Автономная некоммерческая организация высшего образования
«Университет Иннополис»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
(БАКАЛАВРСКАЯ РАБОТА)
по направлению подготовки
09.03.01 - «Информатика и вычислительная техника»

GRADUATION THESIS
(BACHELOR'S GRADUATION THESIS)
Field of Study
09.03.01 – «Computer Science»

Направленность (профиль) образовательной программы
«Информатика и вычислительная техника»
Area of Specialization / Academic Program Title:
«Computer Science»

Тема / Topic	**Анализ и визуализация кода для языка программирования Дарт / Code analysis and visualization for the Dart programming language**

Работу выполнил / Thesis is executed by	**Нугаев Тимур Аладдинович / Timur Nugaev**	подпись / signature
Руководитель выпускной квалификационной работы / Supervisor of Graduation Thesis	**Зуев Евгений Александрович / Eugene Zouev**	подпись / signature

Иннополис, Innopolis, 2023

Contents

List of Figures

Listings

Abstract

The successful organization of a large project's codebase demands consistent analysis and refactoring for future scalability and maintenance. Tools such as static code analysis, refactoring tools, dependency graphs, UML diagrams, etc., are used in this process. Most software development environments, like GitHub or JetBrains IntelliJ IDEA, are essentially text editors with integrated compiling systems, combining these tools.

This thesis introduces a different strategy: a single-time static analysis to generate a browser-based Integrated Development Environment (IDE) accessible from any browser-enabled device. The IDE meets modern standards for code analysis, functionality, and design, without requiring installation, project codebase and dependencies, and project indexing. The developed tool, DartBoard, implements this approach, and its evaluation focuses on its effectiveness and usability.

Most current program visualization tools lack portability and have a steep learning curve. This research aims to solve these problems by creating a lightweight, platform-independent, code viewing platform. This innovative approach, with its methodology, evaluation, and results, offers potential benefits and is detailed in this thesis.

Chapter 1

Introduction

1.1. Objective

This chapter introduces the motivation and context for the development of DartBoard, a tool for generating an HTML document from a Dart/Flutter project to facilitate code understanding and maintenance. The chapter is organized into sections discussing the problem insight, task description, and applicability of DartBoard.

1.2. Problem insight

Nowadays, there are so many giant large-scale projects like Windows or the gcc compiler. All of them demand incredible levels of dedication to develop and maintain the thousands or millions of lines of code. So, this introduces a couple of problems that this thesis aims to address. There are three main reasons why projects get complex: - Scale. As the program grows in size, it becomes harder and harder to manage. The codebase may double in size in a matter of weeks. This is inevitable. That is why it is not our goal to prevent this from happen-

ing but rather make the process of understanding how the code in the project is structured as mild, pleasing, and, most importantly, fast as humanly possible. - Teamwork. Many programmers working on the same project may introduce additional complexity to the codebase. The teammates will need to understand each other's code in order for them to work as one code module. This takes time and we are proposing to minimize it, as well. - Legacy. The system may get so complex, that over time even the programmer who wrote a particular piece of code may not understand what it does.

So, this is why people started to visualize their code. There are many ways to visually represent a codebase: from project tree navigation to UML and class inheritance diagrams. There are also ways to represent the function call stack, which allows for a better debugging experience. All of them aim to better the comprehension of the codebase as well as to decrease the time (and subsequently cost of development) needed to fully dive into the project code.

1.3. Task description

The goal for this research is to build a fully autonomous, stand-alone application called DartBoard that should be able to generate an HTML document from a project written using the Dart/Flutter stack. This HTML document should be generated from the existing source code and provide tooling necessary for the visual aid that a programmer most frequently needs while developing a project. Such tooling should include syntax highlighting, jumping to variable and function declarations and other features described in the Functional Requirements section of this paper. The application takes in a path to the source code of the project being analyzed and produces an HTML document described above. The application

should be available to be spun on a web-server as well as in the console of the end-user.

1.4. Applicability

The final application is going to be useful for programmers to review other programmers' code, providing a lightweight and efficient solution for code analysis and visualization. The tool highlights the different code entities based on their syntactic and semantic role (data types in red, variable identifiers in blue, for example). The constructed HTML will consist of tags that, in case of a variable usage, wrap the variable and link to its declaration. The proposed solution requires less computations (therefore, time and cost) to generate the HTML than to spin up a whole IDE, all the processes of the IDE, etc. The document is only generated once, when the end-user launches DartBoard. The whole HTML is then ready to be sent to whomever and operated under almost any conditions, no matter online or offline.

1.4.1. Use cases

These are some potential use-cases for the DartBoard application:

- Code Review.

  The primary use-case for DartBoard is to provide a powerful, stand-alone tool for code analysis and visualization that can be used during code review. With its functionality, DartBoard can help programmers to better understand complex codebases and quickly identify potential issues or bugs.

- Project Collaboration.

Another use-case for DartBoard is to facilitate collaboration among team members working on the same project. With its ability to generate an HTML document that can be shared easily with other team members, DartBoard can help to ensure that all members of the team have a clear understanding of the project's codebase and can work together effectively.

- Debugging.

  DartBoard can also be used as a debugging tool, with its ability to highlight syntax and show documentation on cursor hover helping to identify issues more quickly. By collapsing block scopes and providing a clear project tree, DartBoard can help to narrow down the source of errors and make debugging faster and more efficient.

- Documentation.

  DartBoard can be used as a tool to generate documentation for a project, with its ability to visualize the dependency graph and inheritance tree providing a clear overview of the project's structure. The ability to generate an HTML document that can be easily shared also makes it a convenient way to create project documentation that can be accessed by others.

- Continuous Integration.

  DartBoard can be integrated with CI/CD pipelines, such as Github Actions, to automatically generate the HTML alongside documentation pipelines, linters, testers, builders, and other tools. This use-case can help to ensure that the code is always analyzed and up-to-date, and any issues or bugs are quickly identified and resolved.

1.5. Thesis context

The chapters that follow cover the technique used to create DartBoard, as well as the evaluation of its effectiveness and usability.

The purpose for and significance of static analysis tools in software development will be covered in this introduction chapter, particularly in light of the Dart programming language. We'll also discuss the DartBoard, a brand-new static analysis tool created especially for Dart, and how it answers a need in the present tool environment. The thesis structure will be described in general, describing the chapters' contents. A brief statement about the document's target audience, which includes academic researchers and software engineers interested in static analysis and programming languages, will be made at the end of the Introduction.

In the Literature Review chapter we delve into the theoretical underpinnings of static analysis, its various approaches, and their application to programming languages. Before focusing on how static analysis is applied especially to the Dart programming language, we will first provide a general introduction of static analysis. The functions of several tools now in use will be discussed, along with their advantages and disadvantages. The discussion of the current tools and their shortcomings will therefore be limited to static analysis in the Dart environment, which eventually highlights the need for a tool like DartBoard.

In the Requirements chapter, we shall describe the functional and non-functional requirements for DartBoard . This will cover elements like the features that were essential, the usability, performance, and compatibility standards that served as the tool's design and development roadmap. We will outline the process for obtaining these needs, which will combine a review of the literature, interviews with subject-matter experts, and user feedback. The chapter will conclude with a

review of the determined requirements and a quick discussion of how they may affect the design of the tool.

In the Implementation chapter, we discuss the architecture and design of DartBoard in detail. We will start with a high-level overview of the tool's architecture, describing its main components and their interactions. We then delve into the details of each component, describing its design and implementation. We will discuss the tool's user interface, highlighting the key design decisions that were made to ensure usability. The core analysis engine, which forms the backbone of DartBoard, will also be discussed in detail.

In the Conclusion chapter, I wrap up the thesis by summarizing the key findings and discussing the implications of my work. We will revisit the motivations presented in the Introduction, summarizing how DartBoard addresses the identified needs and gaps. The key features and benefits of DartBoard will be highlighted once more, emphasizing its contributions to the Dart development ecosystem. We will also discuss potential future work, outlining the areas where DartBoard could be further improved or expanded. This chapter will provide closure to the document, bringing together all the threads introduced in previous chapters.

Chapter 2

Literature Review

This chapter reviews the existing literature relevant to our research, focusing on code visualization, source code analysis, and code/markup generation. It aims to provide a comprehensive understanding of the current state-of-the-art in these areas and identify how they inform the development of the DartBoard tool.

As Code Visualization is a creative and broad topic, we want to specify our area of work. For our purposes, we will need to be able to work with the abstract syntax tree (AST), scoping calculus theory [1], and explore different visualization techniques and tools available at the moment [2].

This part is divided into the following sections. Section 2.1 describes the problem of visualization and puts our study in context with the other papers in the field. Section 2.2 presents an overview of analysis methods and everything that has to do with source code analysis. Section 2.3 elaborates on the visualization part of the project research, different approaches to visualization, etc. Section 2.4 discusses the problem of code/markup generation and how it relates to our area of work.

2.1. Overview

Numerous research has been conducted on the topic, studying different approaches to software visualization, such as Line Representation [3], to the right of the screen of the code editor that shows what the file code looks like, zoomed out. A variant of this approach may be seen in many popular text / code editors by default or using plug-ins: for example, in VS Code or Sublime Text. Another visualization technique may be the Summary Representation tool [3] that gives the bird's eye view on the codebase and allows the end-user (the programmer) to see where and how old each of the components of their system is. Alternatively, even 3D code visualization tools exist, such as Code Park [4], that aim to represent the codebase of a software system in 3-dimensional space for better immersion and comprehensiveness. All these methods and approaches aim to give the programmer a complete picture of what is happening with their code, providing as much simplification and understanding of the codebase as possible.

2.2. Source Code Analysis

Program/Code analysis is automatically analyzing the source code to improve it and/or find where problems with it might arise.

These analyzers are utility programs that optimize the program source code and find potential bugs and vulnerabilities. The utilities automatically traverse the codebase in search of fundamental non-runtime errors or provide help writing better code. An example of the former, in the case of Dart, is a fully functional utility maintained by the Google team, called Dart analyze, to make it easier for the end-user to search for errors and bugs statically at the stage of Compilation. The

latter is known as Linters and help people comply with the set of strictly defined rules such as Effective Dart. They analyze the AST and find parts of code that do not meet the standards.

Program analysis is not only about finding where the problem lies but also often solves the auto-correction problem. For example, if a programmer does not like the enforced curly braces in the code, a tool can provide functionality to automatically find the place where it is found in the codebase and then perform a simple code transformation to eliminate the manual labor. Another use-case is to move variables that do not change throughout the whole execution of a loop out of it [5].

In the book "Principles of program analysis," Nielson et al. [5] introduce four main approaches to program analysis: Data Flow Analysis, Constraint Based Analysis, Abstract Interpretation, and Type and Effect Systems. Data Flow analysis is about gathering information about the values computed at multiple points in a computer program. It analyses the data flow in the control flow graph. This analysis allows for optimization facilities. As per constraint-based analysis, it consists of two parts: constraint generation and constraint resolution. Constraint generation outputs a declarative specification of the desired information about the program and resolves it in the second stage [6]. Now, Abstract Analysis is similar and is based on Data Flow Analysis. It abstracts possible values of code chunks without executing the code and is used when actual computation is either impossible or highly expensive. Then, Type and Effect Systems that have been developed for functional, imperative, and concurrent languages [7] and used for associating types to programs. Mostly, it controls the supplied and returned types and ensures type safety. It should be noted that the above-mentioned analysis methods are a subset of the existing ones described in the book. The more interested readers can

read more about it in “Type and Effect Systems” [7].

2.2.1. Static Analysis

One of the most important aspects of this work lies in static analysis. A static code analyzer is a program that looks at the code through the prism of the defined set of patterns and looks for bugs, errors, and vulnerabilities at compile-time [8]. Analyzers can take compiled to machine code programs as well.

Some examples of popular static code analyzers/checkers include Coverity Static Analysis, Fortify, and FindBugs [8].

2.2.2. Name Resolution and Scope Analysis

Name resolution and scope analysis play a crucial role in the process of code visualization and analysis. Accurately resolving variable names and understanding the scoping rules of the programming language are essential for providing an accurate and comprehensive representation of the codebase. Hovemeyer and Pugh [9] discuss the importance of static analysis tools in detecting potential issues in software projects. Although DartBoard is not specifically aimed at finding bugs, the principles of accurate name resolution and scope analysis can still be applied to improve the overall quality of code visualization and analysis offered by the tool. By ensuring precise name resolution and understanding the scoping rules in the Dart programming language, DartBoard can provide valuable insights and aid developers in navigating and understanding the codebase more efficiently.

Name resolution is one of the most important features of this thesis project. In essence, name resolution is the process of finding and binding a variable (and other entities) in the code to its declaration in the same codebase. The declaration may

be located anywhere in the project: from same-file declaration to being in another directory within the same project structure. The algorithm of name resolution should work for all these cases.

In their research "A Theory of Name Resolution," Neron et al. [10] discuss a language-independent theorization of name binding and resolution that would be fit for a modern programming language that has complicated rules of scoping.

The authors specified two stages to name resolution: scope-graph construction and the resolution process. In the scope-graph construction step, using the predefined set of rules specific to the target language, the scope graph is constructed from the AST of the source. Then, using the "language-independent resolution" process, the scope graph is resolved [10].

One other important process that the authors have formalized is "rename refactoring" [10]. It is about the refactoring process that will help us with code and markup generation in the future chapters.

Scoping analysis will also enable us to implement the collapsing of source code blocks which is one of the Formal Requirements for the thesis project.

2.3. Source Code Visualization

Understanding the structure and intricacies of a software project is a critical aspect of modern software development. Traditional text editors, although useful for editing plain text, are not equipped to handle the complexity of code in large-scale projects. Therefore, source code visualization is a vital component for any Integrated Development Environment (IDE) system. It enables developers to analyze and represent code in an accessible, user-friendly manner, improving the overall workflow and efficiency of the development process.

2.3.1. The Essence of Source Code Visualization

Source Code Visualization is the process of analyzing a program's structure and representing it visually, either graphically or textually [1]. This visualization facilitates a more profound understanding of the codebase, enabling developers to navigate and manage the project more effectively. There are two primary types of information that must be analyzed and accounted for in source code visualization:

- Syntactic: This type of analysis focuses on the surface-level aspects of the source code, such as syntax highlighting for keywords, function definition pop-ups, and more [11].

- Semantic: A deeper analysis that considers the meaning and context of the entities within the source code. It includes name resolution, scopes, and other contextual information [11].

In addition to these two aspects, visualizing the system's architecture is essential for gaining a comprehensive understanding of the project. Tools and techniques like architecture diagrams, dependency graphs, and UML diagrams [11] can provide valuable insights into the project's structure and interconnected components.

Popular tools often employ a combination of textual and graphical visualization methods to deliver a versatile and efficient development environment.

2.3.2. Program text visualization

Program text visualization is concerned with the textual representation of source code [11]. It encompasses features like syntax highlighting, code navigation tools, highlights, and block collapsing that aim to simplify the codebase and make it more comfortable to work with.

This section (2.3) serves as an introduction to the concepts of source code visualization within the Literature Review part of the thesis.

2.3.3. Graphical program visualization

Graphical program visualization focuses on visually representing the structure and organization of a software project's codebase. It aims to provide a high-level overview and understanding of the system as a whole, facilitating the comprehension of the project's layout and interconnectivity.

One widely-used example of graphical visualization is UML diagrams [11]. As stated by Xavier Ferré Grau and María Isabel Sánchez Segura, "UML (Unified Modeling Language) is a language that allows modeling, constructing, and documenting the elements that form a software system oriented at objects. It has become the de-facto standard of the industry" [12].

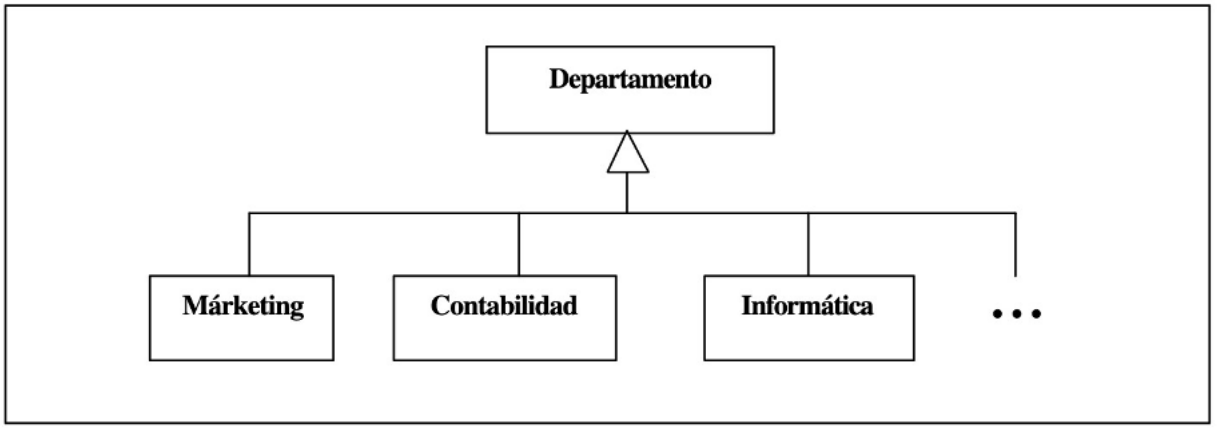

Fig. 1. An example of a UML diagram

An example of a UML diagram is provided in Fig. 1. This example leads us to the topic of Class Inheritance Diagrams (CIDs). They are especially popular and are available to be generated in almost all the modern IDEs available. A CID represents the hierarchy of classes in an object-oriented system, where the nodes

represent the classes and the arrows connecting two classes define the relation between the two classes: the base class is the one on the receiving end of the arrow. Do note that the UML methodology contains many more rules.

Another useful tool for visualizing the structure of a source code file is the Outline View, which is commonly found in modern IDEs. The Outline View provides a hierarchical representation of the file's structural elements, such as classes, functions, and variables, allowing developers to quickly navigate through the code [3].

Block collapsing is an additional method that can be utilized for graphical program visualization. This technique allows developers to hide sections of code, making it easier to navigate and understand the overall structure of the software project [13].

2.4. Generation

Generation is a critical component in the process of code visualization and analysis. It involves creating one or multiple files according to a specific set of predefined rules. The primary purpose of generation in the context of this thesis is to transform source code into a visual representation that aids developers in understanding the codebase more efficiently.

There are various types of generation, including Code Generators, Documentation Generators, and Markup Generators. These generators differ in their production methods, output types, and intended purposes. By understanding the different types of generation and their applications, we can make informed decisions about the techniques and tools we employ in this project.

2.4.1. Documentation Generators

Documentation generators play a vital role in generating documentation from the source code of an application [14]. Common use cases for documentation generators include automatically generating API documentation for back-end server applications, such as those produced by Postman [15], Sphynx **sphynx**, and Swagger [16]. These generators take a generated JSON [17] file of OpenAPI [18] specification and provide developers with detailed information about the application's components, making it easier for them to work with the codebase.

2.4.2. Markup Generators

A markup language is a system for annotating a document in a way that is syntactically distinguishable from the text. It's used to format the text and provide additional information. There exist several markup languages. For example, Markdown [19] or HTML.

For this thesis, I am particularly interested in generating markup code, which is closely related to API documentation generation. Markup generation enables me to create a visual representation of the analyzed code, making it easier for developers to navigate and comprehend the codebase. By incorporating markup generation techniques, I aim to provide valuable insights and facilitate a more efficient development process.

2.4.3. Why HTML?

I chose HTML as the main target markup language for several reasons, which make it an ideal choice for code visualization and analysis in this project:

- Ubiquity. HTML is a widely-used and recognized standard for creating web content. It is supported by virtually all web browsers, making it accessible on a wide range of devices and platforms. This ensures that the generated visualizations can be easily viewed and shared among developers without requiring additional tools or software installations [20].

- Flexibility. HTML offers a rich set of elements and attributes that can be used to represent various types of data and structures. This flexibility allows for the creation of sophisticated textual and graphical visualizations that can effectively communicate the structure and relationships within the codebase [21].

- Interactivity. HTML can be easily combined with JavaScript and CSS to create interactive visualizations. This interactivity enables developers to explore the codebase dynamically, allowing them to drill down into specific sections or elements as needed. This capability can significantly enhance the developer's understanding of the code and facilitate more efficient navigation [22].

- Extensibility. HTML is designed to be extensible, meaning that it can be easily integrated with other web technologies or extended with custom elements and attributes. This extensibility allows for the creation of highly tailored visualizations that can address specific needs or requirements in the code analysis process [23].

- Compatibility. As HTML is continuously evolving, it maintains backward compatibility, ensuring that the generated visualizations remain accessible and usable in the future. This compatibility is essential for long-term maintenance and support of the codebase [24].

In short, HTML is a versatile, powerful, and widely supported markup language that offers significant advantages for code visualization and analysis. By leveraging its capabilities, one can create comprehensive, interactive, and customizable representations of the codebase that aid developers in understanding and navigating the project more efficiently.

2.4.4. Code generation

Code generation is a common practice in software development projects, including those based on the Flutter/Dart stack, which serves as the foundation for our tool. Code generators automate the creation of boilerplate code that can be tedious or time-consuming to write manually. A variety of popular packages are available on pub.dev, such as Auto-route and Freezed, which are widely regarded as standard best practices in the Flutter community [25].

Code generation typically relies on Abstract Syntax Trees (ASTs) or Element Trees to analyze the source code and generate code based on the extracted information. For instance, the json_serializable package (available at https://pub.dev/packages/json_serializable) utilizes the Element Tree to parse annotations [26]. Another example is the Freezed package (available at https://pub.dev/packages/freezed), where developers must annotate specific elements or define factory methods that are subsequently converted into predefined boilerplate code structures that would be tiresome to write manually otherwise (see Fig. 2 and Fig. 3) [27].

```
factory QueueDetailsModel.fromJson(Map<String, dynamic> json) =>
    _$QueueDetailsModelFromJson(json);
```

Fig. 2. An example of the boilerplate substitution.

```
// **************************************************************************
// JsonSerializableGenerator
// **************************************************************************

QueueDetailsModel _$QueueDetailsModelFromJson(Map<String, dynamic> json) =>
    QueueDetailsModel(
      id: json['id'] as int,
      name: json['name'] as String,
      color: json['color'] as String,
      currentUser: UserModel.fromJson(json['on_duty'] as Map<String, dynamic>),
      isOnDuty: json['is_on_duty'] as bool,
      participants: (json['participants'] as List<dynamic>)
          .map((e) => UserModel.fromJson(e as Map<String, dynamic>))
          .toList(),
      trackExpenses: json['track_expenses'] as bool,
      isActive: json['is_active'] as bool,
      isAdmin: json['is_admin'] as bool,
      hash: json['hash_code'] as int,
    );

Map<String, dynamic> _$QueueDetailsModelToJson(QueueDetailsModel instance) =>
    <String, dynamic>{
      'id': instance.id,
      'name': instance.name,
      'color': instance.color,
      'on_duty': instance.currentUser,
      'is_on_duty': instance.isOnDuty,
      'participants': instance.participants,
      'track_expenses': instance.trackExpenses,
      'is_active': instance.isActive,
      'is_admin': instance.isAdmin,
      'hash_code': instance.hash,
    };

```

Fig. 3. Generated boilerplate for the example above.

2.4.4.1. Scanline

In my case, I utilize the Scanline algorithm [28] to keep the order of generation in the HTML generation pipeline. I cover this topic more in details in the Implementation chapter of this thesis.

The scanline algorithm, originally introduced in the context of computer graphics for rendering polygons, can be adapted to process block scoping in a

codebase [28]. The algorithm's core idea is to iterate linearly through the code, keeping track of the current state at each position. This approach enables the efficient handling of block scopes without resorting to nested loops or more complex data structures.

In the context of DartBoard, the scanline algorithm is used to detect the opening and closing of blocks in the code. The algorithm first creates a list of events, where each event corresponds to the starting or ending position of a block. These events are then sorted based on their positions in the code.

The main advantage of the scanline algorithm is its linear time complexity O(N), which allows for efficient processing of block scopes in large codebases. By iterating through the sorted list of events and maintaining the current state at each position, the algorithm can efficiently add the corresponding opening or closing HTML tags to wrap the block content. This process enables the correct visualization of block scopes in the generated HTML document.

So, the Scanline algorithm is necessary for this case as the scopes/blocks are nested inside each other and when generating HTML tags for them, we need to flatten them and iterate over them in the chronological order, which is exactly what the algorithm does.

2.4.5. Documentation generators

Documentation generators most commonly create an HTML project that the end-user can navigate through in the browser independently of dependencies such as the operating system or the browser engine, etc. The most prominent examples of documentation generators are API specification generators, discussed above, such as Postman. The reader can see another example of a documentation generator in Fig. 4 [14].

Swagger [16], ReDoc [29] and other documentation generators typically utilize the OpenAPI specification [18] generated by the backend application itself or by a special utility that takes the backend codebase and coverts it into this large JSON [17] file that follows the OpenAPI specification guidelines. This JSON is then processed by the generators to create a user interface like the reader has seen in Fig. 4.

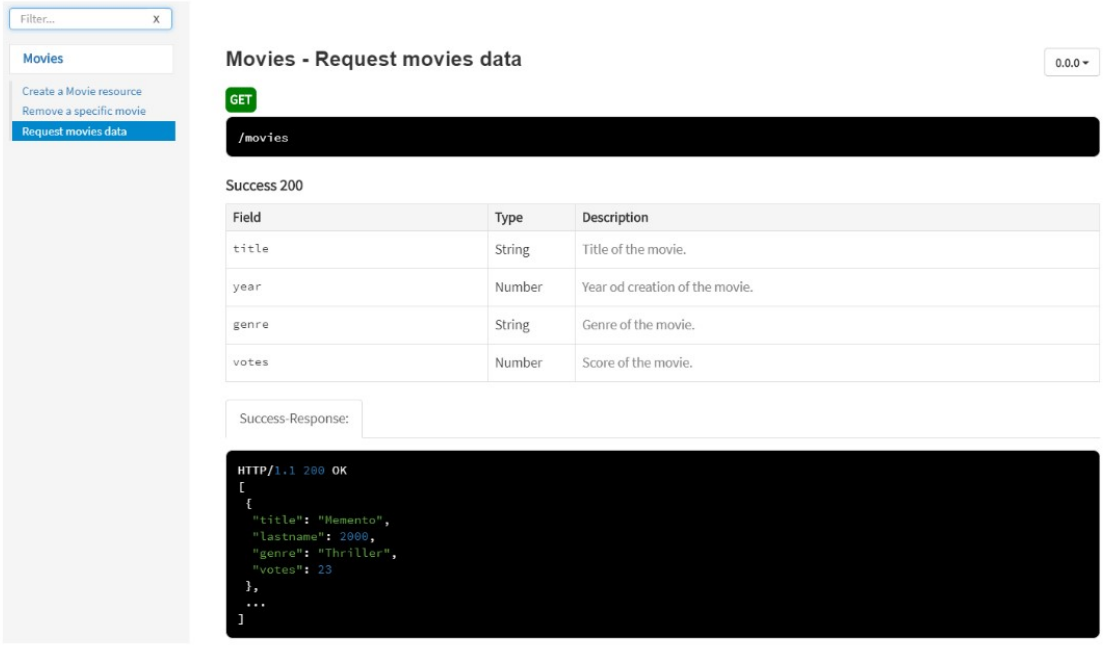

Fig. 4. Styled ReDoc interface.

Chapter 3

Requirements

3.1. Functional Requirements

The Functional Requirements chapter outlines the core features and capabilities that the DartBoard application is designed to provide. These requirements are divided into three priority levels: High, Medium, and Low. The High priority requirements are the core functionality of DartBoard and are considered essential for the application to be effective. The Medium priority requirements provide additional functionality that can enhance the user experience and productivity of the application. The Low priority requirements are optional features that are not considered essential for the core functionality of DartBoard.

By implementing these functional requirements, DartBoard aims to provide a lightweight, fast, and powerful solution for code analysis and visualization for projects written in the Dart programming language.

3.1.1. High priority requirements

The High priority features listed in the Functional System Requirements are essential for the core functionality of DartBoard. These features provide the necessary tools for code analysis and visualization for projects written in the Dart programming language. Given their importance, these features will be implemented in the initial release of DartBoard.

1. Generate HTML files from Dart source code.

   This feature is the core functionality of DartBoard, as it generates the interactable output HTML documents from the source code that can be easily shared and viewed by others.

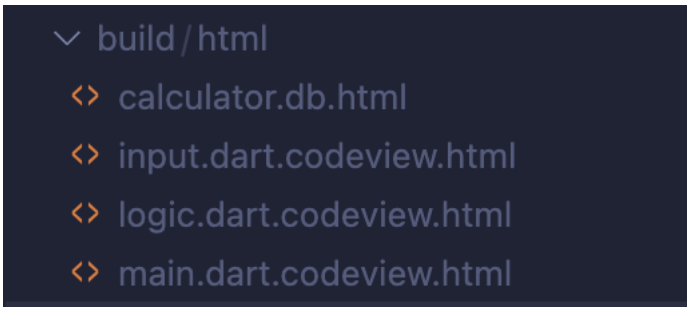

Fig. 5. Output project files structure.

2. Highlight syntax in the generated HTML.

   This feature provides clear and easy-to-read visual cues for different parts of the code, which can help programmers to quickly understand and navigate complex codebases.

```
if (b ≠ 0) {
  result = a ~/ b;
} else {
  print("Division by zero is not allowed.");
  result = null;
}
```

Fig. 6. Syntax highlighting showcase.

3. Jump to declaration.

   This feature allows users to quickly jump to the location in the code where a variable/function or other entity is declared, making it faster to understand the context of that code. Fig. x shows how a variable declaration is highlighted when the user jumps from its usage to the place of declaration.

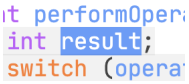

Fig. 7. Declaration entity is highlighted when jumping to declaration.

4. Show documentation on cursor hover.

   This feature provides users with an easy way to view documentation for a particular function or class by simply hovering the cursor over it. In Fig. 8 an example of class specification is shown revealing the information about the annotations associated with this class as well as its name, fields, methods, constructors. This also works with instances of this class showing the documentation for the class that this variable is of: one can see an example in Fig. 9. It pulls the information from the class specification and adds additional information about the variable.

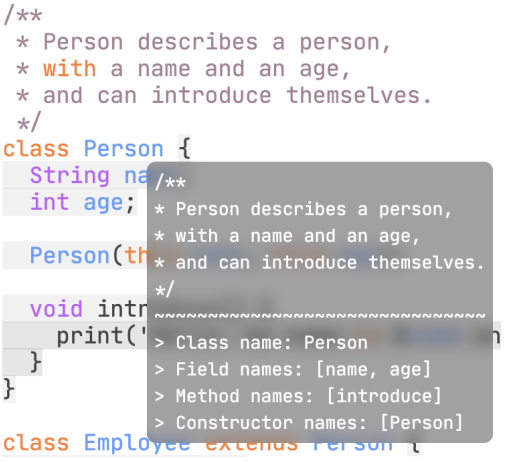

Fig. 8. Class description on cursor hover.

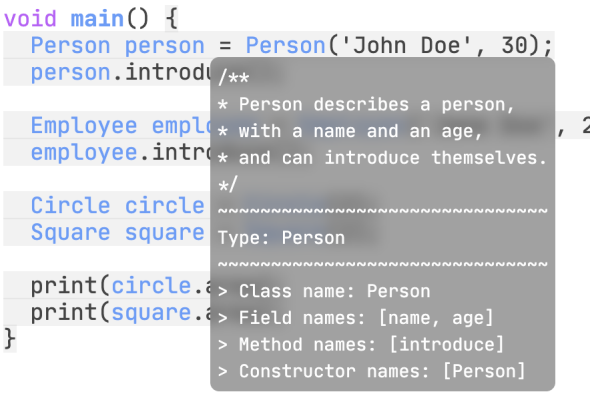

Fig. 9. Description of the class of a usage on hovering the said usage.

5. Collapse block scopes.

   Helps simplify the visualization of code by collapsing block scopes that are not currently being worked on, reducing visual clutter and allowing programmers to focus on the code that is most relevant to them.

Fig. 10. Collapsed state.

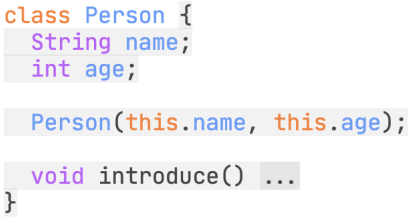

Fig. 11. Unfolded state with folded nested structure inside.

6. Side-bar file navigation.

   This feature allows users to easily switch between different source files within a project, enabling them to navigate between different parts of the codebase.

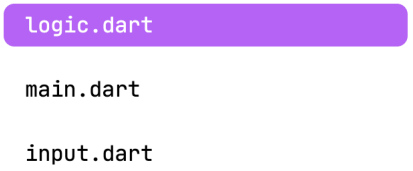

Fig. 12. File navigation interface.

7. Input project into DartBoard.

   Enables users to upload their source files for analysis and visualization. The application takes in the path to the project as a positional parameter and then processes the project as a whole.

Fig. 13. How to pass code as input into DartBoard.

Fig. 14. Test projects for context for the previous Figure.

8. Inter-file connectivity.

   The application allows to have the aforementioned features work cohesively across the whole project. In Fig. 15 a usage of a function in the ‘main.dart’ file of the input project is depicted: `getInput()`. However, it is defined in the `input.dart` file of the project and DartBoard is able to jump from usage to declaration across files within the scope of the project.

Fig. 15. getInput() usage function is located in main.dart.

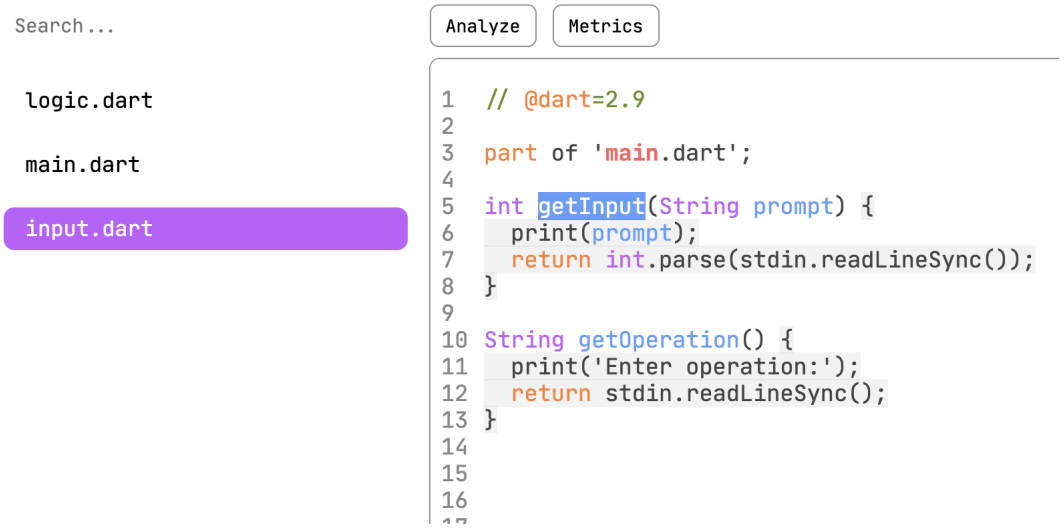

Fig. 16. getInput() declaration function is located in input.dart.

9. Show usages list.

   Upon clicking a declaration entity a list should pop up and show all the places where it is used in the code and be able to jump there from this pop-up window.

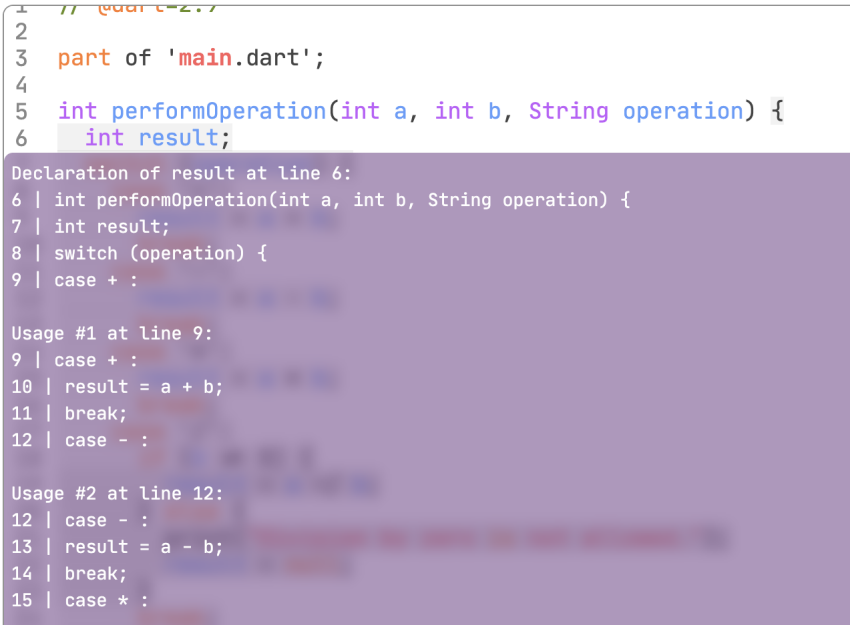

Fig. 17. Pop-up with usages.

3.1.2. Medium priority requirements

The Medium priority features listed in the Functional System Requirements provide additional functionality that can enhance the user experience and productivity of DartBoard. While not essential for the core functionality of the application, these features could be highly beneficial for the users of the application. These features may be implemented in the initial release of DartBoard, or added in future updates depending on their level of complexity and user demand.

1. Export as archive.

   Enables users to export their entire project as a compressed .zip file, making it easier to share and distribute their work.

2. Search by regex expression.

   Allows users to search for specific patterns in the source code using regular expressions, making it easier to find and navigate to specific sections of the

codebase.

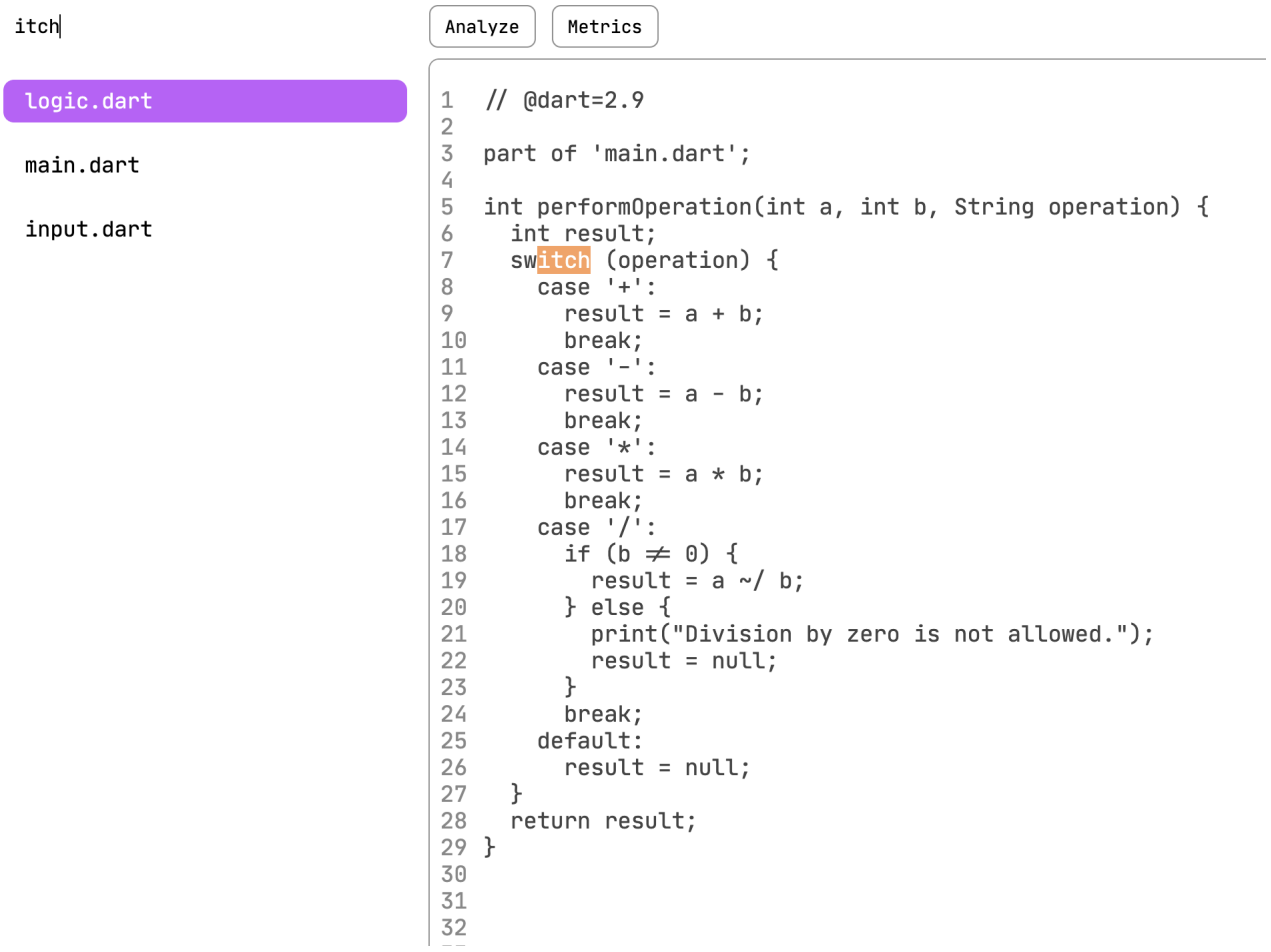

Fig. 18. String search.

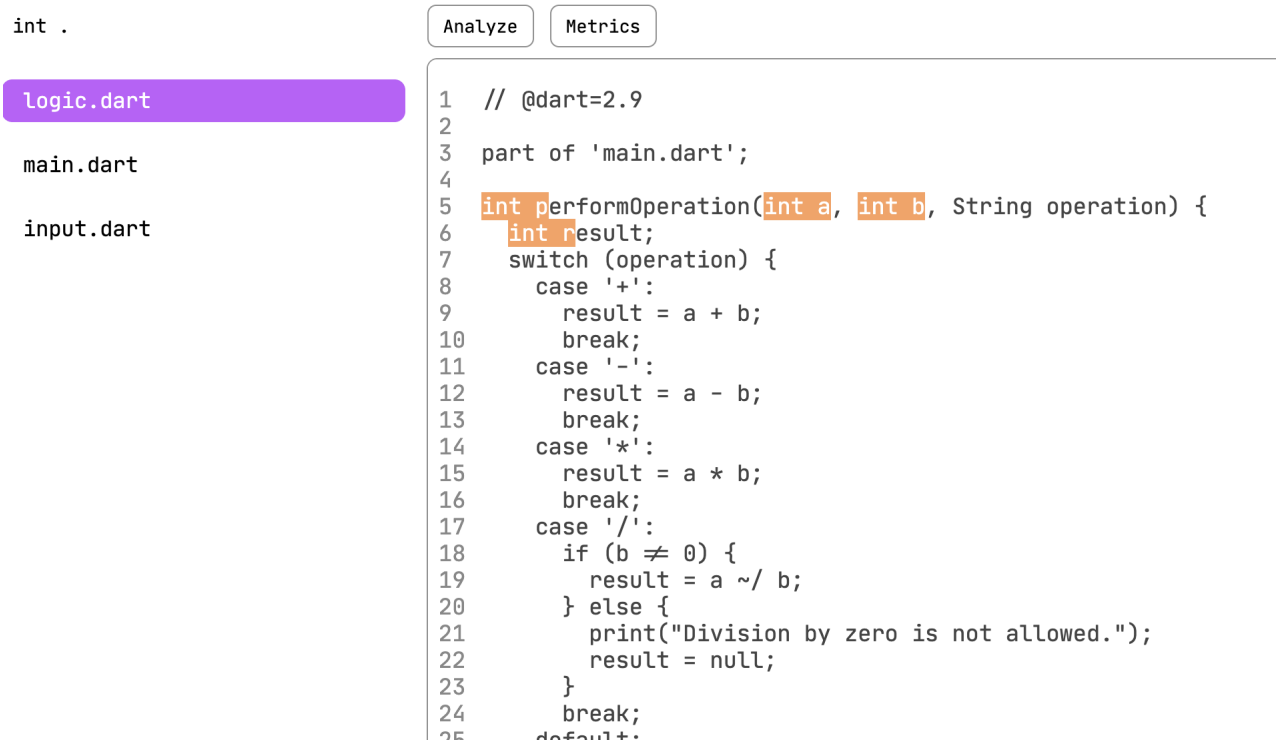

Fig. 19. Regular expression search.

3. Line numbers.

   Adds line numbers to the source code in the generated user interface, assisting users in quickly locating specific sections of code.

Fig. 20. Example of line numbering in the output project.

4. Import zip project (To be implemented in the future).

   Enables users to upload an entire project as a compressed zip file, simplifying the process of importing large codebases into the DartBoard.

5. Import GitHub repo (To be implemented in the future).
   Allows users to upload their project directly from their GitHub repository, streamlining the process of code analysis and visualization.

6. Show Static analysis results on 'dart analyze' (To be implemented in the future).
   Displays the results of Dart's static analysis tool directly within DartBoard, providing users with valuable insights into potential issues in their code.

3.1.3. Low priority requirements

While these features are not essential to the core functionality of DartBoard, they could provide useful additional functionality for the users of the application. However, given the low priority of these features, they may not be implemented in the initial release of DartBoard, but could be added in future updates.

1. Add support for external plugins.
   This feature would enable programmers to extend the functionality of DartBoard by using external plugins that can add new features and capabilities to the application.

2. Make my own plugins.
   This feature would allow programmers to create their own plugins for DartBoard, further customizing and extending the functionality of the application.

3. Highlight the second parenthesis/bracket.
   When the cursor is placed on an opening bracket or parenthesis, the corresponding closing bracket or parenthesis is highlighted, making it easier for

the users to match pairs of brackets in complex code.

4. Highlight the second parenthesis/bracket.

   When the cursor is placed on an opening bracket or parenthesis, the corresponding closing bracket or parenthesis is highlighted, making it easier for the users to match pairs of brackets in complex code.

5. Scrollable minified code tab to the right of the screen.

   This feature provides a high-level overview of the entire code file at a glance, and allows users to quickly navigate to different sections of the code.

Fig. 21. Example of how the minified code stripe could be implemented.

6. Check code for errors.

This feature would provide a rudimentary code validation capability, highlighting syntax and other errors in the code.

7. Check linting errors.
   This feature would check the code against established Dart style and best practice guidelines, highlighting any deviations.

8. Suggest linting fixes.
   In addition to highlighting linting errors, DartBoard could also provide suggested fixes for these issues, helping users to improve the quality of their code.

9. Flutter screen and widget graphical hierarchy.
   This feature would visualize the structure of Flutter screens and widgets, providing a graphical representation of the UI hierarchy.

10. Visualize dependency graph.
    This feature would provide a graphical representation of the dependency relationships between different elements of the code.

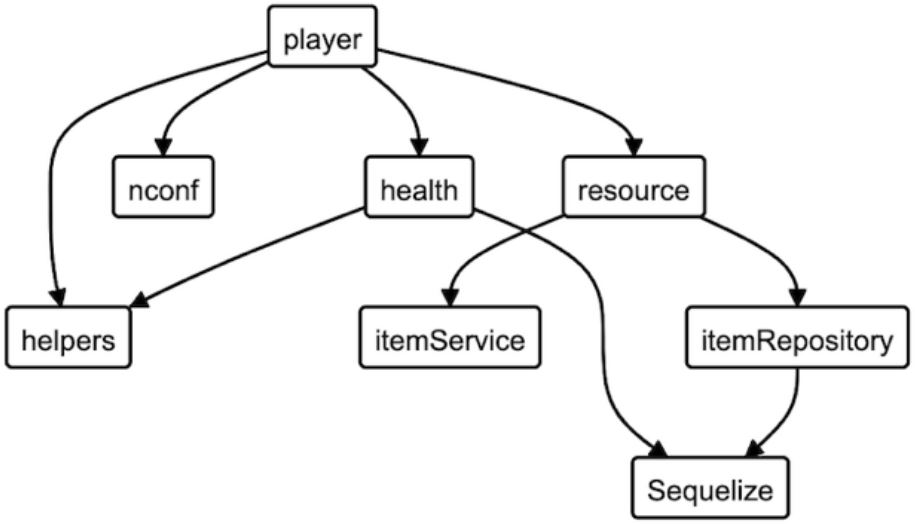

Fig. 22. Example dependency graph.

11. Visualize inheritance tree.

    For object-oriented Dart code, this feature would visualize the class inheritance hierarchy.

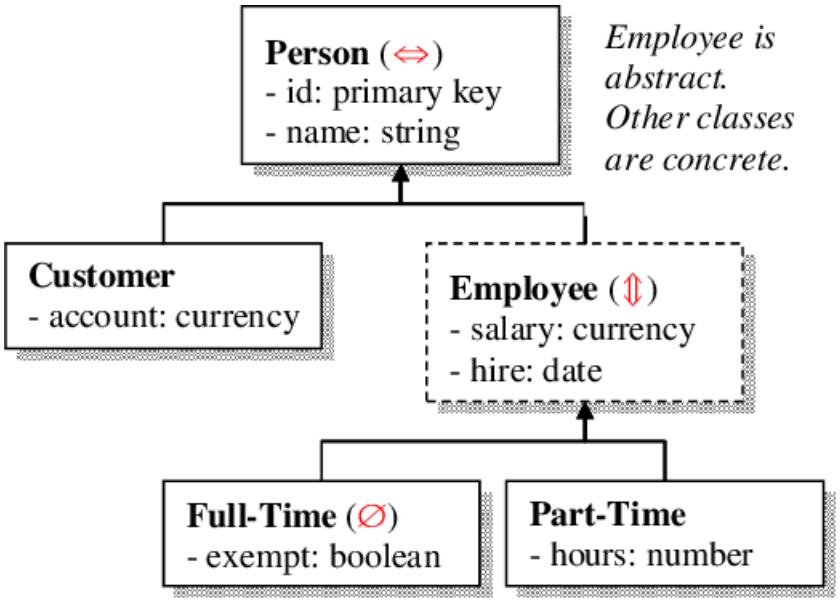

Fig. 23. Example inheritance tree.

12. Pipelines.

    This feature would enable the integration of DartBoard into CI/CD pipelines, automating the generation of HTML documentation as part of the software development process.

13. Rename var/function (refactor).

    This feature would allow users to quickly and easily rename variables and functions throughout the entire project, making it easier to maintain and update the codebase.

14. Replace by regex expression.

    This feature would allow users to perform global search and replace operations using regular expressions, which can help to quickly make changes to large sections of the codebase.

15. Github Action.

    This feature would automate the generation of the HTML alongside the documentation pipelines, linters, testers, builders, etc.

16. Integration with pub.dev.

    make a pub.dev package This would facilitate the distribution and installation of DartBoard as a Dart package.

17. Project tree as an actual tree & Project tree folder collapsing.

    This feature would present the project tree in a more visually intuitive format and allow users to collapse and expand folders within the project tree, making it easier to navigate large codebases with many files and directories.

18. Gray out unused vars and functions.

Highlights variables and functions in the code that are not being used, making it easier for programmers to identify and remove unnecessary code.

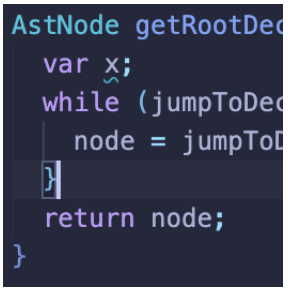

Fig. 24. Identifying unused variables in VS Code.

19. Gray out inaccessible code.

    Highlights code that is not currently accessible due to conditional statements or other control flow structures, making it easier for programmers to understand the context of the code and its dependencies.

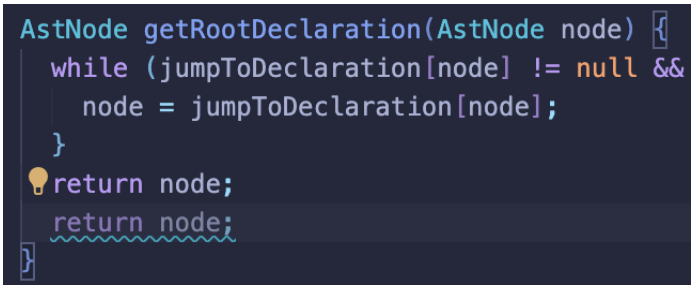

Fig. 25. Identifying unused chunks of code in VS Code.

3.2. Non-functional Requirements

The Non-Functional Requirements chapter outlines the technical and usability requirements that DartBoard must meet in order to be an effective code analysis and visualization tool. These requirements, while not directly related to the core functionality of the application, are essential for ensuring usability, performance, and compatibility across various platforms and environments.

To provide a powerful and flexible solution for code analysis and visualization, DartBoard aims to meet the following non-functional requirements:

- **The HTML generated by the application should have no external dependencies.** This is essential for the output's independence and portability. Users can use DartBoard in a variety of situations without worrying about the availability of external dependencies since it does not rely on them. It streamlines deployment and gets rid of possible issues brought on by missing or incompatible dependencies.

- **The HTML should run on the last three major versions of Chrome, Edge, Firefox, Opera, and Safari.** DartBoard must support a wide range of web browsers in order to be a solution that can be used by everyone. We can reach a larger audience and enhance the user experience by making sure that the tool is compatible with a variety of browsers, enabling people to utilize it in their favorite browsing environment.

- **The HTML should run on MacOS, Windows, and Linux.** In order to make DartBoard available to as many users as possible, support for these three popular operating systems is essential. It ensures that users won't be barred depending on their operating system, expanding the tool's entire audience.

- **DartBoard should have all the necessary functionality of modern code editors except read-only.** This is essential for giving users a full-featured, rich experience. We make sure that users don't feel constrained by DartBoard's features, increasing the likelihood that they will adopt and frequently use the tool. This is accomplished by aligning with the capabilities of contemporary code editors.

- **DartBoard should have the look of a modern IDE in terms of design.** User acceptability can be significantly influenced by the design's aesthetics and familiarity. DartBoard's design is made to resemble contemporary IDEs so that we may take advantage of users' prior expertise to shorten the learning curve and make the tool more appealing.

- **DartBoard should be available as a command-line interface tool.** DartBoard's CLI version offers greater versatility and flexibility in a variety of development scenarios. Due to their efficiency, portability across several platforms, and simplicity, command-line tools are frequently used by developers. This prerequisite makes sure that DartBoard can be utilized in automated procedures and fits neatly into current workflows.

These non-functional requirements are essential to make DartBoard an effective and user-friendly code analysis and visualization tool. By meeting these requirements, DartBoard can provide a high-quality user experience and performance, making it an essential tool for programmers who work with Dart.

Chapter 4

Implementation

This chapter delves into the development and implementation of the DartBoard application. The structure is as follows: Section 5.1 discusses the technology stack and the rationale behind choosing Dart and Flutter. Section 5.2 presents the trade-offs that I had to resolve during development. Section 5.3 highlights the best practices used in DartBoard. Section 5.4 explores the project architecture and some of the design decisions made. Section 5.5 explores the key features and implementation details. Section 5.6 outlines the testing and validation methods employed to ensure the application's quality and reliability.

4.1. Tech stack

DartBoard is built using the Dart programming language and Flutter framework. Dart is a modern, object-oriented programming language that is designed for web and client-side development. It has features that make it well-suited for large-scale applications, including the ability to perform asynchronous operations and efficient garbage collection. Flutter is a mobile app SDK that is built using the Dart language and provides a rich set of pre-built widgets for building native

mobile apps.

Dart and Flutter were chosen as the technical stack for DartBoard for several reasons. First, Dart is the primary language used for developing Flutter applications, which means that the two technologies integrate well together. This ensures that the application can take full advantage of the features and capabilities of both languages. Second, both Dart and Flutter have a strong community of developers and are actively maintained and updated, which means that the technology stack will remain current and up-to-date.

DartBoard also utilizes the internal properties of the Dart programming language to analyze and present the software project as a whole. This allows for efficient code analysis and visualization without the need for external tools or plugins. By leveraging the features of the language itself, DartBoard is able to provide a lightweight and fast solution for code analysis and visualization.

Overall, the choice of Dart and Flutter as the technical stack for DartBoard provides a strong foundation for building a powerful, stand-alone tool for code analysis and visualization.

4.2. Development Trade-offs

In this section, let us delve into various trade-offs in the project development decisions and their implications. We discuss why we chose not to use a templater and/or a JavaScript framework, emphasizing the preference for higher control and quicker deployment. We justify using HTML and explain the decision to generate separate HTML files for better code organization and structuring. We also underline the reasons to forego server usage. This section aims to provide insights into these strategic choices, all of which are crucial for the application development

and the end product itself.

4.2.1. Templating

In my project, I deliberately chose not to employ a templating library for generating HTML and JS code from within Dart. Instead, I opted for a more hands-on approach, working directly with the low-level HTML and JS code. While there are several templating libraries available in the Dart ecosystem, such as Mustache Template, Jaded, and Mason, I made a conscious decision to forgo their usage.

In doing so, I intend to maintain a higher level of control over the output. By directly working with the HTML and JavaScript, I can shape it to my needs without the constraints and abstractions provided by these templating libraries. This allows me to leverage my existing knowledge and experience with HTML and JS, enabling me to work faster and more efficiently.

Plus, bypassing using a templating library, I am able to avoid the overhead in time with learning a new syntax or API. The decision allows me to focus on the project itself and ship it without investing time in understanding and integrating a templating library.

It also reduces the number of dependencies in the project, making it more portable and less reliant on third-party packages.

Ultimately, my choice to forego a templating library reflects my desire for greater control, faster development, and reduced reliance on external tools. By working directly with HTML and JS, I can tailor the project to the specific requirements.

4.2.2. JavaScript / Frameworks

When considering the use of a JavaScript framework for my project, I carefully weighed the advantages and drawbacks before making a decision. Ultimately, I chose not to use a JS framework for several reasons:

First and foremost, opting for a low-level approach without a framework allows me to have a higher level of control over the project. By working directly with JavaScript, I have the freedom to customize and fine-tune every aspect of the code according to my specific requirements. This level of control is especially important when dealing with unique or complex functionality that may not align with the conventions and limitations of a framework.

Another factor that influenced my decision is the time it takes to compile and set up a framework project. Frameworks come with their own build processes, which can add an additional layer of complexity and overhead to the development workflow. By eschewing a framework, I can streamline the development process and avoid potential delays caused by compiling and setting up the framework.

Moreover, using a JavaScript framework often introduces a multitude of dependencies. These dependencies can increase the project's bundle size and reduce its portability. Having too many dependencies can complicate the deployment process and make it harder to ensure consistent functionality across different environments. By keeping the project free from unnecessary dependencies, I can maintain a leaner and more portable solution.

In summary, this should provide greater control, faster development without compilation overhead, and a reduced reliance on external dependencies.

4.2.3. No server?

I decided to not host a server for the output contents even though the output content is a kind of web-site. Here are the explanations as to why this should result in the solution being simple to use and implement, portable, fast, reliable, and less dependable on external libraries utilities:

This project primarily deals with static content, so it does not involve any complex operations that would need a server, like AJAX requests, form submissions, or database interactions. Because of this, we can bypass the usual route of setting up a (local) server and serve the static content directly from the local disk (just open the output files in the browser).

Using this method makes it much simpler, both for the end-user and the developer. For the end-user, the content is delivered swiftly and smoothly, without the need for any additional processing. There is no waiting for server responses, and even the slight delay that might come from local network communication is gone. The result is a quick, responsive experience that delivers the needed static content efficiently.

For the developer, this approach saves time and effort. There is no need to deal with the complexities of server setup and maintenance, and no need to worry about server-side scripting. This streamlines the development process, leaving more time to focus on creating high-quality content and improving user experience.

Another major benefit is the increased portability of the project. Because the project does not rely on a server, it is not tied to any specific location or machine. It can be easily moved, copied, or distributed, and will work the same way wherever it is opened, as long as there is a web browser available.

This approach also boosts performance. By removing the need for server-side

processing and network communication, we are cutting down on the time it takes for the content to reach the browser. This not only speeds up content delivery but also makes the project less dependent on network conditions, which can vary from place to place.

Finally, serving files directly from the local disk makes the project less dependent on external factors. There is no need for Node.js, or other server-side technologies, and no need for additional packages that might add complexity and potential points of failure to the project. This minimizes the risk of dependencies causing problems and makes the project simpler and more robust. Also, one of the most important requirements for this project includes it not being dependant on external dependencies as much as possible.

In conclusion, serving static content directly from the local disk is an efficient and straightforward solution for this project. It offers benefits: simplicity, portability, improved performance, reduced dependency on external technologies, making it a good fit for this project's needs.

4.2.4. HTML

This is a brief recap of the "Why HTML?" section in the Literature Review chapter.

I chose to work with HTML for this project for a variety of reasons, each highlighting the strengths and benefits of this widely used markup language.

Firstly, HTML is ubiquitous. It is the standard language for structuring and presenting content on the web. By using HTML, I ensure compatibility across a wide range of devices and platforms, making the project accessible to a larger audience.

HTML's flexibility is another key factor. It provides a versatile and adaptable

framework for organizing and displaying content. With its wide array of tags and attributes, I have the freedom to structure the project in a way that best suits the needs of the project. This flexibility allows me to create visually appealing and engaging user interfaces.

Interactivity is a crucial aspect of modern web development, and HTML provides the foundation for building interactive elements. By incorporating JavaScript and CSS into HTML, I can create dynamic and responsive features that enhance user engagement and interactivity. From form validation to interactive menus and animations, HTML empowers me to bring the project to life.

HTML's extensibility is also noteworthy: through the use of custom attributes and data attributes, one can extend the functionality of HTML elements and create custom behaviors. This allows me to tailor the project to specific requirements and incorporate additional functionality beyond the standard HTML features.

Lastly, HTML's compatibility with various web technologies is a significant advantage. It seamlessly integrates with CSS for styling and layout, JavaScript for interactivity and dynamic behavior, and other web technologies like SVG and multimedia elements. This compatibility ensures a cohesive and integrated development experience.

In summary, the choice to use HTML for this project is driven by its ubiquity, flexibility, interactivity, extensibility, and compatibility. HTML provides a solid foundation for building web applications that are accessible, visually appealing, interactive, and compatible across different devices and platforms.

4.2.5. Output structure

Separating HTML files into distinct units offers several advantages.

It allows for more logical and structured code organization, promoting better

maintainability and collaboration. By nesting output files and folders within each other, we can represent the project more clearly. This modular approach simplifies development.

Overall, generating separate HTML files facilitates efficient code management and promotes a clean and organized development workflow. Also, should be easier to debug the output this way.

4.3. Best practices applied

Best practices I utilized building this project played a crucial role in creating a robust and efficient application that successfully meets the desired functional and non-functional requirements. By adhering to these best practices, I ensured that the application:

Maintains a high level of code quality. Making sure that the code is readable, modular, and well-organized code. This leads to easier maintenance, debugging, and future enhancements. It also promotes a better understanding of the codebase among team members, if the work is collaborative.

Encourages code reusability and modularity. Following the DRY principle and using modular architecture helps me minimize redundancy, making the codebase more manageable and easier to maintain. Additionally, reusable code components can save time and effort when implementing new features or making updates.

Ensures a timely and efficient development process. By adhering to best practices, I can deliver a high-quality application within the project's time and resource constraints. This should in turn result in a better product for the end user.

4.4. Project architecture and Design Decisions

The DartBoard application is a command-line interface (CLI) utility program designed to provide code analysis and visualization for projects written in the Dart programming language. The application is built using the Dart programming language and relies on several Dart libraries and tools to generate the HTML files from the source code.

The architecture of DartBoard follows the principles of Clean Architecture, which separates the code into layers based on their responsibilities and dependencies. The application consists of three main layers: the presentation layer, the domain layer, and the data layer. The presentation layer is responsible for generating the HTML files and providing the user interface for interacting with the code. The domain layer is responsible for performing the static analysis of the code, while the data layer is responsible for reading the code from the filesystem and caching analysis results for better performance.

The architecture relies on several libraries and tools, including the analyzer library for static analysis, the html library for generating HTML files, and the mustache library for templating. The architecture also includes several custom scripts and tools for optimizing performance and resource usage, including caching analysis results and using code generation to reduce the time required for HTML generation.

One of the key components of the architecture is the use of the Dart Analysis Server and the Dart AST library. The Analysis Server provides a powerful tool for static analysis of Dart code, while the AST library provides an efficient and flexible way to work with the code's abstract syntax tree. These tools are used extensively throughout the application to perform the static analysis of the code

and generate the HTML files.

Overall, the architecture of DartBoard is designed to provide a simple and lightweight solution for code analysis and visualization in the Dart programming language, with minimal overhead and efficient resource usage. By relying on Clean Architecture principles and several powerful Dart libraries and tools, the application can provide powerful and flexible code analysis and visualization capabilities, while remaining fast, efficient, and easy to use.

4.5. Key Features and Implementation Details

A project tree feature, syntax highlighting, and going to variable and function declarations are just a few of the many features that DartBoard offers. Programmers can better grasp complex codebases by using these characteristics, which also help programmers spot possible problems and encourage teamwork. DartBoard may be incorporated into continuous integration/deployment processes to keep up-to-date code analysis and also outputs an HTML document for simple sharing.

4.5.1. Block Scoping

This section focuses on the use of the Abstract Syntax Tree (AST) and the scanline algorithm for effective code analysis as I talk about how scoping is implemented in DartBoard. Block scoping and module scoping implementation were already addressed in earlier chapters. Here, I examine the general strategy for AST walking and the arrangement of data for efficient code analysis in an effort to provide a wider view on how scoping is accomplished in DartBoard.

The AST represents the syntactic structure of source code as a tree-like struc-

ture, with each node denoting a specific language construct, such as a function, class, or variable declaration. In order to better process and view codebases, DartBoard uses the AST to locate and examine scopes within the code.

Using a two-step process, DartBoard achieves effective scoping.

AST Walking: DartBoard moves across the AST, stopping at each node to collect pertinent data about scopes, including variable declarations, function parameters, and constructs linked to modules. It keeps track of the current scope and links variable usages with their corresponding declarations using a customized AST visitor. The basis for both block and module scope analysis is the AST walking mechanism.

Data organization: Following the AST traversal, DartBoard groups the data into various data structures, like maps and lists (Fig. 5), to facilitate quick retrieval and processing. Based on the files in the input project, the data is divided into various buckets of the maps responsible for each of the files in the input project.

```
List<AstNode> contextStack = [];
List<AstNode> blocksBuffer = [];
Map<String, List<AstNode>> blocks = {}; // blocks of particular file
Map<SimpleIdentifier, AstNode> jumpToDeclaration = {};
Map<AstNode, List<SimpleIdentifier>> jumpToUsages = {};
```

Fig. 26. Data organization.

The scanline approach, which was only briefly addressed earlier, is a linear traversal method used to process block scoping quickly. It is important to remember that the scanline technique is crucial to preserving the efficiency and scalability of DartBoard's scope analysis, even though the specifics of its implementation were covered in a previous chapter.

DartBoard uses the scanline method, a linear traversal approach, to handle block scoping quickly. Its main function is to label and enclose distinct sections of

code so that users may collapse or expand them as necessary for easier navigation and code reading. The scanline method is thoroughly explained in this part, with special emphasis placed on DartBoard's practical uses for it and its significance to the project.

For this algorithm specifically, I introduce two new data types: EventType and Event (Fig.7).

```
enum EventType { closes, opens }

class Event {
  int i;
  int x;
  EventType type;

  Event({this.i, this.x, this.type});
}
```

Fig. 27. Data types for Scanline.

In the addBlockCollapsers method I implement the scanline algorithm. The function accepts two parameters: a list of Ast Nodes (blocks) that represents the code blocks to be processed and a codeString that represents the code as a string.

The function goes through these three key actions: event creation, event sorting, and event processing:

1. The function produces two events for each block in the blocks list, one for when the block opens (i.e., the beginning of the block) and one for when it closes (i.e., the end of the block). A list named e contains these occurrences.

```
e.add(Event(
  i: i,
  x: blockStart,
  type: EventType.opens,
));
e.add(Event(
  i: i,
  x: blockEnd,
  type: EventType.closes,
));
```

Fig. 28. Adding opening and closing events.

2. Each event's x value is used to order the events in the e list. The x value designates the location in the code (token index in the code file string) where the event takes place. The events are sorted in the order they come in the code so that we can then iterate over all the events in e chronologically. This is done by the built-in STD .sort() method for lists, it takes a comparator to define the important feature I want to order the list elements according to (Fig. 9).

```
e.sort((Event a, Event b) => a.x - b.x);
```

Fig. 29. Event sorting, comparator.

3. Event processing. My code loops over the sorted list of events, adding tags to the associated code blocks that contain details like Block ID, CSS styles, and event handlers (oncontextmenu for right mouse click or onclick for left mouse click, for example). This is essentially where the HTML generation happens, however it is not going to the resulting HTML file just yet. These tags are then added to a tags data structure, sorted by e[i].x position.

```
for (int i = 0; i < e.length; i++) {
  final b = (i == 0
      ? codeString.substring(0, e[i].x)
      : codeString.substring(e[i - 1].x, e[i].x));
  if (e[i].type == EventType.opens) {
    String content = b;
    final id = "block-${e[i].i}";
    const classes = "block";
    final events = "oncontextmenu='collapse(`$id
`)'";
    final tag = "<span id='$id' class='$classes' $
events>";
    tags[currentFile].putIfAbsent(e[i].x, () =>
 []);
    tags[currentFile][e[i].x].add(tag);
  } else {
    String content = b;
    const tag = "</span>";
    tags[currentFile].putIfAbsent(e[i].x, () =>
 []);
    tags[currentFile][e[i].x].add(tag);
  }
}
```

Fig. 30. Creating tags for blocks.

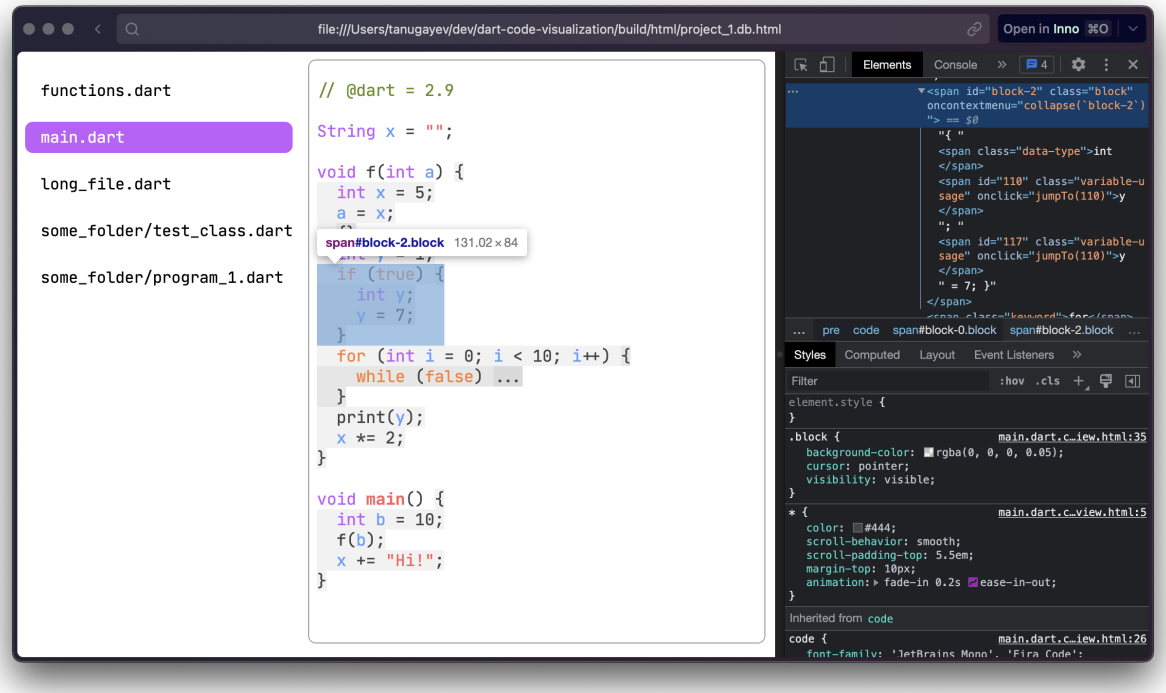

Fig. 31. Blocks in the output file.

Figure 30 illustrates how the scanline technique is applied to the processing and wrapping of code blocks in DartBoard. As a result of the algorithm's effective identification and wrapping of the blocks, users can easily collapse or extend the blocks for better code navigation and readability.

Wrapped code blocks are also shown visually in Fig. 31, and each block can be expanded or compressed by clicking. This functionality allows users to concentrate on particular sections of the code while obscuring less important ones, greatly improving the user experience when navigating complicated codebases. One can see how the scope for the 'while' scope is collapsed whereas the block for the if statement highlighted (block-2) is expanded. This is controlled by the event handler on the html tags that DartBoard puts scanlining the blocks and putting the tags in place.

Also, notice how the end user is able to collapse or extend nested blocks

independently of their parent blocks.

In summary, this section gives a comprehensive overview of DartBoard's approach to scoping, emphasizing the application of AST walking and data organizing techniques. DartBoard's scanline algorithm and these techniques work together to efficiently analyse complex input codebases. The algorithm (and the methods used in this section) is connected to the HTML Generation chapter as well, as it describes how to put the HTML tags in the right places in the template facilitating the interactivity of the resulting static HTML page.

4.5.2. HTML Generation

In this thesis, I present an HTML generation process for Dart code, which aims to improve code readability and facilitate navigation through an interactive and visually enhanced representation. This process incorporates other features as well: such as syntax highlighting, tooltips for documentation, code folding, and declaration bindings for variable usage. The HTML generation process consists of several components, including parsing and analyzing Dart code, processing the code through an HTML generation pipeline, and creating the final HTML output. These components interact through a very important step of pipelining to produce an HTML representation of the Dart code, which incorporates the additional features mentioned above.

The HTML generation pipeline is one of the most important and crucial parts of the process, responsible for transforming the Dart code to incorporate enhanced features. It is designed with maintainability, flexibility, and extensibility in mind. One of its key strengths lies in the layering of features in a smart and efficient manner. Each step in the pipeline focuses on a specific enhancement, and the order in which these enhancements are applied generally does not matter. This modular

approach allows for easy adjustments and improvements without impacting the overall pipeline and/or the end result.

Maintainability. The pipeline's modular design promotes maintainability, as each enhancement is implemented separately from the others. This separation of concerns ensures that changes to one feature do not inadvertently affect others, making it easier to maintain and update the pipeline over time. Additionally, the pipeline uses a consistent approach for adding HTML tags and CSS classes across all steps, streamlining the development and maintenance process.

Flexibility. The pipeline's flexibility comes from its ability to handle changes in the order of the enhancements or the addition of new features without significant alterations. Since the enhancements are applied sequentially and independently, the order can be easily changed or customized to suit specific needs or preferences. This flexibility means that the pipeline can be tailored to different requirements without requiring substantial code modifications.

Extensibility. The extensible nature of the pipeline allows developers to easily introduce new enhancements or modify existing ones. By following the same modular approach, new features can be incorporated into the pipeline without disrupting its core functionality. This extensibility means that the pipeline can continue to grow and adapt to new requirements, ensuring that it remains relevant and effective in enhancing Dart code.

Smart layering. The pipeline's smart layering of features enables each enhancement to be applied independently, without affecting the others. This layering approach ensures that the enhancements are applied in a consistent and efficient manner, resulting in a clean and organized final output. The layering of features also allows for easy troubleshooting and debugging, as issues can be isolated to specific steps in the pipeline.

This pipeline is executed for each file in each directory of the input project. It consists of four main steps: adding block collapsers for code folding, adding declaration bindings for variable usage, adding documentation tooltips for comments, and adding simple syntax highlighting. Each step plays a critical role in creating an interactive HTML file.

Let us consider an example of a step in this pipeline that is already in use in DartBoard: `addDeclarationBinding` (Fig. 12). It is located in `var_binding.dart` and is responsible for converting raw code string into an html markup that is interactable.

```dart
// @dart=2.9

part of '../../../main.dart';

String _wrapUsage(String usage, int uPos, int dPos) {
  const classes = "class='variable-usage'";
  return "<span $classes id='$uPos' onclick='jumpTo($dPos)'>$
usage</span>";
}

void addDeclarationBinding(String codeString, List<
SimpleIdentifier> usages) {
  for (int i = 0; i < usages.length; i++) {
    final declarationPos = getRootDeclaration(usages[i]).offset;

    const classes = "class='variable-usage'";
    final events = "onclick='jumpTo($declarationPos)'";
    final usagePos = usages[i].offset;

    final tag1 = "<span id='$usagePos' $classes $events>";
    const tag2 = "</span>";

    tags[currentFile].putIfAbsent(usagePos, () => []);
    tags[currentFile].putIfAbsent(usagePos + usages[i].length,
 () => []);

    tags[currentFile][usagePos].add(tag1);
    tags[currentFile][usagePos + usages[i].length].add(tag2);
  }
}
```

Fig. 32. Add declaration binding step.

The example pipeline step provided (Fig. 32) demonstrates how to add declaration binding to the HTML generation process. This is how simple it is to add a new step/feature to the existing pipeline: Let us analyze the structure and logic of the existing step. The given example consists of two main parts:

1. A helper function (`_wrapUsage`)

2. The main function that implements the pipeline step (`addDeclarationBinding`)

One can create a new pipeline step by following these steps:

1. Identify the functionality you want to add or modify in the HTML output. When I design a new feature, I am trying to think whether there already exists a pipeline step that I can add this new functionality or extend on the existing ones and only create a new step if it is indeed something of a different concern and I need to separate the logic.

2. Implement the main function for the pipeline step. Create a new function that will apply the desired changes to the code string. This function should take the code string and any other relevant input (e.g., a list of nodes, usages, or blocks) as its arguments. It should then go over the entities one wants to work with (in the example, the `addDeclarationBinding` function iterates through the list of usage identifiers) and then add HTML tags to the `tags` mapping that I use later, in the second part of executing the pipeline. In the example, the `tags` map is updated with the new HTML tags that need to be added to the code string.

3. Update the pipeline execution: Finally, incorporate the new pipeline step into the main pipeline execution function (`codeviewPipeline()`). This will ensure that your new step is applied during the HTML generation process.

4. Then the `executePipeline()` is called inside the `codeviewPipeline()` function as the last step, working out the tags placements in the resulting HTML file.

The `codeviewPipeline()` function contains the pipeline steps and ends with a call to `executePipeline()` on return to form the overall resulting HTML string and, subsequently, the file.

The execution part is the one where we transform the `tags` mapping that gives us the information as to where the tags have to be placed in the code string to form the correct structure of an HTML file.

```
String executePipeline(String codeString) {
  // sort tag keys
  final keys = tags[currentFile].keys.toList();
  keys.sort();
  String newCode = codeString;
  // iterate over keys from end to beginning
  for (int i = keys.length - 1; i >= 0; i--) {
    final pos = keys[i];
    final ts = tags[currentFile][pos];
    // if class doc then print
    for (int j = 0; j < ts.length; j++) {
      final t = ts[j];
      newCode = newCode.substring(0, pos) + t + newCode.
substring(pos);
    }
  }
  return newCode;
}
```

Fig. 33. Pipeline execution step.

Fig. 33. shows how the process of placing these HTML tags is organized:

1. `tags` is defined as follows: `Map<String, Map<int, List<String>>> tags`, where the key of the outer-most map is the file where these tags should be put. `tags[filePath]` is a mapping that stores all the tags at each position in the file (`filePath`).

   For example, we may have something like this:

`<span><span>x</span></span>`, then the position of x will be the key for the inner-most map giving this list: `["<span>", "<span>"]` and the list for `x.position + 1` will return this list: `["</span>", "</span>"]`.

Note, the tags may be different from `spans` and they might include more information. In the real pipeline steps, that I have in the application, the tags might include attributes like `id`, `class`, `data-content`, and/or event listeners – see Fig. 34.

2. It sorts the tags by position so as to then consequently iterate over them.

3. It iterates through all the keys of the `tags` mapping in the reverse order. I need it to be in reverse order because otherwise I would need to readjust the indexes of each tag position in the mapping by the number of characters in the string I am adding for each tag. For example: if I had a string like "abacaba" and wanted to insert "XX" in position 2, then the string will look like "abXXacaba" which pushes all the characters to the right of "XX" by two, so then if I want to add "YY" on position 3, this position is no longer valid as the string has changed, so I need to recalculate it for each of the positions in the `tags` mapping accordingly. To avoid it, I go backwards and eliminates the problem altogether.

4. At each iteration of the algorithm, I take the list of the tags that need to be placed at this particular place (`pos`), and iterate over this list, inserting each of these tags at that position (`pos`). This loop is executed forwards which ensures the correct order of tag placement at the same position.

5. At the last step the function returns the modified string of HTML markup.

6. It is then processed further to form the complete HTML file that is interactable and executable.

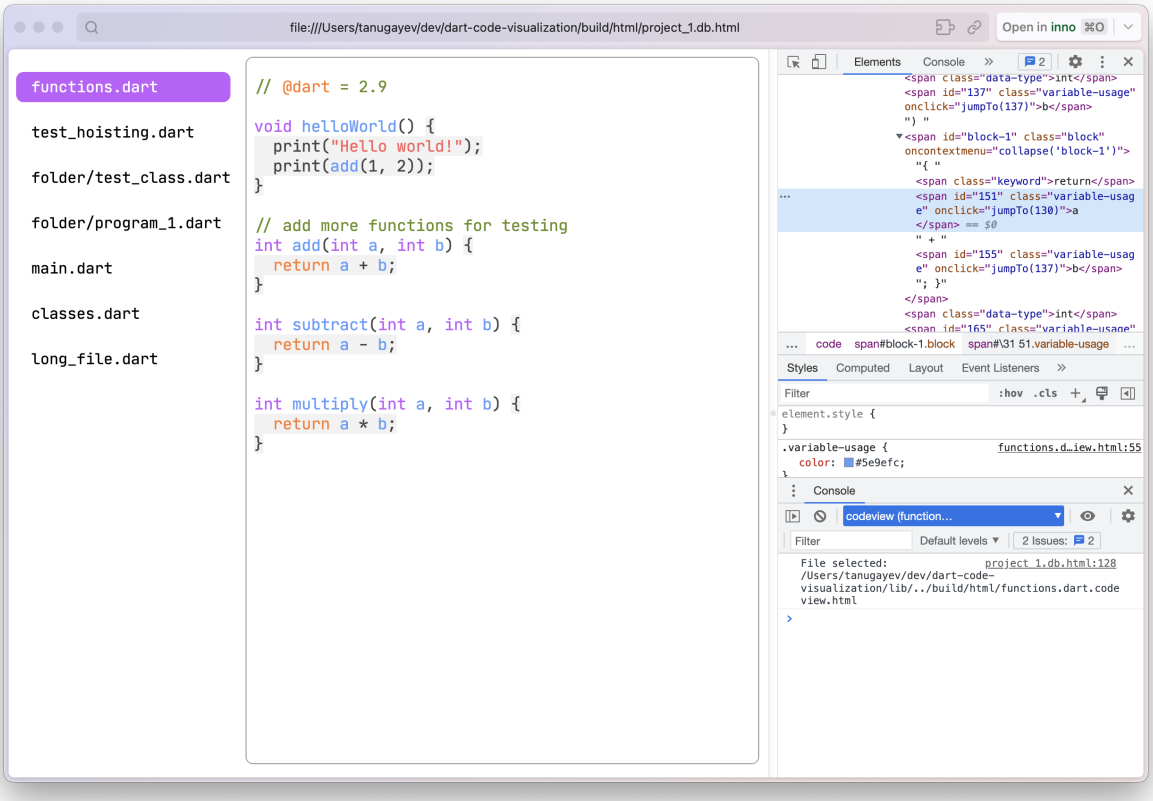

Fig. 34. An example of a tag generated.

4.5.3. Syntax Highlighting

Syntax highlighting is a feature that displays source code in different colors and fonts according to the category of terms. This feature is ubiquitous in most modern text editors and IDEs, and for a good reason. It can significantly improve the readability and understandability of code, especially in complex or large codebases.

In the context of DartBoard, implementing syntax highlighting for Dart code would involve:

1. Color-coding Different parts of the code are color-coded based on their semantic role in the programming language. For instance, keywords could be colored in blue, constants or literals in green, strings in red, etc. This color-coding enhances the visual perception of code, helping developers to quickly recognize the syntactical structure of the codebase.

The simpler color coding is implemented through substitution by regex. This means I go over the source code in search of words from the list of keywords in the Dart language and substitute it with itself but wrapped with a corresponding tag. The more interested reader can look at the code in more details in Fig. 35.

```
void addSimpleSyntaxHighlighting(String codeString) {
  for (RegExp regExp in SyntaxHighlighting.codePatterns.keys) {
    List<RegExpMatch> matches = regExp.allMatches(codeString).toList();
    final color = SyntaxHighlighting.codePatterns[regExp];
    for (int i = matches.length - 1; i >= 0; i--) {
      final classes = "class='$color'";

      final tag1 = "<span $classes>";
      const tag2 = "</span>";

      tags[currentFile].putIfAbsent(matches[i].start, () => []);
      tags[currentFile].putIfAbsent(matches[i].end, () => []);

      tags[currentFile][matches[i].start].add(tag1);
      tags[currentFile][matches[i].end].add(tag2);
    }
  }
}
```

Fig. 35. Simple syntax highlighting pipeline step.

This also includes more complex color-coding problems that I needed to tackle: for example, there is syntax highlighting for entities that the user can click to jump and this is determined using the AST analysis that I do prior to generating the actual HTML. This is implemented as a part of the corresponding steps in the pipeline.

Another example comes from the documentation and annotations pipeline step. It needs to be color coded and styled accordingly as well: when a part of code is considered an annotation, it should be highlighted as a separate entity according to the theme styling defined in the predefined css template: see Fig. 36.

```
3  /**
4   * Person describes a person,
5   * wit a name and an age,
6   * and can introduce themselves.
7   */
8  class Person {
9    String name;
10   int age;
11
12   Person(this.name, this.age);
```

Fig. 36. Annotation highlighting.

2. Font variations Apart from color, variations in font style or weight (like bold, italic) could be used to differentiate between different parts of the code. For example, class names could be bold, comments could be italicized, etc. This provides an additional layer of differentiation for easier code comprehension. I do not use it in DartBoard but it could become useful in the future.

3. Consistent scheme The color and font scheme should be consistent across different files and projects. This ensures that the developer doesn't have to re-learn the color-coding each time they switch projects.

This is why I have chosen the colors that go well together according to the Color Theory (color theory research?).

Note, that the order of processing regexes matters and it may be an issue to get it right. In the code snippet in Fig. 37, one can see how these regexes map to the entity class they represent and this will then map in the css of the generated html file to style it properly: assign colors, change font, etc.

```
static get codePatterns => {
      RegExp(r"\bmain\b(?<!\/\/.*)"): "main",
      RegExp(r'\".*\"(?<!\/\/.*)'): "quote",
      RegExp(r"\/\/.*"): "comment",
      RegExp(r"\/\*[\s\S]*?\*\/"): "comment",
      RegExp(r"@[A-Za-z]+\b|(?<!@)\b(" +
          keywords.join("|") +
          r")\b(?<!\/\/.*)"): "keyword",
      RegExp(r"\b(" + dataTypes.join("|") + r")\b(?<!\/\/.*)"): "data-type",
    };
```

Fig. 37. The mapping of regex to css classes for syntax highlighting.

It is important to note that syntax highlighting can be a non-trivial task, particularly for complex languages or large codebases. It requires careful handling of edge cases (like multi-line comments or string literals) and performance optimization to ensure that the highlighting doesn't slow down the rendering of the code. Moreover, regular expressions, while powerful, can also be complex and error-prone, so they need to be used judiciously and tested thoroughly.

4.5.4. Line Numbers

This feature adds an integral part of most text editors and IDEs - line numbers - to the code displayed in the DartBoard user interface. When viewing source code, each line will be preceded by a number representing its position in the file, starting with 1 and increasing incrementally for each subsequent line.

Having line numbers provides several benefits:

1. Easy navigation.

   Line numbers can help users quickly locate specific parts of the code. They can directly go to a particular line by typing its number, especially useful in large files.

2. Efficient debugging.

   When error messages include line numbers, as they often do, developers can quickly navigate to the problematic line of code.

3. Code reference.

   Line numbers make it easier to discuss specific parts of the code. When collaborating or seeking help, developers can simply refer to the line number instead of copying and pasting the code or describing its location.

4. Readability.

   Line numbers can improve the readability of the code, making it easier to track the start and end of code blocks, especially in languages like Dart that use indentation to mark these blocks.

The line numbering feature can also include capabilities like highlighting the current line number or range of line numbers to further aid in code navigation and readability.

However mostly, these points comprise the use scenarios in the scope of Dart-Board.

Implementation The most suitable approach to implementing this line numbers feature was to do it after the generation itself. Before I integrate the dynamic part to the layout of the ‘codeview’ html template, I add markup to support the introduction of line numbers line as a separate html element (see Fig. 38.).

```
<div id="code-container">
    <div id="line-numbers-half"></div>
    <div id="code-half">
<!-- <pre><code>$codeString</code></pre> -->
```

Fig. 38. Codeview template.

I then add the necessary JavaScript in the same template above so that when the generated html document loads, it performs something to populate the line numbers div element in the html template dynamically.

Namely, I have an `addLineNumbering()` function in the body of the template that is launched on window load (Refer to Fig. 39.). This function is responsible for finding the element of the code section that contains the actual code in the current file: i.e., the source code of the current file the user is looking at at the moment. It then takes that code and figures how many lines of code there is and then creates the element tags that it then puts inside the Line Numbers stripe to the left of the actual codeview, right beside the code itself.

```
//////////////////////////
// Window Setup

window.onload = () => {
    originalCodeSnapshot = document.getElementById("original-code-section").innerText;
    createTooltip();
    createUsagesListContainer();
    addLineNumbering();
}


//////////////////////////
// Line Numbering

function addLineNumbering() {
    // get code element
    let codeElement = document.getElementById("code-section");
    // split code into lines by \n as well as by <br/>
    let lines = codeElement.innerHTML.split(/<br>|<br\/>|<br \/>|\n/);
    console.log("lines: ", lines);
    // for each line, add a line number to line-numbers-half
    let lineNumbersHalf = document.getElementById("line-numbers-half");
    for (let i = 0; i < lines.length; i++) {
        let lineNumber = document.createElement("div");
        lineNumber.innerHTML = i + 1;
        lineNumber.classList.add("line-number");
        // position it at the top of the line
        lineNumber.style.position = "absolute";
        lineNumber.style.top = (i * 20) + "px";
        // add it to line-numbers-half
        lineNumbersHalf.appendChild(lineNumber);
    }
}
```

Fig. 39. Line numbering div population.

Because I have the line numbers defined post-generation, I need to make sure I align the line stripe exactly so that I have the lines coincide with the actual code. This is done through careful CSS and JS manipulations with pixel heights (Fig. 39). This is acceptable to base the aligning on pixel sizes because we have the same font size and a monospace font.

4.5.5. Class Documentation

This feature consists of several steps:

- AST analysis.

  Find class nodes: in the `ClassInfoVisitor`, which is a recursive AST visitor, I visit all the class nodes in the AST in the current file for all files in the input project.

- Then, I extract meta-data about the class: fields, methods, whether the class is abstract, etc (for full list of information I gather, see Listing 4.1). The process itself is shown in detail in Listing 4.2.

- After I extract the meta-data, I store it in a `ClassInfo` object (see Listing 4.1).

- Put tags in the main codeview pipeline. The process is the same for all the features, so the more interested reader may see the previous section for details on how this works.

  Except, this time I add additional information that I extract from the class nodes (`ClassInfo`) and incorporate it into the tags, so that it will be shown on hover in the output HTML. This is accomplished via storing it in the dictionary that maps a node by its unique identifier to its respective `ClassInfo` object (see Listing 4.3) and subsequently utilizing this information when forming the HTML. Using this mapping, in the `addDocumentationTooltip` function of the `documentation_tooltip.dart` file, I can pull this information from the mapping and form the HTML tags accordingly. Little part of the code for the whole process of formation the HTML tag is shown in Listing 4.4.

Listing 4.1: Class meta information.

```
// ...
ClassInfo({
    this.name,
    this.modifiers,
    this.extendedClass,
    this.implementedInterfaces,
    this.mixins,
    this.fieldNames,
    this.methodNames,
    this.constructorNames,
});
// ...
```

Listing 4.2: Gathering class meta information.

```
String extendedClass;
if (node.extendsClause != null) {
  extendedClass = node.extendsClause.superclass.name.name;
} else {
  extendedClass = '';
}

List<String> implementedInterfaces;
if (node.implementsClause != null) {
  implementedInterfaces = node.implementsClause.interfaces
      .map((interface) => interface.name.name)
      .toList();
} else {
  implementedInterfaces = [];
}
```

```
List<String> mixins;
if (node.withClause != null) {
  mixins =
      node.withClause.mixinTypes.map((mixin) => mixin.name.name).
        ↪ toList();
} else {
  mixins = [];
}
```

Listing 4.3: Class description mapping.

```
classDecription[node] = ClassInfo(
  name: className,
  modifiers: modifiers,
  extendedClass: extendedClass,
  implementedInterfaces: implementedInterfaces,
  mixins: mixins,
  fieldNames: fieldNames,
  methodNames: methodNames,
  constructorNames: constructorNames,
);
```

Listing 4.4: Forming class documentation tag.

```
// ...
docText += "Type: ${typeName.name.name}\n";
docText += separator;

if (node != null) {
  docText += classDecription[node].toString();
}
// ...
```

4.6. Testing and Validation

Several testing and validation techniques are used to guarantee DartBoard's quality. These tests emphasize functionality, compatibility, and performance to ensure that the application satisfies the necessary standards and offers a positive user experience. DartBoard's goal is to provide a robust and easy-to-use tool for code analysis and visualization in the Dart programming language. To that end, it conducts thorough testing.

This chapter concludes by highlighting the numerous facets of DartBoard's implementation, from the architectural design, major features, and testing techniques to the technological stack selection. DartBoard strives to offer a powerful and effective solution for code analysis and visualization in the Dart programming language by concentrating on these elements, making it a crucial tool for developers.

I tested the product manually: there are multiple test projects that test different functionality of the application. Some of these test one can see in Fig 40. The goal is to test out all functionality from all possible angles in order to find and eliminate inconsistencies or potential bugs, if any.

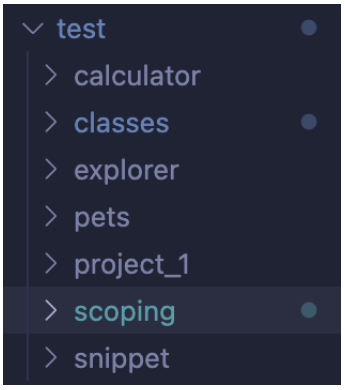

Fig. 40. Several test projects.

One of the most important features of this project and a non-functional requirement for the system is for the application to be cross-platform, which is what I have tested: the application works correctly on different machines, operating systems, and all the required browsers are supported, too. So, the system meets said non-functional requirement.

In future, to better the maintainability and potentially introduce new team members to the development process, I might need to add automated tests: unit tests, etc. This would ensure that I do not introduce any new bugs while working on new functionality for DartBoard.

Chapter 5

Conclusion

5.1. Application Value

The value of this project is manifold, primarily stemming from its potential to simplify comprehension of complex codebases through effective code visualization techniques. By representing large code structures in an easily digestible format, it can greatly reduce the cognitive load for developers, particularly those new to a project or non-developers trying to understand a software system.

In addition, this tool introduces a novel approach to structuring software projects. By bringing clarity to the organization of code and providing an overarching view of the project structure, it aids developers in better navigating and managing their codebase.

This project also contributes significantly to the area of Integrated Development Environments (IDEs). By incorporating innovative functionalities and capabilities, it enhances developers' productivity and code management efficiency.

A standout feature of this tool is its inherent portability and accessibility. The generated output, being platform-independent, can be easily shared among team

members or with management. This is particularly advantageous in professional environments where not everyone might have the specific programming language environment, such as Dart, installed on their computers. For instance, a supervisor wishing to review the progress of a Dart project doesn't need to install or understand Dart. They can review the visual output generated by this tool, which provides a comprehensive overview of the project, its structure, and its coding patterns. This not only fosters better communication within a team but also allows non-technical stakeholders to gain insights into the project's status and progress. This feature also facilitates remote collaboration, which is increasingly becoming the norm in many workplaces.

In essence, this project is a powerful tool that supports and promotes efficient code management, effective communication, and inclusive understanding of codebases in various work environments.

5.2. Application Purpose

Continuing from the previous section, the objective of this initiative is not only to elevate the productivity of software development but also to enrich the developer experience by introducing a dynamic instrument for code visualization. This tool has successfully managed to convert intricate lines of code into a more approachable format, enabling developers to comprehend and traverse through large-scale projects with much greater ease.

The project is designed with the developers' needs in mind, aiming to alleviate the burden that comes with handling extensive codebases. By introducing an accessible visual interface, it provides a clear and concise overview of the entire project, thereby reducing the time and effort spent on understanding and modify-

ing the code. This contributes greatly towards improving the overall code quality and maintaining a consistent and clean codebase.

Furthermore, the purpose of the project also extends to fostering better collaboration among team members. By providing a visual representation of code, team members can easily share and discuss different components and aspects of the project, regardless of their technical expertise. This inclusive approach broadens participation and facilitates a more efficient exchange of ideas, leading to more innovative solutions and improved problem-solving capabilities.

Finally, another key aspect of this project's purpose lies in its commitment to advancing the capabilities of Integrated Development Environments (IDEs). By integrating new functionalities that aid in code visualization, it enhances the programming environment, allowing developers to interact with their code in novel and intuitive ways. This not only improves the workflow and efficiency but also enriches the user experience, making software development more engaging and less daunting.

In conclusion, the purpose of this project goes beyond just offering a tool for code visualization - it is about improving the software development process, promoting efficient teamwork, and enhancing the programming environment. Through this project, we have taken a significant step towards making software development more approachable, efficient, and enjoyable.

5.3. Metrics and measurements

The success of this project was measured in terms of its usability and the impact it had on development efficiency. Preliminary feedback from users indicates that my tool has improved their ability to understand and navigate complex

codebases, suggesting that our metrics for success have been met.

5.3.1. Performance

The system performed well during testing, showing high performance in code visualization. Future improvements could include optimizing performance for larger codebases.

Several measurements have been conducted to benchmark the performance of the system on small projects, averaging around 6.5 seconds.

Individual measurements (that can be viewed in Listing 5.1) have been performed 10 times and averaged out to be about 6.5 seconds which is more than acceptable for our goals: this is fast enough to not detriment the user experience processing one input project.

More tests might be needed on larger projects that I do not have at the time of writing this paper, but it is enough to judge that the system is performant and stable.

Listing 5.1: Measure time to compile on test project 'pets'

```
% Compile time measurement for Pets test project:
$ dart main.dart pets
$ dart main.dart pets  6.72s user 0.62s system 131%
cpu 5.566 total

% Compile time measurement for Calculator test project:
$ dart main.dart calculator  6.54s user 0.62s system 159% cpu
   ↪ 4.499 total
```

5.3.2. Memory usage

The output .zip archives for the Pets and Calculator test projects come out to be 50KB each and do not take much space on the user's hard-drive.

This means it can be freely exchanged in the work environment even over a low-speed internet connection, which satisfies our requirements and solidly fulfills one of the use-cases: to be able to send the code to another team member in a development team for a quick overview of the project.

5.4. Code Distribution

The code developed for this project uses several non-trivial techniques for code parsing and visualization. It's been designed with modularity and scalability in mind, allowing for future extensions and improvements.

My code is freely available for further development and contribution by the wider software development community. I believe that open source is a powerful tool for collective improvement and innovation and encourage any feedback and contributions.

Here is the link to my repository on GitHub: `https://github.com/al1ych/dart-code-visualization`

5.5. Future work

As I continue to build and enhance my idea, I am faced with the exciting possibility and unlimited advancements that the future offers. My goal is to ensure this tool remains a useful resource for developers, aiding in streamlining and demystifying the software development process. This section provides readers with

a glimpse into the potential enhancements and advances for this project, giving them a sense of the direction I envision for the future.

The landscape of the software development business is dynamic and ever-evolving. To keep pace with the changing requirements and practices of software development, this tool too must adapt and advance. I foresee potential upgrades to not only refine the existing capabilities but also to introduce new functionalities. There are several areas that require further research and development, each possessing the potential to contribute to the efforts of providing a comprehensive tool for code visualization. These improvements could range from incorporating additional programming languages, improving visualization capabilities, enhancing user interface design, and integrating with different platforms and programs.

Despite the significant progress I have made on this project, I am aware that I am at the beginning of an exciting journey. The field of code visualization offers extensive scope for exploration and experimentation. I eagerly anticipate the advancements and discoveries that future work will bring. The constant evolution of this project reflects my commitment to creating a tool that remains relevant, practical, and innovative as software development processes continue to advance.

In this section I do not focus on the implemented features as the full list of features implemented and unimplemented is available in the Requirements chapter of this paper. Rather, I will focus on the features that are yet to be implemented in DartBoard.

5.5.1. Unimplemented

In this subsection I describe the features that have been planned for but I did not complete as of the day of writing this paper.

The following list presents the aforementioned features and potential ap-

proaches to implement them:

1. **Import zip project.** This could be achieved by using the dart:io library's ZipDecoder class for zip file extraction.

2. **Import GitHub repo.** Integration with the GitHub API would be required for this feature. A GitHub API client in Dart could fetch and parse the repository data.

3. **Show Static analysis results on 'dart analyze'.** This can be accomplished by running the Dart analysis server in the background and accessing real-time analysis data.

4. **Highlight the second parenthesis/bracket.** Dart's syntax parsing library could be used to find matching parentheses and brackets for highlighting.

5. **Scrollable minified code tab to the right of the screen.** Integrating a code minimap into the DartBoard code editor could be achieved with a package like flutter_codemirror or using the CodeMirror library with dart:html.

6. **Check code for errors.** A basic code validation feature could be added by extending the Dart syntax parsing library to highlight syntax errors.

7. **Visualize dependency graph.** Dart's analyzer package could be used to generate a model of the code's dependency graph.

8. **Visualize inheritance tree.** The analyzer package could also be used to build a model of the class inheritance hierarchy in the code.

9. **Pipelines.** DartBoard could be packaged as a command line tool, making it easy to integrate into CI/CD pipelines.

10. **Rename var/function (refactor).** The dart_language_server package provides functionality for renaming symbols across a codebase, which could be integrated into DartBoard.

11. **Replace by regex expression.** Dart's RegExp class could be utilized to enable search and replace operations using regular expressions.

12. **Gray out unused vars and functions.** Dart's analyzer package could be used to identify unused code symbols, which could then be visually de-emphasized in the DartBoard editor.

13. **Gray out inaccessible code.** Dart's analyzer package could also be used to identify unreachable code blocks, which could then be visually de-emphasized in the DartBoard editor.

Below I overview the features and investigations needed that have not been accounted for in the beginning but arose while developing DartBoard.

5.5.1.1. Features to be implemented

1. **Github Action.** DartBoard could be packaged as a GitHub Action to automate the generation of the HTML as part of the CI/CD pipeline.

2. **Integration with pub.dev.** Packaging DartBoard as a Dart package and publishing it on pub.dev would make it easy for developers to install and use.

3. **Project tree as an actual tree & Project tree folder collapsing.** The flutter_treeview package could be used to create a visual, collapsible tree representation of the project.

4. **Flutter screen and widget graphical hierarchy.** Flutter's Inspector package could be utilized to generate a visual representation of the widget tree.

5. **Support for external plugins.** Implementing a plugin architecture in DartBoard would be required. Defining a Plugin interface to be implemented with Dart dynamic libraries might be an approach.

6. **Creation of own plugins.** This would need detailed documentation on creating plugins for DartBoard and providing an API for interacting with the DartBoard tool.

7. **Check linting errors.** Dart's linter package could be integrated into DartBoard to automatically check code against Dart style and best practice guidelines.

8. **Suggest linting fixes.** Integrating a library like Dart Fix could provide automatic fixes for common linting errors.

5.5.1.2. Opportunities to improve the software quality

In the Testing and Validation section of the previous chapter, I have discussed the tests and benchmarks I have been conducting to determine the stability and overall performance of the system.

Yet, there still are opportunities for further testing and improving the quality of the software. Here are some elaborations on the parts you requested:

Automatic Testing: Unit Tests. Automatic testing is crucial for maintaining code quality, ensuring code correctness, and preventing regressions as the project evolves. In the context of DartBoard, implementing a comprehensive suite of unit tests can be beneficial. Unit tests isolate parts of code and check if individual units

(like functions or methods) behave as expected. The Dart ‘test‘ package provides a powerful, flexible way to write these tests. By writing unit tests for different parts of DartBoard - such as parsers, code analyzers, or visualization functions - I can ensure that changes and additions to the codebase don’t introduce unexpected behavior or break existing functionality.

Tests on Larger Codebases. Testing DartBoard on larger codebases can uncover performance and scalability issues that may not be noticeable with smaller, simpler projects. These tests could involve importing and visualizing large Dart projects, perhaps even large open-source projects available on GitHub. Such tests would give valuable insights into how DartBoard performs under heavier loads, which would be critical if it is to be used in large-scale, enterprise-level software development.

More Real-life Codebase Testing. While synthetic tests and benchmarks are useful, there’s no substitute for testing with real-world code. This could involve getting feedback from real users, perhaps by releasing a beta version of DartBoard to a group of early adopters. Real-life testing could also involve using DartBoard in my own software development workflow. This way, I would experience first-hand any issues or inconveniences that users might face and get insights into how to improve the tool.

Bibliography cited

[1] G.-C. Roman and K. C. Cox, "A taxonomy of program visualization systems," *Computer*, vol. 26, no. 12, pp. 11–24, 1993.

[2] O. T. L. Bedu and F. Petrillo, "A tertiary systematic literature review on software visualization," in *2019 Working Conference on Software Visualization (VISSOFT)*, 2019.

[3] T. A. Ball and S. G. Eick, *Software visualization in the large*, `http://www.cs.kent.edu`, Accessed: 13-Nov-2022. Available: `http://www.cs.kentedu/Hmaletic/softvis/papers/BallEick1996.pdf`, 1996.

[4] P. Khaloo, M. Maghoumi, E. Taranta, and D. Bettner, "Code park: A new 3d code visualization tool," in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 2017.

[5] H. R. N. F. Nielson and C. Hankin, *Principles of program analysis*. Berlin: Springer, 2005.

[6] A. Aiken, "Introduction to set constraint-based program analysis," *Science of Computer Programming*, vol. 35, no. 2-3, pp. 79–111, 1999.

[7] F. Nielson and H. R. Nielson, "Type and effect systems," in *Lecture Notes in Computer Science*, 1999, pp. 114–136.

[8] A. G. Bardas, "Static code analysis," *J. Inf. Syst. Qpg & Manage.*, vol. 4, no. 2, pp. 99–107, 2010.

[9] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*, Vancouver, BC, Canada, Oct. 2004, pp. 132–136. DOI: 10.1145/1028664.1028706.

[10] P. Neron, A. AAN, E. Visser, and G. Wacholt, "A theory of name resolution," in *Programming Languages and Systems*, 2015, pp. 205–231.

[11] A. G. Bakhanov and M. V. Kuskov, *Go language program analysis: Portable visualization and navigation tool*, Thesis, 2022.

[12] X. F. Grau and M. I. S. Segura, *Desarrollo orientado a objetos con uml*, Thesis, 2011.

[13] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman, "Execution patterns in object-oriented visualization," in *Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998, pp. 219–234.

[14] R. Queirós, *Kaang: A restful api generator for the modern web*, Jul. 2018.

[15] *Postman*, https://www.postman.com/, Accessed: May 17, 2023, 2023.

[16] *Swagger open source and pro tools*, https://swagger.io/tools/swagger-ui/, Accessed: May 17, 2023, 2023.

[17] *Json (javascript object notation)*, https://www.json.org/json-en.html, Accessed: May 17, 2023, 2023.

[18] *The openapi specification*, https://www.openapis.org/, Accessed: May 17, 2023, 2023.

[19] *Markdown*, https://daringfireball.net/projects/markdown/, Accessed: May 17, 2023, 2023.

[20] T. Berners-Lee, *Html: A representation of textual information and metainformation for retrieval and interchange*, World Wide Web Consortium (W3C), 1995.

[21] D. Raggett, A. L. Hors, and I. Jacobs, *Html 4.01 specification*, World Wide Web Consortium (W3C), 1999.

[22] D. Flanagan, *JavaScript: The Definitive Guide*. O'Reilly Media, 2020.

[23] D. Crockford, *HTML5: Up and Running*. O'Reilly Media, 2010.

[24] I. Hickson, *Html living standard*, Web Hypertext Application Technology Working Group (WHATWG), 2021.

[25] *Flutter packages*, https://pub.dev/flutter/packages, Accessed March 23, 2023.

[26] *Json_serializable*, https://pub.dev/packages/json_serializable, Accessed March 23, 2023.

[27] R. Nicoletti, *Freezed*, https://pub.dev/packages/freezed, Accessed March 23, 2023.

[28] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, 1996, ISBN: 0-201-84840-6.

[29] *Redoc: Openapi/swagger-generated api reference documentation*, https://redoc.ly/redoc, Accessed: May 17, 2023, 2023.