**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 is where we decide to put forth a different approach to the problem: we perform the static analysis and all the other procedures just once and we generate the Integrated Development Environment itself, so that the end-user could access it on any computer with a browser. The generated IDE ensures total compliance with today’s standards on the modern IDEs in terms of quality of code analysis, functionality, and design at the same time providing full immunity to most of the time-consuming chores like the installation of an IDE, project codebase and dependencies, and project indexing.

**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 at the same time leaning more towards source code viewing platform rather than stand-alone code editing application, providing light-weight, platform-independent and compact solution to analyze software projects.

1. **Introduction**

**Problem insight**

Nowadays, there are so many giant large-scale projects like Windows or the gcc compiler. All of them demand incredible level of dedication to develop and maintain the thousands, millions of lines of code. So, this introduces a couple of problems that this paper aims to eliminate.

There are many reasons why projects get complex. We will emphasize only 3 main reasons for our purposes here:

* 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 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. 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 whoever and operated under almost any conditions, no matter online or offline.

1. **Literature Review**

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 & Scope Analysis**

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**

Regular text editors work well for regular text, but code is much more than just text, so the default text editor cannot cover it. Because the structure of a software project can be so complex and unimaginably enormous, program visualization is essential for our project. It is crucial for an IDE system to present information in a comprehensive and user-friendly manner. Source Code Visualization performs analysis of a program and represents it visually: graphically or textually [3].

Computer scientists distinguish two main types of information that we need to account for and analyze to perform visualization:

* Syntactic – the shallower analysis that does not account for the meaning of the entities in the source code. Examples of this could be syntax highlighting for keywords etc., and function definition pop-ups [2].
* Semantic – everything that has to do with the semantics of the source code. Name resolutions, scopes, etc. To visualize semantic information, we need to solve many complex problems. It takes into account all sorts of contextual information. Example: tooltip on hover to show where the declaration for the variable usage is. Moreover, the analyzer should distinguish between multiple declarations of variables with the same name; it should understand and treat them like different variables. This cannot be done using just regular expressions.

The architecture of the system also needs to be visualized. There are tools and techniques for that, too: architecture diagrams, dependency graphs, and UML diagrams [2] provide an overview of the codebase and allow one to see the whole of the project, its complexity, and interconnectivity.

Most popular tools utilize both textual and graphical visualization methods.

**2.3.1 Program text visualization**

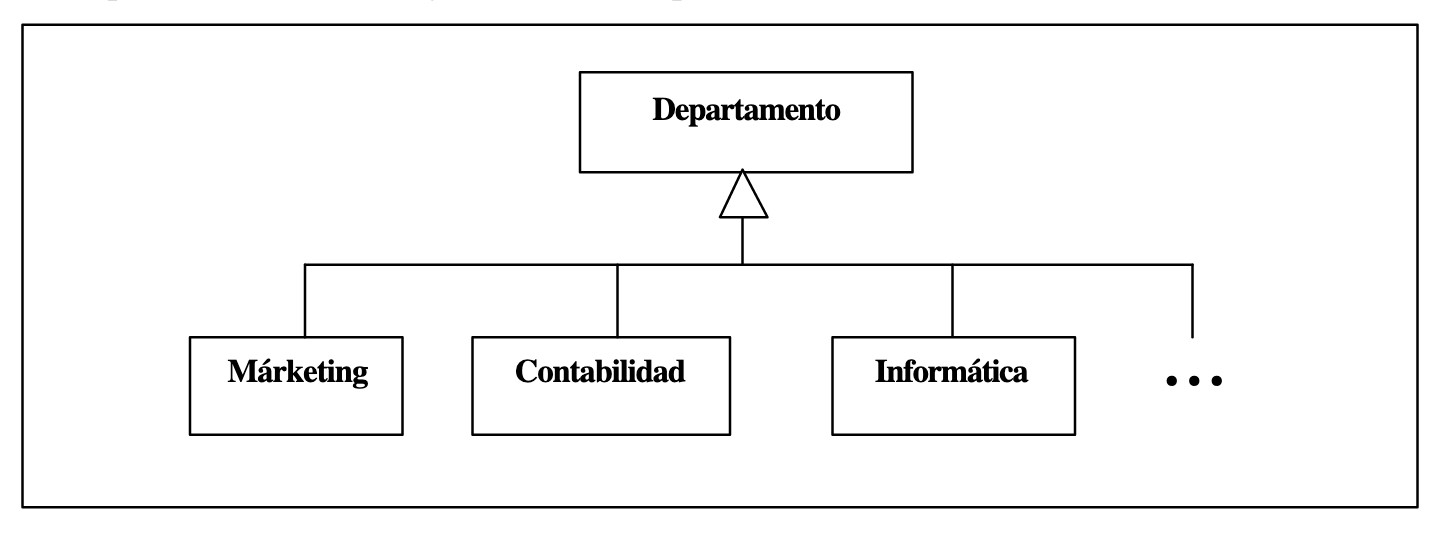
Text program visualization is about everything that has to do with the source code of the program [2]. An example of that could be syntax highlighting. For keyword highlighting, for example, the method consists of simply mapping regular expressions for all the keywords with the corresponding colors from the current color scheme we chose. This mapping formally defines a set of rules for syntax highlighting.

Text visualization is used to simplify the source code to make it more comfortable and optimized to work with the codebase. It is mostly about code navigation tools, highlights, block collapsing, etc.

**2.3.2 Graphical program visualization**

Graphical program representation is more about the structure of the system codebase. It is aimed at providing the big picture and an overview of the project.

One of the most used examples of graphical visualization is UML diagrams [2]. According to Xavier Ferré Grau and María Isabel Sánchez Segura: “UML (Unified Modeling Language) is a language that allows to model, construct and document 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.

**2.4 Generation**

Generation, in broad terms, defines a process of creating one or multiple files according to a specific set of predefined rules.

Numerous classifications of generation are known: Code Generators, Documentation Generators, Markup Generators. They vary in the method of production, type of the output, etc.

Documentation generators generate documentation from the source code of an application [15]. The most common use-case is the automatic generation of the API documentation for back-end server applications such as Postman, Sphynx, and Swagger.

We are most interested in markup generation. Hence the closest issue to ours is the API documentation generation.

**2.4.1 Why HTML?**

We chose HTML as the main target markup language because it provides enough functionality to work with textual and graphical visualizations. It is also crossplatform and available on most modern computers via pre-installed software like browsers. It is versatile and powerful, compared to Markdown or PDF formats, and also allows for interaction and stylization using JavaScript and CSS.

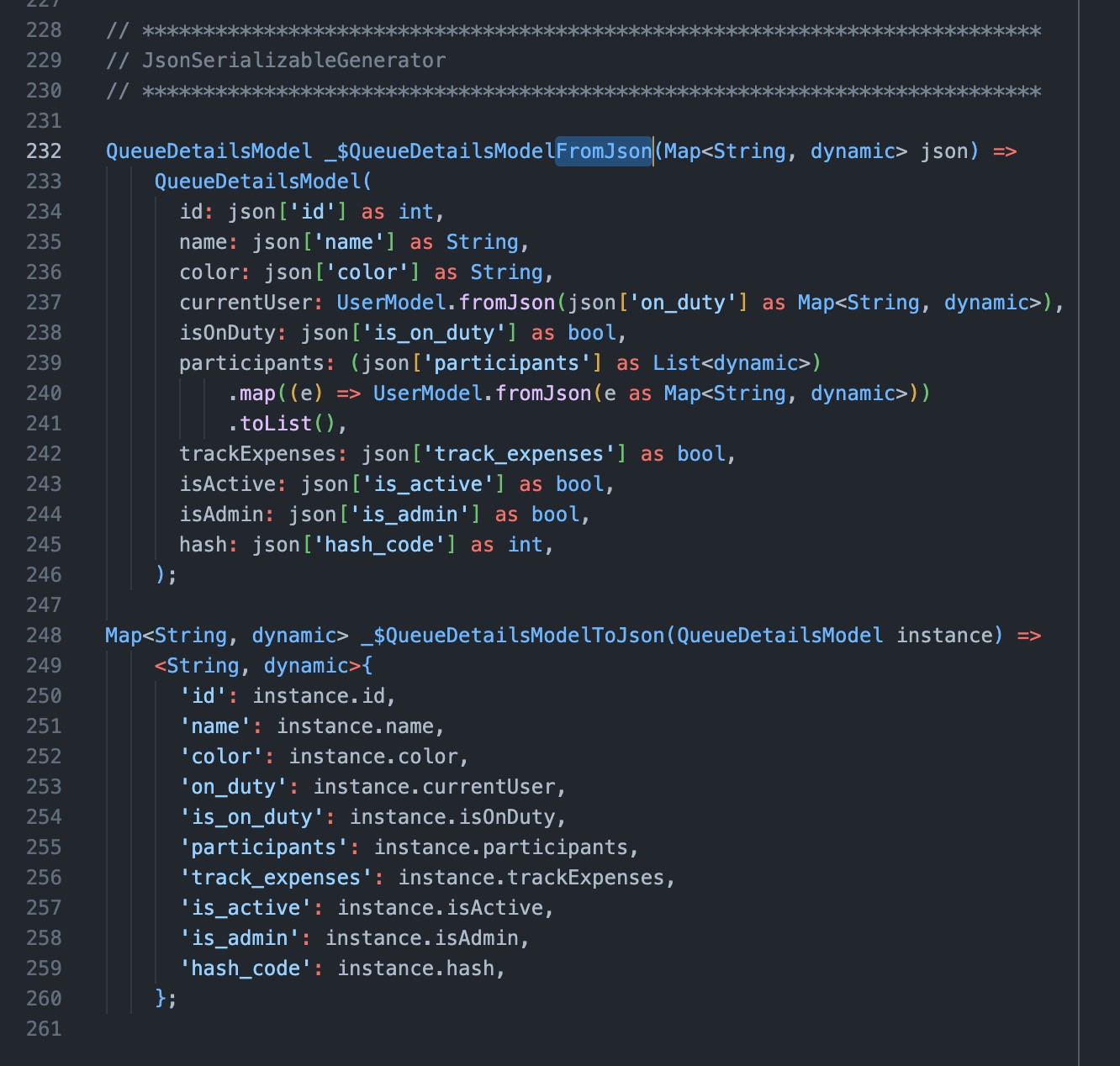
**2.4.2 Code generation**

Code generation is prevalent in Flutter projects, which is what we base our tool around. Code generator packages create boilerplate code that is otherwise very tricky or time-consuming to write. Many popular packages are available on pub.dev, including Auto-route, Freezed, etc. Many of those are considered the de-facto standard and best practice in the world of Flutter.

Code generation commonly uses AST or the Element Tree under the hood to analyze the contents of the source code and then generate some boilerplate based on this information. For example, json\_serializable (available at https://pub.dev/packages/json\_serializable) uses the Element Tree to extract the annotations. Another example, in Freezed (available at https://pub.dev/packages/freezed), the programmer should put annotations that convey a certain meaning or specify factory methods that are then unraveled into the predefined code (see *Fig. 2 and Fig. 3*).



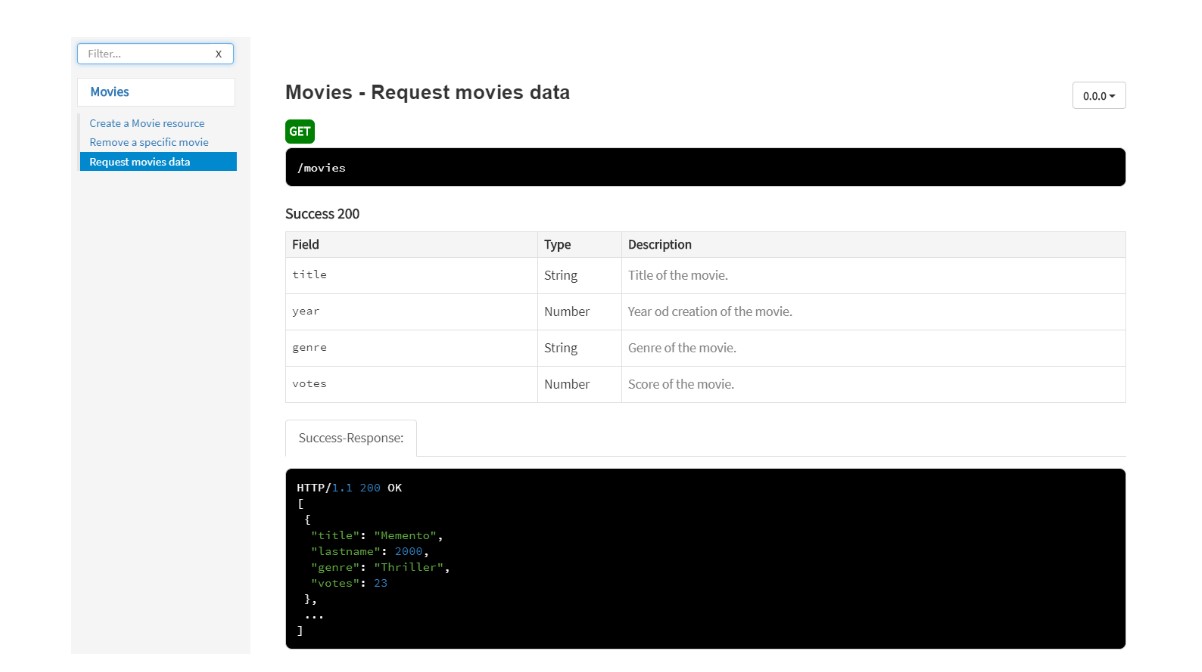
*Fig. 2*



*Fig. 3*

**2.4.3 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. **Methodology**

This chapter is about the technical stack of the project being built, its functional and non-functional system requirements, and use-cases.

* 1. **Tech stack**

The project is written in Dart, Flutter and uses internal properties of the language itself to analyze and present the system as a whole.

**3.2 Functional system requirements**

High priority:

- [ ] Generate HTML files from source code in Dart

- [ ] Highlight syntax in the generated HTML

- [ ] Jump to declaration for a variable

- [ ] Jump to declaration for a function

- [ ] Show documentation on cursor hover

- [ ] Collapse block scopes

- [ ] Upper tab bar to switch between tabs with source files

- [ ] Project tree

Medium priority:

- [ ] Theming

- [ ] Search by regex expression

- [ ] Replace by regex expression

- [ ] Rename var/function (refactor)

- [ ] Upload zip project

- [ ] Upload via github repo

- [ ] Project tree folder collapsing

Low priority:

- [x] Open generated HTML automatically for MacOS

- [ ] Open generated HTML automatically for Windows

- [ ] Open generated HTML automatically for Linux

- [ ] Gray out vars and functions that are not used

- [ ] Gray out parts of code that are inaccessible

- [ ] Add support for external plugins

- [ ] Make my own plugins

- [ ] Highlight the second parenthesis/bracket

- [ ] Scrollable minified code copy tab to the right of the screen (like in Sublime)

- [ ] Check code for errors

- [ ] Check linting errors

- [ ] Suggest linting fixes

- [ ] Flutter screen and widget graphical hierarchy

- [ ] Visualize dependency graph

- [ ] Visualize inheritance tree

- [ ] Pipelines.

Use-case 1: automatically send the generated HTML to the code reviewer.

Use-case 2: automatically host the result on Heroku or other hosting provider and share the link.

- [ ] Github Action. Use-case: automate generating the HTML alongside the documentation pipelines,

linters, testers, builders, etc.

- [ ] Integration with pub.dev (make it a pub.dev package)

**3.3 Non-functional system requirements**

- HTML should have no external dependencies.

- HTML should run on the last three major versions of Chrome, Edge, Firefox, Opera, Safari.

- 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 cli tool.

**Bibliography:**



**Code**

Latest prototype available at:

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