**Code analysis and visualization for the Dart programming language**

Timur Nugaev, Innopolis University

t.nugaev@innopolis.university

**Abstract**

**Background**

To properly organize the codebase for a large project, it is vital to constantly analyze and refactor the code, build and structure the project to maintain and scale it in the future. There are many methods software engineers utilize to achieve this goal: static code analysis, refactoring tools, visualization of dependency graphs, inheritance trees, UML diagrams, etc. Various software development environments combine the aforementioned toolbox and are essentially text editors that have the integration with the compiling system at their core: GitHub, JetBrains IntelliJ IDEA, etc.

**Objective, Novelty & Design approach**

This thesis proposes a different approach to the problem: performing static analysis and other procedures just once to generate a browser-based Integrated Development Environment (IDE) that can be accessed on any computer with a browser. The generated IDE ensures total compliance with modern standards for code analysis, functionality, and design while providing full immunity to time-consuming chores like IDE installation, project codebase and dependencies, and project indexing. The methodology includes the development of a tool, DartBoard, that implements this approach, and the evaluation focuses on its effectiveness and usability.

**Conclusion**

Many of the program visualization tools available at this time lack portability and/or user experience. They demand time to install and sometimes have a very steep learning curve. The product of this research eliminates many of these problems, leaning more towards a source code viewing platform rather than a stand-alone code editing application, providing a lightweight, platform-independent, and compact solution to analyze software projects. The thesis further discusses the methodology, evaluation, and results, demonstrating the benefits and potential of this novel approach.

1. **Introduction**

**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.

**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 happening 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 in the future.

So, this is why people started to visualize their code. There are many ways to visually represent the 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.

**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.

**Applicability**

The final product (DartBoard) 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 the 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.

**Thesis context**

The following chapters will describe the methodology used to develop DartBoard, the evaluation of its effectiveness and usability.

1. **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) and scoping calculus theory [3] and explore different visualization techniques and tools available at the moment [4].

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 [1], 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 [1] 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 [5], 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 as possible.

**2.2 Source Code Analysis**

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

These analyzers are utility programs that optimize the program source code and find potential bugs and vulnerabilities. These utilities automatically traverse the codebase in search of fundamental non-runtime errors or provide help writing better code. The former, in the case of Dart, is a fully functional utility maintained by the Google team 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 the auto-correction problem. For example, if a programmer does not like the enforced curly braces in our code, our tool can be utilized 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 [6].

In the book “Principles of program analysis,” Nielson *et al.* [6] 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 [7]. 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 [9] 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” [9].

**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 compile-time [10]. 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 [10].

**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 [16] 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/Flutter 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 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.* [11] 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 [11].

One other important process that the authors have formalized is “rename refactoring” [11]. It is about the refactoring process that will help us with code and markup generation in future chapters.

Scoping analysis will also allow 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.

* + 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 [3]. 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 [2].
* Semantic: A deeper analysis that considers the meaning and context of the entities within the source code. It includes name resolutions, scopes, and other contextual information [2].

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 [2] 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.

**2.3.1 Program text visualization**

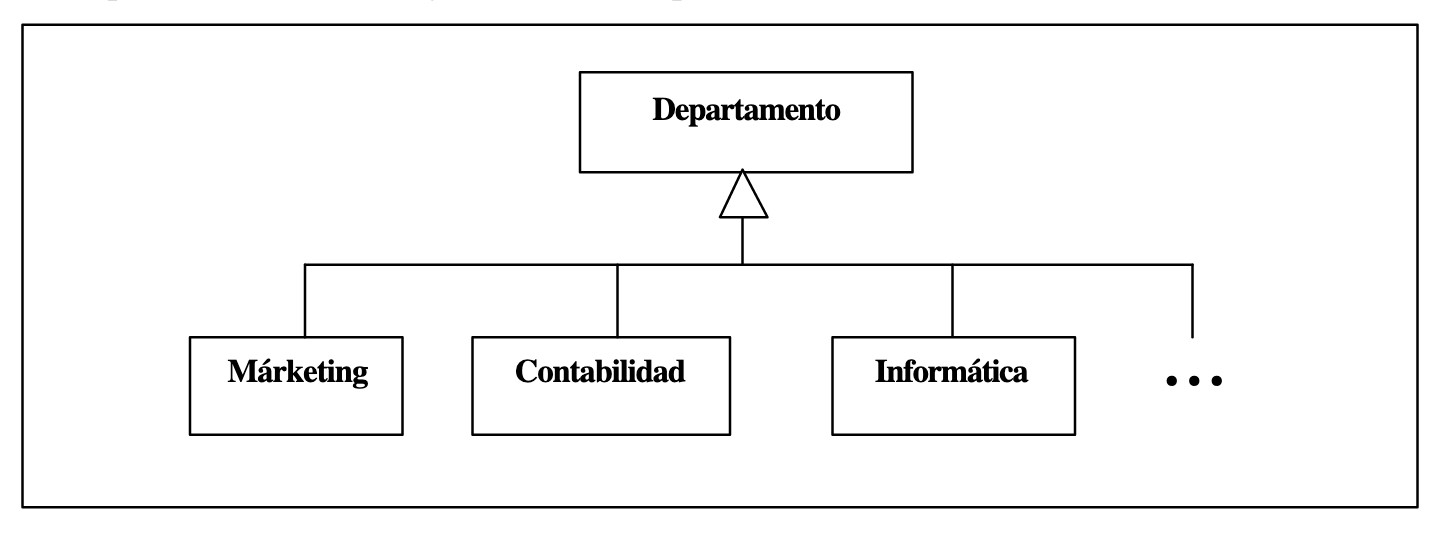
Program text visualization is concerned with the textual representation of source code [2]. It encompasses features like syntax highlighting, code navigation tools, highlights, and block collapsing that aim to simplify the codebase and make it more comfortable to work with.

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

**2.3.2 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 [2]. 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” [13].



*Fig. 1*

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 [18].

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 [17].

**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 [15]. Common use cases for documentation generators include automatically generating API documentation for back-end server applications, such as those produced by Postman, Sphynx, and Swagger. These generators take a generated JSON file of OpenAPI 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**

For this thesis, I am particularly interested in markup generation, which is closely related to API documentation generation. Markup generation enables us 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 [19].
* 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 [20].
* 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 [21].
* 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 [22].
* 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 [23].

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.

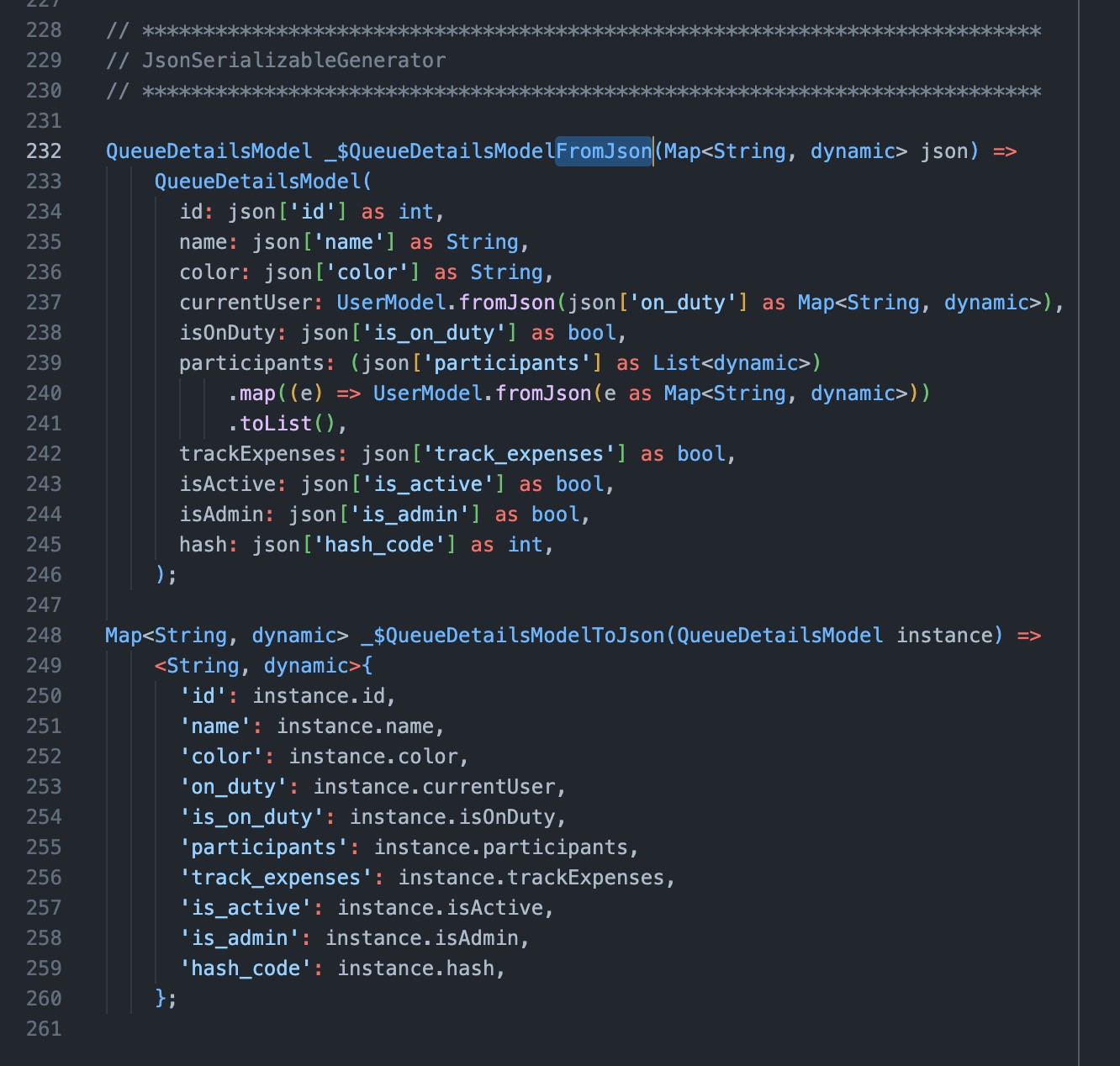
**2.4.4 Code generation**

Code generation is a common practice in software development projects, including those based on the Flutter framework, 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 [24].

Code generation typically relies on abstract syntax trees (ASTs) or Element Trees to analyze the source code and generate boilerplate 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 [25]. 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 code structures (see Fig. 2 and Fig. 3) [26].



*Fig. 2*



*Fig. 3*

In my case, I utilize the Scanline algorithm [27] 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 [27]. 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.

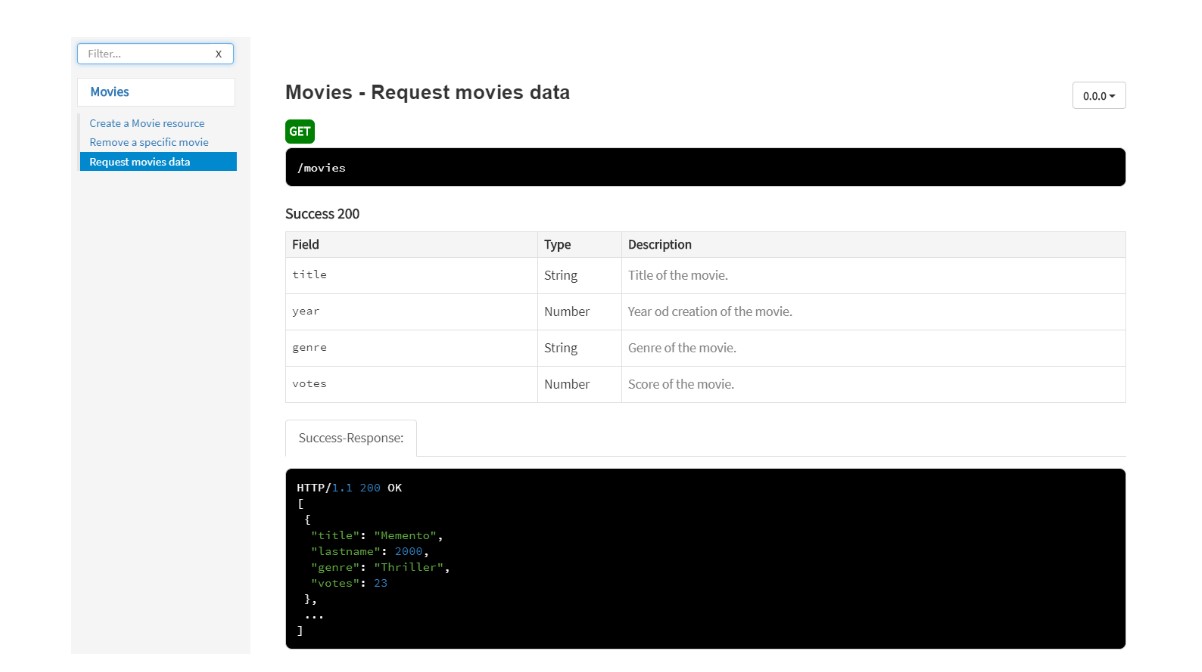
In the context of DartBoard, the scanline algorithm is used to detect the opening and closing of blocks in the code. The algorithm first creates a list of events, where each event corresponds to the starting or ending position of a block. These events are then sorted based on their positions in the code.

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 independently of the operating system in the browser. 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* [15].



*Fig. 4*

1. **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.
* The HTML should run on the last three major versions of Chrome, Edge, Firefox, Opera, and Safari.
* The HTML should run on MacOS, Windows, and Linux.
* DartBoard should have all the functionality of modern IDEs.
* DartBoard should have the look of a modern IDE design-wise.
* DartBoard should be able to deal with programs that contain a syntax error.
* DartBoard should be available as a command-line interface tool.

These non-functional requirements are essential for making 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.

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.

**High priority:**

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.

* 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.

* 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.
* Implement the ability to jump to variable and function declarations: This feature allows programmers to quickly jump to the location in the code where a variable or function is defined, making it easier to understand the context of that code.
* Show documentation on cursor hover: This feature provides programmers with a quick and easy way to view documentation for a particular function or class by simply hovering the cursor over it.
* Collapse block scopes: This feature helps to 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.
* Implement an upper tab bar to switch between tabs with source files: This feature allows programmers to easily switch between different source files within a project, making it easier to navigate between different parts of the codebase
* Add project tree functionality: This feature provides a hierarchical view of the project's source code, allowing programmers to easily navigate and understand the structure of the codebase.

**Medium priority:**

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 programmers who use 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.

* Theming: This feature allows programmers to customize the look and feel of DartBoard to their liking, making it more personalized and easier to work with.
* Search by regex expression: This feature enables programmers 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.
* Replace by regex expression: This feature allows programmers to perform global search and replace operations using regular expressions, which can help to quickly make changes to large sections of the codebase.
* Rename var/function (refactor): This feature enables programmers to quickly and easily rename variables and functions throughout the entire project, making it easier to maintain and update the codebase.
* Upload zip project: This feature allows programmers to upload a compressed zip file of their project to DartBoard, making it easier to analyze and visualize large projects.
* Upload via GitHub repo: This feature enables programmers to upload their project directly from their GitHub repository, streamlining the process of code analysis and visualization.
* Project tree folder collapsing: This feature allows programmers to collapse and expand folders within the project tree, making it easier to navigate large codebases with many files and directories.

**Low priority:**

While these features are not essential for the core functionality of DartBoard, they could provide useful additional functionality for programmers who use 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.

* Gray out vars and functions that are not used: This feature would highlight variables and functions in the code that are not being used, making it easier for programmers to identify and remove unnecessary code.
* Gray out parts of code that are inaccessible: This feature would highlight 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.
* 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.
* 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.
* Highlight the second parenthesis/bracket: This feature would highlight the corresponding closing bracket or parenthesis when the cursor is placed on an opening bracket or parenthesis, making it easier for programmers to match pairs of brackets in complex code.

1. **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 project architecture, design decisions, and best practices. Section 5.3 highlights the best practices used during development. Section 5.4 explores the key features and implementation details. Section 5.5 outlines the testing and validation methods employed to ensure the application's quality and reliability.

* 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.

* 1. **Trade-offs**

- Why no Templater?:

- low level => high level of control

- Why HTML in general?:

- ubiquitous

- Why not JS framework?:

- low level => high level of control

- time to compile for a framework

- more dependencies = less portatibility & bundle size

- Why generate/build separate HTML files:

- to work with code more logically

- to build structured HTML, nesting

- Why no server:

- to keep it simple

- to keep it portable

- to keep it fast

* 1. **Best practices**

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, the DartBoard development team ensured that the application:

Maintains a high level of code quality: Readable, modular, and well-organized code leads to easier maintenance, debugging, and future enhancements. It also promotes a better understanding of the codebase among team members, resulting in more efficient collaboration.

Enhances reliability: By employing Test-Driven Development (TDD) and continuous integration, the team can quickly identify and fix any issues, minimizing the likelihood of introducing bugs or regressions. This results in a more stable and reliable application for end-users.

Encourages code reusability and modularity: Following the DRY principle and using modular architecture helps 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.

Facilitates knowledge sharing and collaboration: Regular code reviews and discussions within the team lead to a collective understanding of the codebase and its design decisions. This knowledge sharing fosters collaboration and helps maintain a unified vision of the project.

Ensures a timely and efficient development process: By leveraging continuous integration and adhering to best practices, the development team can deliver a high-quality application within the project's time and resource constraints. This efficient development process ultimately results in a better end product for the users.

In summary, the implementation of these best practices has been instrumental in shaping DartBoard into a robust, efficient, and user-friendly code analysis and visualization tool, capable of meeting the desired functional and non-functional requirements while providing a seamless experience for developers working with the Dart programming language.

* 1. **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.

Sidenote: <not many architecture decisions made yet, will fill it more as I go with the project>

* 1. **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.

* + 1. **Block Scoping**

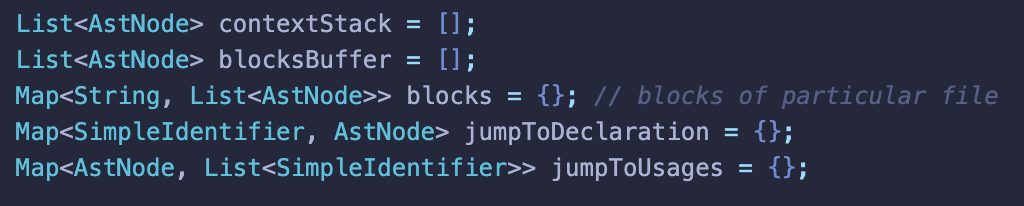
This section focuses on the use of the Abstract Syntax Tree (AST) and the scanline algorithm for effective code analysis as I talk about how scoping is implemented in DartBoard. Block scoping and module scoping implementation were already addressed in earlier chapters. Here, I examine the general strategy 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 structure, 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.



*Fig. 5. 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).



*Fig. 7. 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.

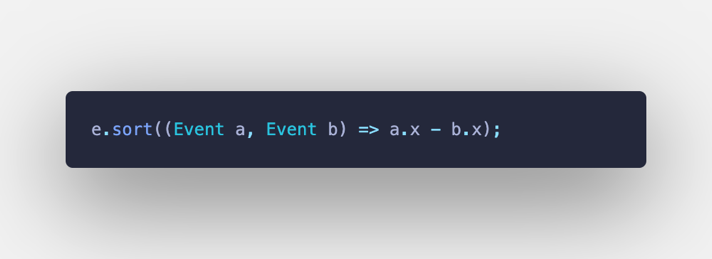
The function goes through these three key actions: event creation, event sorting, and event processing:

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.



*Fig. 8. Adding opening and closing events.*

1. 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).

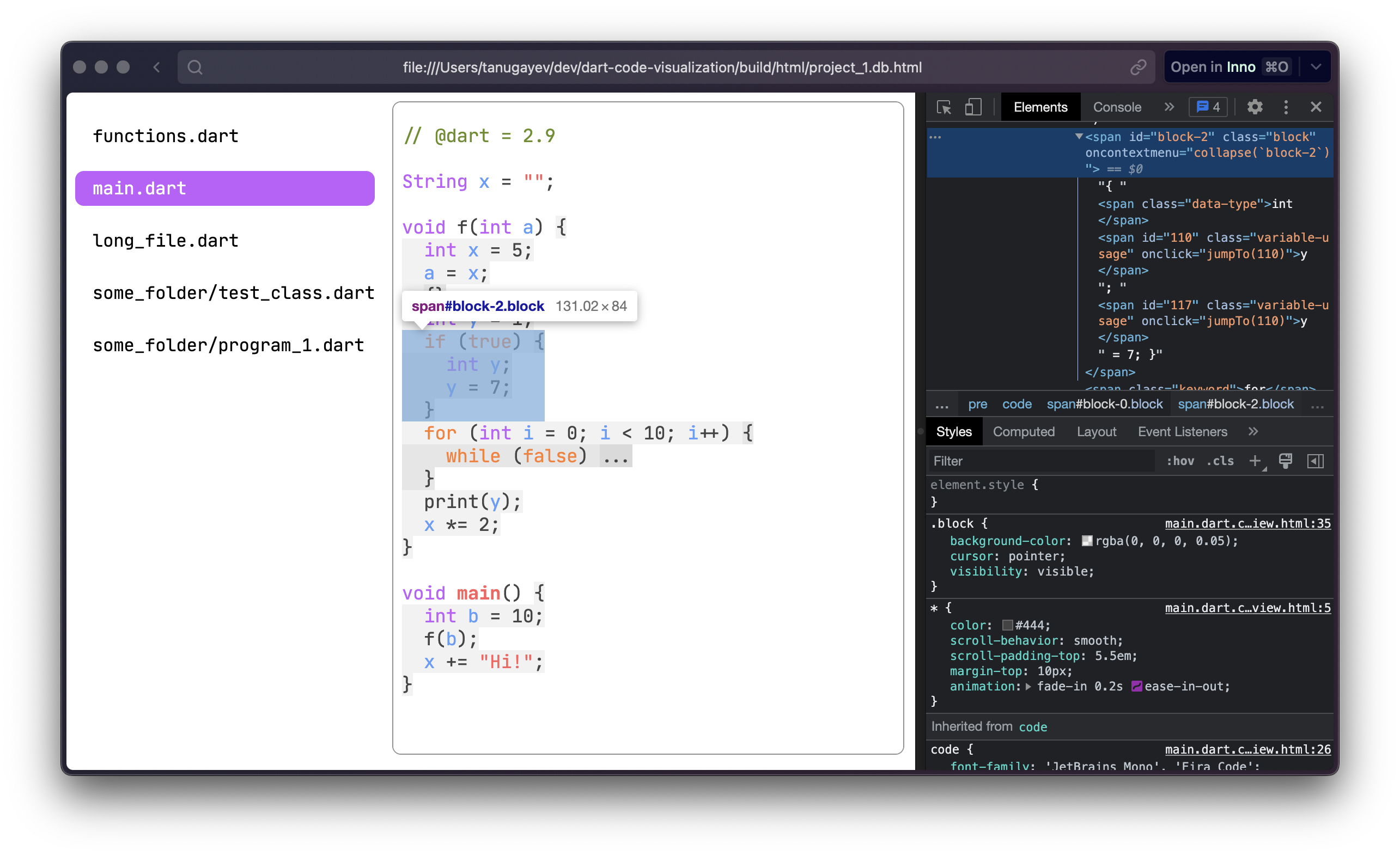


*Fig. 9. Event sorting, comparator.*

1. 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.



*Fig. 10. Creating tags for blocks.*



*Fig. 11. Blocks in the output file.*

Figure 11 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. 11, 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.

* + 1. **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 pipeline**

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.



*Fig. 12. Usage-declaration binding step.*

The example pipeline step provided (Fig. 12) 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 of 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.



*Fig. 13. Pipeline execution step.*

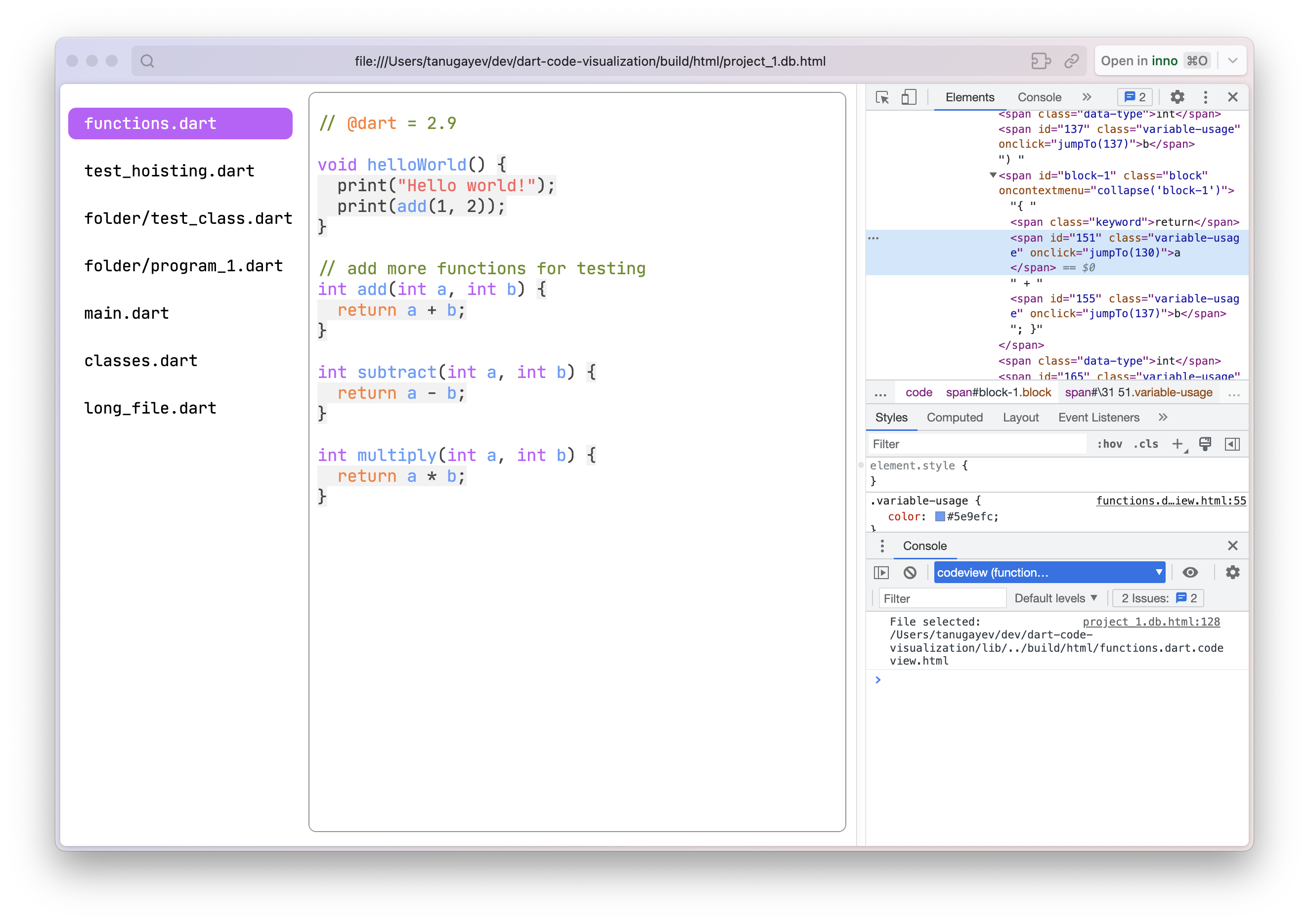
Fig. 13. shows how the process of placing these HTML tags is organized:

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 `span`s 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. 14.

1. It sorts the tags by position so as to then consequently iterate over them.
2. 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.
3. 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.
4. At the last step the function returns the modified string of HTML markup.
5. It is then processed further to form the complete HTML file that is interactable and executable.



*Fig. 14. An example of a tag generated.*

* + 1. **Syntax Highlighting**
    2. **Line Numbers?**
    3. **Comment Documentation?**
    4. **Top-level entities and global scoping**

**- Block scoping:**

- identify scopes using AST

- scanline algorithm

- **Syntax highlighting (regex order)**

**- Line numbers (?)**

**- HTML generation:**

- modular pipeline:

- initial approach (code = step1(code); code = step2(code))

- tag collision problem

- better approach (step1(); step2(); code = pipe(code))

- modular html layout (codeview, explorer, layout)

* 1. **Testing and Validation**

Several testing and validation techniques are used to guarantee DartBoard's quality and dependability. 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.

- test that all files in input project folder are analyzed

- test that with classes

- test that with functions

- test that with variables

1. **Value / Purpose**

**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 syntax highlighting, ability to jump to variable and function declarations, and project tree 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/Deployment: DartBoard can be integrated with continuous integration/deployment 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.

**Bibliography:**



[16] 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

[17] W. De 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.

[18] T. Ball and S. Eick, "Software Visualization in the Large," IEEE Computer, vol. 29, no. 4, pp. 33-43, 1996.

[19] T. Berners-Lee, "HTML: A Representation of Textual Information and MetaInformation for Retrieval and Interchange," World Wide Web Consortium (W3C), 1995.

[20] D. Raggett, A. Le Hors, and I. Jacobs, "HTML 4.01 Specification," World Wide Web Consortium (W3C), 1999.

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

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

[23] I. Hickson, "HTML Living Standard," Web Hypertext Application Technology Working Group (WHATWG), 2021.

[24] Flutter, "Flutter packages," pub.dev, [Online]. Available: https://pub.dev/flutter/packages. [Accessed March 23, 2023].

[25] Google, "json\_serializable," pub.dev, [Online]. Available: https://pub.dev/packages/json\_serializable. [Accessed March 23, 2023].

[26] R. Nicoletti, "Freezed," pub.dev, [Online]. Available: https://pub.dev/packages/freezed. [Accessed March 23, 2023].

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

**Code**

Latest prototype available at:

<https://github.com/al1ych/dart-code-visualization>

