CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Creating an Extendable and Developer-Friendly Framework for Visualizing Data Structures and Algorithms

Bachelor's thesis in Computer science and engineering

JUSTUS FORSBERG
ISAK HAMMARLUND
DENNIS HOLMSTRÖM
JOSEF RASHEED
ALIREZA SHARIFI
FRIDA SUNDELIN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Bachelor's thesis 2025

Creating an Extendable and Developer-Friendly Framework for Visualizing Data Structures and Algorithms

JUSTUS FORSBERG
ISAK HAMMARLUND
DENNIS HOLMSTRÖM
JOSEF RASHEED
ALIREZA SHARIFI
FRIDA SUNDELIN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Creating an Extendable and Developer-Friendly Framework for Visualizing Data Structures and Algorithms

Project source: `https://github.com/Qweting/dsvis`

Supervisor: Jonas Duregård, Department of Computer Science and Engineering
Examiners: Patrik Jansson and Arne Linde, Department of Computer Science and Engineering
Graded by teacher: Birgit Grohe, Department of Computer Science and Engineering

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Creating an Extendable and Developer-Friendly Framework for Visualizing Data Structures and Algorithms

JUSTUS FORSBERG, ISAK HAMMARLUND, DENNIS HOLMSTRÖM,
JOSEF RASHEED, ALIREZA SHARIFI, FRIDA SUNDELIN

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The learning of data structures and algorithms is a necessary part of many computer science programs, and visual aids are a good way of enhancing the comprehension of these abstract concepts. In this thesis, we examine how one such visual aid can be made more developer friendly, extendable, and improved upon. This was done through the refactoring, addition to, and conversion into TypeScript of the existing data structure visualization library `dsvis`. Several data structures and algorithms were added before and after the translation into TypeScript to be able to compare the developer experience. Refactoring and translation of the codebase provided improvements to the developer experience and readability, evaluated through both our own experience and statistical analyses of code comprehension. Finally, we conclude that while we were unable to entirely reach our ambitions (in particular concerning testing and visualization logic), we were indeed able to reduce code complexity and duplication while simultaneously introducing reusable code structures, placing the framework in a good position extendability-wise.

Keywords: data structures; algorithms; visualization; TypeScript; Webpack; refactoring; developer experience; extendability; readability; understandability

Sammandrag

Datastrukturer och algoritmer är ett ämne som lärs ut på många linjer inom datavetenskap, och visuella hjälpmedel är ett bra sätt att underlätta förståelsen för dessa abstrakta koncept. I detta arbete undersöker vi hur ett sådant visuellt hjälpmedel kan göras mer utökningsbart, utvecklarvänligt och läsbart. Detta har gjorts genom omstrukturering, tillägg på, samt omskrivning till TypeScript av det redan existerande visualiseringsbiblioteket `dsvis`. Ett flertal datastrukturer och algoritmer blev tillagda både före och efter översättningen till TypeScript för att möjliggöra jämförelse av utvecklarupplevelsen. Omstruktureringen och översättningen av koden gav förbättringar till utvecklarupplevelsen och läsbarheten, utvärderat genom både vår egen upplevelse samt statistiska analyser av kodförståelse. Vi drar slutsatsen att trots att vi inte helt nådde upp till våra ambitioner (framförallt rörande testning och logiken kring visualisering), så lyckades vi minska komplexiteten och duplicering av kod samtidigt som som vi introducerade återanvändbara strukturer i koden, vilket placerar ramverket i en god position angående utökningsbarhet.

Acknowledgements

We thank our supervisor Jonas Duregård who has provided us with advice and suggestions to help us move forward. In addition, we are grateful to Peter Ljunglöf, the author of the library that we have based our project on. Finally, we thank our families and friends who have continuously supported us during this project.

Justus Forsberg, Isak Hammarlund, Dennis Holmström, Josef Rasheed, Alireza Sharifi, Frida Sundelin, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

The Data Structures and Algorithms course at Chalmers University offers an introduction to fundamental concepts in data management and algorithm design. Covering trees, graphs, sorting methods, and complexity analysis, it provides students with essential skills to solve computational problems and optimize software performance in practical applications [1].

Understanding these concepts is a non-trivial task for students taking the course, especially those relatively new to programming. Visualizations of the processes involved can give students a better grasp on how they work, and research shows that the use of visual programming languages such as Scratch to learn programming is beneficial and facilitates learning [2]. By illustrating the algorithmic steps involved, it can help students understand the fundamentals of a data structure or algorithm.

To this end, the library *Visualizations of data structures and algorithms* (`dsvis`) was developed by Peter Ljunglöf, to visualize various data structures [3]. The design was inspired by David Galles' JavaScript visualization application from 2011 [4].

1.1 Purpose

The goal of this project is to refactor and extend Ljunglöf's library into a framework that is more developer-friendly, extendable, and maintainable. In other words, the project aims to provide a more structured tool for users interested in data structures and algorithms, while also ensuring that the framework is easier to develop and maintain in the long run. A secondary goal is to confirm the extendability of the framework by implementing additional data structures and algorithms that are visually distinct from those already implemented.

1.2 Problem

The existing library works well as is, but an obvious future improvement is to add support for more kinds of data structures such as sorting and graph algorithms. One obstacle to this is the lack of modularity in the codebase, which has caused tight coupling between the logic and the visualization. This limits the scalability of the library. Additionally, the library is written in JavaScript, which is not inherently problematic but does introduce certain challenges.

The primary focus of this project is on improving the existing codebase to enhance its extendability and developer-friendliness. In other words, should the framework need to be extended with additional features, the goal is for this process to be able to be done as painlessly as possible.

This project will thus examine several questions related to its purpose:

- How can we improve the framework's developer experience and extendability over the original library?
- In what ways does code testing contribute to ensuring code integrity and reducing run-time errors?
- What are the benefits of separating logic from visualizations in terms of architecture and future extendability of the framework?
- In what ways can the framework be extended with new data structures while maintaining a clear visual distinction between them?

1.3 Limitations

As our primary focus is on enhancing the code architecture, improvements to the user interface (UI) and the user experience (UX) are not within the scope of this project. This is mainly because the current website already fulfills the purpose of being user-friendly. Additionally, evaluating the outcome of UI or UX changes requires extensive testing and time.

The extension of visualized data structures and algorithms to the project has been limited to implementing a few sorting algorithms and basic data structures. This means that graph algorithms, hash maps, or algorithms outside of the Data Structures and Algorithms course are not covered by the scope of this project.

1.4 Related Work

There are many other online visualizations of data structures. According to a research team at the National University of Singapore (NUS), most online visualizations are hard-coded and non-interactive, using old and outdated technologies such as Flash or Java Applet, or are not publicly available [5].

There are still plenty of more modern publicly available visualization tools [6], including one that the same team from NUS built [5], a more modern implementation made by the CS 1332 team [7], and the project made by David Galles which inspired the original library [4].

1.5 Social and Ethical Aspects

When planning this project, we considered whether it would have any social or ethical aspects that we should be aware of. We considered the ethical aspects of

doing good and not infringing on the anatomy and integrity of others. We also considered who our project would affect. We came to the conclusion that our project did not harm or impact anyone at all. Therefore, we concluded that there are no social or ethical aspects that this project needs to account for.

2

Technical Background

This chapter contains the technical information on which this project is built, and provides the necessary context for understanding the decisions made throughout this project.

2.1 JavaScript

JavaScript is one of the most popular programming languages. According to Octoverse 2024, an annual report published by GitHub regarding the state of programming as seen in GitHub repositories, JavaScript has only recently lost its spot as the most popular programming language to Python, a spot it held for 10 years [8]. JavaScript is a dynamic, weakly typed, and object-oriented programming language and is based on the principles of ECMAScript and its standards [9]. It is mainly used for client- and server-side programming in order to create dynamic web pages through HTML [10]. JavaScript is known for its rapid prototyping capabilities, which make it a go-to language for quickly adding features [11].

2.2 TypeScript

TypeScript is a statically typed programming language built on JavaScript and addresses some of the challenges involved in developing using it [12]. A developer survey published by Stack Overflow in 2020 reveals that TypeScript was the second most loved programming language [13]. TypeScript code can be compiled into JavaScript by simply removing the type definitions, providing compile-time checks without affecting how the code is executed [13]. Some benefits of TypeScript include better tooling and code editor/IDE support, error prevention, and bug detection. Additionally, TypeScript provides support for compiling the latest JavaScript features into older JavaScript standards (down-compiling), allowing execution on older browsers [12], [13]. Furthermore, applications using TypeScript generally have better code readability, understandability, and code quality than JavaScript applications [12], [14].

Some important features in TypeScript are:

- Generics, which can be used with functions, classes, types, and interfaces to make them reusable and flexible.
- Type aliases, which are used to add clear names to more complex types.
- Union types, which can be used to combine multiple primitive types or to allow only specific values.

Together, these features can be used to create complex but flexible types with clear identifiers [12].

Additionally, TypeScript adds additional keywords that are used to further improve the type safety with variables. The first is `as const` that makes a variable deeply read-only and the second is `satisfies Type` which ensures that the variable satisfies a specific type without using the more generic type [13]. All of the above can be combined together to ensure type safety. Finally, TypeScript adds functionality to control the visibility of methods and properties in classes [13], similar to how it is handled in Java.

2.3 Current Library

The pre-existing library is written in JavaScript and supports the visualization of some data structures, such as binary heaps and binary search trees[1]. The library also contains a prototype for integrating course exercises with visualizations of AVL trees. It is written in JavaScript and is presented using a combination of multiple HTML pages that all use the same single CSS file.

At the foundation of the library lies the `Engine` class that handles most of the general logic, with various subclasses handling the algorithm-specific logic. The engine class uses the SVG animation library `SVG.js` to create and animate the different elements that are used for the visualizations.

The `Engine` class provides several methods that are used by subclasses or initialization files that contain multiple data structures. Some important methods are highlighted in the following list:

- `submit` – Takes a method name and an HTML text field. Parses the values from the text field and calls the `execute` method with the method name and the parsed values.
- `execute` – Takes a method name as a string and any potential arguments to the method. Adds them to an action list and then calls the `runActionsLoop` method.
- `runActionsLoop` – Loops through all actions in the action list, visualizing each step of the process.

[1] Specifically: binary search trees, AVL trees, red-black trees, splay trees, B-trees and binary heaps.

- `getObjectSize`, `getNodeSpacing`, and `getStrokeWidth` – Helper methods that are used to obtain consistent sizing and scaling across all the different SVG elements.

There are many additional methods and properties that are used internally and by subclasses to enable smooth animations and stepping forward and backward. Furthermore, some internal helper methods are used for debugging and cookies management.

2.4 Webpack

Webpack is a bundler that is used by a variety of JavaScript and TypeScript applications. It can bundle many types of assets and combine them into one large bundle or multiple smaller chunks for optimized deployment [15]–[17]. Based on the configuration, there can also be multiple bundles depending on the application. Webpack is an important part of modern web development, especially due to the way it handles complex dependencies while improving performance [15]–[17].

The handling of dependencies by using custom internal Webpack functions enables the developer to write more modular code, making it easier to manage and maintain, allowing Webpack to handle dependencies [17]. Webpack can also optimize and minimize code for production, reducing the bundle size and load times [15]–[17]. Configuration can be done in many ways, depending on the use case and how complex the project is. Furthermore, Webpack utilizes both pre-installed, installable, and custom plugins and loaders to add extra functionality or optimize bundling. Some common Webpack plugins are `HtmlWebpackPlugin` (which is used to generate HTML files) and `TerserWebpackPlugin` (which is used to minimize compiled JavaScript files) [15], [16].

Additionally, Webpack can be used to enhance the developer experience by providing a development server with source maps for debugging, Hot Module Replacement, and Live Reloading, providing instant feedback and faster development times by allowing the website to update instantly after saving changes [15], [16]. TypeScript can seamlessly integrate with Webpack using the TypeScript loader and compiler to check types and compile the code into JavaScript [15].

2.5 Code Comprehension

A concept often discussed alongside maintainability and developer-friendliness is code comprehension [18], or the act of reading and understanding code. Many attempts have been made to model code comprehension by measuring the readability of code [19], [20].

Despite the use of readability as a general term, there is a distinction to be made between *readability* and *understandability*. In general, readability refers to the legibility of code on a local level, working with individual code snippets and statements, while understandability refers to the legibility of code on a global scale, regarding

entire classes, files, or projects [18], [19], [21], [22]. To put it simply, readability refers to low-level code quality, while understandability refers to high-level code quality.

2.5.1 Buse and Weimer's Metric

One of the first code readability metrics created is the metric derived by Buse and Weimer [23]. The metric, hereafter referred to as B&W, takes the shape of several code characteristics, such as "average number of periods" or "maximum line length". These code features are associated with positive or negative relative predictive power (RPP) of readability, as seen in Figure 2.1. The RPP values are, in turn, based on the readability judgments of 120 developers on 100 Java code snippets (making 12000 judgments in total), meaning it is an opinion-based metric.

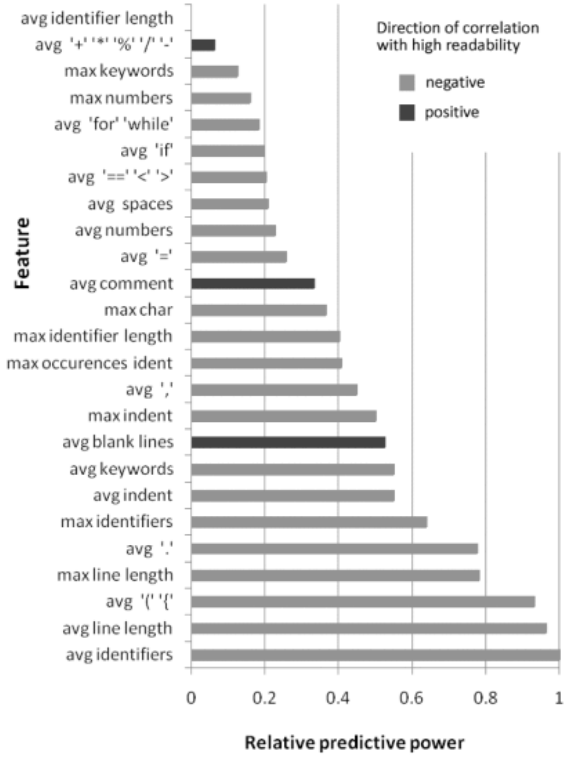

Figure 2.1: Relative predictive power of code features with the B&W metric [23]. © 2010 IEEE

2.5.2 Cognitive Complexity

A popular way to measure understandability is the Cognitive Complexity (CC) metric, introduced by Campbell for use at SonarSource [24]. It is based on and aims to modernize the older Cyclomatic Complexity [25]. The main principle behind CC is to measure complexity by counting breaks in the flow of the code. The code is analyzed line by line and a tally is incremented each time the control flow is broken, such as on `if` or `else` statements and `for` or `while` loops. The final tally is the understandability score for the analyzed code, where a lower score is better.

Additionally, there is an additive penalty for nesting, where certain control flow-breaking code located deep inside nested structures contributes an additional point to the tally for each nesting level. For example, a `if` statement within two nesting levels would lead to a tally increase of 3 instead of 1.

2.6 Refactoring

In computer science and software design, code refactoring is the process of reconstructing the internal structure of code without altering its external behavior. The goal of the process is to further improve the internal structure of the codebase, reducing its overall complexity and making it more maintainable, extendable, and easier to understand [26]. This also makes future changes easier, safer, and more efficient to implement.

Refactoring can be performed by applying different techniques to solve specific challenges or issues in a project. A common technique utilized when refactoring is the single responsibility principle, which revolves around a requirement that each class should have only one specific responsibility, this principle simplifies code comprehension and maintenance [27]. Another common focus with refactoring is code documentation and comments which simplifies future development, as developers can easily maintain and expand on the current codebase [28].

2.7 Comments

Comments in code can generally be seen as a way to enhance the legibility of the code by highlighting sections, clarifying complex parts, and organizing. However, this has not been confirmed by research [29]. There is no standard rigid structure for comments, which can lead to greater variability, unlike source code where there are standard structures. In most cases, comments are written alongside source code, providing documentation and additional information, but this varies between languages [29].

```
/**
 * Multiplies two numbers together
 *
 * @param num1 - The first number to multiply.
 * @param num2 - The first number to multiply.
 * @returns The product of the two numbers
 */
function multiplyNumbers(
    num1: number,
    num2: number
) {
    return num1 * num2;
}
```

Figure 2.2: A JSDoc comment on the `multiplyNumbers` function.

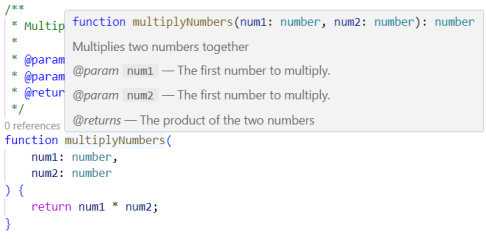

Figure 2.3: The JSDoc-based preview shown in Visual Studio Code when hovering over the `multiplyNumbers` function.

One common type of comment is the header comment. These are comments placed above class, function, method, or variable definitions and are used to explain their functionality in further detail. Compared to source code comments, which generally only explain the code at the lowest level. In JavaScript and TypeScript, header comments often come in the form of JSDoc comments, as shown in Figure 2.2. These are used to describe code and how it is expected to behave, which can be compared to JavaDoc for Java or phpDocumentor for PHP [29], [30]. The primary use of JSDoc comments is to automatically create a documentation page by parsing them [31], but many IDEs support additional features, such as an automatically generated preview when hovering the cursor over a documented object. An example is shown in Figure 2.3. JSDoc supports types, but when used with TypeScript, it becomes redundant because TypeScript provides better and more advanced tooling.

2.8 Testing

Software testing can be performed at several different levels depending on the project requirements. In general, the three levels of testing can be described as follows:

- **Unit tests** verify the smallest units of code, and the verification process is done in isolation and is performed on individual functions or classes. By focusing on a single "unit", one can ensure that the code is behaving as intended before the code is integrated into to the codebase [32].
- **Integration tests** ensure that multiple units work together as expected. For comparison, in unit testing individual functions or classes are tested and verified in isolation, whereas in integration testing the focus is on testing the data exchange and the interactions between these units [33].
- **End-to-end (E2E) tests** emulate a real user's interaction with with the application. This method of testing focuses on ensuring that every part of the application is working as expected for the end user [34].

Throughout all levels, robust test suites provide multiple benefits, guarding against code regression when refactoring, documenting the expected behavior of the software, and enabling automated verification of correctness. In particular, they help ensure that implementation of new features do not break existing functionality. This is valuable, especially in projects where the components have a complex interaction between them. Consequently, systematic testing supports maintainability and facilitates safer long-term evolution of the software [32]–[34].

3

Methodology

This chapter outlines the key strategies employed to enhance the framework's structure, maintainability, and functionality. To accomplish these objectives, we will use the following approach:

1. Transition from JavaScript to TypeScript to enhance reliability through type safety
2. Refactor the existing codebase to improve readability and, if possible, performance
3. Extend the framework with new visualizations, to demonstrate possible future development

Each step is designed to ensure more robust and scalable code while facilitating future development and collaboration.

3.1 Migrating to TypeScript

Migrating a JavaScript project to TypeScript can be approached in various ways depending on the current state of the code. For example, one strategy is to gradually adopt TypeScript by using the TypeScript compiler to check JavaScript files using the types in existing JSDoc comments [13]. Due to the smaller size of the framework, with few dependencies and no JSDoc comments, we did not utilize gradual adoption and instead followed the migration guide provided by the TypeScript documentation [13].

Firstly, the project directories were organized to separate source files from the compiled output, preventing any overwriting issues. The Node Package Manager (npm) is a tool used to manage JavaScript packages and initialize projects [15]. Using this, we initialized a new project and then installed Webpack and TypeScript as dependencies. Additionally, we needed type declarations for the SVG library to ensure type safety. Therefore, the library and its type declarations were installed using npm, in favor of having the library file in the source code.

Secondly, TypeScript and Webpack were configured to define compiler options and specify which files should be included in the compilation process. TypeScript was initially configured to follow the migration guide. The configuration was later improved to better utilize the benefits of TypeScript by enabling stricter type checks.

The Webpack configuration was initially based on the same migration guide. This configuration provided an output bundle with the compiled source code. It was then improved to provide a development server with source maps, Hot Module Replacement, and Live Reloading enabled. Further improvements were also made to simplify the developer experience by utilizing the plugins built into Webpack. The final configuration provided one bundled output file per HTML page and generated the HTML with the correct dependencies.

Finally, existing JavaScript files (`.js`) were renamed to TypeScript files (`.ts`), allowing the TypeScript compiler to process and validate them. This step was followed by systematically resolving the type errors that surfaced, primarily by introducing appropriate type annotations. Whenever possible, the most specific types were used to maximize type safety and ensure more reliable code. We used a structured approach to this part of the migration, starting with individual SVG element classes, as these were relatively self-contained and had minimal dependencies. Subsequently, the main engine class was converted, followed by the various data structures that had already been implemented.

3.2 Refactoring

The refactoring process was performed in multiple smaller steps after the codebase was migrated to TypeScript, with the goal of slowly improving the extendability and developer experience. By refactoring after the migration to TypeScript, we could utilize its features.

The original implementation used a single global object named `DSVis` as a top-level namespace, which contained all functions, constants, and classes as key-value pairs. Thus, all parts of the library were visible to each other. The first step of the refactoring process was the removal of `DSVis` in favor of importing and exporting. This functionality was enabled by using the down-compile feature from TypeScript.

Another step in the refactoring process was extracting related functions together into smaller helper classes that are used and initialized by the main engine class. These new classes were built to be extendable. When smaller classes depended on each other, we passed the class instance as a property to avoid having multiple instances.

In order to ensure that certain prerequisites are met, guard clauses were used to check values before executing the main part of functions [35]. The original library did not use any checks to ensure this. We added guard clauses to narrow down the types and ensure that the properties were initialized. If the conditions were not met, we threw errors at runtime for checks that could not be made at compile time.

The last step of the refactoring process was to clean up the code and improve code quality. Readability and maintainability are highly connected concepts, and improvements in one measure usually result in improvements in the other [18], [20], [23]. To improve code quality, readability, and understandability, we added extra lines between statements, improved variable names, and added comments inside the

function bodies. Furthermore, JSDoc comments were added to classes, functions, and variables to further improve code quality.

3.3 Sorting Implementation

The implementation process followed a structured workflow: first designing the HTML page, then developing and integrating the sorting algorithms, and finally refactoring and converting the codebase to TypeScript.

First, we created a new HTML page dedicated to sorting algorithms, using the existing priority queue page depicted in Figure 3.1 as a template. This was done because the priority queue page already had a functioning array structure that supported insertion (called `DSArray`), meaning we had to spend less time working out how to utilize the arrays.

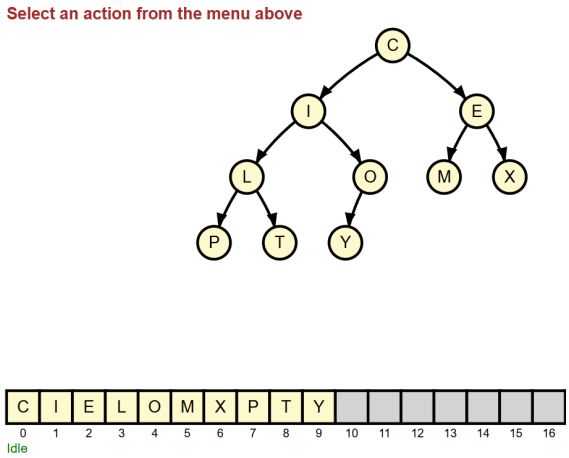

Figure 3.1: The original priority queue page.

The interface was modified to replace the delete button with a sort button, a dedicated JavaScript file was developed to handle the page logic, and the `DSArray` was reused for the array to be sorted. Separate files were created for each of the four sorting algorithms (selection sort, insertion sort, quicksort, and merge sort), where we defined both the logic of the algorithms and their corresponding animations. These animations include element highlighting and a step-by-step explanation of the currently chosen algorithm.

The sorting algorithms were built with adaptability in mind. The array that holds the list is a dynamic list that grows when values are inserted. This allows for the sorting of lists of any size. Since this made it so that the array can be longer than the width of the screen, the canvas was adapted with scrolling and zooming capabilities. Merge sort in particular has the ability to expand both to the left and to the right, which means it can expand into negative coordinates of the SVG canvas. To solve this, a `compensate` function was added that predicts the leftmost coordinate of the branching structure, inverts the negative value, and adds a margin that moves the entire structure so that it fits on the canvas.

The visualizations of the sorting algorithms required differently colored highlights in order to distinguish different parts of the sorting mechanism. This was not possible in the original library, as the pre-existing `highlight` function only allowed for a single highlight color. We solved this by extending `highlight` to allow for multiple.

After verifying everything was working correctly, the code was translated into TypeScript by changing the file endings and resolving all the resulting errors, similarly to the process described in Section 3.1. Finally, we isolated common parts of the sorting algorithms into their own class, which all specific implementations of sorting algorithms inherit. This base class provides utility methods for animations and array access such as insertions and swaps of elements. This class is then extended by specific sorting implementations.

3.4 Linked List Implementation

The implementation of the linked list data structure was based on the Chalmers course book on data structures and algorithms [36]. This involved defining a generic node structure that stores data and maintains a reference to the subsequent node in the sequence. The node structure was designed to accommodate flexibility and broad applicability within the framework.

Standard operations such as insertion, deletion, and search were implemented to provide users with a complete understanding of the linked list's behavior. These operations were made interactive using visualizations.

Throughout the development process, strong emphasis was placed on usability and clarity. The visual elements of the framework used animations and visual indicators to clearly convey the effects of each operation.

Just like the pre-existing data structures, the visualizations of linked list operations were accompanied by step-by-step textual messages that describe each phase of the operation. The insertion, deletion, and searching operations are all animated with a similar style to the pre-existing visualizations.

3.4.1 Separation of Logic and Visualization

To pursue our goals of modularity, maintainability, and extendability, we conducted a test implementation that emphasized the separation of concerns by adopting a clear separation between the algorithmic logic of the data structures and their visual representation. The purpose of this test was to evaluate whether we could effectively decouple these components within our current time constraints and architectural setup.

Although we were uncertain whether we had the resources and structural planning to fully support this approach, we saw it as an important step toward building a more modular system.

The core logic of the linked list was encapsulated within a dedicated class, operating independently of any visual concerns and responsible solely for managing the

structure's internal behavior. This included implementing the standard operations as well as maintaining the internal state of the list.

In parallel, a separate visualization class manages the visual representation of the linked list. This class handles tasks such as computing node positions, animating transitions, and providing interactive visual feedback. It is responsible for rendering elements such as nodes, directional arrows, and highlights that respond to changes within the linked list.

3.4.2 Adaptive Scaling and Layout

New node and link elements were created that utilize the scaling values from the existing library to maintain a unified appearance. To prevent overcrowding and ensure visual clarity, a method was added to dynamically calculate the maximum number of nodes that could be displayed on the screen. This method does the following:

1. Computes the number of nodes that could fit horizontally within the given viewport width.
2. Determines the vertical row capacity based on the current scaling.
3. Multiplies these two values to establish the maximum number of nodes that can be rendered without compromising readability.

To make optimal use of the available screen space while maintaining readability, a zigzag pattern was used for node arrangement. This positioning logic follows a bidirectional layout pattern, alternating the direction of node placement in successive rows. Specifically, nodes placed on odd-numbered rows are oriented from left to right in a standard configuration. In contrast, nodes on even-numbered rows are arranged from right to left and are displayed in a mirrored orientation. An illustration of the zigzag pattern can be found in Figure 3.2.

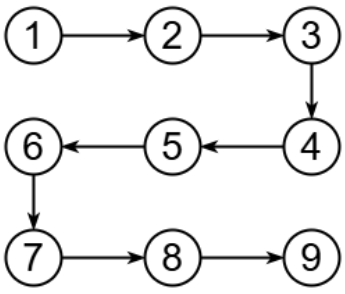

Figure 3.2: The zigzag pattern used for the linked list implementation.

3.5 Testing

Initially, we planned to implement all three types of tests described in Section 2.8 (unit, integration, and E2E tests) to ensure that each level of the application works and integrates properly. However, all algorithms handle their own animations, which

means that they all depend on the HTML element initialized in `Engine` that contains the SVG canvas. In turn, for `Engine` to be able to initialize this element, it needs to be run in a browser. This fact made the implementation of unit and integration tests difficult to accomplish as the algorithms fundamentally couldn't be run in isolation, so most of our focus was laid on E2E testing.

To implement E2E tests, we used the `jsdom` library [37], which replicates a browser environment to be able to run tests directly in Node.js. We used this in combination with `ts-jest` [38], a TypeScript testing library. Tests were created to simulate starting the application, inserting into an array, running selection sort on that array, and then asserting that the array had been properly sorted.

3.6 Comprehension Analysis

The primary statistical method used to evaluate the project was comprehension analysis, performed separately as readability and understandability analysis, utilizing the two metrics presented in Section 2.5. As the notion of "readable code" is inherently subjective to the person reading [21], it was important to base the analysis on mathematical and statistical models.

We used existing readability and understandability metrics to analyze the codebase since the design of such metrics is outside the scope of the project. Due to the syntactic similarity of JavaScript and TypeScript, the metrics were applied both to the JavaScript code and to the refactored TypeScript code.

The analyzes were performed on three different versions of the file that defines the `Engine` class to compare the readability and understandability scores before and after refactoring. Primarily, we analyzed `engine.js` from the original project and `engine.ts` from after the refactoring process. Since many of our refactoring steps involved splitting off parts of `engine.ts` into separate files, we also performed the analysis on `engine.ts` combined with its major dependencies, termed "auxiliary files". This could be seen as a fairer comparison to the original `engine.js`. These auxiliary files are:

- `cookies.ts`
- `info.ts`
- `state.ts`
- `helpers.ts`
- `algorithm-controls/engine-algorithm-controls.ts`
- `general-controls/engine-general-controls.ts`
- `debugger.ts`

Both analyzes were performed using a simple Python script, which can be found in Appendix A.

3.6.1 Readability Analysis

We used Buse and Weimer's metric to measure readability. Despite being based on readability judgments of Java snippets, we considered the syntax of JavaScript and TypeScript to be similar enough for B&W to remain a useful metric.

The analysis was simply performed by measuring each of the code features defined in B&W. This was done on a line-by-line basis, so for the "avg # blank lines" code feature, for example, a line was counted only if it was blank, and for "avg # comparisons", every comparison operator on a line was counted, and so on.

3.6.2 Understandability Analysis

We used the Cognitive Complexity metric to measure understandability, due to its relatively large-scale focus on control flow.

CC is a very general algorithm and can be adapted to many languages. However, the exact definitions of the algorithm needed to be adapted to our TypeScript and JavaScript use case. The following structures were defined as control flow-breaking structures leading to a tally increase equal to the current nesting level.

- `if`, `switch` and ternary operators
- `for`, `while` and `do` (part of `do/while`)
- `catch` (part of `try/catch`)

The following structures were defined as leading to a tally increase of 1, regardless of the nesting level.

- `else` and `else if`
- sequences of like binary operators (specifically of `||` and `&&`)

The following structures were defined as nesting level-increasing structures. Note that nesting increments were defined to occur after tally increments.

- `if`, `else`, `else if`, `switch` and ternary operators
- `for`, `while` and `do`
- `catch`
- anonymous function definitions

Unlike B&W, CC does not measure code features popularly thought to be connected to unreadability, such as line length or variable names. This is by design [24], and by using CC the overarching code structure could be measured, while the B&W metrics were instead used for the more minute details of the code.

3.7 Evaluation

To determine the success of TypeScript migration and code refactoring, and to what extent our goals were reached, the evaluation of this project was conducted based on several different aspects, which are going to be covered in this section.

To evaluate the extendability of our refactored framework, we used our own personal experiences as the primary metric. The evaluation process consisted of integrating new extensions, such as the attempt to implement new data structures, and observing how the framework accommodates these new additions. This was performed in separate from the refactoring process, in order to gain a more accurate reflection of the developer experience.

Additionally, the extendability was evaluated by comparing the size of the JavaScript bundles for the different HTML pages at different stages of the process. These sizes will be gathered by taking the file sizes of the files that are imported into the HTML pages at different stages of the project.

The developer experience has been evaluated with the help of the improvements we have made to the framework, and with the results of the comprehension analysis. With code refactoring, we sought to improve the overall structure of the code base as described in Section 3.2. We have also evaluated the impact on the framework's developer experience and code extendability by assessing how the new changes impacted our development.

The evaluation of the new data structures and algorithms has been done in multiple steps. To verify the correctness of each data structure and algorithm, we have followed the same implementation that is used in the Data Structures and Algorithms coursebook [36]. Furthermore, we have also performed a step-by-step verification of the algorithms to ensure their correctness (which can be thought of as manual E2E testing). In doing so, we have also ensured the accuracy and visual clarity of the textual feedback that is displayed to users.

4

Results

This chapter outlines the most important results from the changes and additions to the framework. There is some overlap between the different sections, we have structured the sections to make them more readable.

4.1 Developer Experience Improvements

In this section, we will show some results of the changes and how they have improved the developer experience (DX) of the framework. DX is important when the framework grows larger and multiple different developers work with the framework.

4.1.1 Types and Documentation

The conversion to TypeScript improved DX by eliminating uncertainties about the parameters that a function expects and what it returns. As shown in Figure 4.1, the method previously accepted any inputs for the `left` and `node` parameters. This was improved, as shown in Figure 4.2, by renaming the function parameters and adding JSDoc comments that further explain the function without the developer having to read the implementation.

By applying as specific type annotations as possible, a higher level of type safety is achieved. Consequently, it reduces the likelihood of runtime errors. Examples of custom type definitions used to improve type safety are illustrated in Figure 4.3. The first type, `InputValues`, is used to represent that inputs may be strings or numbers. Following this, the type `InfoStatus` restricts the valid values to one of three specific string literals, thereby ensuring strict control over the possible statuses. The third type utilizes the string literals defined in `InfoStatus` as keys in a key-value object named `statusText`. This object is typed to be deeply read-only, and the type system enforces that each key corresponds to a value in the object. Any missing key or incorrect values result in a compile-time error. The final custom type is a recursive interface named `MessagesObject`, which is used to define the structure of valid message objects.

4.1.2 Higher-Order Functions

A higher-order function is a function that returns or accepts a function as an argument. Replacing the original string-based function references with direct function

```
async doubleRotate(
    left: any,
    node: any
): any {
    // Note: 'left' and 'right' are variables
    // that can have values "left" or "right"!
    const right =
        left === "left"
            ? "right"
            : "left";
    // Same code below
}
```

Figure 4.1: The `doubleRotate` method without types and a comment inside the body.

```
/**
 * Performs a double rotation on the given node t
o maintain AVL tree balance.
 *
 * @param firstDir - The primary rotation directi
on. Same as first part of "left-right" or "right-
left"
 * @param node - The node to rotate.
 * @returns A promise resolving to the new root o
f the rotated subtree.
 * @throws Error If the necessary child nodes for
the rotation is missing.
 */
async doubleRotate(
    firstDir: BinaryDir,
    node: Node
): Promise<Node> {
    const secondDir =
        firstDir === "left"
            ? "right"
            : "left";
    // Same code below
}
```

Figure 4.2: The `doubleRotate` method with better types, variable names and JSDoc documentation.

```
type InputValues = string | number

type InfoStatus = "running" | "paused" | "inactive";

const statusText = {
    running: "Animating",
    paused: "Paused",
    inactive: "Idle",
} as const satisfies Record<InfoStatus, string>;

interface MessagesObject {
    [key: string]:
        | string
        | ((...args: any[]) => string)
        | MessagesObject;
}
```

Figure 4.3: Some custom TypeScript types used in the framework.

references passed as parameters to the `submit` and `execute` methods. This resulted in a higher degree of type safety and eliminates the reliance on string literals, which are prone to typographical errors and do not offer validation at compile time. Instead, by leveraging TypeScript's static type system, function signatures can be enforced at compile time, leading to more predictable behavior, easier debugging, and improved code maintainability. With the modified implementations of `submit` and `execute` compile-time checks were used to ensure that the correct type and number of parameters were passed in. The previous implementation, shown in Figure 4.4, lacked type safety, as there was no guarantee that `this[action.oper]` referred to a callable function. Assuming that it referred to a callable function, there was no check in the `submit` or `execute` methods ensuring that `action.args` were valid pa-

rameters. The revised implementation, shown in Figure 4.5, ensures type safety by assigning a function to `action.method`. The `apply` function binds the correct `this` context to the method and invokes it with the specified `action.args`. Furthermore, the `submit` or `execute` methods ensure that `action.args` are valid arguments for the function. Notably, the renaming from `action.oper` to `action.method` was used to further clarify that a method reference was stored.

```
async runActionsLoop(): Promise<void> {
    for (
        let nAction = 0;
        nAction < this.actions.length;
        nAction++
    ) {
        // Same code here
        await this[action.oper](
            ...action.args
        );
    }
}
```

Figure 4.4: Previous implementation of `runActionsLoop` using string-based function references.

```
async runActionsLoop(): Promise<void> {
    for (
        let nAction = 0;
        nAction < this.actions.length;
        nAction++
    ) {
        // Same code here
        // Bind this to method and call it
        await action.method.apply(
            this,
            action.args
        );
    }
}
```

Figure 4.5: New implementation of `runActionsLoop` as a higher-order function.

Previously, when invoking the methods `submit` or `execute` that in return call the `runActionsLoop` method. The methods were error-prone without giving an error at compile time. Examples of how `submit` or `execute` were called in the previous implementation are shown in Figure 4.6. The new implementation shown in Figure 4.7 introduces checks at compile time, since we now know that `this.engine.insertRight` is a function and what parameters it expects. The `submit` method is restricted to arguments of type `string | number`, while `execute` can accept functions with arbitrary arguments passed as the second parameter.

4.1.3 Removal of Global Variable

The original implementation used a global object named `DSVis` that had functions, constants, and classes on different keys, as a simplified version shows in Figure 4.8. In the original library, `DSVis` was defined in the main engine file and treated as global. The new implementation uses a module-based structure in which functions, constants, and classes are exported and then imported by the file that needs them, as shown in Figure 4.9. This enforces better encapsulation and modularity, thereby reducing the risk of accidental name collisions or overwriting, which can occur when multiple files interact with the same global object. It also allows for better types because the values available on the different keys previously depended on what was imported to the HTML file and in what order. Furthermore, it enables modern IDEs to offer enhanced developer support, including intelligent code completion and easier navigation between files and classes.

```
this.createRight.addEventListener(
    "click",
    () => this.engine.submit(
        "insertRight",
        this.insertField
    )
);

this.moveParent.addEventListener(
    "click",
    () => this.engine.execute(
        "moveParent"
    )
);

this.moveLeft.addEventListener(
    "click",
    () => this.engine.execute(
        "moveChild",
        ["left"]
    )
);
```

Figure 4.6: Function calls using string references.

```
this.createRight.addEventListener(
    "click",
    () => this.engine.submit(
        this.engine.insertRight,
        this.insertField
    )
);

this.moveParent.addEventListener(
    "click",
    () => this.engine.execute(
        this.engine.moveParent,
        []
    )
);

this.moveLeft.addEventListener(
    "click",
    () => this.engine.execute(
        this.engine.moveChild,
        ["left"]
    )
);
```

Figure 4.7: Function calls using higher-order functions.

```
const DSVis = {};

DSVis.AVL = class AVL extends DSVis.BST {};
```

Figure 4.8: Definition of `AVL` with the global `DSVis` variable.

```
import { BST } from "./BST";

export class AVL extends BST {}
```

Figure 4.9: Definition of `AVL` utilizing import and export functionality.

4.1.4 Preventing Incorrect Calls

The addition of guard clauses at the beginning of the function and method bodies results in clearer runtime errors, enhancing DX. The previous solution, found in Figure 4.10, relied on documentation (a comment) to assume that the passed node would never have two children, but did not perform an actual check before execution. This lack of validation could lead to unexpected behavior without a clear error if the function was called incorrectly.

The new solution, found in Figure 4.11, explicitly ensures that the node does not have two children, throwing a clear error if this condition is violated. This improves debugging and provides better protection against incorrect usage. Additionally, the inclusion of a JSDoc comment allows this information to be easily accessed without the need to open and read the implementation of the function.

4.2 Extendability

In this section, we will show some results from the changes made to our framework and how they have improved the extendability. Extendability is an important part of this framework, enabling the addition of new small or large features.

```
async deleteNode(node: Node): Promise<{
    success: true;
    direction: BinaryDir | null;
    parent: Node | null;
}> {
    // The node will NOT have two children - this has been taken care of by deleteHelper
    // Same code below
}
```

Figure 4.10: The `deleteNode` function before refactoring.

```
/**
 * Deletes a node from the binary tree.
 * This function assumes that the node has at most one child.
 * Cases with two children should be handled by `deleteHelper`.
 *
 * @param node - The node to delete. Can at most have one child.
 * @returns A promise that resolves to an object containing:
 * - `success`: Always `true` when resolved.
 * - `direction`: The direction from the parent to the deleted node ("left" or "right"), or `null`
if the node was the root.
 * - `parent`: The parent of the deleted node, or `null` if it was the root.
 *
 * @throws Will throw an error if the node has two children, which violates the function's assumpti
on.
 */
async deleteNode(node: Node): Promise<{
    success: true;
    direction: BinaryDir | null;
    parent: Node | null;
}> {
    if (node.getLeft() && node.getRight()) {
        throw new Error(
            "Expected node to only have one or zero children when delete node was called"
        );
    }
    // Same code below
}
```

Figure 4.11: The `deleteNode` function after refactoring.

4.2.1 Adding Elements to the Canvas

To enhance extendability, modifications were made to the way elements are added to the canvas. The previous approach, illustrated in Figure 4.12, involved adding a custom method to `this.Svg`, which in turn invoked another method responsible for adding the element. This solution has been refactored to eliminate the use of the custom intermediary method. Elements are now added directly to the canvas, as shown in Figure 4.13.

This refactoring not only reduces complexity by simplifying the function call stack, but also improves extendability by decoupling individual elements from the main `Svg` class. However, the new solution requires the developer to know that `init` needs to be called after the element has been added to the canvas to avoid potential runtime errors.

```
newNode(text: string): BinaryNode {
    return this.Svg.binaryNode(
        text,
        ...this.getNodeStart(),
        this.getObjectSize(),
        this.getStrokeWidth()
    );
}

Svg.binaryNode(
    text: string,
    x: number,
    y: number,
    size: number,
    strokeWidth: number
): BinaryNode {
    return this.put(
        new BinaryNode(
            text,
            size,
            strokeWidth
        )
    ).init(x, y);
}
```

Figure 4.12: Previous approach for adding elements, using an intermediary custom method.

```
newNode(text: string): BinaryNode {
    return this.Svg.put(
        new BinaryNode(
            text,
            this.getObjectSize(),
            this.getStrokeWidth()
        )
    ).init(...this.getNodeStart());
}
```

Figure 4.13: Refactored approach for adding elements, adding directly to the canvas.

4.2.2 Helper Classes

Refactoring non-central functionality into self-contained helper classes has helped reduce clutter within the main engine class. By extracting, grouping, and organizing related logic, developers can now focus on extending the engine without having to understand the internal implementation details of the helper classes.

A clear example of this is the `cookies` helper class. Previously, custom functions were required to update and retrieve cookies, and the logic was scattered across multiple parts of the main engine class. After the refactor, the cookies functionality is now fully encapsulated within its own class, requiring only initialization in the engine constructor. All loading, saving, and updating operations are handled by the new class, which enhances DX and makes the framework more extendable.

4.2.3 Bundled Output Size

By integrating TypeScript and Webpack into the project, we achieved the advantages of strong typing, enhanced developer tooling, and more maintainable code while preserving a compact output bundle. As shown in Table 4.1, the initial conversion to TypeScript resulted in smaller bundle sizes. This reduction can be attributed to Webpack's bundling mechanisms. The bundling process involved minimization, including eliminating line breaks, comments, and shortening variable names, all of which contributed to the initial size reduction.

Subsequent refactoring introduced improvements, such as code splitting. These changes further reduced the output size by removing unused imports and enhancing

modularity. A major contributing factor was the decoupling of the addition of new elements to the canvas from the central rendering logic, as discussed in Section 4.2.1. This decoupling improved the scalability and maintainability of the codebase, while further reducing the output size of the bundles.

Table 4.1: Output file sizes after different project phases.

	Base Engine	Collections	Prioqueue	AVL-quiz
Original size	113 kB	177 kB	126 kB	139 kB
TS conversion size	95 kB	144 kB	116 kB	122 kB
Refactoring size	76 kB	142 kB	108 kB	115 kB

4.2.4 Sorting Algorithm Base Class

To facilitate the implementation of new sorting algorithms and increase maintainability, a general sorting class was implemented. This class acts as a superclass that all implementations of specific sorting algorithms inherit. It inherits the engine and is responsible for initializing the correct sorting subclass and the array that will be sorted. It also handles all operations on the array, such as insertions, swaps, and highlights. Thus, any specific implementation of a sorting class only needs to include the sorting logic and custom messages for that algorithm. Therefore, this class greatly reduces the time it takes to implement a sorting algorithm and also helps to make the code more readable and maintainable by not having as much duplicate code.

4.3 Readability and Understandability

In this section, the three versions of `Engine` will be referred to as such:

- JS – `engine.js` from the original library
- TS – `engine.ts` from our refactored framework
- TS + aux – `engine.ts` with auxiliary files

In Table 4.2 we present a comparison of readability scores, with relative predictive power (RPP) values from Figure 2.1 included for comparison. We did not include B&W code features with an RPP below an absolute value of 0.3, as they were considered too insignificant. Note that with “avg”, we specifically mean “average per line”.

The scores from the understandability analysis were as such:

- JS: 212
- TS: 43
- TS + aux: 116

Table 4.2: Results of the readability analysis of the three versions of `Engine`.

Code feature	RPP value	JS	TS	TS + aux
avg # blank lines	0.5	0.16	0.13	0.12
avg # comments	0.3	0.045	0.175	0.090
max # occurrences of a character	-0.4	1232	1041	2661
max identifier length	-0.4	19	22	22
max # occurrences of an identifier	-0.4	30	26	44
avg # commas	-0.4	0.24	0.18	0.19
max indentation	-0.5	24	20	28
avg # keywords	-0.5	0.69	0.74	0.68
avg indentation	-0.5	7.34	7.30	7.31
max # identifiers	-0.6	12	13	13
avg # periods	-0.8	0.76	0.66	0.54
max line length	-0.8	140	120	132
avg # parentheses	-0.9	1.87	1.38	1.41
avg line length	-1	30.8	32.8	30.6
avg # identifiers	-1	1.64	1.66	1.47

4.4 Sorting Algorithms

The results for the sorting algorithms were that four new algorithms were added; selection sort (Figure 4.14), insertion sort (Figure 4.15), quicksort (Figure 4.16) and merge sort (Figure 4.17). Each of these algorithms are visually distinct in the way in which they sort the array. Selection sort groups the sorted part on the left side of the array and highlights it. Insertion sort moves one value at a time in a wave-like function to the left. Quicksort has been implemented in a standard upper-lower pointer implementation and can be easily distinguished by its use of pivots and partitions, and merge sort is spotted by its use of multiple arrays that split from the main array. Differently colored highlights were also used to differentiate between different highlighted elements.

Each of the algorithms has custom text messages that display what the current step is doing, allowing users to more easily follow along with the visualization. This is in line with previous visualizations already implemented in `dsvis`.

The algorithms have been implemented on a dynamic canvas, allowing for both horizontal and vertical navigation through scrolling and zooming functionalities. This provides an interactive way to observe the behavior of each algorithm in real time. The dynamic canvas ensures that users can follow each operation closely, regardless of the size of the dataset being sorted. This feature is currently exclusive to the sorting algorithms and is not available for the other algorithms.

Furthermore, they have been implemented with dynamic arrays, allowing larger lists to show the effect of different algorithms on lists of different sizes. For merge sort specifically, we also implemented an extra algorithm that shifts the array tree to the right whenever it crosses the left canvas border.

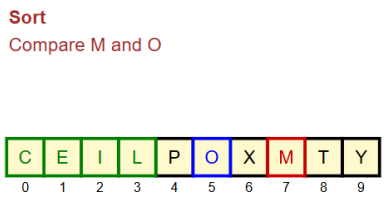

Figure 4.14: Visual representation of selection sort.

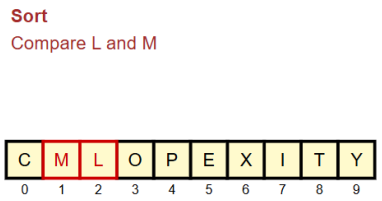

Figure 4.15: Visual representation of insertion sort.

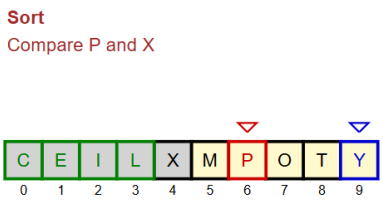

Figure 4.16: Visual representation of quicksort.

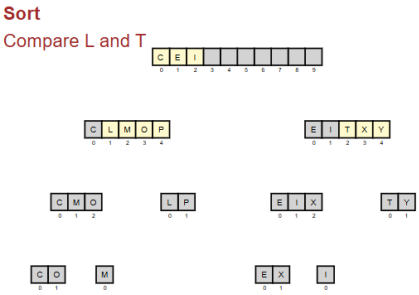

Figure 4.17: Visual representation of merge sort.

4.5 Linked List

The linked list implementation follows a traditional singly linked list data structure model, where each node contains a `data` field storing the value, and a `next` field pointing to the next node.

The `LinkedList` class implements the linked list data structure. The class contains the parameter `head` with the type `Node<T> | null`, which represents the first element of a list. Each node in the `LinkedList` is an instance of the `Node<T>` class, where each node can hold a value and a reference to the next node, forming a chain-like structure.

This linked list implementation supports the standard set of operations:

- **Insertion**: Adds a new node at the end of the list or at the beginning if the list is empty.
- **Deletion**: Removes the first node matching a given value.
- **Find**: Locates a specific node by traversing the list.

4.5.1 Linked List Animation

The `LinkedListAnim` class is responsible for the visualization of the data structure. By extending the `Engine` class, the animation class leverages the inheritance prin-

ciple to reuse core engine functionality, such as positioning, scaling, and drawing elements on the canvas. The class contains methods for the three standard operations, which are animated according to the zigzag layout.

The insertion of a new element at the end of the list is animated to emphasize the dynamic expansion of the data structure. The node is introduced with a slide-in animation. If the inserted node is not the first in the list, a directional arrow is drawn to connect it to the previous tail node. Furthermore, the visualization adjusts for directional mirroring to ensure consistent pointer alignment across alternating rows.

The find operation was visualized through an animated traversal of the list, highlighting nodes and connecting arrows as they are visited. Starting from the head of the list, each node is temporarily emphasized to indicate its inspection. If the target value is located, the corresponding node is highlighted. Each step of the traversal process can be seen in the figures. As nodes are visited their values are displayed as seen in Figure 4.18, if the value is not found then a message is displayed showing that the element has not been found as in Figure 4.19. It displays a message indicating that the algorithm is continuing its search as in Figure 4.20, and finally, it displays a message indicating that the element has been found as in Figure 4.21.

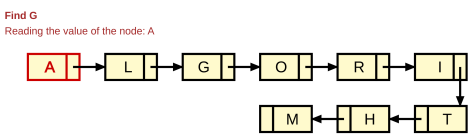

Figure 4.18: Message displaying the current value.

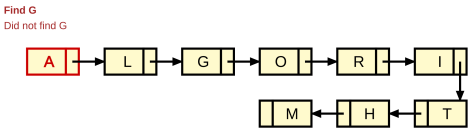

Figure 4.19: Message showing that the value has not been found.

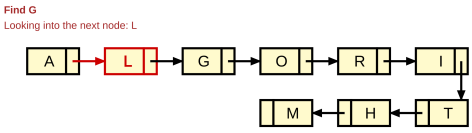

Figure 4.20: Message when continuing searching for the element.

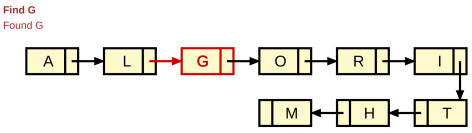

Figure 4.21: Message showing that the value has been found.

To visually represent the deletion of a node, the traversal of the list is first animated using the same highlighting logic applied during the find operations. Once the target node is identified, it is removed. Thereafter, the connecting arrow from the preceding node is updated to bypass the removed node, visually rerouting to the next node in the sequence if applicable. The layout is then recalculated and adjusted to close any gaps, ensuring that the overall structure remains compact and visually aligned.

The adaptive scaling and layout mechanisms ensure that visual representations of the data structure remain legible, visually balanced, and adaptable to a variety of display environments and user-defined settings.

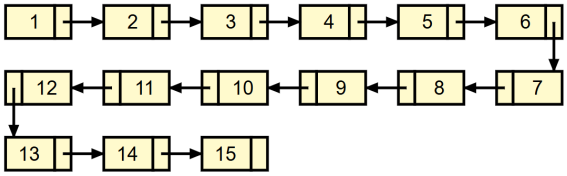

Figure 4.22: The implemented linked list layout using the zigzag pattern.

The zigzag pattern for node placement contributes to a visually balanced presentation. This design choice not only prevents overcrowding in horizontal space but also guides the user's eye through the structure in a clear and logical manner. See Figure 4.22 for an example.

4.5.2 Separation of Logic and Visualization

We aimed to promote modularity by encapsulating the core logic of the linked list in a dedicated logical layer (`LinkedList`), and handling all rendering and animation separately in a visualization layer (`LinkedListAnim`). However, in practice, we found that the current implementation of the visualization class renders the logical layer redundant. All essential structural operations, such as insertion, deletion, and find, are re-implemented directly within the visualization layer. These operations are deeply intertwined with the visual state.

This tight coupling means that the visualization layer effectively implements the data structure itself, albeit indirectly and visually. As a result, the logical layer could be removed entirely without impacting the functionality of the animation. This undermines our initial objective of separating concerns, as the animation layer has absorbed responsibilities that were meant to belong exclusively to the logical layer.

4.6 Testing

During the development process, we intended to implement tests to ensure the proper functioning of the code. However, we faced several significant obstacles that ultimately prevented us from establishing functional tests. Below is a list of the main issues we encountered:

- Initial attempts to implement unit and integration testing were unsuccessful, as we were unable to run methods or classes independently because they all depend on HTML elements.
- After a shift in focus to E2E tests, we encountered several difficulties with installing and configuring the libraries, such as compatibility issues and conflicting settings.
- We discovered that the HTML document generated by `jsdom` did not override the engine's default HTML document, which caused our query selectors to fail to locate the intended objects.

- We tried a workaround to the HTML document problem by redefining the inner HTML of the engines document. However, this caused several other time-consuming problems.

At that point, given the time already spent and the persistent technical issues, we decided to shift our efforts elsewhere and focus on other parts of the project.

5

Discussion

This chapter discusses how the modifications and additions to the framework have helped to reach our goal of a developer-friendly and extendable framework. We will also discuss related work, future work, and the use of AI in the project.

5.1 Developer Experience

This section will discuss how the framework's developer experience (DX) has changed after migrating to TypeScript and refactoring.

5.1.1 TypeScript

TypeScript has improved DX through improved type safety by eliminating the uncertainties about what parameters are expected and what is returned. By enforcing static typing, the risk of run-time errors is reduced and errors can often be found and fixed at compile time. Some errors present in the original library were found and fixed in our framework thanks to these features. Creating complex types can take extra time when developing, but it is beneficial when new developers try to use or extend the code.

The migration additionally enhances DX when developing with an IDE. Some of the features that improved the experience were documentation, autocompletion, and easier navigation between files. Furthermore, TypeScript allows splitting the code into separate files and importing and exporting functions, classes, constants, and types. This is familiar to how other languages and other larger code bases work and simplifies reading and understanding the code.

We believe that migrating from JavaScript to TypeScript was a valuable investment into maintainability, scalability, and DX. Although the migration required some initial work, efforts to understand, extend, or change the code have been reduced. Combined with the benefits from improved type safety, enhanced developer tooling, and better code organization, it made the transition worth it.

5.1.2 Understandability and Readability

From the results presented in Table 4.2, we see a clear improvement in the CC score related to the understandability of the code. The refactored `Engine` file has an ap-

proximately five times better understandability score than the original file. Perhaps unsurprisingly, by moving logic to multiple smaller files, each file is easier to understand. However, even when taking into account `Engine` with all its dependencies, the CC score of our refactoring is still approximately twice as good as the original, showing that our project was successful in improving maintainability in general.

Looking at the readability scores produced by the B&W metric, most code features have improved while others stand out. The number of blank lines (the strongest identifier of readability) has decreased slightly. However, the number of comments has drastically increased, which may balance out the reduction of blank lines. Adding more blank lines and comments would be a simple change and would be beneficial for readability. Another important improvement is a decrease in the average number of parentheses and identifiers, both strongly correlated with unreadability.

However, for many code features of the B&W metric, there were no major improvements. This is explained by the fact that the original project was already quite readable and the code was in general well-formatted. This reinforces our other results in that the main problem we faced was understandability and structural issues. Nevertheless, some improvements in readability were undoubtedly made according to the analysis.

An interesting change in B&W is that the maximum number of occurrences of a single character has increased drastically. This might seem odd at first glance, but in reality, it relates to the fact that the auxiliary files of the `Engine` class frequently reference it through `this.engine`, which has led to a large increase in the amount of the letter E. The increase in the maximum number of occurrences of a single identifier is also due to this.

It should be noted that all understandability and readability values are subjective to the evaluating developer [19]. Although the presented data provides some kind of evidence that improvements were made, the models are not perfect [21]. However, we still believe that the measured improvements were real and beneficial. We would continue to add new comments, blank lines, and make other such improvements if we were to continue working on the framework.

5.2 Extendability

This section will discuss how the extendability of the framework has changed after the migration to TypeScript, the addition of Webpack, and after refactoring.

Webpack handles the bundling of JavaScript and generates HTML output with the embedded link tags. This is more extendable than the previous solution, where the developer needed to manually update the HTML files with multiple imports in the correct order whenever the dependencies changed. Furthermore, the reduction of the base `Engine` size means that when the framework is extended, the pages will continue to load as fast as possible for the end user. This might not be noticeable at this stage of the framework, but it is nonetheless considered good practice.

The refactoring process of moving code into smaller helper classes helped to improve the extendability of the framework. The number of members available in the main engine class was reduced, which also reduced the learning curve for new developers. This could be further improved by utilizing visibility to make members `public`, `private` or `protected`, but would require additional refactoring to how the helper classes are set up. Additionally, a clearer grouping of related code simplifies changing or extending the functionality because the developer can clearly see how the functions are used and how they relate to each other. An example of this could be to change the `Debugger` class in favor of a logger class that could handle all logging logic. The switch could be done in smaller steps, adding one logging level (info, warn, error, etc.) at a time without having to change the logic in multiple places.

5.2.1 Implementing New Algorithms

During this project, we implemented algorithms in both JavaScript and TypeScript. When implementing in different languages, we faced different challenges. If we compare the implementations that we wrote in JavaScript against the ones written in TypeScript, we could notice that while JavaScript was quite a bit faster when implementing new methods due to not having to worry about typing. However, not typing variables, along with other features lost by not using TypeScript, also led to a lot of would-be compile errors turning into hard-to-find runtime errors.

In contrast, while developing after the TypeScript conversion, it was more straightforward to navigate around in the codebase due to being able to jump to the source of a variable or method. This, along with clearer documentation through JSDoc comments, made it easier to work with methods and variables from other files. Due to the inclusion of compile-time errors with TypeScript, debugging became simpler since the code did not have to be compiled and executed to notice any errors. Even when there were runtime errors, it was easier to find and solve them thanks to the inclusion of a source map. The implementation of new algorithms was also made simpler by the fact that `DSVis` was replaced with proper exporting and importing, which made it easier to understand and use important methods.

5.2.2 Decoupling Logic and Visualization

We considered decoupling the visualization code from the actual underlying logic, pivoting from how it was done in the original library. An investigation of how easy and effective this approach would be was performed by implementing the separation with the linked list data structure.

Although our initial design aimed for a clean architectural separation between the algorithmic logic and its visual representation, our experience during development revealed critical limitations in this approach. There is a fundamental challenge in visualizing dynamic data structures, where the visual behavior is inherently tied to the algorithm's behavior. Representing the state of data structures visually often requires duplicating, or even fully embedding, the data structure logic into the visualization code. With our current implementation, this makes a separate logic layer unnecessary. This result highlights the need for a revised architecture.

5.3 Sorting Algorithms

The development and visualization of the sorting algorithms involved a variety of design decisions and technical considerations.

5.3.1 Algorithm Selection

Several sorting algorithms were considered for implementation. The final selection was based on a combination of popularity and visual distinctiveness. The four algorithms chosen (selection sort, insertion sort, quicksort, and merge sort) represent a diverse set of sorting strategies and time complexities, suitable for illustrating different algorithmic behaviors. Among them, merge sort and quicksort are divide-and-conquer algorithms, providing an opportunity to demonstrate recursive visualizations. All four algorithms are included in the Data Structures and Algorithms course, which further motivated their selection and ensured that the algorithms would be relevant to the intended users.

5.3.2 Dynamic or Static Arrays

The decision to use dynamic rather than static arrays was influenced by several factors, including visual clarity, implementation complexity, and flexibility in array sizing. Static arrays offer some clear advantages; it is easier to see that the arrays have unused positions that may be used, and since they are constant in size, they are much easier to implement and less prone to bugs than the dynamic array. However, the static array also comes with a few drawbacks; a maximum size for the array must be predefined, and there will be empty spaces when the array is not fully utilized.

The dynamic array does not explicitly show capacity or growth potential, which can make it difficult to know that it may be added to. This does allow for cleaner visualizations, especially when using an algorithm that splits or creates arrays, since unused positions do not need to be accounted for. It also allows for limitless expansion, which can be useful for visualizing larger arrays. We chose to make the dynamic arrays have an empty space in the beginning, which then gets removed at the sorting stage, which helps with clarity and makes it easier to understand that one may add more to the array.

In summary, dynamic arrays were chosen for their flexibility and their ability to visualize complex algorithms better, which was valued over the lack of complexity and some visual clarity that could have been gained from using static arrays.

5.3.3 Visualization of Quicksort

When planning the new visualizations, we encountered a particular issue regarding whether the visualization of quicksort should use a single array or many arrays. If we had used many, it would have been more similar to merge sort, which shares other similarities with quicksort, such as both being recursive divide-and-conquer algorithms. The problem with using many arrays to visualize quicksort is that even though it uses partitions, it remains in place in the code. Thus, if the visualization

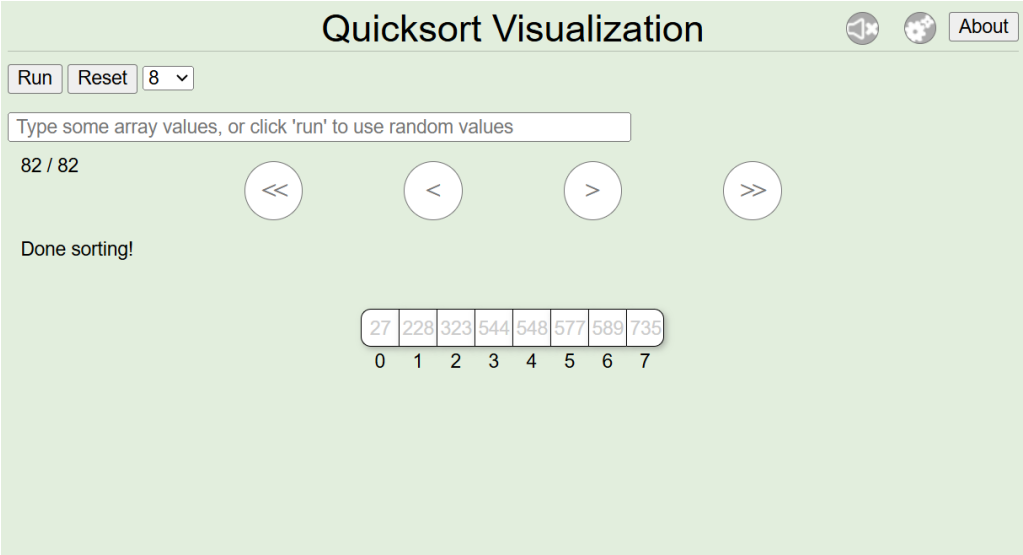

Figure 5.1: Quicksort example from the Chalmers coursebook [36].

used many arrays, it could easily be mistaken for merge sort, since quicksort is usually visualized as a single array as in Figure 5.1 (other examples include [4], [7]).

However, we prioritized being consistent with the Chalmers coursebook, which uses one array to visualize it, and so in the final project the algorithm is also visualized using a single array.

5.4 Linked List

In order to get a visually distinct and recognizable design for the linked list, we opted for a simple sectioned box design, with one section representing the value and the other representing the link to the next node. This visual representation captures the behavior of this fundamental data structure with minimal ambiguity.

5.4.1 Benefits and Drawbacks of Reusing Code

Due to the limited time we had for this project, there was a constant need to evaluate whether to rewrite or replace parts of the pre-existing library where improvements could be made, or whether to reuse them and focus on other, more important parts. While this issue applies across the entire project, we will focus on the linked list implementation as it most clearly exemplifies these problems.

An issue we encountered is that positional variables update concurrently with the animation. This is helpful when animating other related elements that track this changing value, but it becomes a problem whenever we needed to know the final position of an object before its animation finishes. This forced us to work around the issue by having slightly messier code since we had to make separate calculations to predict the position of objects after their animations.

Another notable example of this decision-making process is evident in our approach to implementing the link visualization for the linked list data structure. Instead of developing a new implementation from the ground up, we opted to extend the existing `Connection` class, originally designed for tree structures, by creating a subclass tailored to the linked list's requirements.

However, this approach also meant that certain design decisions and constraints inherent in the original tree-based implementation influenced the visualization of the linked list. While this reuse promoted development efficiency, it occasionally limited the flexibility to fully tailor the visualization to the unique characteristics of linked lists. One specific difficulty arose from the connection class's support for bending arrows, a feature that is particularly suited for hierarchical layouts such as trees. Since our linked list visualization uses a zigzag layout pattern that alternates node orientation across rows, extra work has to be done to get this visual feature working correctly. Attempting to adapt this logic within our constrained timeline proved impractical, and as a result, we decided to postpone implementing this feature.

Our decision to create a subclass of `Connection` exemplifies the trade-offs between code reuse and customization. It highlights the considerations that guided our development process, balancing the benefits of existing resources against the desire for specialized implementations.

5.4.2 Potential Insertion Strategies

To enrich the learning experience, multiple insertion strategies can be incorporated. These include inserting elements at the beginning of the list, and inserting at arbitrary positions within the sequence. Each method could be visualized distinctly to illustrate how different insertion techniques impact the structure and performance of the linked list.

Furthermore, there is the possibility of implementing additional data structure operations. However, during the refactoring of the UI, the focus remained on mirroring the original implementation, which only supported core operations such as insertion, deletion and search. As a result, when we decided to implement additional operations it became apparent to us that it would be challenging to implementing new functionality due to the limitations that is imposed by the HTML and the TypeScript interface, which only supported the three primary operations or other operations that already existed and were unique to a specific data structure. Consistently, adding new buttons to the canvas is currently not possible, at least not without refactoring the HTML files.

5.5 Testing

We faced difficulties when researching testing due to the fact that the framework is (when compiled) built using HTML and JavaScript. Most of the online information on the subject is dedicated to website components built using the React library, which is not applicable to our project and makes finding relevant information difficult

due to the similarity of search terms. As a part of the limited resources we found, the library that we chose to simulate a browser, `jsdom`, has somewhat incomplete documentation. There exists a general overview of the different features of the library, but unlike many other software libraries, there is no list of the classes or methods included in the library. This, in combination with the other factors, also hindered the development.

One reasonable improvement to the conditions to implement tests would be modifying the structure to adhere more to a typical Model-View-Controller (MVC) pattern, which is common in object-oriented programming standards [39]. This would make it possible to implement tests on only the Model and perhaps on the Controller without having to use the View at all. However, this comes with its own set of problems, as discussed in Section 5.2.2.

It is also possible that there are alternatives to `jsdom` with better documentation that more easily might have provided a browser environment. One such example is Playwright [40], which seems to have more extensive documentation. We unfortunately discovered this library too late to use it in our project, but based on our limited testing, Playwright would allow us to conduct E2E testing. However, it would not help with the remaining issues of implementing unit or integration tests.

Of course, one of the primary difficulties we faced regarding testing was a lack of experience writing tests for websites, combined with the time limit of the project. Many of the concepts that we found difficult to understand during the installation and modification of the settings for the packages are naturally easier for a developer more experienced within this specific field. This fact also made the debugging process very time-consuming since we had limited knowledge to debug from.

5.6 Differences to Related Work

Comparing our implementation to the other existing implementations that we briefly mentioned in Section 1.4, we can see that they differ in terms of features and the way they have chosen to implement the different data structures and algorithms. An example of this would be the implementation created by the team from NUS [5], which contains features ranging from arrays to trees and even reductions. If we compare our sorting implementations, they feature arrays that display as bar charts. In addition, they have an extra feature that allows you to toggle on and off depth when using merge sort and quicksort, whereas we have this feature hard-coded to show more visual clarity. They also have a feature we considered implementing, which is to have pseudocode displayed alongside the algorithm while it is sorting (described in Section 5.7.2). Another difference is the variant of quicksort used, as they use a version of quicksort with only one pointer instead of an upper and lower pointer, differing from both our and the original implementation [41]. If we compare our linked list implementations, we can also see a couple of differences, such as that their linked list connections are more dynamic and that they have pointers to both the head and tail, while we only point at the head.

Another visualization tool is the one made by the CS 1332 team and Rodrigo Pontes [7]. This tool also features many algorithms. They have the four sorting algorithms featured in our work as well as some others, such as radix and heap sort. The tool also features some intentionally impractical sorting algorithms such as bogosort and miracle sort. Their solution is similar to the above in that it has many of the same features such as pseudocode, but a stark difference is that it uses the same in-place standard type quicksort that we use, with two pointers. Looking at their linked list implementation, it's similar to the NUS implementation; however, the tail pointer is toggleable. They also have variants of linked lists such as doubly and circular linked lists.

The CS 1332 site is a fork of an older work by David Galles, which is published at the University of San Francisco [4]. This implementation uses a minimal user interface, and it is the inspiration for the original library by Peter Ljunglöf that this project is based on [3]. The main difference between Galles' and our work, apart from the number of algorithms, is that you can't customize the list that will be sorted, and that his array is displayed with a bar chart.

5.7 Future Work

We are happy with what we have accomplished with our framework. With that said, there are many things that could further improve the extendability and usability of it. If there were more time for the project or if someone would like to continue working on the framework, we present in this section some ideas on future improvements.

5.7.1 Separate Engine from Visualizations

We believe that it should be possible to separate the algorithm logic from the visualizations, despite our failed attempts. Some minor separation was made between the SVG elements and `Engine`, but this was not finished either. Accomplishing this would move the framework closer to following the MVC pattern.

To fully separate `Engine` from the visualizations, we believe that the problem should be approached with more planning and structure before starting. One possible solution would be to implement a data structure like we did with linked lists and to then send that into a new class that is only responsible for the visualization of the data structure, representing an MVC View. Combined with a refactoring of how the elements are animated and access their size variable, which currently is either tightly coupled to `Engine` or requires the size to be passed into each method as a parameter.

5.7.2 Pseudocode

The addition of pseudocode was something we considered implementing for this project, but because of time constraints, this was never done. We believe that if the code that is executed is shown to the user together with the animations of the

data structures, the learning process could be improved. Adding this can be done in many different ways. Our approach would be to add it directly to `Engine`, preferably as a helper class that is initialized in the constructor. Each algorithm would then become responsible for changing what code is visible and what line is highlighted.

5.7.3 Additional Data Structures

Adding additional data structures would be beneficial for the framework, because the more data structures implemented, the more useful the framework becomes. If we had time, we would have started implementing the rest of the algorithms from the Chalmers course, and afterwards continued with other useful algorithms. Specifically, we would prioritize implementing hash maps or graph algorithms because we believe they would be more beneficial to add than further sorting algorithms.

Another alternative is to extend the data structures that have already been added. One option is to add a selector that allows for different pivot strategies in quicksort. Although our current implementation uses the middle element as the pivot, the course presents several other alternatives, including using the first element, selecting a random element, choosing the median of three indices, or applying a combination of these strategies based on the size of the list.

5.8 Usage of AI

During our research, we have been careful with the usage of AI tools. During the writing of this paper, we have used AI as a helper tool, primarily for minor tasks such as improving grammar, fixing language, and assisting with formatting. Furthermore, we have utilized AI tools for finding sources such as Scopus AI. Even then, we have analyzed these sources ourselves and have not relied on AI for the interpretation of the research papers. In terms of coding, we have used AI as a debugging tool and have also utilized it as a brainstorming assistant.

6

Conclusion

Through thoughtful refactoring and the integration of the modern tools TypeScript and Webpack, we improved both the developer experience and the extendability of the framework. Type safety and clearer error messaging now enable faster development and easier debugging. By partly decoupling core logic from the data structure functionality and introducing reusable structures, we reduced complexity and duplication. These changes not only streamlined the current codebase but also laid a solid foundation for future scalability and maintainability.

Recurring challenges across our project were a lack of separation between logic and visualization; together with limited prior experience in modern web testing practices, it made testing harder. A better-structured architecture design and the use of more widely supported frameworks could have significantly improved testability and modularity. We should have prioritized early architectural planning and the use of tools with strong community and documentation support.

The implemented data structures are designed to be recognizable at a glance, with visual representations reflecting the structural characteristics of the corresponding algorithm. This clarity helps users quickly identify each structure and grasp its role visually. As we see it, this makes the visualizations more useful for learning and exploration, offering a clear and intuitive experience.

Bibliography

[1] Chalmers University of Technology, *Course syllabus for Data structures and algorithms*, Feb. 2024. [Online]. Available: https://chalmers.se/en/education/your-studies/find-course-and-programme-syllabi/course-syllabus/DAT525.

[2] A Kurniawati, NH Akbar, and D Prasetyo, "Visual Learning on Mobile Phone for Introduction Basic Programming in Vocational High School," in *2018 International Conference on Computer Engineering, Network and Intelligent Multimedia (CENIM)*, Surabaya, Indonesia: Institute of Electrical and Electronics Engineers, Jul. 2018, pp. 186–191. DOI: 10.1109/CENIM.2018.8710873.

[3] P Ljunglöf, *Visualizations of data structures and algorithms*, 2024. [Online]. Available: https://chalmersgu-data-structure-courses.github.io/dsvis.

[4] D Galles, *Data Structure Visualization*, 2011. [Online]. Available: https://www.cs.usfca.edu/~galles/visualization.

[5] S Halim, ZC Koh, VBH Loh, and F Halim, "Learning Algorithms with Unified and Interactive Web-Based Visualization," *Olympiads in Informatics*, vol. 6, pp. 53–68, 2012.

[6] S Halim, "VisuAlgo – Visualising Data Structures and Algorithms Through Animation," *Olympiads in Informatics*, vol. 9, pp. 243–245, 2015. DOI: 10.15388/IOI.2015.20.

[7] RDL Pontes, M de los Reyes, and A McQuilkin, *CS 1332 Data Structures and Algorithms Visualizations*, 2025. [Online]. Available: https://csvistool.com.

[8] GitHub Staff, *Octoverse: AI leads Python to top language as the number of global developers surges*, Oct. 2024. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-2024.

[9] ECMA, *Standard ECMA-262: ECMAScript Language Specification*, 3rd ed. Dec. 1999. [Online]. Available: https://ecma-international.org/publications-and-standards/standards/ecma-262.

[10] J Park, S An, W Shin, Y Sim, and S Ryu, "JSTAR: JavaScript Specification Type Analyzer using Refinement," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Melbourne, Australia: Institute of Electrical and Electronics Engineers, 2021, pp. 606–616. DOI: 10.1109/ASE51524.2021.9678781.

[11] B Peterson and B Vogel, "Prototyping the Internet of Things with Web Technologies: Is It Easy?" In *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, Athens,

Greece: Institute of Electrical and Electronics Engineers, Oct. 2018, pp. 518–522. DOI: 10.1109/PERCOMW.2018.8480268.

[12] S Kushwah, C Bansal, S Rathore, V Kate, D Bargal, and D Vishwakarma, "TypeScript: An Open-Source Programming Language with Options for Robust Development and Large-Scale Applications," in *2024 International Conference on Advances in Computing Research on Science Engineering and Technology (ACROSET)*, Indore, India: Institute of Electrical and Electronics Engineers, 2024, pp. 1–5. DOI: 10.1109/ACROSET62108.2024.10743426.

[13] Microsoft, *TypeScript Documentation*. [Online]. Available: https://typescriptlang.org/docs.

[14] J Bogner and M Merkel, "To Type or Not to Type? A Systematic Comparison of the Software Quality of JavaScript and TypeScript Applications on GitHub," in *MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories*, Pittsburgh, USA: Association for Computing Machinery, 2022, pp. 658–669. DOI: 10.1145/3524842.3528454.

[15] F Zammetti, *Modern Full-Stack Development: Using TypeScript, React, Node.js, Webpack, and Docker*, 1st ed. Apress Media LLC, Mar. 2020. DOI: 10.1007/978-1-4842-5738-8.

[16] M Bouzid, *Webpack for Beginners: Your Step-by-Step Guide to Learning Webpack 4*, 1st ed. Apress Media LLC, Jun. 2020. DOI: 10.1007/978-1-4842-5896-5.

[17] J Rack and CA Staicu, "Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications," in *CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, Copenhagen, Denmark: Association for Computing Machinery, Nov. 2023, pp. 3198–3212. DOI: 10.1145/3576915.3623140.

[18] S Scalabrino, M Linares-Vásquez, R Oliveto, and D Poshyvanyk, "A comprehensive model for code readability," *Journal of Software: Evolution and Process*, vol. 30, no. 6, Jun. 2018. DOI: 10.1002/SMR.1958.

[19] S Scalabrino, G Bavota, C Vendome, M Linares-Vásquez, D Poshyvanyk, and R Oliveto, "Automatically assessing code understandability: How far are we?" In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, USA: Institute of Electrical and Electronics Engineers, Nov. 2017, pp. 417–427. DOI: 10.1109/ASE.2017.8115654.

[20] L Lavazza, S Morasca, and M Gatto, "An empirical study on software understandability and its dependence on code characteristics," *Empirical Software Engineering*, vol. 28, no. 6, pp. 1–24, Nov. 2023. DOI: 10.1007/S10664-023-10396-7.

[21] CEC Dantas and MA Maia, "Readability and Understandability Scores for Snippet Assessment: an Exploratory Study," in *9th Workshop on Software Visualization, Evolution and Maintenance*, Joinville, Brazil: Brazilian Computer Society, Aug. 2021, pp. 46–50. DOI: 10.5753/VEM.2021.17217.

[22] D Oliveira, R Bruno, F Madeiral, and F Castor, "Evaluating Code Readability and Legibility: An Examination of Human-centric Studies," *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 348–359, Sep. 2020. DOI: 10.1109/ICSME46990.2020.00041.

[23] RPL Buse and WR Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, 2010. DOI: 10.1109/TSE.2009.70.

[24] GA Campbell, "Cognitive complexity: An overview and evaluation," in *TechDebt '18: Proceedings of the 2018 International Conference on Technical Debt*, Gothenburg, Sweden: Association for Computing Machinery, May 2018, pp. 57–58. DOI: 10.1145/3194164.3194186.

[25] TJ McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976. DOI: 10.1109/TSE.1976.233837.

[26] M Fowler, *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison-Wesley, 1999.

[27] J Bräuer, R Plösch, M Saft, and C Körner, "Measuring object-oriented design principles: The results of focus group-based research," *Journal of Systems and Software*, vol. 140, pp. 74–90, Jun. 2018. DOI: 10.1016/J.JSS.2018.03.002.

[28] SCB de Souza, N Anquetil, and KM de Oliveira, "A study of the documentation essential to software maintenance," in *SIGDOC '05: Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, New York, USA: Association for Computing Machinery, Sep. 2005, pp. 68–75. DOI: 10.1145/1085313.1085331.

[29] D Kerschbaumer, C Schatz, T Ruprechter, C Gütl, and A Steinmaurer, "Do Comments Matter? Investigating Students' Source Code Comment Behaviour and Its Relation to Academic Success in a CS1 Course," in *Futureproofing Engineering Education for Global Responsibility: Proceedings of the 27th International Conference on Interactive Collaborative Learning (ICL2024)*, vol. 2, Tallinn, Estonia: Springer, Mar. 2025, pp. 519–530. DOI: 10.1007/978-3-031-85649-5_51.

[30] JSDoc contributors, *Use JSDoc*, 2011. [Online]. Available: https://jsdoc.app.

[31] P Nawar, *JSDoc: The ultimate guide to documenting your JavaScript code*, Mar. 2023. [Online]. Available: https://precodes.hashnode.dev/jsdoc-the-ultimate-guide-to-documenting-your-javascript-code.

[32] Y Fang, F Tian, W Wang, and W Li, "Effective computer software unit testing: thinking about unit test-driven development (UTDD) mode," in *Fourth International Conference on Signal Processing and Computer Science (SPCS 2023)*, vol. 12970, Guilin, China: SPIE, Dec. 2023, pp. 721–726. DOI: 10.1117/12.3012428.

[33] S Banitaan, M Alenezi, K Nygard, and K Magel, "Towards Test Focus Selection for Integration Testing Using Method Level Software Metrics," in *2013 10th International Conference on Information Technology: New Generations*, Las Vegas, USA: Institute of Electrical and Electronics Engineers, 2013, pp. 343–348. DOI: 10.1109/ITNG.2013.55.

[34] W Chen, H Cao, and X Blanc, "An Improving Approach for DOM-Based Web Test Suite Repair," in *Web Engineering*, vol. 12706, Biarritz, France: Springer, 2021, pp. 372–387. DOI: 10.1007/978-3-030-74296-6_29.

[35] IG de Wolff and J Hage, "Refining types using type guards in TypeScript," in *PEPM 2017: Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, Paris, France: Association for Computing Machinery, Jan. 2017, pp. 111–122. DOI: 10.1145/3018882.3018887.

[36] P Ljunglöf, C Sattler, and N Smallbone, *Course book in Data structures and algorithms*. [Online]. Available: https://chalmersgu-data-structure-courses.github.io/OpenDSA.

[37] D Denicola, *jsdom: A JavaScript implementation of various web standards, for use with Node.js*. [Online]. Available: https://github.com/jsdom/jsdom.

[38] K Kabra, *ts-jest: A Jest transformer with source map support that lets you use Jest to test projects written in TypeScript*. [Online]. Available: https://github.com/kulshekhar/ts-jest.

[39] RC Martin, "Design Principles and Design Patterns," *Object Mentor*, vol. 1, no. 34, 2000.

[40] Microsoft, *Playwright: Fast and reliable end-to-end testing for modern web apps*, 2025. [Online]. Available: https://playwright.dev.

[41] CAR Hoare, "Algorithm 64: Quicksort," *Communications of the ACM*, vol. 4, no. 7, p. 321, Jul. 1961. DOI: 10.1145/366622.366644.

A

Analysis source code

This appendix contains the source code of the Python script utilized for the code comprehension analysis described in Section 3.6.

```
import re



### SETTINGS
BASE = "~/dsvis/"
FILES = [
    "src/engine.ts"
]
DEBUG = False



# ex: ["//", "//", "&&", "&&", "&&", "//"] -> ["//", "&&", "//"]
def reduceGroups(groups):
    reduced = []
    for i in range(len(groups)):
        if i == 0:
            reduced.append(groups[i])
        elif groups[i] != groups[i-1]:
            reduced.append(groups[i])
    return reduced

def cc(lines):
    nestingLevel = 1
    tally = 0
    binaryOpGroups = []
    ternaryC = 0
    def debug(message):
        if not DEBUG:
            return
        print((nestingLevel - 1) * "\t" + message)

    for lineNumber, line in enumerate(lines):
        if re.search(r"^ *//|^ *\*|^ */\*", line):
            # skip comments
            continue


        # count sequential binary operators
```

```
41          binaryOps = re.findall(r"&&|\|\|", line)
42          binaryOpGroups.extend(binaryOps)
43          # keep searching if line ends with a binary operator
44          if re.search(r"(&&|\|\|)$", line):
45              debug(f"{line.strip()} [binary ops...]")
46              continue
47          if binaryOpGroups:
48              sequentialBinaryOpC = len(reduceGroups(binaryOpGroups))
49              tally += sequentialBinaryOpC
50              debug(f"{line.strip()} [binary ops, +{
                    sequentialBinaryOpC}]")
51              binaryOpGroups = []
52
53
54          # count ternaries
55          ternaryC += len(re.findall(r"\? .+? :", line))
56          # keep searching if line ends with a ternary operator
57          if re.search(r"\? .+ :$", line):
58              debug(f"{line.strip()} [ternaries...]")
59              continue
60          if ternaryC:
61              # ternaryTally = nestingLevel + (nestingLevel + 1) ...
                    + (nestingLevel + ternaryC - 1)
62              ternaryTally = 0
63              while ternaryC > 0:
64                  ternaryTally += nestingLevel + ternaryC - 1
65                  ternaryC -= 1
66              tally += ternaryTally
67              debug(f"{line.strip()} [ternary nest, +{ternaryTally}]"
                    )
68              ternaryC = 0
69
70
71          # decrease nesting level
72          if re.search(r"^ *\}", line):
73              nestingLevel -= 1
74              if nestingLevel < 1:
75                  nestingLevel = 1
76              debug(f"{line.strip()}")
77
78
79          # anonymous function
80          if re.search(r"\(.*\) => \{", line):
81              debug(f"{line.strip()} [anon function]")
82              nestingLevel += 1
83
84          elif re.search(r"else if \(", line) \
85            or re.search(r"else \{", line):
86              tally += 1
87              debug(f"{line.strip()} [simple nest, +1]")
88              nestingLevel += 1
89
90          elif re.search(r"(if|switch|for|while|catch) \(", line) \
91            or re.search(r"do \{", line):
92              tally += nestingLevel
93              debug(f"{line.strip()} [nest, +{nestingLevel}]")
```

```
94
95              # skip nesting level increase on braceless if
                    statements
96              nextLine = lines[lineNumber + 1]
97              if re.search(r";", line) \
98              or (re.search(r";", nextLine) and not re.search(r"\{",
                    line)):
99                  continue
100             nestingLevel += 1
101
102     return tally
103
104
105 def bnw(lines):
106     numLines = len(lines)
107     charCounts = []
108     indentationCounts = []
109     identifierCounts = []
110     keywordCounts = []
111     numberCounts = []
112     numComments = 0
113     numPeriods = 0
114     numCommas = 0
115     numSpaces = 0
116     numParentheses = 0
117     numArithmeticOps = 0
118     numComparisonOps = 0
119     numAssignments = 0
120     numBranches = 0
121     numLoops = 0
122     numBlankLines = 0
123     characterOccurrences = {}
124     identifierOccurrences = {}
125
126     for line in lines:
127         line = line.rstrip() # ignore trailing whitespace
128
129         # line length
130         charCounts.append(len(line))
131
132         # blank lines
133         if not line:
134             numBlankLines += 1
135             continue
136
137         # indentation
138         indentation = re.search(r"^ *", line).group(0)
139         indentationCounts.append(len(indentation))
140         line = line.lstrip() # remove indentation
141
142         # characters, periods, commas, spaces, parentheses
143         for char in line:
144             if char == ".":
145                 numPeriods += 1
146             elif char == ",":
147                 numCommas += 1
```

```
148             elif char == " ":
149                 numSpaces += 1
150                 continue # skip spaces
151             elif char in "()[]{}<>":
152                 numParentheses += 1
153             characterOccurrences[char] = characterOccurrences.get(
                    char, 0) + 1
154 
155         # comments
156         if re.search(r"//|^ *\*|^ */\*", line):
157             numComments += 1
158 
159         # names
160         line = re.sub(r"//.*$", "", line) # remove comments
161         line = re.sub(r"^ *(\*|/\*)", "", line) # remove block
                comments
162         line = re.sub(r"\".*?\"", "", line) # remove strings
163         line = re.sub(r"'.*?'", "", line)
164         line = re.sub(r"/.*?/.?", "", line) # remove regex strings
165         namesInLine = re.findall(r"\b[a-zA-Z_\-$][a-zA-Z0-9_\-$]*\b
                ", line)
166 
167         # keywords, branches, loops
168         KEYWORDS = [
169             "any", "as", "boolean", "break", "case", "catch", "
                    class", "const", "constructor", "continue", "
                    debugger", "declare",
170             "default", "delete", "do", "else", "enum", "export", "
                    extends", "false", "finally", "for", "from", "
                    function", "get",
171             "if", "implements", "import", "in", "instanceof", "
                    interface", "let", "module", "new", "null", "number"
                    , "of",
172             "package", "private", "protected", "public", "require",
                     "return", "set", "static", "string", "super", "
                    switch",
173             "symbol", "this", "throw", "true", "try", "type", "
                    typeof", "var", "void", "while", "with", "yield"
174         ]
175         keywords = [name for name in namesInLine if name in
                KEYWORDS]
176         keywordCounts.append(len(keywords))
177         for keyword in keywords:
178             if keyword in ["if", "else", "switch"]:
179                 numBranches += 1
180             elif keyword in ["for", "while", "do"]:
181                 numLoops += 1
182         # else if and do while fix
183         numBranches -= len(re.findall(r"else if", line))
184         numLoops -= len(re.findall(r"do while", line))
185 
186         # identifiers
187         identifiers = [name for name in namesInLine if name not in
                KEYWORDS]
188         identifierCounts.append(len(identifiers))
189         for identifier in identifiers:
```

```
190             identifierOccurrences[identifier] =
                    identifierOccurrences.get(identifier, 0) + 1
191
192         # numbers
193         numbers = re.findall(r"\b\d+(\.\d+)?\b", line)
194         numberCounts.append(len(numbers))
195
196         # arithmetic ops
197         arithmeticOps = re.findall(r"\s
                (\+|\-|\*|\/|\%|\*\*|\+\+|\-\-|<<|>>|>>>|&|\||\^)\s",
                line)
198         numArithmeticOps += len(arithmeticOps)
199
200         # comparison ops
201         comparisonOps = re.findall(r"\s(==|!=|===|!==|<=|>=|<|>|
                instanceof)\s", line)
202         numComparisonOps += len(comparisonOps)
203
204         # assignments
205         assignments = re.findall(r"\s
                (=|\+=|-=|\*=|\/=|%=|&=|\|=|\^=|<<=|>>=|>>>=|\+\+|--)\s"
                , line)
206         numAssignments += len(assignments)
207
208     print(f"Most common character: {max(characterOccurrences, key=
            characterOccurrences.get)}")
209     print(f"Most common identifier: {max(identifierOccurrences, key
            =identifierOccurrences.get)}")
210
211     results = [
212         ("avg line length", sum(charCounts) / numLines),
213         ("max line length", max(charCounts)),
214         ("avg # identifiers", sum(identifierCounts) / numLines),
215         ("max # identifiers", max(identifierCounts)),
216         ("avg identifier length", sum([len(name) for name in
                identifierOccurrences.keys()]) / len(
                identifierOccurrences)),
217         ("max identifier length", max([len(name) for name in
                identifierOccurrences.keys()])),
218         ("avg indentation", sum(indentationCounts) / numLines),
219         ("max indentation", max(indentationCounts)),
220         ("avg # keywords", sum(keywordCounts) / numLines),
221         ("max # keywords", max(keywordCounts)),
222         ("avg # numbers", sum(numberCounts) / numLines),
223         ("max # numbers", max(numberCounts)),
224         ("avg # comments", numComments / numLines),
225         ("avg # periods", numPeriods / numLines),
226         ("avg # commas", numCommas / numLines),
227         ("avg # spaces", numSpaces / numLines),
228         ("avg # parentheses", numParentheses / numLines),
229         ("avg # arithmetic ops", numArithmeticOps / numLines),
230         ("avg # comparison ops", numComparisonOps / numLines),
231         ("avg # assignments", numAssignments / numLines),
232         ("avg # branches", numBranches / numLines),
233         ("avg # loops", numLoops / numLines),
234         ("avg # blank lines", numBlankLines / numLines),
```

```
235         ("max occurrences of a character", max(characterOccurrences
                .values())),
236         ("max occurrences of an identifier", max(
                identifierOccurrences.values()))
237     ]
238 
239     return [(name, round(value, 4)) for name, value in results]
240 
241 lines = []
242 for path in FILES:
243     path = BASE + path
244     with open(path, "r") as f:
245         lines.extend(f.readlines())
246 
247 print(f"CC: {cc(lines)}\nB&W: ")
248 for name, value in bnw(lines):
249     print(f"\t{name}: {value}")
```