TU Delft
Delft
University of
Technology

CodeStories

Roelof Sol, Salim Salmi, Maarten van Beek

June 19, 2015

Coach | Prof.dr.Elmar Eisemann
Client | Dr. Anna Vilanova
Bachelor Coordinators | Dr.ir. Felienne Hermans & Dr. Martha Larson

Abstract

Understanding code can be cumbersome. CodeStories strives to provide the tools to make code more understandable by creating a tight coupling between code execution and explanatory elements such as visualizations. Currently CodeStories only supports JavaScript, but extending it with different languages would significantly increase its range. During our research phase we have found that solutions exist that partly solve the problem, though none of the solutions are extensive.

CodeStories relies on several third party projects. The main projects are D3 for the visualizations, JS-Interpreter, to make the narratives dynamic and AngularJS as the front-end framework. As is common, the Karma plus Jasmine combination has been used as a testing framework.

Some new concepts had to be introduced to keep the application flexible. The CAST has been introduced as a file/folder tree extended with the Abstract Syntax Tree for JavaScript files. Narrative items are the atoms a CodeStory is build up from and VObjects are the higher level objects that represent visualizations. For the user interface we have chosen an approach inspired by interactive storytelling. The end-product fulfills all core features and a little extra. Improvements can be made mainly in the user experience, language support, and the extensiveness of the VObject library.

Preface

This is the report for the TU Delft Computer Science Bachelor project, CodeStories. We would like thank Elmar Eisseman and Anna Vilanova from the TU Delft Computer Graphics and Visualization group for their guidance and the opportunity to develop this project. We would also like to thank Dennis Bijlsma from the Software Improvement Group for analyzing our code quality and his feedback. This document elaborates on our research, implementation and the conclusions that we have drawn.

Have you ever been lost in someones code, only to conclude with "Of course! I wish someone had shown me this sooner..."

Roelof Sol, Salim Salmi and Maarten van Beek

Contents

I Orientation **6**

1 Assignment **7**
1.1 Problem description . . . 7
1.2 User Stories . . . 8
1.3 Application definition . . . 10

2 Research **11**
2.1 Existing solutions . . . 11
2.1.1 code-guide . . . 12
2.1.2 Python Tutor . . . 13
2.1.3 VisuAlgo . . . 14
2.1.4 Existing documentation frameworks . . . 14
2.1.5 Static code visualizations . . . 15
2.1.6 Interactive storytelling . . . 15
2.2 Technologies and libraries . . . 15
2.2.1 Code editor . . . 15
2.2.2 Javascript interpreter and debugger . . . 16
2.2.3 Visualization libraries . . . 17
2.3 Technical Limitations . . . 18

3 Requirements **19**
3.1 Functional requirements . . . 19
3.2 Non-functional requirements . . . 21

4 Project Planning **22**
4.1 Developer tools . . . 22
4.2 Scrum . . . 22
4.3 Issue tracking . . . 22

4.4 Testing . . . 23
4.5 Time Planning . . . 23

II Implementation 25

5 Concepts Introduction 26
5.1 CAST . . . 26
5.2 Narrative . . . 27
5.2.1 Narrative items . . . 27
5.2.2 Item Hooks . . . 29
5.3 VObject . . . 29
5.4 Narrator . . . 29

6 Relations 31

7 User experience & user interface 33
7.1 Viewer . . . 33
7.2 Writer . . . 34

8 Angular design patterns 37
8.1 Use of state . . . 37
8.2 Modules . . . 38

III Reflection 40

9 Fulfillment 41
9.1 Requirements . . . 41
9.1.1 Functional requirements . . . 41
9.1.2 Non-functional requirements . . . 43
9.2 User experience & user interface . . . 43

10 Usage Example 44
10.1 Writing the narrative . . . 44
10.1.1 The initialization code . . . 44
10.1.2 Visualization Object . . . 47
10.1.3 Adding the visuals . . . 49
10.2 Viewing the narrative . . . 50

11 Recommendations and further Improvements **52**

12 Conclusion **54**

IV Appendix **55**

A Glossary **56**

B Feedback **58**
B.1 First SIG Feedback (Dutch) 58
B.2 Second SIG Feedback (Dutch) 59

C Original project description **61**
C.1 Project description 61
C.2 The project goal 61
C.3 Company description 62
C.4 Auxiliary information 62
C.5 Basic Product Requirements 62

Introduction

The current process of understanding software projects can be a cumbersome practice, especially when complex data structures and algorithms are being used. At these moments a personal guide who can lead you through the code, point you at the relevant details, give you a visual representation of complex concepts and provide you with some extra elaboration where necessary, can be a huge benefit. This Bachelor project strives to provide tools for anyone to create a digital counterpart of this person for his or her project, a code story. We strive to make documentation more enjoyable through interactive and visual storytelling, a practice in which a developer can give a user a guided tour through the project layout and (core) functionality. The user can decide with which information he or she is provided and make it possible to gradually acquaint him- or herself with a project.
This document is divided in three parts. The first four chapters will describe the orientation phase. *Chapter 1* will elaborate on the assignment definition and a brief exploration of its potential user base. *Chapter 2* will elaborate on our research. It will explore some existing solutions and technologies and conclude with some limitations following from it. *Chapter 3* will describe this project requirements and *Chapter 4* elaborates on the time planning. The next four chapters will describe the implementation. In *Chapter 5* we will introduce the concepts we will use, *Chapter 6* describes the product structure in UML and *chapter 7* deals with the user experience. *Chapter 8* describes some design patterns used with our framework of choice, Angular. The document concludes with a reflection. *Chapter 9* describes the realization of the requirements, *Chapter 10* gives a example of how the system can be used to visualize an algorithm. *Chapter 11* proposes future improvements and *Chapter 12* draws a conclusion to this document.

Part I

Orientation

Chapter 1

Assignment

This chapter will focus on an exploration of the problem associated with the assignment. First a brief description of the problem is given. Next the target audience is defined and the details of the problem will be defined through a set of user stories. Lastly the application proposed to deal with the problem is defined.

1.1 Problem description

When confronted with a project or algorithm that one does not understand there are several ways to go about discovering the inner workings of that code. One way is reading the code itself to try and get a mental image of how it would execute. This can get fairly confusing when the algorithm is long or complex. Pairing the knowledge obtained from examining the code with the input and output can result in a clearer image, but this would require an understanding of the data structure and ability to create several mock inputs and compare the results.

The ideal way would be to have somebody who already understands the algorithm to explain the parts that are not understood. However in a real world scenario people can often not have the time to do so, especially when a lot of people want to know a multitude of different things.

The project is about providing coders with the possibility to easily produce an informative experience along with their code and algorithms. A developer is able to write stories for their code and is able to generate visualizations to exemplify execution and helps the reader in understanding the

algorithm. These stories contain for example: a video presentation, sound, text, and custom visualizations of data structures. This is done for the Computer Graphics and Visualization department of TU Delft.

1.2 User Stories

Alice the prof

Alice is a professor at the university in Delft specializing in algorithmics. Next to her work in this field she also teaches students the fundamentals, theory and science behind algorithms. For this reason she often needs to write simple programs to demonstrate these algorithms in class. However she found that often there is confusion when explaining code to her students as the code and the execution are disjoint. For this reason she wants to be able to go through her code in a step by step manner and display the effects of each step.

Bob the new developer

Bob wants to know the inner workings of a software project so he can help add a new feature or fix a bug. He finds himself lost in the code of someone else and wants to quickly gain insight in the project layout and functionality. There are two places he could start reading. Read the code or read the docs. Interactive documentation has been added to the code, so Bob quickly scans the code to find the sections that are relevant for his contributions. Through visualizations Bob is made familiar with the used concepts, and Bob executes harder parts of the code multiple times with different inputs to get familiar with it.

Carol the Computer Science teacher

Carol is a Computer Science teacher and wants to teach her students about BubbleSort. She had a thought on how this could be made more intuitively for her students if they can see the code in action by swapping bars in a bar graph. She loads her BubbleSort code into the program. She adds code to display the original array. She adds code at the swap elements to swap

the location of the visual elements in the canvas. Eventually she shows her students the execution of the algorithm and lets them play around with it.

Dane the interpreter developer

Dane has written a Javascript interpreter. He wants people to quickly see how it works. He codes up a visual display for a stack, a venn-diagram to represent the variables and their scope. At every type of step, the variable used are highlighted and moved to represent the step.

Eve the teacher

Eve wants to show her students how to do matrix multiplication. She takes a simple double nested loop code and loads this into our program. She codes the matrices to be displayed one left, the other on top and an empty matrix in the middle. At the multiplication step two arrows from the original matrices point to the empty spot. Her students can watch multiplication in action and play around with the input values.

A few short use cases

- An algorithms teacher wants to visualize some algorithms for his students so that it is easier to explain
- A student wants step through a visualization of an algorithm so that he understands it better
- A developer wants to show his fellow developers quickly how his code solves a problem in the interest of saving time
- A developer wants to display a certain data structure so that he can show this to his coworkers.
- A developer wants to gain insight in a specific part an algorithm
- A student trying to learn how to code wants to have a visual representation and effect of the simple functions, statements and data structures.

1.3 Application definition

This application serves as a way to help developers introduce their projects in a guided and visual manner.

For convenience and clarity we will call the developer who wants to explain his project and or code the Writer and the user who would want to see the explanation we will call the Viewer

With the aid of this application, the Writer can create a narrative for his or her project. Borrowed from the actual definition of the word: in short a narrative is a sequence of connected events, presented in a sequence of written words, and/or in a sequence of (moving) pictures. The writer can create these narratives by adding elements called narrative items to a part of a project. An example of a narrative item could be a textual explanation, an illustrative image or a video presentation. On top of this, parts of the code itself can be narrated in this manner. The Writer can select a step in the code and provide it with some explanation in the form of these narrative items. Much like a debugger, the Viewer can then go through the code in a step by step manner. However this time the Writer can not only use text or images but can also access the variables in the code to provide the Viewer with an animation based on the state of the variables during execution. To access animations the Writer manipulates visual objects that are either given or self written. When these objects reach their updated state they will display the new state of the animation. An example would be a bar chart. The Writer can specify a bar chart should be used to visualize a certain part of the code, and than make calls to the bar chart object at the desired places in the code.

The Writer can make multiple narratives for his project and can decide for himself or herself on what level of abstraction. Lastly a Writer can link from within a narrative to another narrative which allows for reusing narratives.

To conclude, the aim is to allow the Writer to convey meaning through dynamic story telling by easily being able to couple story telling elements with his code.

Chapter 2

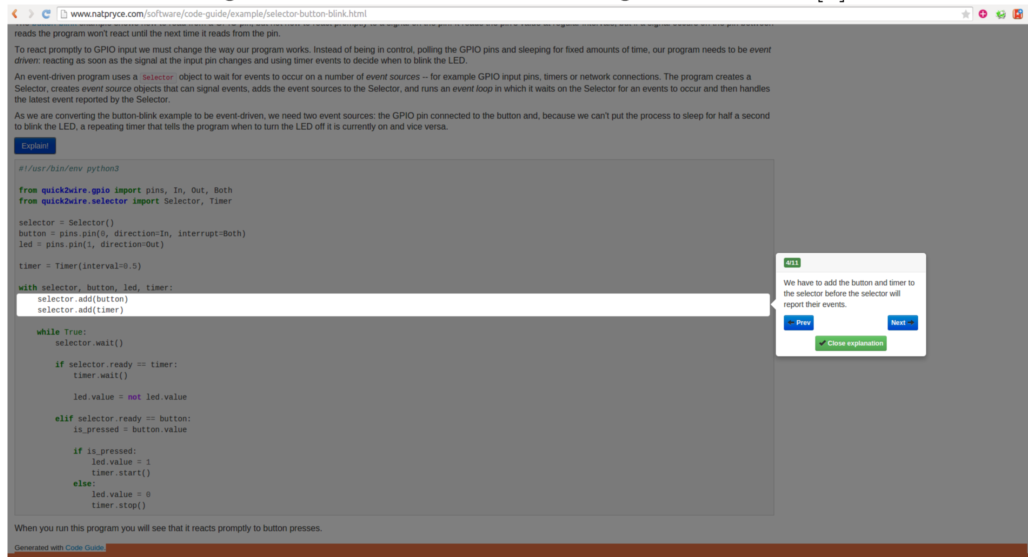
Research

2.1 Existing solutions

We have researched some of the existing solutions for interactive code documentation, interactive storytelling and algorithm visualization. Next to existing software, we have also researched technologies that are related to our application. This chapter lists some notable examples.

2.1.1 code-guide

Figure 2.1: Screenshot of code-guide at work [1]

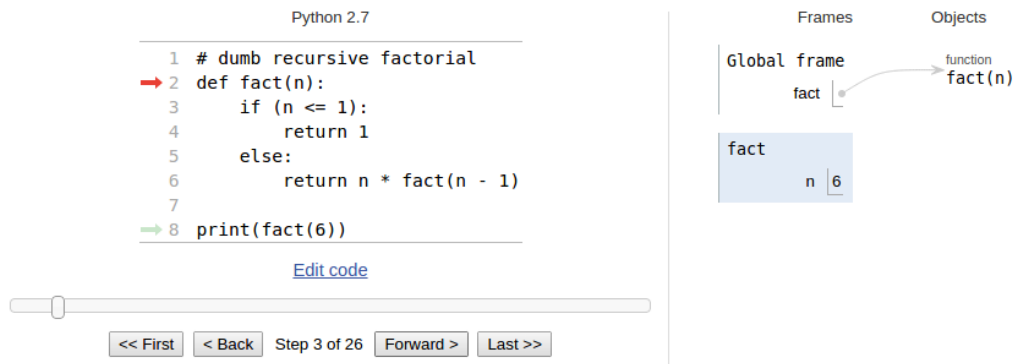


.

Code guide is a solution to generate interactive documentation from python files. It works by writing a string of documentation and linking it to one or more lines of code. When the interactive documentation is started, the lines of code for which documentation is available are highlighted in sequence and its documentation is shown in a text bubble. Highlighting the relevant code is a useful visual element.

2.1.2 Python Tutor

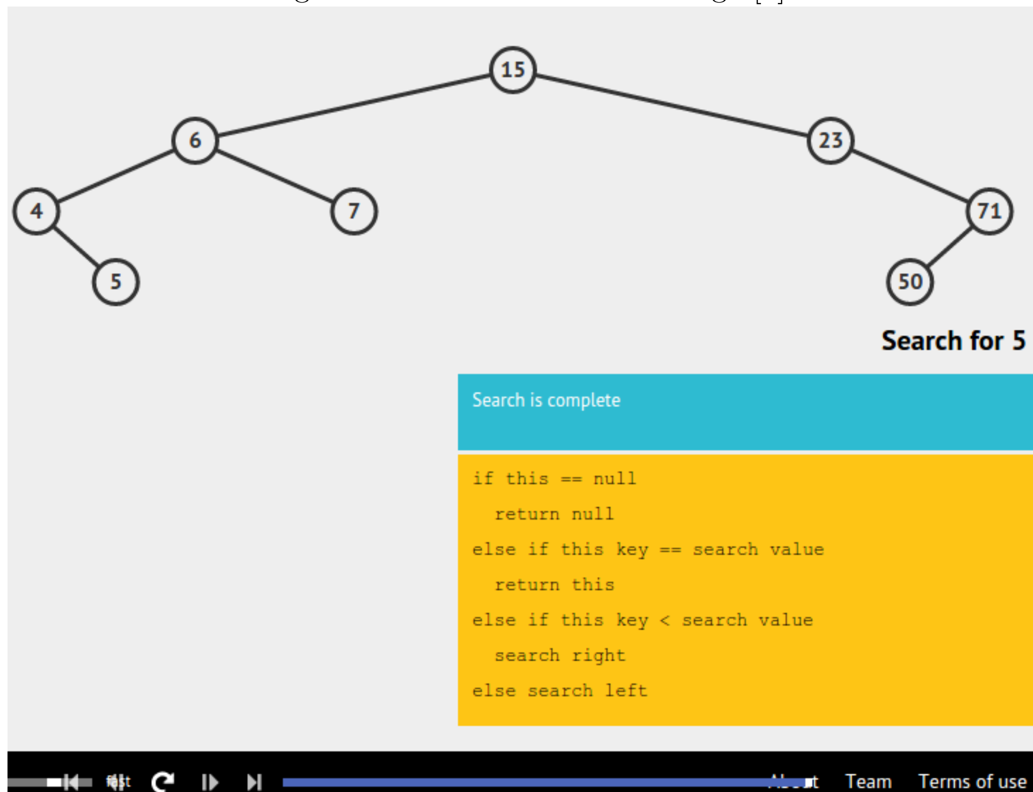
Figure 2.2: Screenshot of python tutor at work [2]



Python tutor is a tool that explains the execution of a python script. It does this by stepping through the code like a debugger and showing the state of the program on the right. It shows fairly low level animations with no possibility to be extended. This tool allows for exploring the code by stepping through its execution, however it does not allow the writer to add a personalized output.

2.1.3 VisuAlgo

Figure 2.3: Screenshot of VisuAlgo [3]



`VisuAlgo.net` is a website that gives dynamic visualizations of existing algorithms. It provides options to play with the input of a certain algorithm, and then visualizes how the algorithm solves the problem for a given input. A downside of VisuAlgo is that it does not run plain code, you can not alter an algorithm or add your own and edit the visuals

2.1.4 Existing documentation frameworks

There are many tools for adding for generating documentation for code. A few examples are Doxygen [4], JSDoc [5] and Sphinx [6]. These documentation tools commonly scan the code to generate a documentation. However,

after the documents are generated there is no further interaction with the project.

2.1.5 Static code visualizations

Handmade algorithm visualizations can function as inspirations for the possible animations that our application can generate. A website that contains many interesting visualizations is the website of Mike Bostocks [7]

2.1.6 Interactive storytelling

Interactive storytelling is a way of article writing in which the article develops as the reader is reading it. Interaction with the document can influence the outcome of the document. This is a method that we can use for our visualization, interaction with the document will the determine what elements the reader needs explanation on. An example is the following New York times article: `http://www.nytimes.com/interactive/2014/09/19/travel/reif-larsen-norway.html`. New views open as the reader scrolls though the document and explanatory visualizations show up as you scroll over them.

2.2 Technologies and libraries

The project is based in the browser. Therefore we have researched the existing browser based solutions for the following three modules for our application, a **Code Editor**, a **Javascript interpreter and debugger** and **Visualization libraries**. For each of these concepts we will state our considerations and conclude with our selection.

2.2.1 Code editor

During the use of the application the user is constantly handling code. Therefore it is important that the process of reading and editing code is a pleasant experience. Creating a pleasant user experience for editing and viewing code is an art in itself and therefore we have researched the existing solutions. The requirements for the editor are that it should

- Support basic features such as syntax highlighting

- Be lightweight
- Be easy to integrate in the application

We have found the following editor implementations to be potential solutions

- Ace [8]
- Code Mirror [9]
- ICE Coder [10]
- Codiad [11]

These options are the commonly used code editors for the web. Ace and Code mirror are popular in web based development platforms like c9.io and jsfiddle.net. These two are the most simple with only the minimal most common features you would expect in a code editor, such as syntax highlighting, automatic indent and outdent and code folding. On top of that they allow to be easily embedded, which gives more freedom for how the document is structured.

The latter two options, ICE Coder and Codiad are full featured web-based IDE's. They have support for a multitude of languages and have their own file manager on top of all the common features the previous two code editors have. These are very complete and feature heavy. Downsides are we have to build our application around these editors instead of embedding them in our application.

We have chosen not to use ICE Coder or Codiad, because they force us to build our entire application around the editor, instead of embedding the editor as a module in our application. This leaves us with the choice, Ace or Code Mirror. An important aspect of our application is that we can programmatically select parts of the code, to visualize what code is being interpreted. We have found that Ace has a better integration of this feature and is also better documented. Therefore we chose Ace as our solution.

2.2.2 Javascript interpreter and debugger

To add meta code to specific statements in a piece of Javascript code it has to be broken down to its logical components. While stepping through and executing this code, the linked meta-code must be executed. This behavior is very similar to an interpreter, with the addition that some extra code can be linked to a statement which is executed when the statement is executed. Therefore we have researched the available Javascript interpreters. We have defined the following requirements

- Written in Javascript, so we can use it in the browser
- Easy access to the scope, so it can be used in the meta code
- Open-source, so that we can easily extend it to fit our needs

We have found JS-Intepreter [12] to fit our needs. It is a simple Javascript interpreter which allows stepping through the code. For parsing it uses a separate project called acorn [13], which outputs the syntax tree as described by Mozilla [14].

2.2.3 Visualization libraries

When picking the visualization library it is important to assess the ease of use of the library. The library should be featured enough to allow any data structure or algorithm to be visualized, yet it should not become a painful experience for the user to create these visualizations. Also the libraries should allow for making the animations dynamic without too much effort. Within the possibilities we have found a distinction between canvas based and vector based visualizations. We have found the following solutions to be relevant.
The HTML5 canvas. The standard HTML5 canvas is very low level. It is this way naturally full featured and it is well documented but not very user friendly. Producing visualizations with only the HTML5 canvas would not yield user-friendly creation of dynamic visuals.
Pixijs. A Canvas type framework that enables easy loading and has some built in features for attaching visualizations and the animation loop. Pixijs [15] is aimed at making animations but not necessarily data visualizations.
SVG DOM element. Just as HTML5 Canvas very powerful but also fairly low level. Writing animations with the SVG DOM element only would lead to a lot of extra code to make it work. Very powerful, but to low level to be user friendly
D3. D3 [16] describes itself as the data-driven approach to DOM manipulation, therefore it works together nicely with SVG DOM elements. D3 is currently the most widely used framework for data visualizations. Examples of its use are the algorithm visualizations of Mike Bostocks [7] and the data visualizations of the New York Times [17]

We have not chosen any of the low level libraries, as these lack in user friendliness. D3 has proven its usability in data visualization and makes the use of

vector graphics possible. Therefore we have chosen to use D3 as the default visualization library.

2.3 Technical Limitations

There are a few limitations that we put on this project. Firstly we will only be supporting Javascript based projects. We presume that adding support for different languages will give us an overhead that will not fit in the scope of this project. Furthermore we will not be focusing on browser support, the browsers that we will support are the latest versions of Firefox and Chrome. We will also assume that the projects being narrated are final and that their form will not change.

Chapter 3

Requirements

In this chapter we will discus the requirements. We created user stories and from this we created a MosCoW overview of the capabilities we would implement in the 10 weeks we had for our project.

3.1 Functional requirements

Must haves

- Be able to load a javascript project
- Project navigator to view the files in the project
- Code editor to view the contents of a file
- Ability to select a file and have it displayed in the code editor
- Be able to parse the javascript files in the project to create ASTs
- Be able to debug javascript
- Be able to attach a narrative to either a file, folder or node in a AST
- Be able to view all narratives attached to a file, folder or node in AST
- An element to display a narrative in a story panel
- Be able to add narrative elements to a narrative, examples:
  - Text
  - Video
  - Images
- Be able to edit narrative elements of a narrative
- Editor to input, write and edit narrative elements

- Visualization: A narrative element to create animations that can make use of:
  - A graphics framework
  - Ability to access program scope
- Control buttons for debugging:
  - play
  - next/previous code statement
  - next/previous narrative element
- Convex hull example
- Code editor for initialization of code
- Example/Mock input

Should have

- Be able to export the project with its narrative
- Be able to share the project with narrative to another user (that user can not edit the narratives)
- A visual tree representation of the narrative structure
- Highlighting of code during execution
- GitHub support for projects
- Ability to make small changes in the project code without losing all of the visualization data
- Default visualizations available in that the user can use to display commonly used data structures

Could have

- Automatic code layout on import for visibility
- Playback speed buttons to control how long each step of the algorithm lasts
- Documentation of basic Javascript functions
- Choose which particular graphics framework is used in visualization

Won't have

- Support for more languages other than Javascript

- 3D objects for use in the visualization
- Relative object placement in the visualization
- Asynchronous visualization

3.2 Non-functional requirements

- Written in javascript
- Completed withing 10 weeks starting from April 20th
- Must be supported in at least latest versions of the javascript and chrome web browser
- Use of git for revision control

Chapter 4

Project Planning

4.1 Developer tools

We have used Angular [18] as front-end framework for building the application. We have found that the most experience exists in the group for using this tool and it is currently the most widely adopted MVC front-end web framework available. We set up our project using Yeoman [19] for building the base of our application, which gives us Bower [20] and npm [21] as package managers and Grunt [22] as task runner. For version control we have used Git, as all three of us have experience with the tool and it has clearly become the de-facto standard in recent years. We use GitHub [23] to host our repository.

4.2 Scrum

Initially we followed the Scrum plan, but quickly discarded the notion of roles and rigid time planning. We met up daily to work on the project. In the morning we would discuss the priorities and division of the tasks for the day, and during lunch we would evaluate the progress.

4.3 Issue tracking

Most improvements were done in separate branches to be merged at the end of the day. We began sprint planning with priorities. As the project

progressed and the major features were implemented we started to rely more heavily on the GitHub issue tracker.

4.4 Testing

Testing was done with Jasmine [24] on top of the Karma [25] framework. Jasmine is a behavior driven development testing framework for JavaScript. Jasmine is the framework that interprets the testing code. Karma produces an environment and web server in which the test-code can be ran. This combination is one of the most widely adopted Javascript unit testing solutions at this moment in time and is for this mainly Javascript based application a perfect choice. A code coverage report is produced by a framework called Istanbul [26] which also works on top of Karma.

4.5 Time Planning

We worked week days except Monday. We started at 9:30 and worked till 17:30. We set the following goals to schedule our work.

Week 1	Conduct research
Friday	Deadline project plan
Week 2	Work on research report, define use cases
Friday	Research report done
Week 3	Start of implementation
	Basic framework and modules
Week 4	Basic narrator
Week 5	Advanced features
	Testing of current code
Week 6	
Tuesday	Hand in first SIG submission
Week 7	Process SIG Feedback
Week 8	Polish user interface
Tuesday	Hand in final SIG submission
Week 9	Work on final report
Friday	Deadline final report
Week 10	Work on presentation
Friday	Final presentation

We have diverted slightly from this time planning. The finishing of the core features have been mixed with adding the more advanced features. The core features were really functional in week 8.

Part II

Implementation

Chapter 5

Concepts Introduction

From the Research phase was concluded that the central elements of the system would allow a *writer* to attach documentation i.e. narratives, at both the file system level (files and folders), as well as the code level in the form of an Abstract Syntax Tree (AST). The AST is generated when code is parsed by a parser and is defined by a programming language. This allows statements to be nested and specifies how declarations are made and more [1]. A *viewer* should then be able to play back these narratives. While playing back narratives on code, the elements on the AST would be called by an interpreter that is running the code. Graphical representations should be able to be generated/updated based on the variables in the scope.

In order to set up a strong foundation for development we divided the necessary elements into a couple of concepts as to give each functionality context. The next sections contain a further description of each of these concepts.

5.1 CAST

First off, in order to associate a narrative to a point in a given project we decided on a data structure that would facilitate this. The CAST , (**C**ontext & **A**bstract **S**yntax **T**ree) is a concept we introduced to create a tree of files, folders and AST's. The three is structured as follows: The root node of the tree is the root directory of the project. It then has for each folder and for each file a folder- or file node respectively as its children. For each folder

[1] https://en.wikipedia.org/wiki/Abstract_syntax_tree

node this process is repeated. Each file node only has a maximum of one child. In the case the file is of a type that can be processed by the interpreter (in the scope of this project this currently only is Javascript), then its child is the AST of its content. The result is one continuous tree that contains all the context that is needed of the makeup of a project. This way the nodes in this data structure correspond to the the locations which can be narrated which in turn allows for the narratives to be easily managed.

5.2 Narrative

Narratives are the main focus of the application. They have to be powerful enough to allow the user to express the information they want to convey and simple enough so that it is not a hassle to create. In order to satisfy both these conditions we defined the content of a narrative to be a list of building blocks.

5.2.1 Narrative items

The different types of building blocks that were implemented are:

- Text item
- Audio item
- Picture item
- Video item
- Link item
- VCode item
- Code item

The first half of the list are standard multimedia elements and fairly self explanatory. However the latter half demands some explanation.

Link item

The link item was devised to allow for reusing other narratives. When a link item is reached during playback it will insert a narrative specified in the link and play this narrative. When the linked narrative is done playing it will continue f the original narrative.

Text item

The text item allows a user to simply display a paragraph, but it also has the additional feature when running a CodeNarrative of injecting the content of a variable. A writer can state [[obj.somekey]] and the content of that variable will be displayed instead. The syntax is limited to the direct .dot notation and obj['somekey'] will not work.

VCode item

VCode (Visual Code) is the name that is used for the item that can be added to create visualizations. When writing a VCode item the user has access to the variables in the scope of where he decides to put the item. These variables will be instantiated by the interpreter and contain the current value of the state of execution. It can read the variables in the active scope of the Javascript that is being interpreted but not write to them. And has access to the graphic libraries and a persistent set of objects created in previous VCode items.

Some standard objects have been defined to easily chart data. But the developer is free to directly manipulate with d3 declarations, or a canvas element, or even create a new object with a simpler interface for visualizing some state.

To display a DOM element in the narrator the developer ends his VCode with the function call $display(DOM_ELEMENT)$

The VCode is evaluated by using javascript's *with* and *eval* statements. This is considered bad practice. A future improvement could augment the JS-Interpreter that was used to evaluate the VCode. But this was considered impractical within the scope of the project.

Code item

Code items are snippets of code that are executed inside the interpreter. The main use for these items are so a *writer* can initialize his code narrative. When a code narrative is about a function, a code item can be used to call that function.

5.2.2 Item Hooks

Narratives on file/folder nodes (FS Narratives) are quite different from narratives on AST nodes (Code Narratives). FS Narratives are simply a list of narrative items. For Code Narratives this is not as simple. Since parts of the narrative should be shown when a certain part of the code is reached, it inherently can not be a simple list of items. Instead code narratives have a list of Item Hooks. Item Hooks are set up as a key/value pair. The key is a path to a sub AST Node that can be reached during execution and the value a list of items. This allows these hooks to be evaluated when the interpreter has executed these nodes. These items will be displayed multiple time when in a loop for example because the statement can be reached multiple times. But because the values of the variables in the scope can change throughout the execution, the items that rely on values in the scope become dynamic.

5.3 VObject

Earlier, VCode items were explained to be items that allow users to write code for visualizing. To avoid users having to rewrite the same code over and over if they want to use a visualization multiple times we introduced VObjects. These are wrappers of VCode the user can create on a project level that can be called in every VCode item. This way the user can define a VObject. For example a graph, and call it in different narratives throughout the project.

5.4 Narrator

For the purpose of handling the actions surrounding a narrative the narrator was conceptualized.

Out of all the elements in the applications the narrator is the most dependent on the state of the application. Depending on what state the narrator is in, it has different functionality. Two main states can be discerned, namely a *writer* state and a *viewer* state, each with 4 main functions. The viewer state has the functions:

- Selecting narratives for playback
- Display the next narrative Items
- Initialize an interpreter for narratives on AST Nodes
- Load the next narratives when it encounters a Link Item.

The writer state has the functions:

- Adding/removing new narratives
- Selecting narrative for editing
- Adding/removing new items
- Selecting the Item Hooks in case of a Code Narrative

Chapter 6

Relations

We use the Model View Controller structure. This is natural to Angular and fits our application. The project consists loosely out of three parts: The project explorer, the narrator and the project manager. The project manager is responsible for saving and loading the CAST, which consist of the project files, folders and narratives. The project manager only comes into play again when the current project needs to be saved or a new one needs to be loaded. Both the Narrator and the explorer use the CAST to gather the required information. The Project explorer gives a representation of the current state of the CAST. The narrator uses it for getting the content of nodes, selecting new nodes and determining narratives that are related to a certain node.

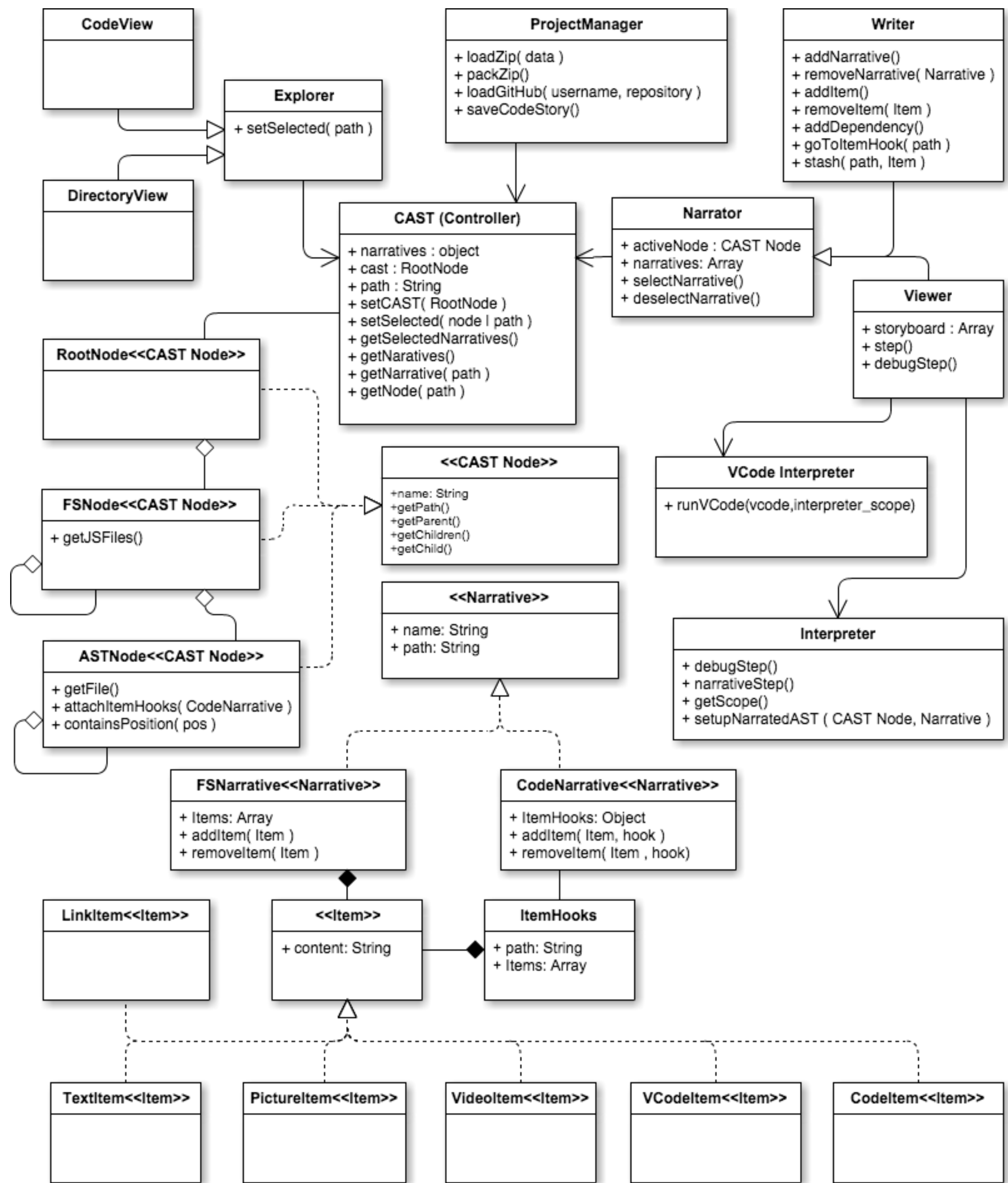


Figure 6.1: CodeStories UML Diagram

Chapter 7

User experience & user interface

For our user-interface we have drawn inspiration from *interactive storytelling*. The main interface is split into three parts. The menu bar, the project view and the narrative view.

Below is a list of simple actions the different users have to be able to perform in the main scenario of the application followed by the result of those actions.

7.1 Viewer

1. Open project → Way to input link to project
2. Select node you want to view → Folder structure and ability to view the code (See related narratives)
3. Select narrative → List of narratives
4. Play back narrative → Step by step display of narrative primitives. Highlights the file/folder/code section and creates animations.

Figure 7.1: Wireframe of viewer mode

This is the basic screen the Viewer will be navigating through. On the left the viewer can navigate through his files and select a file which will then be displayed in the middle. On the right there are the available narratives for the current location. There are buttons for stepping through the narrative and this will also update the location on the left side. The narrative can also initialize the interpreter. This will highlight the code and generate the defined visualizations for the pieces of code.

7.2 Writer

1. Open project → Method to upload
2. Select node you want to narrate → Folder structure and ability to view the code

3. Create/Select narrative → List of already defined narratives and a method to create new narratives
4. Add/edit narrative primitives → List of already defined primitives and a method to add a new primitive
5. Select which primitive → List of available primitives
6. Supply primitive with information → An input field

Figure 7.2: Wireframe of writer mode

Figure 7.3: Wireframe of writer mode editing

The writer has a similar GUI. But different controls. The writer can create new narrative when selecting a CAST node. Here the writer can add a new Narrative. If the Narrative is located on a File or Folder the writer can simply add items to the narrative. If the Narrative is added on a AST Node , the writer is able to select any sub node in the code view and add items on those nodes.

Chapter 8

Angular design patterns

8.1 Use of state

One of the dependencies that the application uses is the routing framework AngularUI Router. This is a more flexible implementation of the standard angular routing service that is build around the URL. Instead of relying on the URL (routes) to define which views are loaded, it makes use of states. This brings a couple of useful features to the way routes are handled.

First off, states are the primary way the application decides what part of the application should be active. Routes can be tied to these states but are inherently optional. Secondly, it allows for multiple views to be tied to a single state. Lastly, it allows for nested states and nested views.

These features allow for the state of the application to be tree structured. Nodes in the tree can be tied to a route but this is not required. This allows for flexible specification of the routes and allows us to switch state's without changing the URL.

This last feature is used to let the application know if it is in a writer or viewer mode without having to clutter up the route or make use of Boolean variables that are difficult to manage in the code. This way the route can be used for specifying a CAST node.

The application's states are structured as can be seen in Figure 8.1.

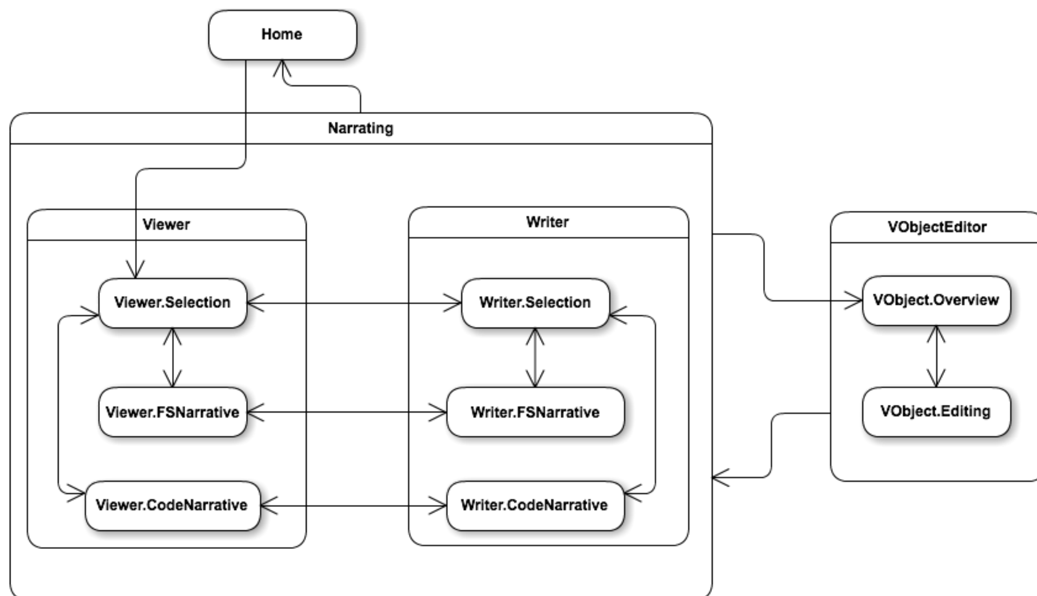


Figure 8.1: State diagram

On the highest level there are three different states which the application can be in, namely Home, Narrating and VObjectEditor. The application starts in the Home state. From here a user can load in a project and start narrating. Naturally this will lead the user to the narrating state. In the narrating state there are a handful of sub states but these can be categorized into two different groups, Viewing and Writing. The viewing state is used when a user wants to playback a narrative and the writing state is used when a user wants to create a narrative. Lastly a user can write some of his own code to use for visualization in the VObjectEditor. The code written here can later on be called when writing narratives. This is separate from writing a narrative so that the user does not have to rewrite code for visualizations but rather can just call functions that he has already written in the VObjectEditor as to prevent cluttering the visualization code.

8.2 Modules

An important aspect of Angular is how it allows to keep front-end Javascript code separate and modular. While it already separates parts of the application into the different controllers, directives, services, etc. it also ties these to

a Module. Modules are Angular's way to wire together the different parts of an application. Some advantages of this approach are, per Angular's website:

- The declarative process is easier to understand.
- You can package code as reusable modules.
- The modules can be loaded in any order (or even in parallel) because modules delay execution.
- Unit tests only have to load relevant modules, which keeps them fast.

In the application modules are used to divide up the following important sections of the application:

- App - The main module that is used to initialize the app, handles states.
- CAST - Logic of the CAST.
- Narrator - Handles playing and editing narratives.
- Explorer - Handles navigating the CAST.
- Navigation - Handles navigation of the application besides the CAST.
- ProjectManager - For importing and exporting of projects.
- VCode - The interpreter for the visual code.
- VObjectEditor - Handles the VObjects.

This partitioning is granular enough as to have all the different parts of the application in different modules. Next to these important modules there are also some extra modules like the module used to display notifications. These features have been assigned their own module because they are used throughout the application.

Part III

Reflection

Chapter 9

Fulfillment

Here we reflect on our progress over the last 10 weeks. How much we achieved of our original goal, the unforeseen problems we have encountered and recommendations for further development.

9.1 Requirements

The majority of our goals have been achieved. We discovered while writing our example that the interpreter required some patches to return the proper scope. Initially the design for the controls called for one item to be displayed at a time. Here the back button would be useful. But we quickly came up with the design were everything was added to a growing list of items so the user could simply scroll back. The visual tree representing the narrative structure was dropped due to time.

9.1.1 Functional requirements

Must haves

- ✓ Be able to load a Javascript project
- ✓ Project navigator to view the files in the project
- ✓ Code editor to view the contents of a file
- ✓ Ability to select a file and have it displayed in the code editor
- ✓ Be able to parse the Javascript files in the project to create ASTs
- ✓ Be able to debug Javascript
- ✓ Be able to attach a narrative to either a file, folder or node in a AST

- ✓ Be able to view all narratives attached to a file, folder or node in AST
- ✓ An element to display a narrative in a story panel
- ✓ Be able to add narrative elements to a narrative, examples:
- ✓ Be able to edit narrative elements of a narrative
- ✓ Editor to input, write and edit narrative elements
- ✓ Visualization: A narrative element to create animations that can make use of:
  - ✓ A graphics framework
  - ✓ Ability to access program scope
- ✓ Control buttons for debugging:
  - ✓ play
  - × next/previous code statement
  - ✓ next/previous narrative element[1]
- ✓ Convex hull example
- ✓ Code editor for initialization of code
- ✓ Example/Mock input

Should have

- ✓ Be able to export the project with its narrative
- ✓ Be able to share the project with narrative to another user.
- × A visual tree representation of the narrative structure
- ✓ Highlighting of code during execution
- ✓ GitHub support for projects
- ✓ Ability to make small changes in the project code without losing all of the visualization data
- ✓ Default visualizations available in that the user can use to display commonly used data structures

Could have

- × Automatic code layout on import for visibility
- × Playback speed buttons to control how long each step of the algorithm lasts

[1] The narrator being a scroll-able list of items is our solution to this requirement

× Documentation of basic Javascript functions
× Choose which particular graphics framework is used in visualization

9.1.2 Non-functional requirements

✓ Written in Javascript
✓ Completed withing 10 weeks starting from April 20th
✓ Must be supported in at least latest versions of the Javascript and chrome web browser
✓ Use of git for revision control

9.2 User experience & user interface

We have been successful in creating the basic layout we set out to make. Next to that we have taken the time to implement basic features we had not defined in our research. The UI steadily improved every week. This was a superficial but enjoyable indication of the progress that we made. Features that were added but had not been defined in our research report include:

- Cut and paste items
- notifications to display errors
- A home screen with github, zip and preset loading functions
- A VObject editor
- A testing area for the VObjects
- An auto-play feature for narratives
- A brief guide for new users

A critical part of our UI is the display area for the narrative items. Initially we had thought this element up to be in analog with a PDF document, where you scroll down to see the next pieces of information. The problem we experienced with this, is that visualizations become very chaotic when they do not animate in one static location. This led to a fair amount of discussion. We have considered several things, such as putting all visualizations in one static location or making a separate tab for the visualizations. The downside to these approaches are that we would lose the scrolling experience. At last we found that having visualizations scroll up the page and than stick to the top gave us the experience we wanted. Visualizations will not scroll off the page, while the scrolling experience will remain intact.

Chapter 10

Usage Example

Here we will give a example of writing a CodeStory to visualize the execution of a convex hull algorithm using the graham scan approach. We wont be adding extra text to the CodeStory for the sake of simplicity. Instead we will focus on the visualization object and Narrative with the goal of explaining the usage of the application and not necessarily the convex hull algorithm itself. Some knowledge of D3 [16] is suggested. We will use the convex hull algorithm as written by GitHub user brian3kb [1].

10.1 Writing the narrative

The plan is to generate a random set of points and add these to the ConvexHull Object and then request it to generate a ConvexHull. We would like to

- Display the points as dots
- Highlight the anchor point
- Visualize the sorting based on the angle
- Visualize the consideration to add a point to the hull.

10.1.1 The initialization code

Before we can start running the code, we have to provide some example input. We do this by adding a code item with the initialization code for the

[1] `https://github.com/brian3kb/graham_scan_js`

algorithm. We add the following code to the code item, that generates some random points for us.

```
var chull = new ConvexHullGrahamScan();

for(var i = 0 ; i < 10 ; i++){
    chull.addPoint(
        Math.floor((Math.random() * 200) - 90) ,
        Math.floor((Math.random() * 200) - 90 )
        )
}
chull.getHull();
```

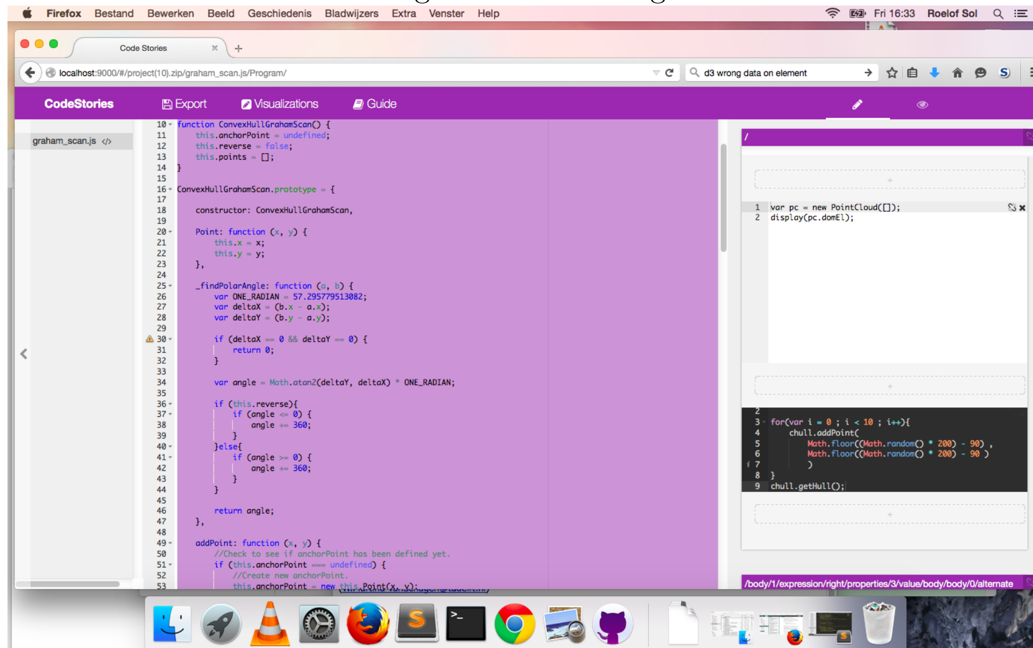


Figure 10.1: Writing

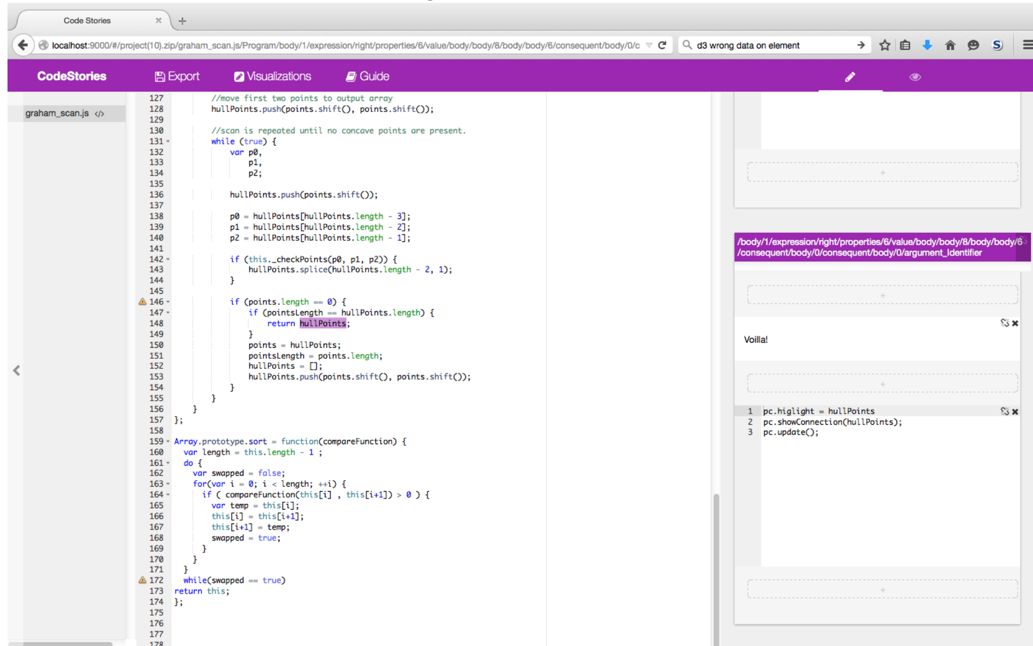
Next we click the 'return hullPoints' in the getHull function. Here we add the text item [[hullPoints]] and we check to see if the code narrative works. we press view and play.

We notice that there are two native Array functions that are not well supported. *sort* will work but we can not attach hooks in the compare function, and *every* is a relative new function that is not supported by our interpreter. We fix this by overwriting our array prototypes in our code item.

```

Array.prototype.sort = function(compareFunction) {
//bubblesort
  var length = this.length - 1 ;
  do {
    var swapped = false;
    for(var i = 0; i < length; ++i) {
      if ( compareFunction(this[i] , this[i+1]) > 0 ) {
        var temp = this[i];
        this[i] = this[i+1];
        this[i+1] = temp;
        swapped = true;
      }
    }
  }
  while(swapped == true)
return this;
};

Array.prototype.every = function(checkFunction){
    var result = true;
    for(var i = 0 ; i < this.length && result; i++ ){
            result = checkFunction(this[i]);
        }
    return result;

}
```

Now the code will run and we see a text item appear with our result.

Figure 10.2: Writer



10.1.2 Visualization Object

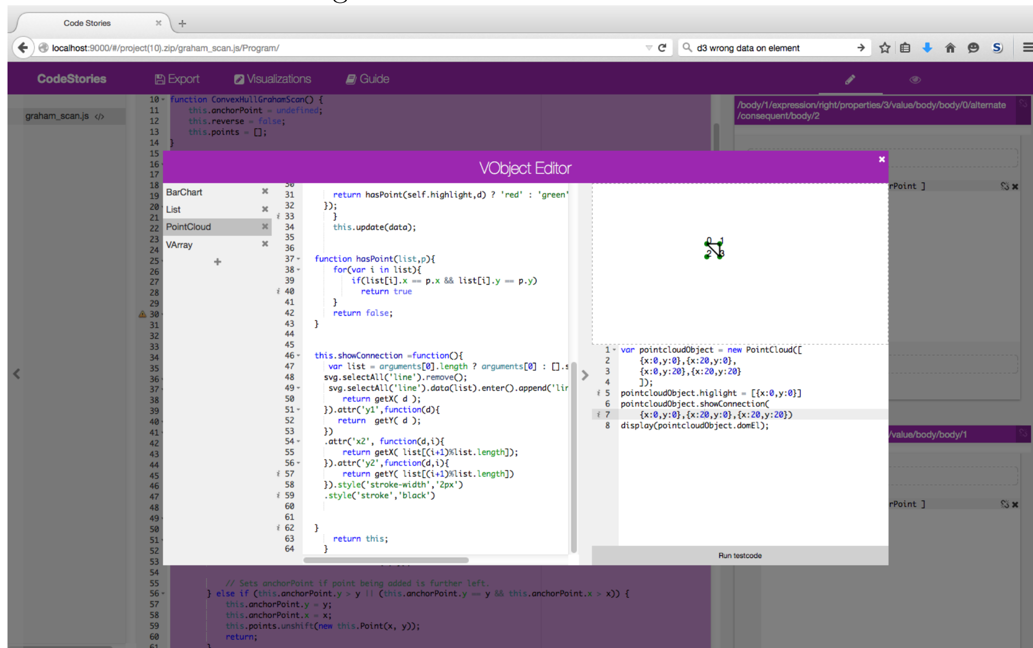
As we do not wish to clutter the code items with D3 code, we will create a Visualization Object. We create a new object called PointCloud in the VOBject editor, and add its basic functionality. We want to show a set of points, highlight some points. We create the following D3 code and generate an interface for it with three functions, *update*, *showConnection* and *highlight*.

```
function (data) {
  var self = this;
  this.domEl = document.createElement('div');
  var svg = d3.select(this.domEl).append('svg');
  var height = this.height, width = this.width, c = this.center, data;
  this.highlight = [];
  var data = data;

  function hasPoint(list,p){
    for(var i in list){
      if(list[i].x == p.x && list[i].y == p.y)
        return true
```

```
13      }
14      return false;
15    }
16
17
18    function getY(d,i){
19      return c.y+d.y
20    }
21    function getX(d,i){
22      return c.x+d.x;
23    }
24
25    this.update = function(newData) {
26      data = newData || data
27      var group = svg.selectAll('g').data(data);
28      group.exit().remove(); //exit
29      var enter = group.enter().append('g')
30      enter.append('circle').attr('r',4) //enter
31      enter.append('text');
32
33      group.attr('transform', function (d, i) {
34        return 'translate(' +  getX(d) + ','+ getY(d) +')';
35      })
36      group.select('text').text(function(d,i){return ''+i})
37      group.select('circle').attr('fill', function (d, i) {
38
39        return hasPoint(self.highlight,d) ? 'red' : 'green';
40      });
41    }
42    this.update(data);
43
44
45
46
47    this.showConnection =function(){
48      var list = arguments[0].length ? arguments[0] : [].slice.
           call(arguments);
49      svg.selectAll('line').remove();
50      svg.selectAll('line').data(list).enter().append('line')
51      .attr('x1',function(d){
52        return getX( d );
53      })
54      .attr('y1',function(d){
55        return  getY( d );
56      }).attr('x2', function(d,i){
```

```
57        return getX( list[(i+1)%list.length]);
58      }).attr('y2',function(d,i){
59        return getY( list[(i+1)%list.length])
60      }).style('stroke-width','2px')
61      .style('stroke','black')
62
63    }
64    return this;
65  }
```

Figure 10.3: Visualization editor



10.1.3 Adding the visuals

Now we have the Visualization ready to go. We can expand our narrative. We prepend a visualization element on the root node with the content

```
1 var pc = new PointCloud([]);
2 display(pc.domEl);
```

This will display the DOM element in the narrator and allow us to call it in other visualization items

Next we add two new visualization calls at the addPoint function

```
pc.highlight = [ this.anchorPoint ]
pc.update(this.points);
```

This will update the points in the narrator VCode item.

Lets also add a visualization item in the sort function. The If check will always execute so lets add the visualization item

```
pc.showConnection([a,self.anchorPoint,b]);
```

This will show lines between these points.

next we add 2 more visualization items to the checkPoint call inside the getHull function. on the check we highlight them with

```
pc.showConnection(p0,p1,p2);
```

and on the consequent we highlight the hullpoints again

```
pc.higlight =hullPoints
pc.update();
```

Next we could add some intro text. some more text on the checks to elaborate on what it is checking. More information on how it calculates the angles , etc. He can even add a narrative to the file node and give a general introduction. From inside this narrative he can then link to the code narrative that was created.

Time to publish.

Now the writer can export the story. He can either export the zip and give it to the viewer or he can export the .codestories file, add the file to his GitHub repository and tell the viewer about his repository. The viewer can load the project using the writer's GitHub's user name and repository name.

10.2 Viewing the narrative

The viewer loads the project. He sees in the left hand that there is a narrative on a program node. He clicks this and presses play and the narrative starts playing. The viewer can change the play speed by adjusting the slider under the play button. If the viewer wants to take the time to inspect every step he can press the next button, next to the play button, which will add the next narrative item. Likewise the step button next to the next button will only show the following step in the interpreter.

Figure 10.4: Viewer

Chapter 11

Recommendations and further Improvements

The biggest improvements are gained on the user experience and extended functionality of the application. Currently all core features are supported. One such extended functionality is **language support**. At the moment the application supports JavaScript. This is convenient as we are working with a web application, but optimally one would like to support a broad range of languages. At githut [27] one can for example see that Python and Java are two language that, if supported, would greatly increase the range of the application. Currently the user needs to inform him- or herself by examining existing examples and create one accordingly.

A bottleneck in the use of our application are the **VObjects.** To create a custom VObject, a good understanding of the D3 library is necessary. Creating an extensive and complete library of standard visualizations that cover most use-cases of the program would make this problem less present.

In the current implementation **projects can not change**. It is expected that the configuration of a project (the files, their locations and their content) are final. To support changing projects, differences must be detected and either automatically resolved or resolved by the user. This feature would make narrating projects in development possible, but is not included for simplicity.

A more extensive **user manual** and/or set of tutorials would lower the boundary to get started with the application. Giving the user a quick run through the program along with some tutorials that show how to make a basic narrative, would polish the user experience.

Text items only support the [[obj.subobj]] syntax of objects and can not evaluate an expression. This could be helpful when for instance you would want to output the index of a value in an array with array.indexOf(obj);

Lastly, giving the writer of a Narrative more specific **error feedback** might spare the writer some debugging time.

Chapter 12

Conclusion

We set out to make a system to help programmers tell the story of their code and create a web based code visualization platform. In the process, several design decisions had to be made. First of all we chose to power our application with the AngularJS front-end framework. Given that this framework was new to two of the team members implied that the framework had to be learned alongside the development. This posed challenges but we believe that in the end it gave us a well structured maintainable application. To give our system the set of features that were desired, alongside with the possibility for new features to be added later, some concepts had to be introduced. First of all the CAST. The data structure that has the file system and code structure integrated in one tree. This allowed us to add story elements in any part of a project. Next to that we had to find a way to hook into the execution of JavaScript, so that we could implement our own visualization hooks. The solution we found was to use JS-Interpreter, for its simplicity. The user interface had to be designed in a way that made watching a code story natural. Improvements to this can be made, but overall the scrolling view with locking visualizations is a fair implementation. We believe the application has become a useful tool for giving and getting an inside into code. The concepts we introduced have provided a flexible system, that allows for a more advanced system to be build.

Part IV

Appendix

Appendix A

Glossary

The definitions used throughout the document:

- **AST**: Abstract syntax tree. A tree that represents the structure of source code written in a programming language. Nodes in this tree can be constants or variables (leaves) and operators or statements (inner nodes). File/Folder tree. Basic file and folder structure.

- **CAST**: Context abstract syntax tree. Combination of File/Folder tree & AST. A node in the CAST can be a file, a folder or a node that is available in the AST.

- **Narrative**: A Sequence of Narrative Primitives attached to a node in the CAST.

- **Code Narrative**: A Narrative that is located in an AST node. It has access to the scope of the AST node and it can specify locations in the AST that, when reached, display a Narrative Items.

- **Narrative Items**: A step/element in a Narrative that a user can step through. Possible primitives are Text, Video, Image, Link to a narrative and Visualization call.

- **Standard VObjects**: Object with functions to animate/update a graphic. examples: Graph, table, pie-chart.

- **Narrative graph**: A graph representing related narratives.

- **Viewer**: A person that views/follows Narratives or selects a CAST Node to gain more insight about that node.
- **Writer**: A person that writes Narratives for nodes attached to the CAST.

Appendix B

Feedback

B.1 First SIG Feedback (Dutch)

[Analyse]

De code van het systeem scoort vier sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code bovengemiddeld onderhoudbaar is. De hoogste score is niet behaald door een lagere score voor Unit Size en Duplication.

Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld de 'UnpackZip'-methode, zijn aparte stukken functionaliteit te vinden welke ge-refactored kunnen worden naar aparte methodes. Commentaarregels zoals bijvoorbeeld '//Loop through files that are packed in the zip' of de verschillende if-blokken zijn een goede indicatie dat er een autonoom stuk functionaliteit te ontdekken is. Het is aan te raden kritisch te kijken naar de langere methodes binnen dit systeem en deze waar mogelijk op te splitsen.

Een tweede punt wat meespeelt bij de Unit Size is de relatief grote 404 pagina, hier staat de CSS van de pagina inline. Dit zou eventueel ook verplaatst kunnen worden naar een apart bestand. Mocht dit niet mogelijk zijn dan is het goed om de reden hiervoor duidelijk te communiceren in de documentatie. Dit helpt toekomstige ontwikkelaars om fouten te voorkomen.

Voor Duplicatie wordt er gekeken naar het percentage van de code welke

redundant is, oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code doorgaans op meerdere plaatsen moet gebeuren. In dit systeem is er (bijvoorbeed) duplicatie te vinden tussen 'link.html' en 'text.html', dit zou opgelost kunnen worden met een apart template. Ook tussen de 'viewer'- en de 'writer'-controllers is code hetzelfde. Het is aan te raden om dit soort duplicaten op te sporen en te verwijderen.

Over het algemeen scoort de code bovengemiddeld, hopelijk lukt het om dit niveau te behouden tijdens de rest van de ontwikkelfase. De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd wordt.

Response

De project loader en de unpack en pack zip functies moeten inderdaad gerefactord worden. De 404 pagina hadden wij zelf over het hoofd gezien en is gegenereerdt bij het opstarten van het project. De items link.html en text.html waren nog niet feature complete en zal de volgende keer diverser zijn dan nu. Hier hebben wij dus express al plaats voor gemaakt. De writer en de viewer is een soort gelijk verhaal. In writer mode moet de state net even anders gezet worden dan in viewer mode. In een eerdere versie was dit allemaal in dezelfde file gedaan met veel 'if else' statements. Dit is volgens ons een betere oplossing.

We hopen bij de volgende review de fouten er uit gehaalt te hebben maar we zullen inderdaad alert moeten zijn op de test code zodat deze up-to-date blijft en begrijpbaar.

B.2 Second SIG Feedback (Dutch)

[Hermeting]

In de tweede upload zien we dat het codevolume sterk is gestegen, terwijl de score voor onderhoudbaarheid ongeveer gelijk is gebleven. Bij zowel Duplication als Unit Size zien we dat jullie een aantal voorbeelden hebben

weggewerkt, maar omdat de nieuwe code soortgelijke problemen heeft gaan de deelscores er uiteindelijk niet op vooruit.

Het is wel goed om te zien dat jullie naast nieuwe productiecode ook nieuwe testcode hebben geschreven, al worden sommige onderdelen beter getest dan andere.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige evaluatie deels zijn meegenomen in het ontwikkeltraject.

Appendix C

Original project description

C.1 Project description

Ever been completely lost in the code of someone else? Or you wished that someone would have provided you with more information about the functioning, pitfalls and more of the code? In many cases you will have to go over each line, think about example executions and consider various cases. This process is time consuming and in the end, it is either unsuccessful or finishes often with "Oh, of course! ..... Well I wish somebody had told me sooner".

WEBOS-Narrative is about providing coders with the possibility to easily produce an informative experience along with their code and algorithms.

C.2 The project goal

Build a web-based Javascript code editor and dependency navigator.

With the aid of a parser, a default narrative for a function is created, much like a debugger. Additionally a developer is able to extend this narrative with code that might exemplify execution branches and helps the reader in understanding the algorithm. For instance, a couple of examples, a video presentation, a web page, and custom visualizations of data structures and methods should be possible to add directly.

C.3 Company description

"Company" and "supervisor" are from the Computer Graphics and Visualization Group @ TU Delft (see below)

C.4 Auxiliary information

To give a narrower focus to the project, the methodology should be exemplified for a convex hull algorithm and the implementation should allow the easy integration of visualization tools for this algorithm.

C.5 Basic Product Requirements

- Save and Load.
- Syntax highlight
- Interactive dependency tree view
- Tutorial/example/narrative mode

Code Stories

26/06/15

Dynamic code documentation at your fingertips

The current process of understanding software projects can be a cumbersome practice, especially when complex data structures and algorithms are being used. At these moments a personal guide who can lead you through the code, point you at the relevant details, give you a visual representation of complex concepts and provide you with some extra elaboration where necessary, can be a huge benefit.

This Bachelor project strives to provide tools for anyone to create a digital counterpart of this person for his or her project, a code story. We strive to make documentation more enjoyable through interactive and visual storytelling, a practice in which a developer can give a user a guided tour through the project layout and (core) functionality.

During the process of developing this application we dove into the inner workings of several topics like code interpretation and Abstract syntax tree data models, to name a few.

The most important technologies we used were Angular and D3. The testing was done through Karma.

Comissioned by

TU Delft Computer Graphics And Visualization Group

Contact: e.eisemann@tudelft.nl

WebOS Narrative Team

Maarten van Beek

Salim Salmi

Roelof Sol

Special thanks to

Anna Vilanova

Elmar Eisemann

The final report for this project can be found at: http://repository.tudelft.nl

Bibliography

[1] Nat Pryce. Code guide. http://www.natpryce.com/software/code-guide/example/selector-button-blink.html, 2011.

[2] Philip Guo. misc python tutor. http://pythontutor.com/.

[3] Dr Steven Halim. Visualgo. http://visualgo.net/.

[4] http://www.stack.nl/~d.imitri/doxygen/.

[5] Contributors JSDoc 3 documentation project. Jsdoc. http://usejsdoc.org/.

[6] Georg Brandl and the Sphinx team. Sphinx python documentation generator. http://sphinx-doc.org/.

[7] Mike Bostock. Visualizing algorithms. http://bost.ocks.org/mike/algorithms/, 2014.

[8] Mozilla & Cloud9. Ace (ajax.org cloud9 editor). http://ace.c9.io/.

[9] CodeMirror Team. Codemirror. http://codemirror.net/.

[10] ICEcoder Ltd. Icecoder. http://icecoder.net/.

[11] Fluidbyte. Codiad web ide. http://codiad.com/.

[12] Neil Fraser. Js-interpreter. https://github.com/NeilFraser/JS-Interpreter/.

[13] Marijn Haverbeke. Acorn. https://github.com/marijnh/acorn/.

[14] Spider Monkey Development Team. Spider monkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API.

[15] Goodboy Digital Ltd. pixi.js. http://www.pixijs.com/.

[16] Mike Bostock. D3, data-driven documents. http://d3js.org/.

[17] Shan Carter and Kevin Quealy. Home prices in 20 cities. http://www.nytimes.com/interactive/2014/01/23/business/case-shiller-slider.html?_x=0.

[18] Google. Angularjs. https://angularjs.org/.

[19] The Yeoman Team. Yeoman, the web's scaffolding tool for modern webapps. http://yeoman.io/.

[20] Alex MacCaw & Jacob Thornton. Bower, a package manager for the web. http://bower.io/.

[21] Node.js Foundation. Nodejs. http://nodejs.org/.

[22] Grunt Development Team. Grunt. http://gruntjs.com/.

[23] GitHub. Github. http://github.com/.

[24] Pivotal Labs. Jasmine, dom-less simple javascript testing framework. http://jasmine.github.io/.

[25] KarmaJS Development Team. Karmajs. http://karma-runner.github.io/.

[26] Krishnan Anantheswaran. Istanbul, js code coverage tool. https://github.com/gotwarlost/istanbul/.

[27] Carlo Zapponi. githut. http://nodejs.org/, 2014.