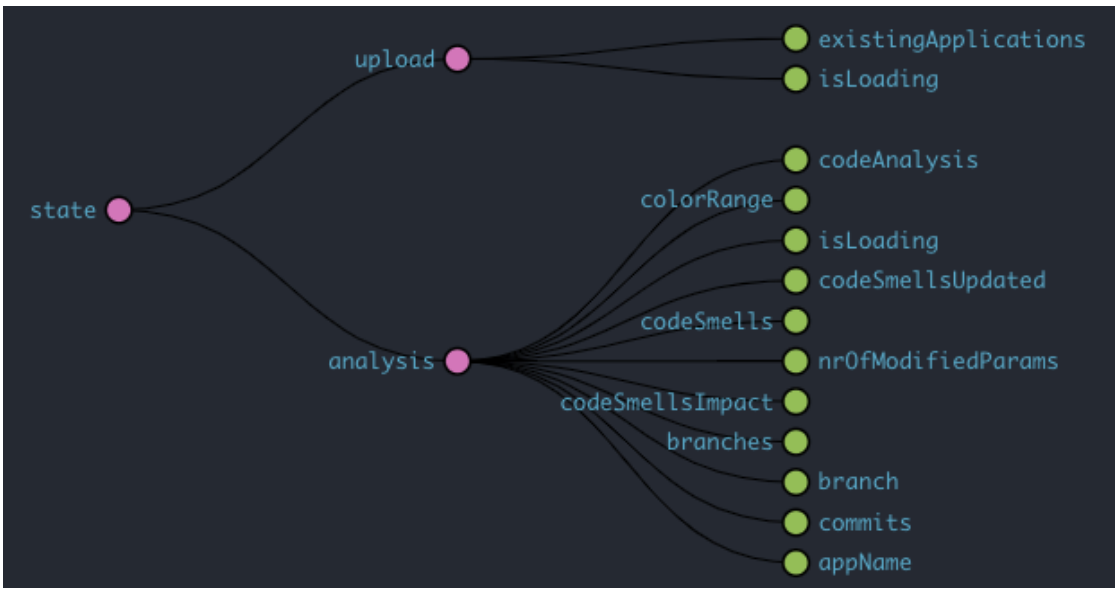

Figure 31. Screenshot from NgRx Store Devtools displaying empty state of the application.

Bootstrap and Material Dashboard. Material Dashboard[33] was used on top of Bootstrap[34] in order CSS of the components. Bootstrap was selected based on the author's personal preference, so no additional frameworks were considered. Additional components from Material Dashboard are used for simplicity and a better user interface. The selection of additional components is based on the author's personal preference,

[32]https://www.mulesoft.com/resources/esb/what-is-single-source-of-truth-ssot
[33]https://www.creative-tim.com/product/material-dashboard
[34]https://getbootstrap.com/

quality of documentation, and availability of required components. CoreUI was viewed as an alternative for UI components. Table 9 concludes the comparison between two libraries. As it can be seen, both libraries had good UI in the author's opinion and decent documentation. Material Dashboard was preferred over CoreUI because it has more available components.

Table 9. Comparison of Material Dashboard and CoreUI

	Material Dashboard	**CoreUI**
Documentation and examples	Decent documentation with few examples	Good documentation with many examples
Multi-select component	yes	only in paid version
Input field component	yes	yes
Dropdown select component with autofill	yes	yes
Dark mode	yes	only in paid version
Authors opinion about UI	Clean and beautiful components, great color selection	Clean components, nice colors

4.7.2 Architecture of code-smell-visualizer-web

code-smell-visualizer-web is separated into 6 main libraries: api, app-load, code-analysis, d3, http-common and code-smell-visualizer. Figure 32 shows source tree of the application. In addition figure 33 shows an HTML tree of the application.

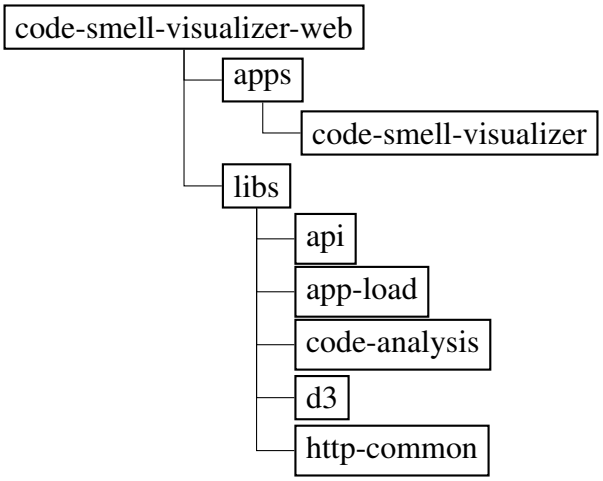

Figure 32. Architecture of code-smell-visualizer-web done with Nx

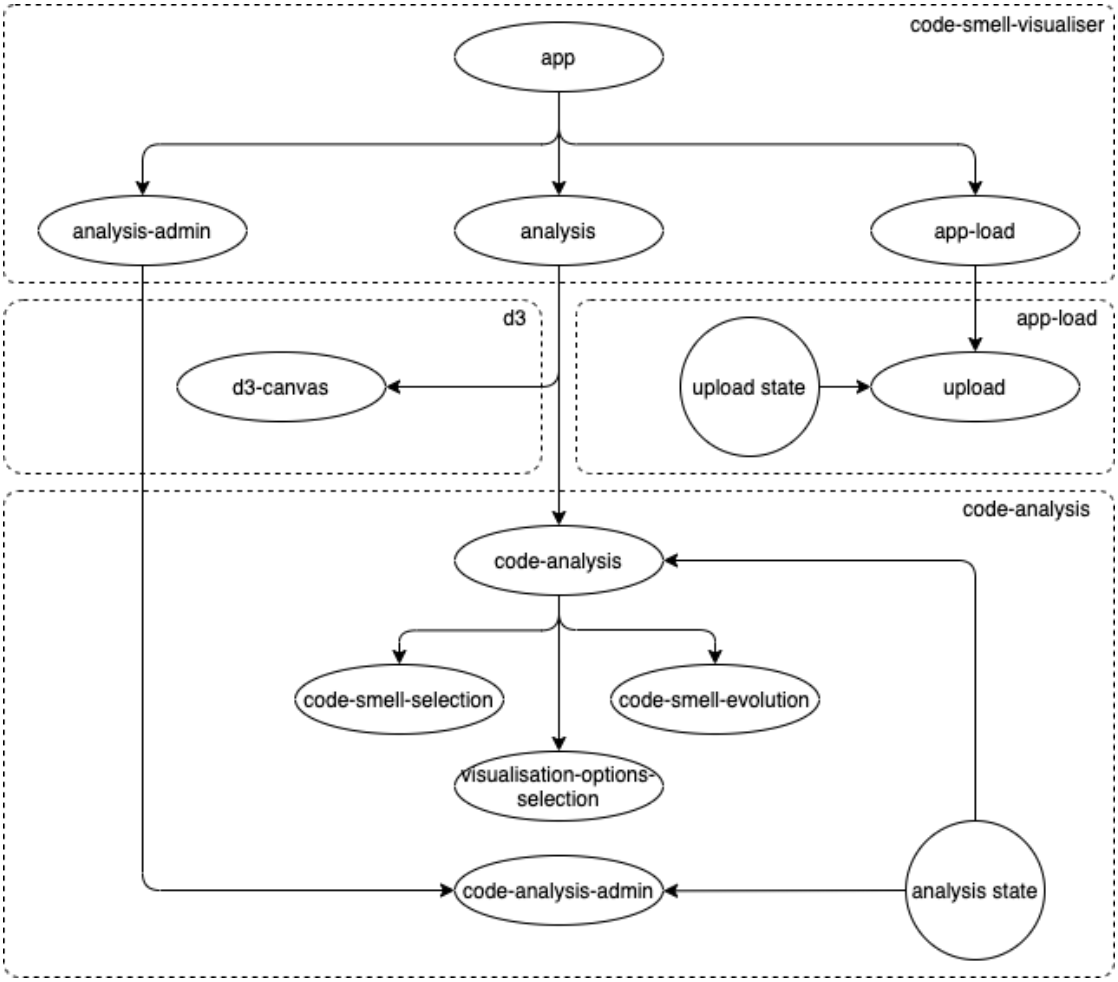

Figure 33. HTML tree of code-smell-visualizer-web.

api. Api library contains models and enumerations that are used for code-smell-visualizer-api API calls. All models and enumerations of the library are exported to other libraries. Api library does not have any components, services, or access to the global state. As it can be seen from figure 34 ,api library does not use any other libraries of the application as dependencies.

http-common. Http-common is another library that does not have any dependencies (figure 34). Its main purpose is to wrap incoming and outgoing requests from TypeScript models to JSON objects and vice versa.

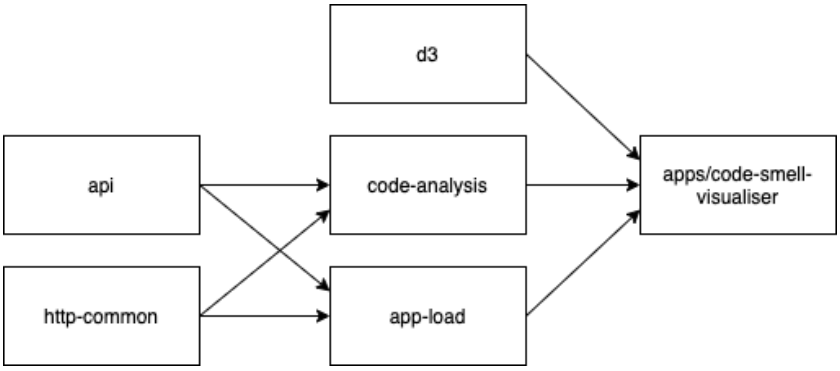

Figure 34. Dependencies between code-smell-visualizer-web libraries.

app-load. App-load is a library responsible for uploading the source code of the application under analysis into the code-smell-visualizer. The uploading of the source code is described in section 4.3.1. The library depends on *api* and *http-common* libraries (figure 34). As it can be seen from the HTML tree in figure 33, the app-load library has only one component which interacts with its own state called upload state. Based on the HTML tree it can be seen that the upload component of the app-load library is exported to other libraries and used by code-smell-visualizer library.

code-analysis. Code-analysis is one of the largest libraries in the application based on number of components. The library handles selection of code smells (chapter 4.3.2), modification of code smell parameters (chapter 4.3.3), branch and commit selection (chapter 4.3.6) and color slider modifications (4.3.5). Figure 33 shows that library has its own state called analysis state, which is accessed by code-analysis and code-analysis-admin components. From the HTML tree it can be seen that these components are exported to other libraries and used by code-smell-visualizer library. In addition to multiple external libraries, code-analysis library depends on *api* and *http-common* libraries (figure 34).

d3. D3 library implements the visualization logic of the application (chapter 4.3.4). From the HTML tree in figure 33 it can be seen that the d3 library has only one component, which is exported to other libraries and is used by code-smell-visualizer. D3 does not use any other libraries of the application as dependencies.

4.7.3 Example flow

The following example flow demonstrates fetching code smells from the backend application and forwarding it to the component. The purpose of the example is to give a small

overview of the architectural choices in code-smell-visualizer-web and to showcase the usage of NgRx. This example flow is selected as it is one of the simpler use cases in the application, and it covers all important aspects of the state management in the application. Figure 35 showcases the sequence diagram of the flow.

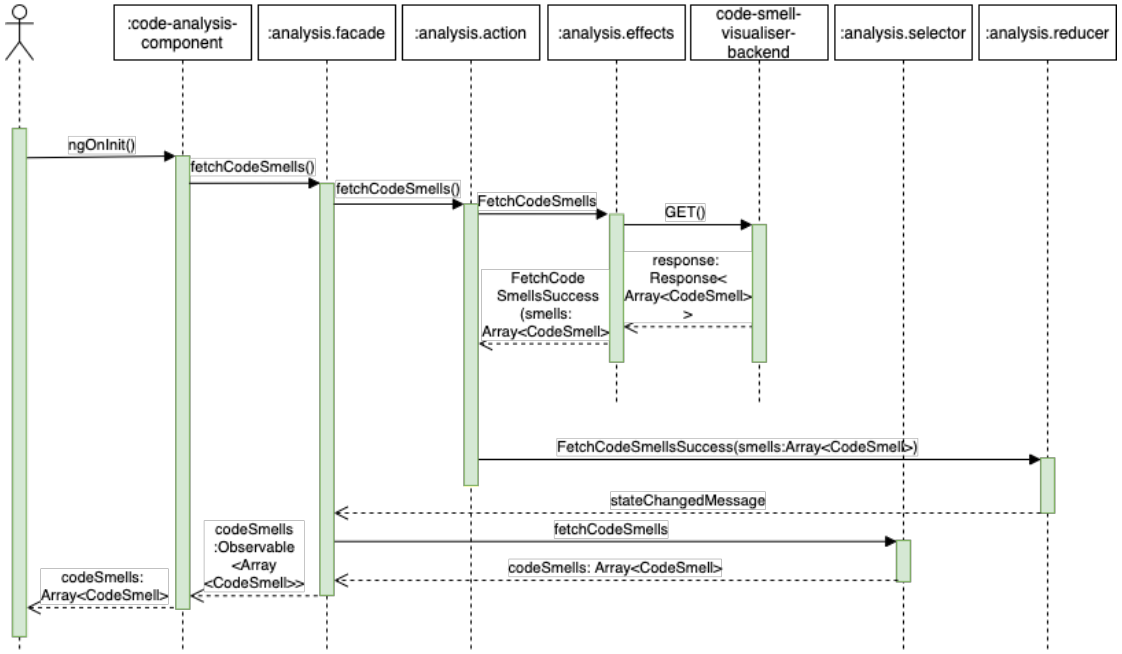

Figure 35. Sequence diagram of fetching code smells.

analysis.facade. Once the component is initialized from *ngOnInit* method, the component fetches the code smells from the facade. As it can be seen from the abstract representation of state management in figure 36, the facade serves as an intermediate layer between component and the state management. Based on the component call, the facade either retrieves data that is sent by the selector or calls an action. In the current case analysis.facade calls an analysis.action to fetch the code smells since there are no code smells in the store at the moment.

analysis.action. Each action in the application represents a unique event. Action serves as a center piece of the NgRx state management[35]. In figure 36 it can be seen that action interacts with effect and the reducer. In current case action simply calls the effect to gather the information. Code snippet from figure 37 shows an example of two

[35]https://ngrx.io/guide/store/actions

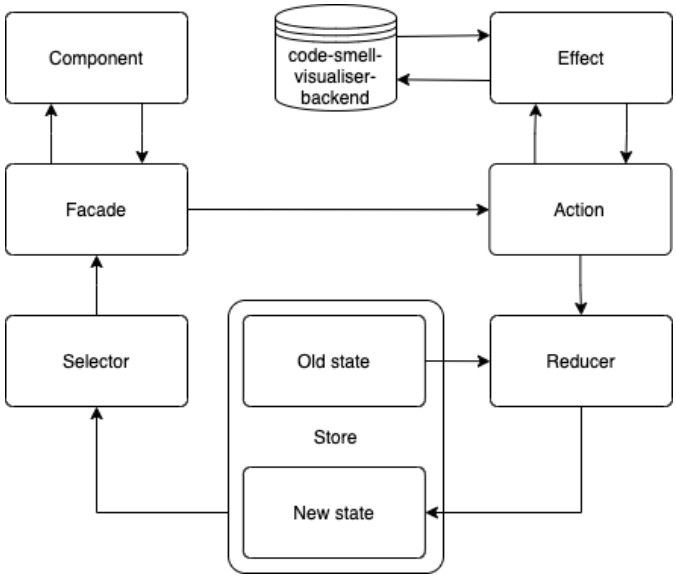

Figure 36. Abstract visualisation of state management in code-smell-visualizer-web.

actions: *FetchCodeSmells* and *FetchCodeSmellsSuccess*. Both actions can be also seen in sequence diagram (figure 35). Action *FetchCodeSmellsSuccess* will be described in next paragraphs.

```
export enum CodeAnalysisActionTypes {
  FetchCodeSmells = '[CodeSmells] Fetch Code Smells',
  FetchCodeSmellsSuccess = '[CodeSmells] Fetch Code Smells Success',
}

export class FetchCodeSmells implements Action {
  readonly type = CodeAnalysisActionTypes.FetchCodeSmells;
}

export class FetchCodeSmellsSuccess implements Action {
  readonly type = CodeAnalysisActionTypes.FetchCodeSmellsSuccess;

  constructor(
    public queryResult: Array<CodeSmell>,
  ) {
  }
}
```

Figure 37. Code snippet from analysis.action

analysis.effect. As it can be seen from figure 36, the effect interacts with the backend of the application. In the current case analysis.effect makes an API GET request to the code-smell-visualizer-api and receives the response with corresponding data. Figure 38 shows an example of GET request made to the code-smell-visualizer-api. Note that on line 2 of the figure, the type of the calling action can be seen, and on line 9, it can be seen that *FetchCodeSmellsSuccess* action is triggered with the response that came from the code-smell-visualizer-api. The response is passed as the constructor parameter to the action (line 13-16 in figure 37). In the current case, *FetchCodeSmellsSuccess* action calls analysis.reducer to modify the state with the results.

```
1  @Effect() fetchCodeSmells$ = this.dataPersistence.fetch(
2    CodeAnalysisActionTypes.FetchCodeSmells,
3    {
4      run: () => {
5        return this.httpClient
6          .get(this.baseUrl + '/code-smell')
7          .pipe(
8            map(resp => {
9              return new FetchCodeSmellsSuccess(
10               resp as Array<CodeSmell>,
11             );
12           })
13         );
14     }
15   }
16 )
```

Figure 38. Code snippet from analysis.effect

analysis.reducer. Reducer is the only function in state management that can modify the store of the application. As shown in figure 36, it takes an old state of the application, modifies it, and sets it as a new state of the store. Figure 40 presents an example of the reducer function for *FetchCodeSmells* and *FetchCodeSmellsSuccess* actions. Note that on line 15, the *codeSmells* variable in the store is modified to the data received from the action. Figure 39 shows the state after the analysis.reducer changed code smells.

analysis.selector. Once the state of the store is updated, the analysis.facade is notified. After that, the analysis.selector is called to retrieve the required data from the store. As it can be seen from the figure 36, the selector accesses the data from the store and sends it to the facade. In the current case analysis.selector retrieves code smells from the store wrapped into the observable and passes it to the analysis.facade, which later passes it

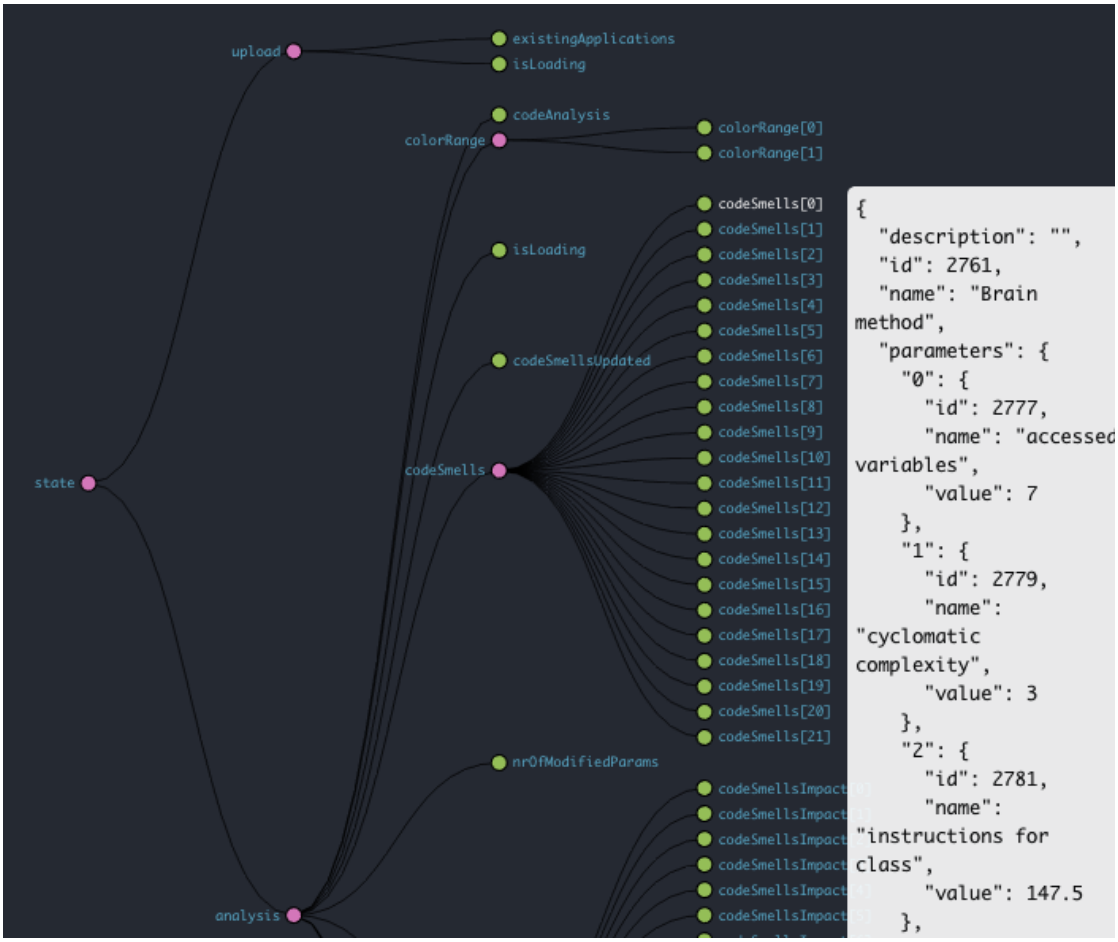

Figure 39. Screenshot from NgRx Store Devtools displaying state of application.

to the component. Figure 41 gives an example of the *fetchCodeSmells* selector, which selects code smells from the current state of the store.

```
export function analysisReducer(
  state: CodeAnalysisState = initialState,
  action: ClassesAction
) {
  switch (action.type) {
    case CodeAnalysisActionTypes.FetchCodeSmells:
      return {
        ...state,
        isLoading: true
      };
    case CodeAnalysisActionTypes.FetchCodeSmellsSuccess:
      return {
        ...state,
        isLoading: true,
        codeSmells: action.queryResult
      };
    default:
      return state;
  }
}
```

Figure 40. Code snippet from analysis.reducer

```
export const fetchCodeSmells = createSelector(
  getCodeAnalysisState,
  (state: CodeAnalysisState) => state.codeSmells
)
```

Figure 41. Code snippet from analysis.selector

4.8 Evaluation

Evaluation of the application is done with tests and interviews. The purpose of the evaluation is to verify that all necessary requirements are implemented and to collect feedback about the usefulness of the developed tool.

4.8.1 Tests

The application is tested using unit, integration, and system tests. Unit tests are used to test domain logic and adapters separately from each other to verify that these modules work on their own. Integration tests verify that all flows in the application work properly and that all components of the application (domain logic and adapters) work together. System tests check that developed use cases satisfy functional and non-functional requirements described in section 4.1. Testing results are based on the test run as of 30th of April 2021 and may differ from the test results generated after that.

Unit testing. There is a total of 29 unit tests that cover 97 classes of the application. Since the code-smell-visualizer-web does not have any domain logic of the application, unit tests are only covering the code-smell-visualizer-api application. Mockito [36], JUnit [37] and assertJ [38] libraries were used as main external testing frameworks. All of the libraries were selected based on the author's personal preference, and no other libraries were considered.

All the unit tests follow a Given-When-Then notation [39], however, it is important to note that the tests are not written using Behaviour Driven Development[40] structure, nor any BDD syntax supporting tools (e.g., Gherkin or Cucumber [41]) are used. Given-When-Then notation is used only in the naming of the tests. The pattern is used to improve tests readability. Figure 42 gives an example of the web adapter unit test. The name of the test defines that when code smells are requested, then the response is correctly mapped. The test verifies only the correct behavior of the web adapter since the behavior of the domain logic is mocked (line 3 on figure 42). This example is selected as it has all the important aspects of the unit test. Additional results about the execution of unit tests can be seen in figure 55 in the Materials. There it can be seen that all unit tests are passing.

Integration testing. There is a total of 11 integration tests that cover nine use case flows. Similar to unit tests, integration tests do not cover code-smell-visualizer-web part

[36] https://site.mockito.org/
[37] https://junit.org/junit5/
[38] https://assertj.github.io/doc/
[39] https://martinfowler.com/bliki/GivenWhenThen.html
[40] https://cucumber.io/docs/bdd/
[41] https://cucumber.io/docs/gherkin/

```
1     @Test
2     void WHEN_code_smells_requested_THEN_data_mapped_correctly() {
3         when(usecase.execute()).thenReturn(TestData.mockResponse());
4
5         var response = controller.getCodeSmells();
6
7         verify.getCodeSmellsResponseMappedCorrectly(response);
8     }
```

Figure 42. Code snippet web adapter of the code-smell-visualizer-api

of the application. However, unlike unit tests, integration tests are verifying that the application communicates with the database. The main technologies used in integration tests are Testcontainers [42] and mockMvc [43]. Testcontainers is a Java library that simplifies integration tests by running an application in temporary containers that contain all necessary dependencies, and an empty database [31]. MockMvc simplifies Spring MVC (Model-view-controller[44]) testing by encapsulating all spring beans and making them available for tests [32]. Both tools were chosen based on the author's personal preference, so no other frameworks were considered. Figure 43 shows an example integration test. An example is selected because it is relatively short and covers all the important points of the test. Similar to unit tests, integration tests follow the Given-When-Then naming convention. Additional results about execution of integration tests can be seen in figure 56 in the Materials. There it can be seen that all integration tests passed.

Test coverage. In order to check whether the non-functional requirement CSV-NFR2 is satisfied, test coverage of code-smell-visualizer-api unit and integration tests was measured. Figure 44 shows the results of test coverage measurements. It can be seen that in total 92% line coverage was achieved for the application. The image also shows test coverage results for all modules of the code-smell-visualizer-api. It can be seen that application and persistence modules have the lowest line and method coverage. The reason is that the application module contains all the domain objects, and the persistence module contains all of the database entity objects. Most of these objects are annotated with Lombok[45] $@getter$, $@setter$, $@builder$, and $@data$ annotations, meaning all fields of the classes have getters and setters that are not used in the application, and their testing is irrelevant. However, not covering these methods with tests reduces the statement coverage.

[42] https://www.testcontainers.org/
[43] https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/test/web/servlet/MockMvc.html
[44] https://www.codecademy.com/articles/mvc
[45] https://projectlombok.org/

```java
@Test
@Transactional
public void
    GIVEN_app_is_created_WHEN_code_smells_impact_requested
_with_one_code_smell_THEH_max_impact_returned() throws Exception
    {
     CodeSmellImpactInputModel input = TestData.
         createCodeSmellImpactInputModelWithOneCodeSmell();

     mvc.perform(post("/api/code-smell-impact")
             .content(objectMapper.writeValueAsString(input))
             .contentType(MediaType.APPLICATION_JSON))
             .andExpect(status().isOk())
             .andExpect(jsonPath("$", hasSize(1)))
             .andExpect(jsonPath("$[0].name", is("Long method")))
             .andExpect(jsonPath("$[0].impact", is(100.0)));
}
```

Figure 43. Code snippet of integration test from the code-smell-visualizer-api

100% classes, 92% lines covered in package 'ee.ut.codevisualiser'

Element	Class, %	Method, %	Line, %
application	100% (42/42)	81% (117/144)	93% (380/407)
common	100% (1/1)	100% (1/1)	100% (4/4)
graphify	100% (1/1)	100% (1/1)	100% (2/2)
persistance	100% (23/23)	79% (136/171)	90% (506/562)
web	100% (30/30)	92% (116/125)	96% (244/253)

Figure 44. Unit and integration tests coverage in code-smell-visualizer-api

System testing. System tests are done manually by the author of the thesis, and they cover all parts of the application. The test verifies that all requirements of the system are satisfied. Each functional requirement is tested based on its Primary scenario. Non-functional requirements are tested based on the Success Criteria of the requirement. Requirement CSV-NFR1 is tested together with functional requirements. The requirement is marked as satisfied if all functional requirements are met and all functionalities can be used in the Google Chrome web browser. Requirement CSV-NFR2 is marked satisfied based on test coverage results described in the previous chapter. Requirement CSV-NFR3 is evaluated by analyzing code-smell-visualizer-api application using code-smell-visualizer. The analysis results show that out of 97 classes, 4 have intensive coupling code smell, meaning that 95.8% of the classes don't have the code smell. Figure 57 in the Methods chapter shows the results of code-smell-visualizer-api module visualization. Note that only Feature envy code smell is displayed in the figure,

meaning that other code smells are turned off. In comparison, figure 58 displays the analysis of the code-smell-visualizer-api with all available code smells turned on.

Since the tests need to be as close to the live environment as possible, a real project is analyzed for the tests. The project selected for the analysis is project A described in section 2.1. Project A is selected over project B based on the author's personal preference. Table 10 captures results of system testing. There it can be seen that two functional requirements are not satisfied. The reason these requirements are not satisfied is that they have been prioritized as 'Will not have' requirements and are not implemented.

Table 10. System tests results

Requirement ID	**Satisfied**	**Explanation**
CSV-FR1	YES	
CSV-FR2	YES	
CSV-FR3	YES	
CSV-FR4	YES	
CSV-FR5	YES	
CSV-FR6	YES	
CSV-FR7	YES	
CSV-FR8	YES	
CSV-FR9	YES	
CSV-FR10	YES	
CSV-FR11	YES	
CSV-FR12	YES	
CSV-FR13	YES	
CSV-FR14	YES	
CSV-FR15	YES	
CSV-FR16	NO	Requirement priority marked as Will not have
CSV-FR17	NO	Requirement priority marked as Will not have
CSV-NFR1	YES	
CSV-NFR2	YES	
CSV-NFR3	YES	

4.8.2 Interviews

Evaluation interviews are conducted with developers and scrum masters from the two projects described in chapter 2.1. Product owners and Quality assurance are not interviewed as it was found in requirements gathering interviews (section 4.1.1) that these stakeholders will not benefit from the code-smell-visualizer application. The interviews are conducted with seven stakeholders in total: five developers and two scrum masters. It is important to note that all available developers and scrum masters are interviewed from both projects meaning that no additional selection is made within the group of available stakeholders.

For all interviews source code of projects A and B are analyzed. The analysis of the source codes is done to show interviewees that the tool works and to center the interview around the project that the interviewee works on. This way, interviewees are presented with the domain they already know, and thus interviewees will have a better understanding of the visualization. All analysis was prepared before the interview to save time.

Interviews consist of 6 parts: introduction, presentation of existing work, discussion about existing work, presentation of application developed, questions about technology acceptance, and discussion about the software. Figure 59 presents an evaluation interview plan with time estimates of each part. In the introduction part, the topic of the thesis is described, stakeholders are informed why they are selected for the interview, and the flow of the interview is described. At the beginning of the interview, interviewees are informed that discussions are not recorded, and that author of the thesis will take some notes. In the presentation of existing work, interviewees are introduced to work already done in the current field. Works that are presented are described in section 2.2. Interviewees are presented with existing work in order to better explain what code and code smell visualization is. After the existing work is presented, interviewees are asked about their opinion on the implementations and the concept of code smell visualization in general. The opinion is asked to get the interviewee more involved in the interview and to understand whether the stakeholder might benefit from the existing tools. In the presentation of the developed application, the code-smell-visualizer application is shown. The presentation includes a demonstration of existing functionalities and a description of what is presented by the visualizer. Explanation of existing functionalities is done because not all stakeholders participated in requirements gathering interviews and might not understand what is visualized. In addition, currently, the application has no tool-tips that help users to easier understand concepts done in the application. In the Questions about technology acceptance part, interviewees are asked various questions about perceived usefulness and perceived ease of use. Evaluation interview questions are shown in figure 60. Finally, in the discussion about the software part, interviewees are asked about what changes and features they would like to see in the application. The goal of this part is to gather requirements for the future development of the application.

4.8.3 Quantitative interview analysis

In this part of the interview analysis, the responses to questions 1 to 14 mentioned in figure 60 are analyzed. The questions are conducted based on the technology acceptance model described in section 3.4.

Overall ease of use. Interviewees were asked whether they understand what the purpose of the application is. All developers and scrum masters answered that they understood what the purpose of the application is. When asked about the ease of use of the application, the answers were more varied. A chart in figure 45 shows that two interviewees (one scrum master and one developer) said that the usage of the application is very easy, four interviewees (one scrum master and three developers) said that the usage of application is easy, and one developer said that the application is not so easy to use. The reasoning for these answers can be found in the responses to the following questions. A lot of change proposals and requests for additional features were made that would simplify the overall usage of the application in general. All the proposals are discussed in the qualitative analysis section (chapter 4.8.4).

Overall usefulness. When it comes to the usefulness of the application, five interviewees answered that the application is useful, and two interviewees answered that the application is very useful. Results can be seen in the chart displayed in figure 45.

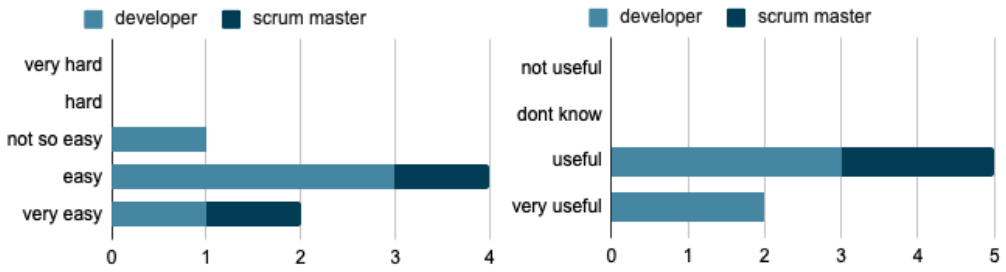

Figure 45. Persisted ease of use (left) and persisted usefulness of application in general

In addition, stakeholders were asked whether the application will improve the code quality of the project, whether their everyday job would benefit from the tool, and whether the tool has everything necessary for code smells recognition (questions 13-15 in the evaluation interview described in figure 60). Interviewees said unanimously that the usage of the application would improve the code quality of the software. It was mentioned that the application could be used as an additional metric for evaluating the code quality of the software (other metrics are, for example, passing of all tests and test coverage) and that developers would look at the image from time to time before the releases. When asked whether the everyday job of an interviewee would benefit from

the tool, most of the answers were no. The most common explanation for the response is that the interviewee would not use the tool on a daily basis but rather once a week or even less often. When asked whether the application provides everything necessary for recognition of code smells, only one interviewee answered that the tool has everything needed. Other interviewees mentioned that the tool should have some additional features that would improve the usage of the tool. Additional features are described in section 4.8.4.

Navigation between branches and commits. Interviewees were asked whether branches and commits selection is easy to use and whether this feature is necessary. The first question targets persisted ease of use and the second question targets persisted usefulness of the application. The results of the responses can be seen in figure 46. All of the developers and scrum masters answered that navigation between branches and commits is very easy. Out of seven stakeholders, three said that this feature is useful, and four said that the feature is very useful.

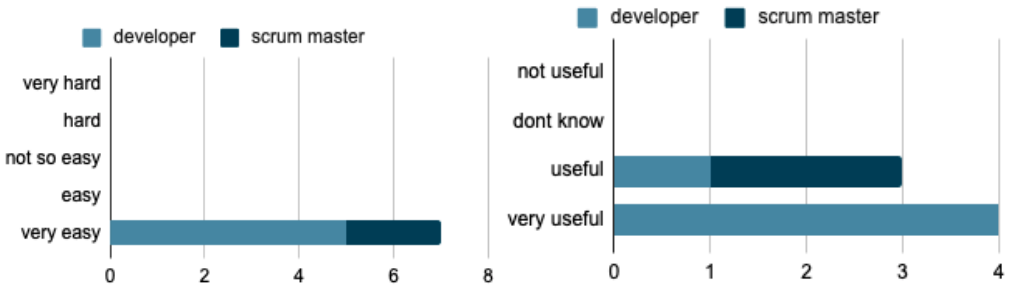

Figure 46. Persisted ease of use (left) and persisted usefulness of selection branches and commits

Selection of code smells. Here once again, interviewees were questioned about persisted usefulness and persisted ease of use of the feature. Based on the analysis of the responses visualized in figure 47 it can be seen that the feature is relatively easy to use, but the feature does not seem useful. Three interviewees answered that feature is very easy to use, three responded that it is easy to use, and one answered that it is not so easy to use. The interviewee who answered that feature is not so easy to use explained that code smells have no additional description of what they actually do and how they affect the code. The proposal to improve this feature was to add tooltips that explain the definition of code smells. When it comes to persisted usefulness of the feature, many interviewees pointed out that the feature is not useful. Two developers said that the feature is not useful, two developers and one scrum master said that they are not sure whether they would use this feature. The main reason why many developers said that

the feature is not useful is because the definition of some code smells is unclear. In addition, it was mentioned that this feature should not be used often as deselecting code smells may result in incorrect visualization. The solution to fix these concerns is to move this feature to a separate tab that is accessible only to the advanced user. However, one developer marked this feature as useful, and one scrum master marked it as very useful. The explanation was that this feature allows excluding false positives from the results of the visualization. If some code smell appears in a class and it is known that there is no actual code smell there or the refactoring of the code smell has already been addressed, then it is a good idea not to include this code smell in the visualization. For example, during the presentation, it was found out that many interfaces are colored red because they all have Lazy class[46] code smell. It turned out that the static analysis tool didn't recognize that the interfaces are supposed to not have any method bodies in java[47].

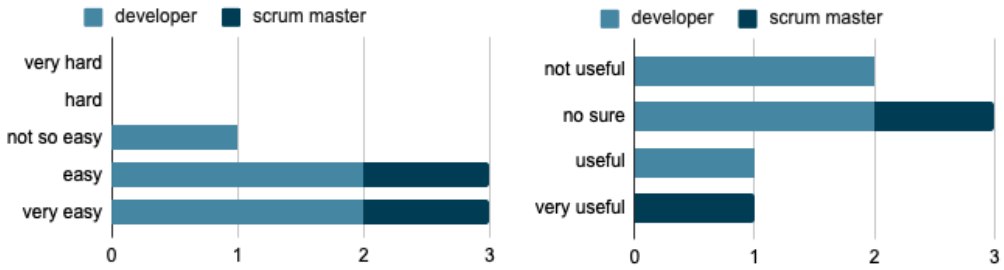

Figure 47. Persisted ease of use (left) and persisted usefulness of code smells selection

Adjustment of code smell parameters. This feature was not discussed as extensively as other features since the primary user of the feature is neither a regular developer nor a scrum master. As it can be seen from the figure 48 all interviewees agreed that adjustment of the code smell parameters is essential, and thus the feature is useful. Persisted ease of use responses, however, were not so unanimous. The main reason why some interviewees marked this feature as not so easy to use is that parameters lacked additional information about them. The users wanted to see links to web pages or tooltips explaining what the purpose of the parameter is and how it contributes to code smell.

Visualization. Visualization of the source code is evaluated based on how easy it is to find classes that need refactoring. Persisted usefulness is not measured as visualization of the source code is a core feature of the code-smell-visualizer. Based on the analysis displayed in figure 49 it can be concluded that interviewees found it easy to discover classes that need refactoring. A lot of suggestions were made for how to make this feature more usable. These features are described in section 4.8.4.

[46]https://refactoring.guru/smells/lazy-class
[47]https://docs.oracle.com/javase/tutorial/java/concepts/interface.html

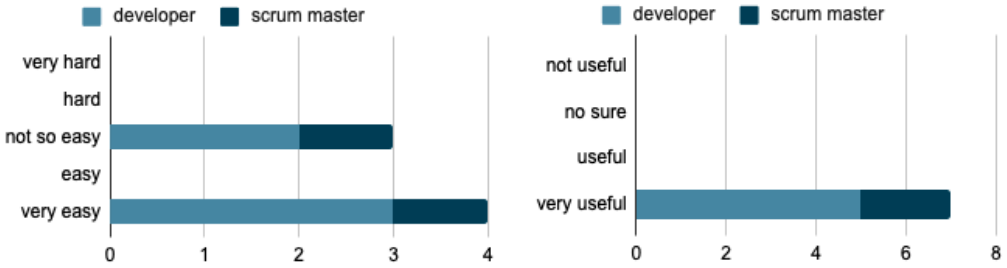

Figure 48. Persisted ease of use (left) and persisted usefulness of code smell parameter adjustment

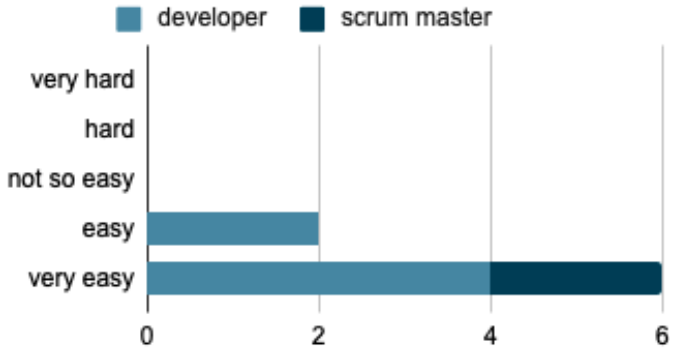

Figure 49. Persisted ease of use of the application visualization

Adjustment code smell severity level. Based on the feedback gathered, adjustment of the code smells severity is the most confusing and impractical feature in code-smell-visualizer. Based on the feedback visualized in figure 50 it can be concluded that most of the interviewees didn't understand how the slider works and what its purpose is. The complicated logic behind the slider is also the main reason why some interviewees said that overall application usage is not very easy. Some suggestions were made on how to improve this functionality and make it more understandable. The proposals are described in qualitative analysis of the feedback (section 4.8.4).

Based on the interviewees' feedback, it is concluded that the application is practical and easy to use. code-smell-visualizer is beneficial both for developers and scrum masters. The application needs improvements in the code smell severity modification slider and code smell selection. Future enhancements of the software are described in section 4.8.5.

4.8.4 Qualitative interview analysis

After the technology acceptance questions, interviewees were asked to give feedback about the application (point 6 in Evaluation interview plan described in figure 59). From

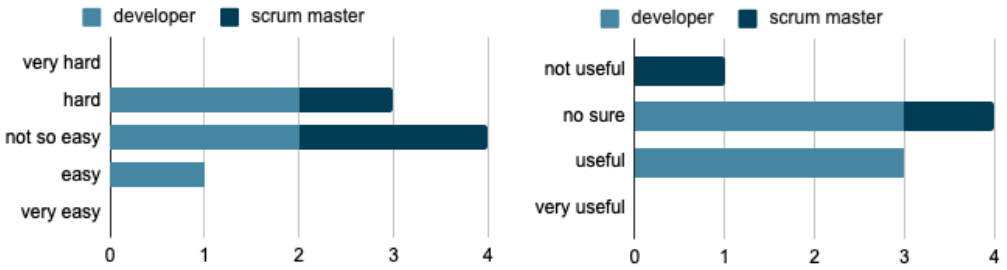

Figure 50. Persisted ease of use (left) and persisted usefulness of code smells severity modification

the responses, it can be conducted that code-smell-visualizer is a useful application and can be used in production. Some developers mentioned that the application could be used after every commit after branches are merged or for the sprint reviews. Many developers and scrum masters noted that project architects would benefit from the tool as they can quickly see the quality of the code without the need to look into static analyzers. Scrum masters mentioned that the application could even be used by users who are not very familiar with the development of the software. One of the proposed use cases was that product owners or senior developers could use the tool to add refactoring tasks to the backlog. Interviewees liked that the scope of the application is small and easy to comprehend. It was mentioned that the application has a pleasant user interface.

On the downside, many interviewees mentioned that, at first, the application seemed complicated. It was mentioned that after the introduction, the tool became much more understandable, and functionalities seemed logical. This means that additional information needs to be provided in order for users to use the tool without further explanation. It was also mentioned that the tool lacks in providing more detailed information about the code. Many developers said that in order to use the tool much more effectively, the application needs to show additional data about the classes, like which code smells affect each class. In addition, there were many minor improvements that interviewees wanted to see in the tool. All feature requests and changes suggested by the interviewees can be seen in figure 51.

1. Add more explanatory text to code smells selection
2. Add tool tips for code smells explaining what they are and how they are calculated
3. Add explanatory tool tips to code smell parameter adjustments explaining what each parameter does and how it contributes to the formula
4. In visualization window, show additional information about the class when circle is clicked. Show name of the class, methods of the class and color each method based on the code smell severity
5. In additional information window (created in previous point) show code smell score of the class
6. In additional information window add links to static code analysis that will show where exactly the code smell is
7. In branch and code smell selection change current selection to git tree structure displaying when each branches were merged together.
8. Hide code smell selection from average user.
9. Hide the slider from average user.
10. Add numbers to the slider that show code smell score and display code smell scores also in the additional information window.
11. Add explanatory text to the slider explaining what it actually does and how code smell scores are calculated
12. In the visualization highlight classes with the code smell when code smell is clicked in code smell selection component.
13. Add user management and payment system
14. Add more statistics about how code smells developed. For example, show when code smell first appeared in the class.
15. Add animations for visualizations that can be played to see what how code smells developed in time.

Figure 51. Requested features.

4.8.5 Improvements based on interview feedback

All suggestions received during the interviews were reviewed, and new requirements were formulated based on the recommendations. Many of the suggestions did not end up as requirements, but that doesn't mean that the requests are ignored. These requests will be rechecked in the next iteration. Based on the interviews, the most requested features are additional tooltips with explanations (suggestions 1, 2, 3 and 11 in figure 51) and additional information windows in the visualization (suggestions 4, 5 and 6 in figure 51). Requirements for the most requested features can be seen in table 11. Note that requirement CSV-RF8 is an existing requirement and is slightly modified by changing the main actor of the requirement to advanced user. This means that regular users will not be able to use the slider. This decision is made based on the interviewees' suggestion number 8 shown in figure 51). Furthermore requirements for additional information window already exist (Requirements CSV-FR16 and CSV-FR17), but since they are prioritized as 'will not have' requirements (table 5) they were not implemented. Based on the feedback, the requirements priority is raised to 'must have' requirements. Priorities of the new requirements for future iteration can be seen in table 11.

Table 11. Functional requirements gathered from interviewees feedback

ID	**CSV-FR8**
Name of the requirement	Adjust color coding
Description	Advanced user should be able to adjust coloring of the code analysis
Actors	Advanced user
Precondition	1. Code analysis is loaded
Primary scenario	1. Actor sets a numerical range for Green, Yellow and Red colors 2. Color of classes is adjusted based on code smell severity in the class and range that user has selected
Postcondition	Advanced user successfully adjusted color coding of the code smell severity in the class

ID	CSV-FR18
Name of the requirement	Tool tips
Description	User should be able to see additional explanation about code smells
Actors	User
Precondition	List of code smells is available
Primary snecario	1. User hovers over code smell 2. Additional information about code smell is shown in tool tip: - Definition of code smell - How does code smell impact the code - Link to read more about code smell
Postcondition	User sees additional explanation about selected code smell

ID	CSV-FR19
Name of the requirement	Numeric values in color coding adjustment
Description	User should be able to see code smell score when adjusting color coding
Actors	Lead developer
Precondition	Code analysis is loaded
Primary snecario	1. In color adjustment slider Lead developer sees the exact value code smell score on the slider
Postcondition	User sees code smell score when adjusting color coding

Table 14. New Functional Requirements prioritized based on MoSCoW model

Requirement ID	Priority
CSV-FR18	M - must have
CSV-FR8	M - must have
CSV-FR19	M - must have
CSV-FR16	M - must have
CSV-FR17	M - must have

5 Discussion

This section covers comparison with other applications, restrictions, future work of the application, and lessons learned during the thesis. Firstly chapter compares code-smell-visualizer with already existing solutions described in chapter 2.2. Then the chapter explains what were the main limitations while implementing the application and what are the future developments of the code-smell-visualizer. Finally, the author of the thesis describes the lessons that were learned during the development of the code-smell-visualizer.

5.1 Comparison with existing tools

As mentioned in chapter 2.2, there already have been similar applications developed. This chapter discusses some of the key differences between the applications.

2d vs. 3d. All the existing tools mentioned in chapter 2.2 are using 3d or even VR modeling for visualizing the application. The code-smell-visualizer uses simpler 2d visualization. In both requirements gathering and evaluation interviews, interviewees mentioned that 3d modeling is fun to look at, but the applications cannot be used in software development as it is hard to understand what is going on in the image. In VR applications, the developers would have to acquire special headsets to interact with the code. So, in conclusion, the interviewees preferred a simpler 2d visualization.

Code evolution. None of the previous tools allow users to view the history of code or visualize the code smells evolution. Since GraphifyEvolution analyses the entire history of the application, code-smell-visualizer provides an opportunity to navigate between the branches and commits of the app. As mentioned in chapter 4.8.3, interviewees noted that the feature is useful.

Additional information. CodeHouse and CodeCity allow users to see additional information about each class of the software. As mentioned in the qualitative analysis of the evaluation interview (chapter 4.8.4), an additional information window is one of the most requested features for the code-smell-visualizer. At the moment, the feature is not implemented, but it is assigned high priority for future development (chapter 4.8.5).

In conclusion, the key differences between the code-smell-visualizer and other applications are modeling dimensions (2d and 3d), visualization of code evolution, and additional information about the code. In the author's personal opinion, the code-smell-visualizer is more suitable for the real production environment, but some already existing applications are better in terms of providing additional information about the code.

5.2 Restrictions of the applications

The main restrictions of the application came from the integration with GraphifyEvolution. Since GraphifyEvolution is a command-line tool, it is challenging to use it in web development as communication cannot be done via REST API. In addition, deployment of the application is more complicated since the tool uses local paths as an input, meaning that it cannot be deployed on the web. For example, in section 4.3.1 it is described that in order to analyze the application, the path of the project is required. This means that the source code of the application needs to exist locally on the computer. However, it is essential to note that both of these problems are taken into consideration and are already being worked on. Once these problems are solved, the uploading of the source code will be much easier, and it will be possible to deploy the application.

5.3 Future work

Part of the future work is already described in section 4.8.5. There it is discussed that some new requirements were elicited based on the feedback gathered from the interviews. Implementing these requirements is the first priority since these requirements are requested a lot. These requirements should significantly improve the usability of the application. At the moment of writing, the thesis requirements are already being developed. After the requirements are implemented, the next step will be to once again to conduct the interviews with potential users and ask for feedback and new requirements. Since it was conducted that application has the potential to be used in some projects by multiple stakeholders, the code-smell-visualizer application should be promoted to other projects as well. In addition to the features proposed by interviewees, there are many important features that need to be added in order for code-smell-visualizer to be useful. Some of the ideas are listed below:

1. Currently the application works only locally. The application needs to be deployed so users can use the tool without the need to download the source code.

2. In order to deploy the application authentication and authorization of users should be added. This must be done for user protection and to provide more customized user experience.

3. In order to make application more usable by developers, integration with more popular static code analysis tools like SonarQube should be added. Code analysis from other sources should be combined with existing code analysis that comes from GraphifyEvolution.

4. The application can encounter some performance issues when code base of the application is very large. For example analysis of the current application took over two hours. This means that asynchronous analysis should be added.

5.4 Lessons learned

There were many lessons learned while planning, developing, and evaluating the application. It is important to share these lessons in case someone decides to build a similar application in the future.

The most important lesson is that research of existing technologies is essential. The selection of the proper framework or tool can make the development process much more manageable. It is important to compare different technologies also based on the documentation and community size. In addition, it is important first to gather the requirements and only then look for existing technologies that can be used to implement the requirements. In the thesis, a comparison between three visualization frameworks was made before the d3 library was selected. In the user interaction section (table 8) it is explained that functionalities like zooming and moving around are very easy to implement in d3. Since one requirement was to allow users to zoom and move in the visualization, the d3 library was the heavy favorite to be selected for the implementation of visualization. Had this research not been done, implementation of the zooming and navigation functionalities could have been much more complicated than it turned out.

Another lesson learned is that before planning the development, it is a good idea to find a focus group for whom the application will be developed. Before gathering the requirements, it was thought that the software would benefit Developers, Quality Assurance, and Product Owners. However, after conducting the interviews and talking with representatives of different stakeholders, it turned out that Quality Assurance and Product Owners were not that interested in the software, but Scrum Masters and Developers thought that the product could benefit their job. With that in mind, the requirements were gathered only from Developers and Scrum Masters, and the application was developed based on their needs.

Finally, whenever possible, developers should prefer agile software development [48]. It was mentioned in chapter 3 that the development process for building the application was based on Waterfall methodology[49]: firstly, the prototype was created, and requirements were gathered, then software was developed, and finally it was tested and evaluated. The main problem encountered during this process was that no end-users feedback was gathered during the implementation. If the agile methodology was used, then maybe it would have been noticed that use cases CSV-FR16 and CSV-FR17 are more important than use case CSV-FR8, and they would have been implemented instead. This would have resulted in better user feedback.

[48] https://www.guru99.com/agile-scrum-extreme-testing.html
[49] https://www.guru99.com/what-is-sdlc-or-waterfall-model.html

6 Conclusion

The thesis is focusing on the visualization of code smells. It is researched what data about the code should be analyzed and how to present it to the users with and without the knowledge of the code. The requirements for the application are collected by conducting interviews with different stakeholders from two Nortal projects. The application for visualization is built based on the requirements gathered. The application is using GraphifyEvolution code smell recognition tool, but the architecture of the application allows to easily change static code analyzers. Application is built in a way that it is easy to add or modify features. The usefulness and ease of use of the application are evaluated by conducting interviews with multiple stakeholders from two previously mentioned projects. For the testing and evaluation of the software, the source codes of the two previously mentioned projects are analyzed and visualized. The interviews are conducted using the analysis of the projects. In conclusion, it can be said that the main goal of the thesis is achieved. Application for visualization called code-smell-visualizer is built and evaluated. The answers to the research questions are the following:

- Q1. Code smell visualisation tools are beneficial for Software Development. From the responses of the persisted usefulness questions it can be concluded that application is useful and has a lot of potential.
- Q2. Based on the research done in the thesis it can be concluded that developers and scrum masters benefit the most from the code smell visualisation application.

The code-smell-visualizer application has a lot of potential, and there are many features and requests for future development of the application. New requirements are made based on the feedback gathered from the evaluation interview. The author of the thesis will continue to work on the improvements of the application.

References

[1] B.A. Wichmann, A.A. Canning, D.L.Clutterbuck, L.A. Winsborrow, N.J. Ward, and D.W.R. Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69–75, 1995.

[2] D. Savchenko, T. Hynninen, and O. Taipale. Code quality measurement: Case study. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1455–1459, 2018.

[3] P. Anderson. The Use and Limitations of Static-Analysis Tools to Improve Software Quality. *CrossTalk-Journal of Defense Software Engineering*, 21:18–21, 2008.

[4] J.O. Bakare. Key Factors That Determines Internalization of Service Firms: Evidence From Estoina SMEs – Bolt and Nortal. Master's thesis, Tallinn University of Technology, 2019.

[5] R. Knaster and D. Leffingwell. *SAFe 5.0 Distilled; Achieving Business Agility with the Scaled Agile Framework*. Pearson Education, 2020.

[6] L. Apke. *Understanding The Agile Manifesto; A Brief & Bold Guide to Agile*. Lulu Publishing, 2015.

[7] R. Mili and R. Steiner. Software Engineering - Introduction. *Revised Lectures on Software Visualization, International Seminar*, pages 129–137, 2001.

[8] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola. Code Park: A New 3D Code Visualization Tool. *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 2017.

[9] A. Schreiber and M. Misiak. Visualizing Software Architectures in Virtual Reality with an Island Metaphor. In *Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation: 10th International Conference*, pages 168–182. Springer International Publishing, 2018.

[10] N. Capece, U. Erra, S. Romano, and G. Scanniello. Visualising a Software System as a City Through Virtual Reality. In *International Conference on Augmented Reality, Virtual Reality and Computer Graphics*, pages 319–327. Springer International Publishing, 2017.

[11] A. Hori, M. Kawakami, and M. Ichii. CodeHouse: VR Code Visualization Tool. *2019 IEEE Working Conference on Software Visualization (VISSOFT)*, 2019.

[12] K. Rahkema and D. Pfahl. GraphifyEvolution - A Modular Approach to Analysing Source Code Histories. *Proceedings of MobileSoft 2021*, 2021. (accepted).

[13] D. Hennig. N-Tier Application Design. *CODE Magazine: 2000 - Summer*, 2000.

[14] I. Cooper. Decoupling from ASP.NET - Hexagonal Architectures in .NET. https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net#video, 2014. (Last visited: 10.03.2021).

[15] J. Stenberg. Exploring the Hexagonal Architecture. https://www.infoq.com/news/2014/10/exploring-hexagonal-architecture/, 2014. (Last visited: 11.03.2021).

[16] H. Graça. DDD, Hexagonal, Onion, Clean, CQRS, ... How I put it all together. https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-togethe#connecting-the-tools-and-the-application-core, 2017. (Last visited: 11.03.2021).

[17] A. Cockburn. Hexagonal architecture. https://alistair.cockburn.us/hexagonal-architecture/, 2005. (Last visited: 12.03.2021).

[18] T. Hombergs. *Get Your Hands Dirty on Clean Architecture: A hands-on guide to creating clean web applications with code examples in Java*. Packt Publishing Ltd., 2019. (Last visited: 11.03.2021).

[19] J. Palermo. The Onion Architecture : part 1. https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/, 2008. (Last visited: 16.03.2021).

[20] M. Kramer. Best Practices in Systems Development Lifecycle: An Analyses Based on the Waterfall Model. *Review of Business & Finance Studies*, 9:77–84, 2018.

[21] M. Vestola. A Comparison of Nine Basic Techniques for Requirements Prioritization. https://www.semanticscholar.org/paper/A-Comparison-of-Nine-Basic-Techniques-for-Vestola/2ff7d2c0fe7da8fd3dcd22e635335a33b675fcd2, 2010. (Last visited: 12.04.2021).

[22] J.A. Khan, I.U. Rehman, Y.H. Khan, I.J. Khan, and S. Rashid. Comparison of Requirement Prioritization Techniques to Find Best Prioritization Technique. *I.J. Modern Education and Computer Science*, 11:53–59, 2015.

[23] F.A.C. Pinheiro. Requirements Traceability. In *Perspectives on Software Requirements*, pages 91–113. Springer, Boston, MA, 2004.

[24] Y. Li and W. Maalej. Which Traceability Visualization Is Suitable in This Context? A Comparative Study. In *Requirements Engineering: Foundation for Software Quality*, pages 194–210. Springer-Verlag, 2012.

[25] F.D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13:319–340, 1989.

[26] N. Marangunić and A. Granić. Technology acceptance model: a literature review from 1986 to 2013. *Universal Access in the Information Society*, 14:81–95, 2015.

[27] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[28] V. Savkin. Nrwl Nx — An open source toolkit for enterprise Angular applications. `https://blog.nrwl.io/nrwl-nx-an-open-source-toolkit-for-enterprise-angular-applications-38698e94d65l`, 2017. (Last visited: 30.04.2021).

[29] F. Cheng. *State Management with NgRx*, pages 205–241. Apress, Berkeley, CA, 2018.

[30] D. Bugl. *Learning Redux*. Packt Publishing Ltd., 2017.

[31] T. Koleoso. Test Quarkus Applications. In *Beginning Quarkus Framework*, pages 257–293. Apress, Berkeley, CA, 2020.

[32] J. Acetozi. Integration Tests. In *Pro Java Clustering and Scalability*, pages 127–133. Apress, Berkeley, CA, 2017.

Appendix

I. Materials

1. Have you ever used code analysis tools? If yes:
   (a) Which tools did you use?
   (b) Why these tools?
   (c) What did you like or dislike about the tools?
2. Have you ever tried code visualisation tools? If yes:
   (a) Which tools did you use?
   (b) What did you like or dislike about the tools?
3. Do you think that visualizing entire application will have some benefits for your job?
4. What do you think might be the advantage of visualization tool over static code analysis?
5. What would you want to see in the visualisation of the code?

Figure 52. Requirements gathering interview questions.

Table 15. Functional requirements

ID	**CSV-FR1**
Name of the requirement	View code smells
Descriotion	Developer should be able to view all existing code smells according to their type
Actors	Developer
Precondition	1. Application is chosen
Primary scenario	1. Developer is on the main page 2. Developer sees class based and method based code smells in separate views
Postcondition	Developer sees existing code smells separated by their type

ID	**CSV-FR2**
Name of the requirement	View code smell impacts
Description	User should be able to see impacts of indi-vidual code smells on the application
Actors	User
Precondition	1. User is on the main page 2. List of code smells is available
Primary scenario	1. Impact of each code smells is shown as colour coding: - Green: No impact -Yellow: Medium impact -Red: High impact
Postcondition	User sees impacts of individual codesmells

ID	**CSV-FR3**
Name of the requirement	Adjust code smell parameters
Description	Advanced user should be able to adjust code smell parameters
Actors	Advanced user
Precondition	1. Application is chosen
Primary scenario	1. Lead developer navigates to parameter modification view 2. Lead developer adjusts parameters 3. Lead developer clicks ‘Save’ button
Postcondition	Code smell parameters are adjusted

ID	CSV-FR4
Name of the requirement	View code analysis
Description	User should be able to view code analysis
Actors	User
Precondition	1. Application is chosen 2. User has made desirable adjustments
Primary scenario	1. User clicks on ‘Get code analysis’ button
Postcondition	Code analysis is shown to the user

ID	CSV-FR5
Name of the requirement	Distinguish severity of code smells
Description	User should be able to distinguish between severity of code smell in different classes of the application
Actors	User
Precondition	1. Code analysis is loaded
Primary scenario	1. Classes in code analysis are distinguished by color based on code smell severity: - Green: No code smells in the class - Yellow: Some code smells in the class - Red: Too many code smells in the class
Postcondition	User can distinguish between severity of code smells in the classes

ID	CSV-FR6
Name of the requirement	View usage of class
Description	User should be able to distinguish between often used and rarely used classes
Actors	User
Precondition	1. Code analysis is loaded
Primary scenario	1. Classes in code analysis are distinguished by size based on how widely the class is used
Postcondition	User can distinguish between often and rarely used classes

ID	CSV-FR7
Name of the requirement	Group classes
Description	User should be able to see classes in the same packages
Actors	User
Precondition	1. Code analysis is loaded
Primary scenario	1. Classes in code analysis are grouped based on the package
Postcondition	User sees classes that belong to the same packages

ID	CSV-FR8
Name of the requirement	Adjust color coding
Description	User should be able to adjust coloring of the code analysis
Actors	User
Precondition	1. Code analysis is loaded
Primary scenario	1. User sets a numerical range for Green, Yellow and Red colors 2. Color of classes is adjusted based on code smell severity in the class and range that user has selected
Postcondition	User successfully adjusted color coding of the code smell severity in the class

ID	CSV-FR9
Name of the requirement	Navigating in visualization
Description	User should be able to navigate in the application
Actors	User
Precondition	1. Code analysis is loaded
Primary scenario	1. User navigates in the code smell analysis using mouse or trackpad 2. Image is adjusted based on user navigations
Postcondition	User can navigate in the application

ID	CSV-FR10
Name of the requirement	Viewing branches
Description	User should be able to view branches in the application
Actors	User
Precondition	1. Application under analysis has branches
Primary scenario	1. User clicks on branch selection 2. Branches of the application under analysis are displayed
Postcondition	User sees branches of the application

ID	CSV-FR11
Name of the requirement	Branch selection
Description	User should be able to select branches in the application
Actors	User
Precondition	1. Application under analysis has branches 2. User views available branches
Primary scenario	1. User clicks the branch 2. Application analysis is reloaded based on the selected branch
Postcondition	Code analysis is done based on the latest commit of the selected branch

ID	CSV-FR12
Name of the requirement	View commits
Description	User should be able to view commits under the branch
Actors	User
Precondition	1. Application under analysis has commits 2. Branch is selected
Primary scenario	1. User is shown List of commits made in the specified branch

ID	CSV-FR13
Name of the requirement	Select commit
Description	User should be able to select commits
Actors	User
Precondition	1. Application under analysis has commits 2. Branch is selected
Primary scenario	1. User selects commit by clicking on it
Postcondition	Code analysis is done based on the selected commit of the selected branch

ID	CSV-FR14
Name of the requirement	Upload application
Description	User should be able upload application
Actors	User
Precondition	1. User is on upload page
Primary scenario	1. User inserts the location of the project 2. Project is uploaded
Postcondition	User uploaded the project

ID	CSV-FR15
Name of the requirement	Select project
Description	User should be able to select the project
Actors	User
Precondition	1. User is on upload page
Primary scenario	1. User clicks on existing projects tab 2. User selects projects by clicking on it 3. User clicks 'Confirm' button
Postcondition	User has selected the project and is redirected to code analysis view

ID	CSV-FR16
Name of the requirement	Additional class data
Description	User should be able to see methods that belong to the class
Actors	User
Precondition	1. Code analysis is loaded
Primary scenario	1. User hovers over the class 2. Information about class is displayed: Name of the class Methods that are in the class
Postcondition	User sees additional information about the classes

ID	CSV-FR17
Name of the requirement	Method code smells
Description	User should be able to see code smells severity in the methods
Actors	User
Precondition	1. Code analysis is loaded 2. User hovered over the class
Primary scenario	1. Methods are colored based on the severity of code smells: - Green: no code smells - Yellow: 1-2 code smells - Red: >2 code smells
Postcondition	User sees severity of code smells in the method

Table 32. Non-functional requirements

ID	**CSV-NFR1**
Name of the requirement	Portability - Browser support
Description	User should be able to see code smells analysis in Google Chrome
Success criteria	User can see code smell analysis in Google Chrome

ID	**CSV-NFR2**
Name of the requirement	Test coverage
Description	Application is covered with tests
Success criteria	Application has at least 90% test coverage

ID	**CSV-NFR3**
Name of the requirement	Module separation
Description	Domain logic, web layer and database layer should be separated from each other
Success criteria	95% of classes don't have intensive coupling code smell

```
version: '3'
services:
  neo4j:
    image: neo4j:3.5
    restart: unless-stopped
    ports:
      - 7474:7474
      - 7687:7687
    volumes:
      - ./conf:/conf
      - ./data:/data
      - ./import:/import
      - ./logs:/logs
      - ./plugins:/plugins
    environment:
      # Raise memory limits
      - NEO4J_dbms_memory_pagecache_size=1G
      - NEO4J_dbms.memory.heap.initial_size=1G
      - NEO4J_dbms_memory_heap_max__size=1G
```

Figure 53. Code snippet from Neo4j container compose file.

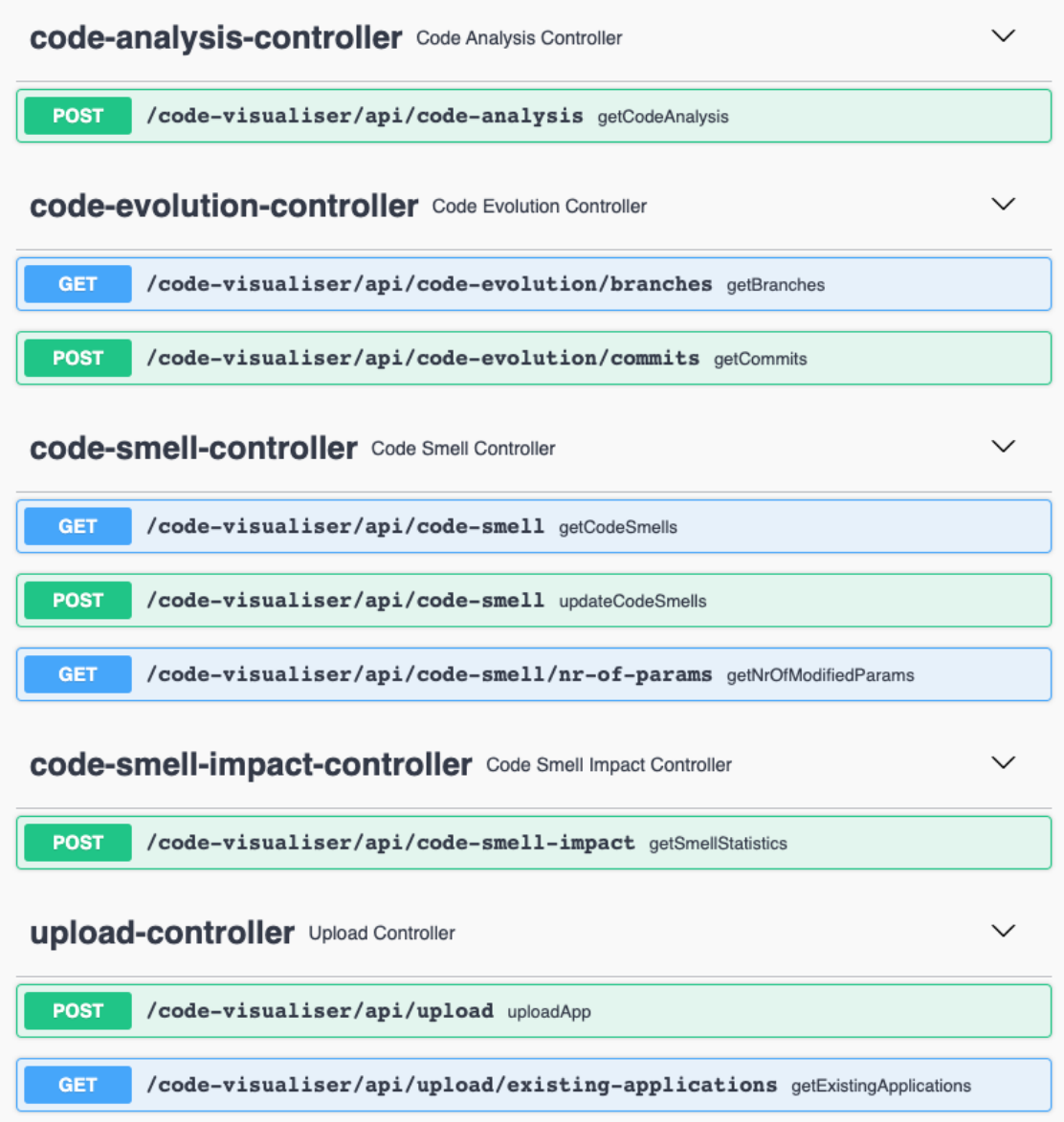

Figure 54. Screenshot from swagger displaying API endpoints of the application.

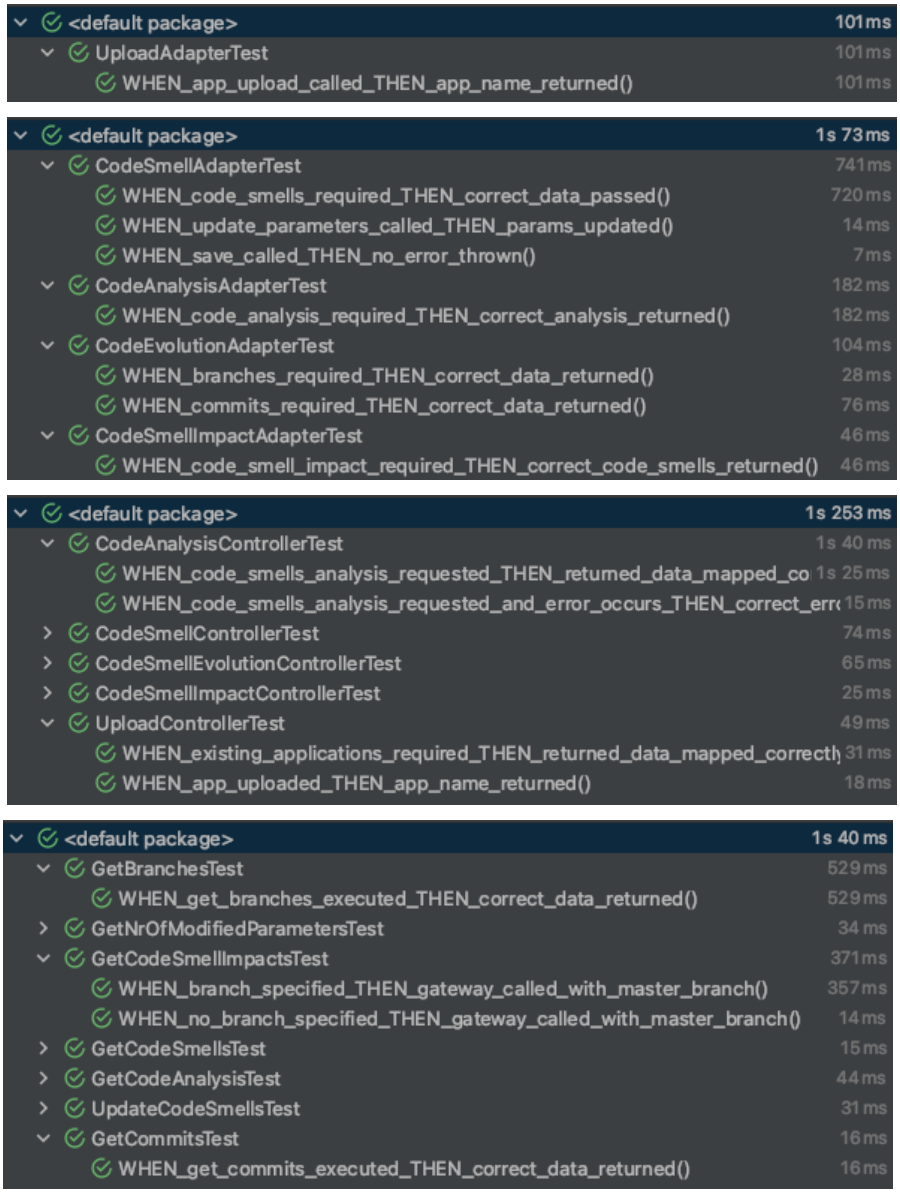

Figure 55. Screenshots of unit test execution results in code-smell-visualizer-api

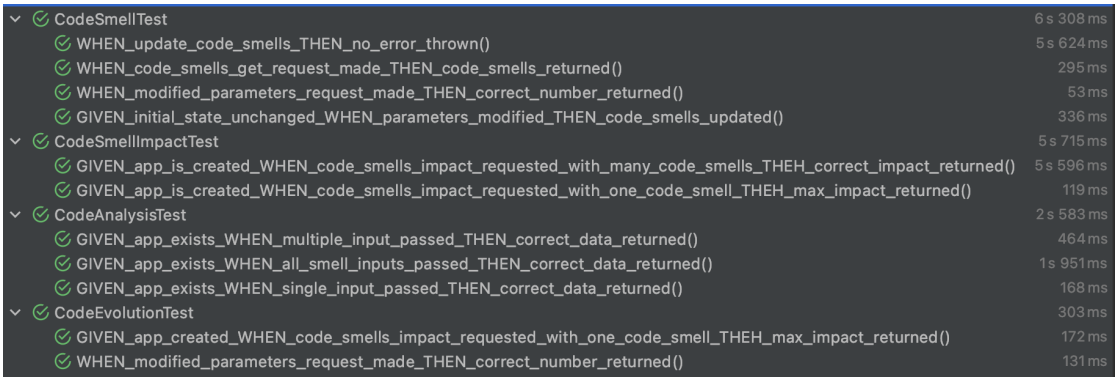

Figure 56. Screenshots of integration test execution results in code-smell-visualizer-api

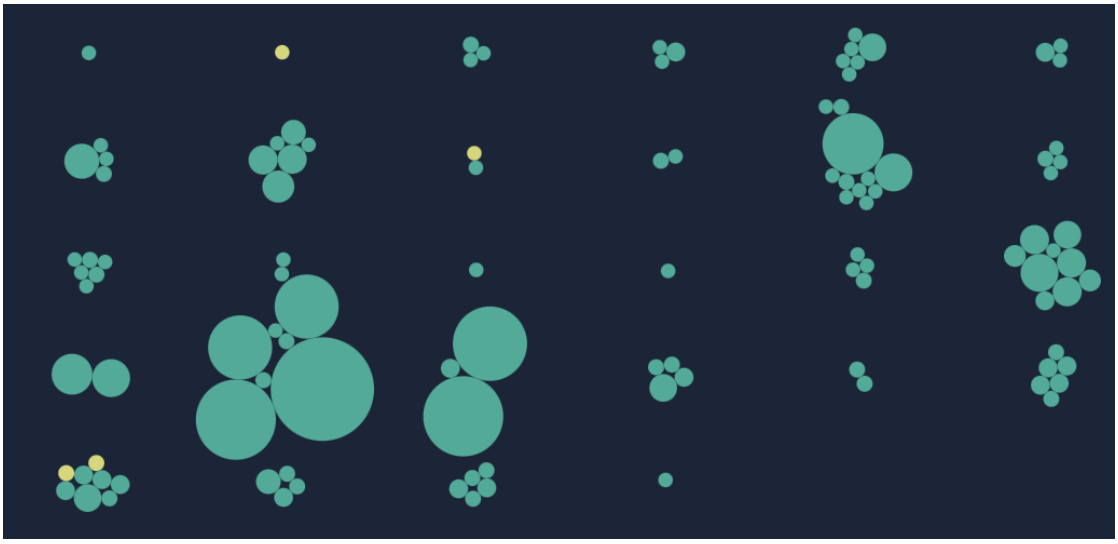

Figure 57. Screenshot of code-smell-visualizer-api feature envy code smell analysis

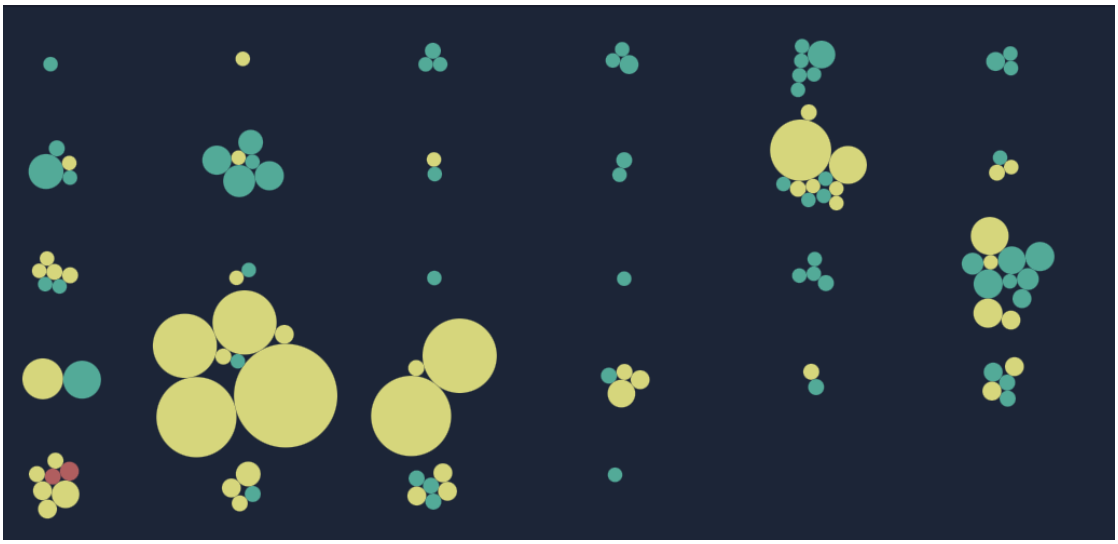

Figure 58. Screenshot of code-smell-visualizer-api code smell analysis

1. **Introduction:** (2 min)
   (a) Introduce the topic of the thesis. (30s)
   (b) Tell stakeholder why they are being interviewed (30s)
   (c) Describe that interview is not recorded and approximate length of the interview is 30 minutes. (30s)
   (d) Describe the flow of the interview: (30s)
      i. Background information presentation
      ii. Presentation of the developed application
      iii. Questions about usability of the application
      iv. Open discussion, suggestions for new functionalities
2. **Presentation of existing work:** Present already existing work with visualizations. Show images of existing applications (2 min)
3. **Discussion about existing work:** Ask for interviewees opinion about existing work: would the interviewee use the applications? (3 min)
4. **Presentation of application developed:** Present the application that is developed. Briefly describe different functionalities of the application. Describe what does visualization represent. (2 min)
5. **Questions about technology acceptance:** Ask interviewee questions about the application (7 min)
6. **Discussion about the software:** Open discussion about the usages and improvements of the application (10 min)

Figure 59. Evaluation interview plan.

1. Do you understand what the application is about?
2. How easy it is to use the application in general?
3. How useful is the application in general?
4. How easy is the navigation between branches and commits?
5. How useful is the navigation between branches and commits?
6. How easy is the selection of code smells?
7. How useful is the selection of code smells?
8. How easy is adjusting code smell parameters?
9. How useful is adjusting code smell parameters?
10. How easy is it to find classes that need refactoring?
11. How easy is it to adjust the level of severity of code smells?
12. How useful is the level adjustment of code smell severity?
13. Would the application improve the code quality of your project?
14. Does the application provide everything necessary for recognition of the code smells?
15. Would the usage of the application be beneficial for your everyday job?

Figure 60. Evaluation interview questions based on technology acceptance model.

II. Repository links

- Full project group: https://gitlab.com/code-smell-visualizer
- code-smell-visualizer-api: https://gitlab.com/code-smell-visualizer/code-smell-visualizer-api
- code-smell-visualizer-web: https://gitlab.com/code-smell-visualizer/code-smell-visualizer-web
- dev-scripts: https://gitlab.com/code-smell-visualizer/dev-scripts

III. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Miron Storožev**,
(author's name)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Exploration of Techniques to Visualise the Code Quality,
   (title of thesis)

   supervised by Dietmar Alfred Paul Kurt Pfahl and Kristiina Rahkema.
   (supervisor's name)

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 licence 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 licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Miron Storožev
14/05/2021