University of Glasgow | School of Computing Science

Palgo - Algorithm Animator

Andrei-Mihai Nicolae

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2017

Abstract

Understanding algorithms is both very common and hard for developers in general, regardless of their level of expertise. Even the fundamental ones, such as Dijkstra's algorithm for finding the shortest path between two nodes in a graph, are quite complicated to grasp. Many studies show that visualizing an algorithm and its steps make understanding it much easier. In this report, we will present an Algorithm Animator built specifically for solving this problem in a modern, responsive and efficient manner. Among others, we will also show why certain design decisions (e.g. making it a native desktop app instead of a basic jar, using material design for the user interface), the implementation choices and the evaluation results make this tool a viable option for software engineers when it comes to learning different kinds of algorithms.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ———————————— Signature: ————————————

Contents

Chapter 1

Introduction

Firstly, the starting point should be defining what an algorithm is.

NOUN

A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

'a basic algorithm for division'

Figure 1.1: Algorithm definition taken from the Oxford English Dictionary.

As we can see above, it is a "process or set of rules" followed by a computer when trying to solve a problem. As a machine is, at its core, composed of 1s and 0s, a human being needs to visualise what is going under the hood in order to comprehend the steps that are undertaken. In a study conducted by Cristopher D. Hundhausen et al. [7], it was proven that even though algorithm visualisation is not the "perfect" solution, it is definitely effective in teaching students and experienced developers.

There are many tools out there that already provide this functionality, but one thing was a key factor in deciding how to proceed with planning the whole software process: there are barely any standalone applications. Even though one can find many websites which let people visualise algorithms' steps (e.g. a great example is VisuAlgo [13]), this is not such a good solution as the user cannot use the application at his/her commodity, an Internet connection being required.

Therefore, the plan was to create an app that can be run offline and would feel natural to the user regardless of the operating system of choice. There are many Java implementations (i.e. resulting in jar executables), but one of the main drawbacks is the lack of proper GUI tools that can build native-feeling applications.

As such, the decision was made to use the Electron framework originally built by the team behind the Atom text editor. This framework uses only web tools to generate executables for all 3 main OS families (i.e. macOS, Windows and Linux) that provide the user with a native, modern and responsive feel.

The planning was made using the best agile practices [3], eventually deciding on following a variation of XP programming. Designing and coming up with an architectural plan was a major milestone to be reached as it took a considerable amount of time to be put in place. However, as it will be discussed in future chapters, the implementation of the animation engine was a crucial and time consuming challenge that required the highest amount of effort.

The report also presents how the app was test in many various ways, as well as how it was evaluated using potential end users. In the end, we shall discuss about the roadmap of the project, why open source is and will

be vital for the development of the animator as well as some final thoughts and lessons learned throughout the process.

1.1   Aims

The goals of the project were set and subsequently refined throughout many project meetings, as well as meeting with some fellow classmates to get feedback along the way.

The main aims of the Algorithm Animator, however, have always been:

- Create an efficient and easy-to-use animator.

- Make the animator a cross-platform application that would run natively on the main operating system families: macOS, Windows and Linux.

- Provide a user-friendly interface that would make the user want to enjoy the product.

- Create at least 5-6 fully functional algorithms.

- Make the application scalable and easy to maintain.

- Test all functionality and ensure quality above quantity.

- Evaluate the product throughout the software process to meet acceptance criteria.

- Build a roadmap that would allow the animator grow even after the level 4 project has finished.

1.2   Motivation

Even if we have previously mentioned that visualising algorithms is one of the best ways of understanding how they perform, there are also other key motivational aspects behind the need of building an animator.

One such motivation is the increasing demand of software engineers throughout the industry as well as the rise in level of expertise. Developers, and students in particular, will benefit tremendously from a good grasp on how to implement correct and optimal algorithms when applying for a new job. Thus, an algorithm animator would be a good component in one's tool belt.

Another major aspect was the opportunity to create a modern tool that would make users enjoy working with it. At the moment, one of the most popular animator toolkits is the one presented in John Morris' paper [15]. Below a screenshot was taken from the app.

Figure 1.2: John Morris' Animator Toolkit.

It was written in Java, which comes with certain advantages: multi-threading support, enhanced familiarity due to university coursework and projects etc. However, the lack of proper graphical user interface components makes this a not so viable option for regular users when adopting it.
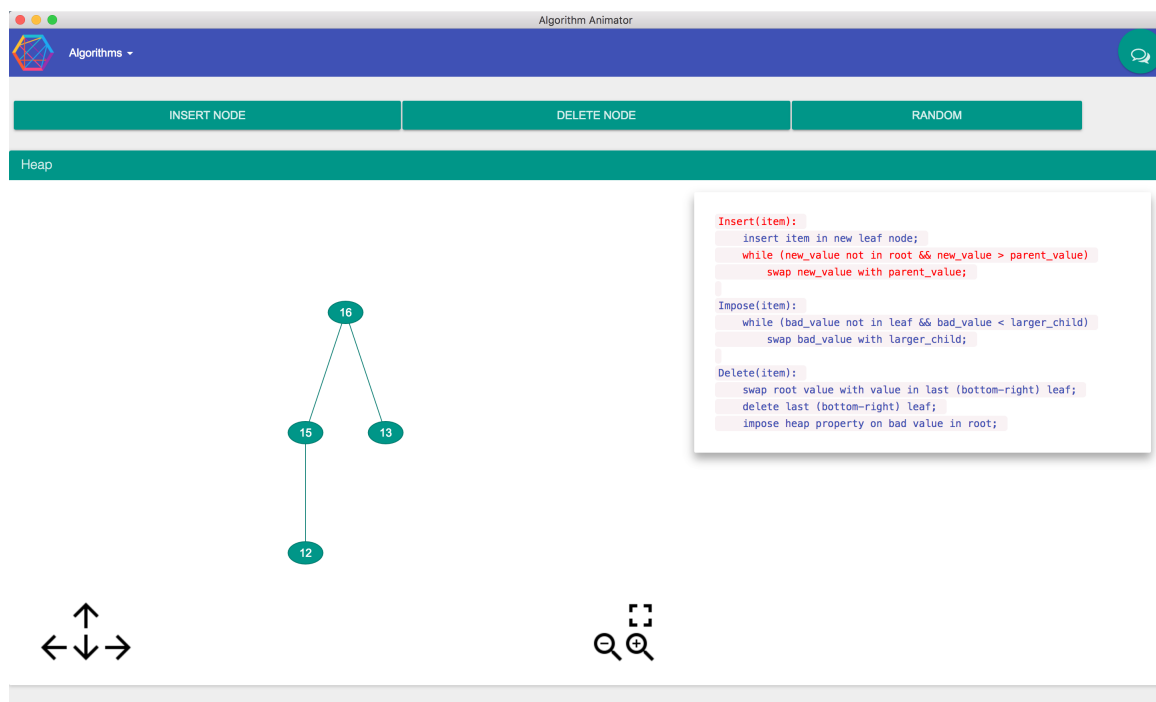

Figure 1.3: The animator presented in this report.

We can see a clear difference between the two and why one might choose the latter option due to familiarity (native feel) and easy-to-use user interface.

3

1.3   Contributions

This report serves to deliver certain main contributions:

- Shows the whole software development lifecycle of an algorithm animator.

- Presents various technologies used to build native and modern desktop apps [10] [21]

- Provides a brief overview of current animators available and background knowledge on their benefits and efficiency in teaching.

1.4   Report Content

The rest of the report will analyse the background of animators and why they were proven useful, as well as cover all the steps in gathering requirements, designing, implementing, testing and evaluating the tool.

- Chapter 2 covers work related to the purpose of algorithm animators and why they are useful

- Chapter 3 goes into how the problem was analysed and what requirements were gathered through project meetings and discussions with Algorithmics students.

- Chapter 4 shows the steps undertaken to follow the best agile methodology principles.

- Chapter 5 explains the design decisions behind the tool and illustrates various lessons learned and problems faced along the way.

- Chapter 6 goes into the implementation details of the animator.

- Chapter 7 show how extensive unit, integration and other types of testing (e.g. smoke, end-to-end) were undergone and why they were essential to the development of the application.

- Chapter 8 details the overall results of the project.

Chapter 2

Background

Firstly, we need to define an algorithm. We will use the above-mentioned definition taken from the Oxford dictionary [8]: "A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer". In our case, we will only relate to set of rules followed solely by computers. Thus, put simply, an algorithm is a set of steps defined by someone in order to solve some problem. As an algorithm can be written by anyone, there is no known number or catalog of them. However, the most important algorithms (e.g. Huffman tree encoding/decoding) are taught in any Computer Science course at universities around the globe.

Having defined what an algorithm is, let's take an actual example - Dijkstra's algorithm for finding the shortest path between two nodes [9], first published almost 60 years ago (the pseudocode was formally checked by Dr. Norman):

```
1  // S is set of vertices for which shortest path from u is known
2  // d(w) represents length of a shortest path from u to w
3  // passing only through vertices of S
4  S = {u}; // initialise S
5  for (each vertex w) d(w) = wt(u,w); // initialise distances
6  while (S != V) { // still vertices to add to S find v not in S with d(v) minimum;
7    add v to S;
8    for (each w not in S) // perform relaxation
9    d(w) = min{ d(w) , d(v)+wt(v,w) };
10 }
```

Listing 2.1: Pseudocode for Dijkstra's shortest path algorithm

As one might observe, this is not trivial to grasp. Therefore, many solutions have been created in order to help students and engineers in general understand them better. One such solution is a rather creative one, developed by students at Spatienta University, which tries to teach different sorting algorithms by dancing.
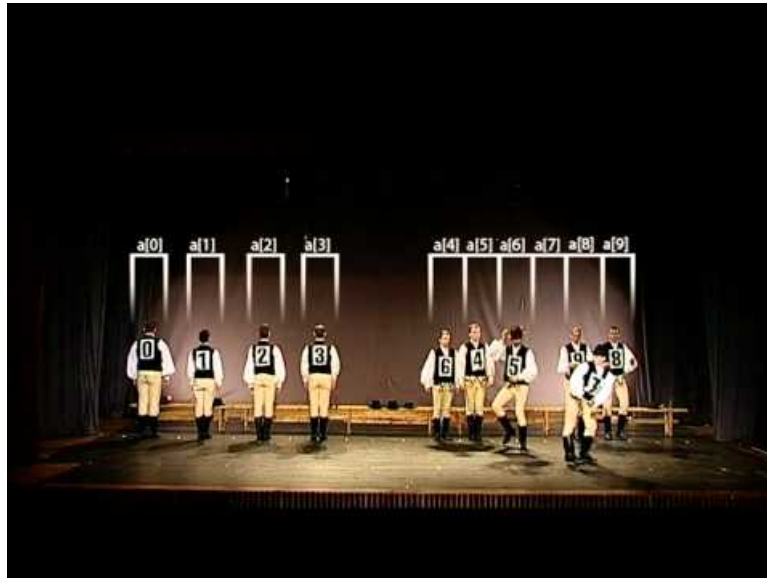
Figure 2.1: Sorting algorithms learning through Hungarian folk dancing.

This was featured on many websites and gained enormous traction among computer science enthusiasts. However, a more traditional approach that is adopted formally in universities as well is the use of animators.

Firstly, let's define what an animator is. Again, taking the definition from the Oxford dictionary [8], it is "a person who makes animated films". In our case, the person is the computer and the film is the sequence of steps required to produce an output after feeding input to an algorithm.

Usually, most algorithm animators are divided into 2 sections: one side will contain the graphics in order to represent the data structures used (e.g. in graph traversal algorithms the nodes and edges comprising the graph will be drawn) while the other side will have the code lines, the actual rules the computer needs to follow in order to produce the desired output. In general the current code line that is executed at a certain moment is highlighted, while the data structure visualized (e.g. some box that represents an array's element) changes and the user can see the output visually and immediately, making a mental connection between the code line and the change it produces.

But why an animator precisely? There are many studies that show visualization in any area can enhance learning capabilities drastically. Brenda Parker and Ian Mitchell's work [17] shows clearly how seeing the steps can greatly improve a student's understanding of what is happening "behind the scenes". On the other hand, even if animations were proven to be generally useful and beneficial for the learner, the need for seeing algorithms animated declines over time, as the study shows.

Another study by Susan Palmiter and Jay Elkerton [16] tried to find out which method was the most effective in learning new computer-based tasks. Even though animations for complex mechanisms were found not to be as effective as for some easier one (e.g. parallel programming vs. graph traversal), they are very helpful for the algorithms generally taught at university courses or used most often.

Having discussed methods of learning new algorithms better and why animators are useful, we need to take a look at previous solutions developed by other people. Apart from John Morris' tool shown above, there are many other applications available, including Galant [19] and the visualizer built by D. Galles [11].
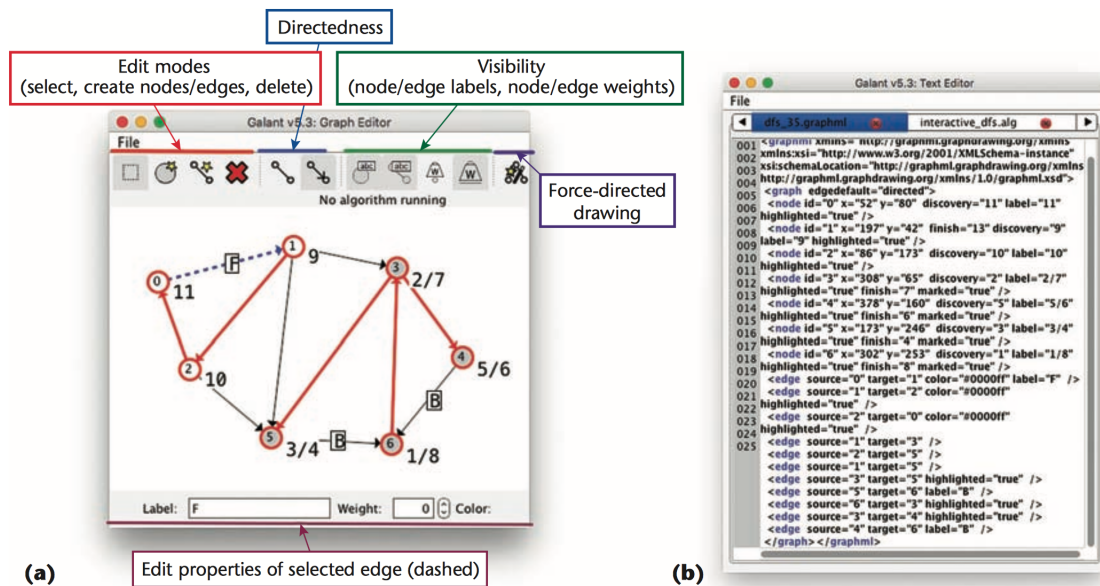
Figure 2.2: Galant Algorithm Animator.

Here we can see one of the 2 main approaches of building an animator, which is a standalone JAR application. Thus, it is written completely in Java and, as the language is pre-installed on most electronic devices, it can run on the major operating systems families.

There are obvious advantages to this approach:

- end product can run on a variety of machines with only one codebase

- large number of Java libraries facilitate an easy development workflow

- many tutorials and resources available

- java is taught at most universities in introductory courses, making it easier for newcomers to create their own application rapidly
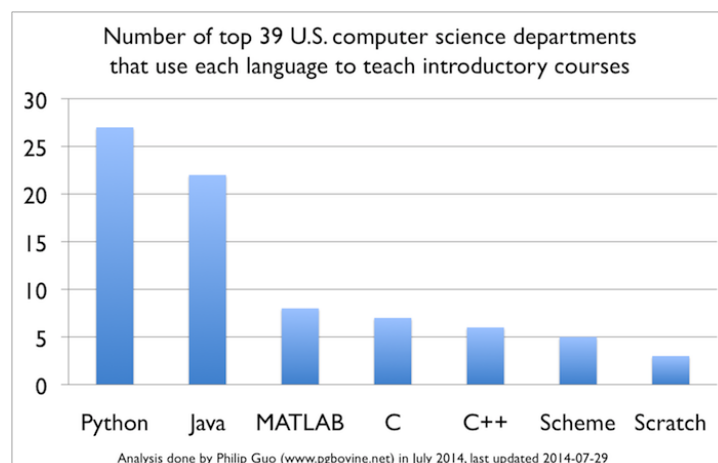
Figure 2.3: Languages used in introductory courses at top US universities.

However, there are also drawbacks:

- the most popular Java libraries for producing standalone graphical JAR applications output old-fashioned user interface

- the code for drawing on the canvas and animating in Java (using Swing or JavaFX) can become extremely verbose and unreadable

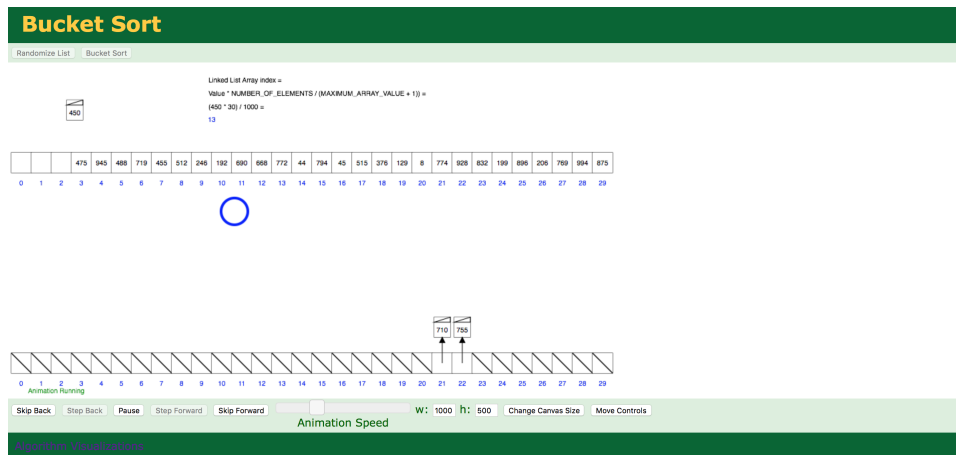On the other hand, we also have the web approach to building animators, one of them being D. Galles' application [11].

Figure 2.4: Algorithm animator built by D. Galles at University of San Francisco.

Some advantages of this solution are the following:

- the developer can use a wide variety of JavaScript tools (e.g. drawing directly on HTML5 canvas, VisJs, SigmaJs) to draw and animate objects

- the app can have a very modern and responsive feel which can make users enjoy the product

- virtually all users have at least a browser installed on their machine, thus making the animator available to everyone

However, the main drawback is that it is still a website, thus it need to be hosted (which implies additional costs) and it cannot be reached by anyone at all times due to the fact that not everyone has constant Internet access.

We have covered all the necessary background and related work before diving into how Palgo was designed and implemented. We believe that our solution is a combination of the two main approaches listed above, merging the benefits of both into a single tool.

Chapter 3

Requirements

As for every software built with best practices, requirements need to be gathered, defined, analyzed and then prioritized. As we tried to follow an Agile structure, the requirements have changed every week through student-supervisor weekly as well as end user meet-ups.

We tried to keep the requirements SMART [14], which implies they are specific, measurable, attainable, realizable and time bounded. Thus, making them follow this well established format made the software development lifecycle much faster and cleaner.

Below is a screenshot from a Trello [20] board used during one of the iterations (a separate board was used for tracking non-functional requirements as well).
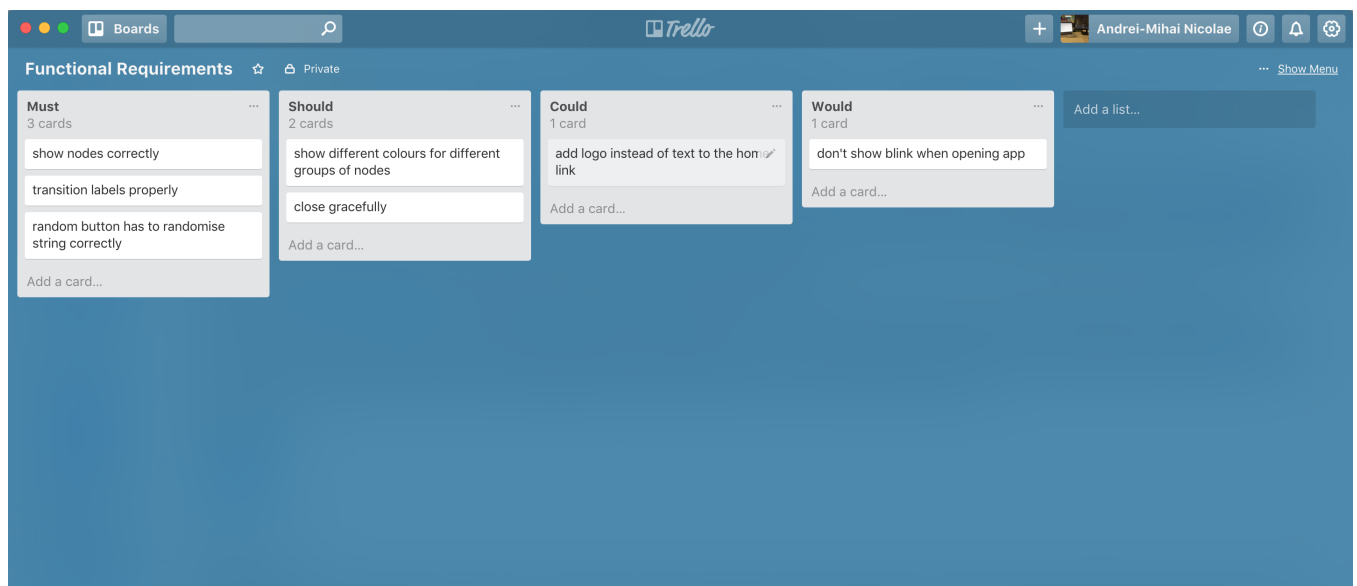
Figure 3.1: Trello board for functional requirements during one of the iterations.

The requirements followed a pipeline through which the requirement together with its rationale would get broken down into pieces continuously until user stories would be produced. One example of such a requirement would be: "Show me a correct and easy-to-understand animation for the insert function in a heap data structure.". This would then be broken down in multiple user stories, such as "Color current node that is inserted and add its label" and "Highlight code line corresponding with changing the label".

We also tried to follow the MoSCoW [5] guideline for prioritizing the requirements. The technique has

proven to be very effective and it is adopted widely in industry, even though the paper is more than 20 years old. This facilitated a smooth workflow and concise goals to be targeted.

3.1   Problem Analysis

We have analyzed the problem systematically, devising it into multiple main questions to be answered:

- who are our target users?

- what type of format will the application have (i.e. web/desktop)?

- what algorithms should be animated?

- how would the main layout be structured?

During the analysis, we needed to take into account various factors, including how much experience with algorithms and computer science in general will our users have or what algorithms would need visualization the most. As previous animators built under the supervision of Dr. Norman are used for educational purposes in the level 3 Algorithmics I course, we believe that having mainly graph and tree algorithms would benefit the project the most. However, we tried to keep the structure simple and the codebase easy-to-maintain such that other families of algorithms will be straightforward to implement. Because we want to project to be continued in the open source space, users other than students can use the animator, thus we tried to keep the requirements more general.

Also, there was much discussion around whether to use a web or a desktop approach. For reasons explained above, we resumed to implementing a desktop application using the technologies mentioned in the subsequent chapters.

We have gathered requirements and split them into functional and non-functional ones, dividing them further into smaller tasks to be implemented. At the end, we were left with user stories that could be translated directly into work tasks.

3.2   Requirements Gathering

As mentioned above, the requirements gathering was a continuous process that lasted from first week of 4th year up until the end of the project's development lifecycle. Not only that we had the weekly regular student-supervisor meetings were we could add or refine requirements, but there were also meetings with possible end users.

Every two or three weeks there were meetings scheduled with a couple of classmates from University of Glasgow's Computer Science course that would give continuous feedback after testing the newest version of the application. We met in one of the laboratories in Boyd Orr building where they were provided with an executable that could be run on their personal laptops. They were given around 15 minutes to try the features and, afterwards, they provided invaluable feedback without which the animator would not perform as efficiently as it does today.

3.3 Functional Requirements

Functional requirements target specific functionalities that the system should perform. They started with a couple of algorithms that needed to be animated. Then, functionalities such as current code line highlighting and specific button behaviors began to stack up. Eventually, we can categorize them as follows:

- animate tree/graph depending on what the user chose to do (e.g. insert/delete node)

- highlight current code line for every algorithm

- define specific behavior for every button displayed on the UI

- each window interaction should perform as expected depending on the operating system the user is using the application on

Many smaller ones appeared along the software process, but the above mentioned are the main categories.

3.4 Non-Functional Requirements

A non-functional requirement is a type of requirements that judges the system as a whole and how it operates rather than checking for specific criteria. The main non-functional requirements were rather set from the beginning. We decided to have an algorithm animator that would:

- be easy to use and let the users familiarize rapidly

- make users enjoy playing with the product

- be installed/fetched with ease

- allow end users to use it on any platform

- be efficient and not consume excessive resources

- be maintainable and scalable (i.e. let future possible contributors add algorithms easily)

Chapter 4

Agile Software Development

Good software practices are key to successful projects. We have followed Agile methodologies, especially what was taught last year in the Professional Software Development class, as well as through Ian Sommerville's "Software Engineering" book [18].

We eventually decided to follow an Extreme Programming path and we implemented the best advice taken from industry experts. Some of the key principles we implemented throughout the whole process are:

- good planning strategy that would allow us to follow a schedule; our cost estimation efficiency improved with every iteration

- frequent commits and releases that would generate continuous feedback from our potential end users

- customer involvement (i.e. student meetings every couple of weeks as well as weekly student-supervisor meetings)

- heavy refactoring as opportunity arose

- emphasis on simplicity in design and implementation

- tracking bugs and issues in general allowed us to be always informed and include necessary items for specific milestones so that the quality would always have high standards

All these ensured a good workflow with results that could be remarked immediately. We believe that a major part of the success of the project was embracing the above Agile methodologies.

4.1   Extreme Programming

We have followed the approach first coined by Kent Beck called Extreme Programming [1]. It is an Agile technique through which good practices are most times pushed to "extreme" levels. XP has proven to be very effective as many companies in the industry use it in their development lifecycle. Some principles it promotes which we have followed are:

- frequent releases based on requirements gathered incrementally in meetings (e.g. supervisor-student meetings)

- constant refactoring to promote simplicity

- customer involvement throughout the whole process (i.e. getting feedback every 2-3 weeks from students that might use the application)

- people over processes, which means collective ownership of the code as well as avoiding exhausting long working hours

Burndown charts are another great tool used by people using XP in their schedule to track down progress. You start off by defining the number of user stories you want to complete in an iteration and, then, divide that number by the number of days the sprint has (in our case it was 7 days). Therefore each day you should implement a certain number of user stories, constantly, such that at the end of the iteration you are left with none. That is the most productive outcome. Also, because this was a student project, we will count weekend days as working days.

During the entire process we tried to monitor the progress and see how we are performing. Below you can see a burndown chart from one of the November weekly iterations.
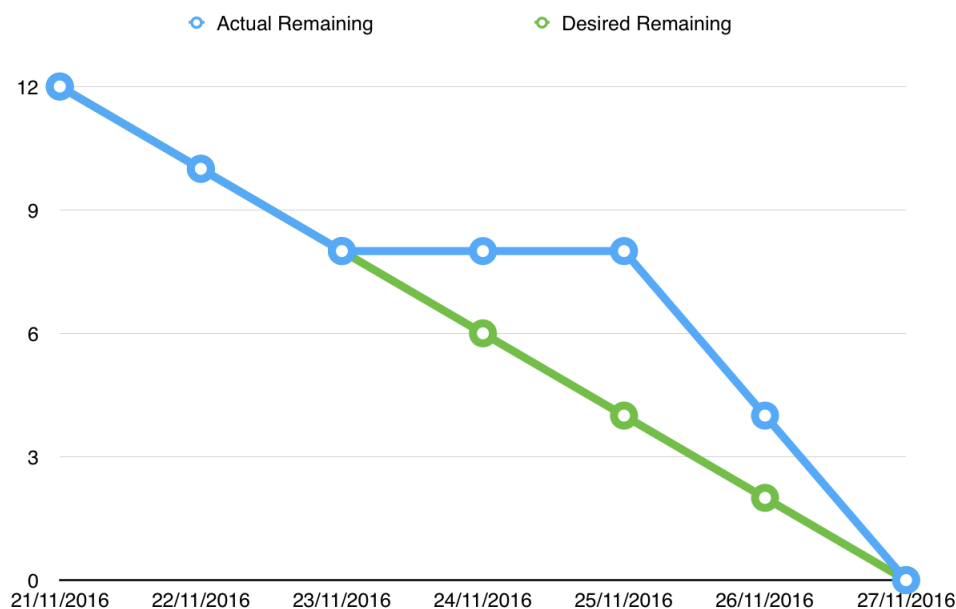
Figure 4.1: November iteration burndown chart.

As one can notice, there is quite a discrepancy between the desired number of user stories per day and what we actually completed. This was because of lack of consistency, shown in the graph between days 23 and 26, where there were no user stories finished. However, below is a burndown chart corresponding to one of the last iterations of the project.
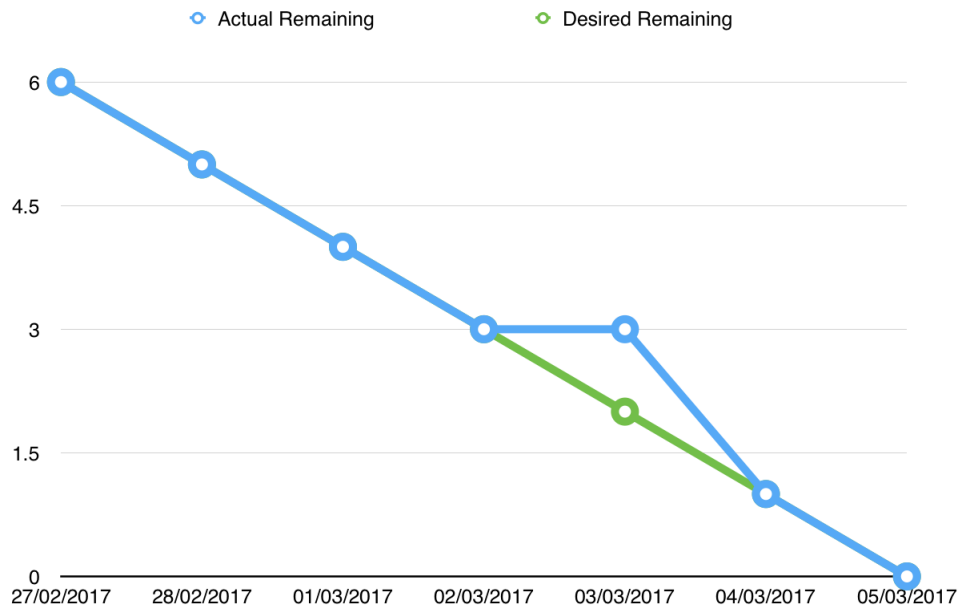
Figure 4.2: February-March iteration burndown chart.

There are noticeable difference between the 2 iterations: firstly, we started with fewer user stories to be completed in the latter sprint which was much more feasible to finish and it didn't make us work extra hours (like in the first burndown chart example), thus violating one of the extreme programing principles. Moreover, there was only a slight discrepancy from the schedule on March 3rd where there were no user stories completed. However, this is relatively minor and we believe that for a two people project, this was an amazing achievement that could only be reached gradually.

Summing up, following all the good practices of XP has brought us great benefits and a working schedule that allowed us to implement all the requirements gathered in user and supervisor meetings.

4.2 Planning

Planning is a vital part of every Agile project - it involves setting deadlines for tasks needed to be implemented by developers. However, as some tasks are not trivial to estimate, multiple strategies have been invented so that team members can effectively set number of days (or other units of work) for every user story.

In our case we adapted the most effective Agile method, which is Planning Poker [12]. In this practice, all developers are given decks of cards and when a user story is brought up, every developer shows a card which has a value. That value will represent the number of units of work needed to complete the story. Then, after everyone shows their estimation, discussions emerge among the developers until a consensus is reached. In our situation, we did not use a deck of cards during the student-supervisor meetings, but we both estimated the number of days needed to implement a feature and, if our opinions diverged, we discussed until we had the same duration in mind.

Below it can be observed how we kept track of all the user stories using Trello cards. We added the estimation in the description, as well as a description using the popular format "As a.../I want to.../So that..." [6] to always know the rationale behind every implemented feature.
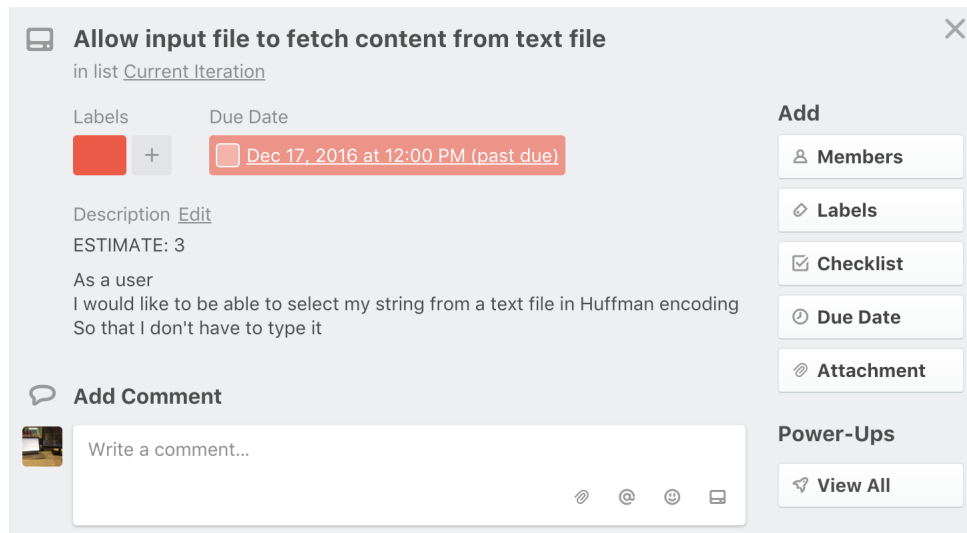
Figure 4.3: Trello card for estimating a user story.

This method was very effective and as we gained more experience after several iterations, we would come to a conclusion much faster and the user story cost estimations became more accurate.

4.3 Continuous Integration

We have used another great tool heavily promoted by Agile methodologies which is Continuous Integration. CI means merging all working copies of everyone who is developing on the project to a shared mainline multiple times a day [2]. This is an amazing tool for checking that the builds do not break regardless of the changes made and that quality is of primary importance.

In our case, we have used CircleCI [4] as the tool for CI. It was configured such that after every commit, the code would be merged into the main branch, tested to see it everything is working as expetec and, only after, the code would be pushed.

Figure 4.4: CircleCI status showing red/green depending if the build was successful on GitHub.

As it can be seen above, there is a small green check (or a red x if the build did not pass) on the GitHub repository of the project telling the developers if anything went wrong. This was a great asset to the development lifecycle, informing us whether we needed to add some bugs to our tracking system to be solved later or everything performed exactly as it should have.

Not only that we applied continuous integration to our workflow, but we also used it properly: as mentioned above, the definition of CI means the developers should push their local changes to the main repository multiple

15

times a day. Every day of development meant multiple commits and multiple pushes such that we made sure the builds were all passing and that our code was not broken.
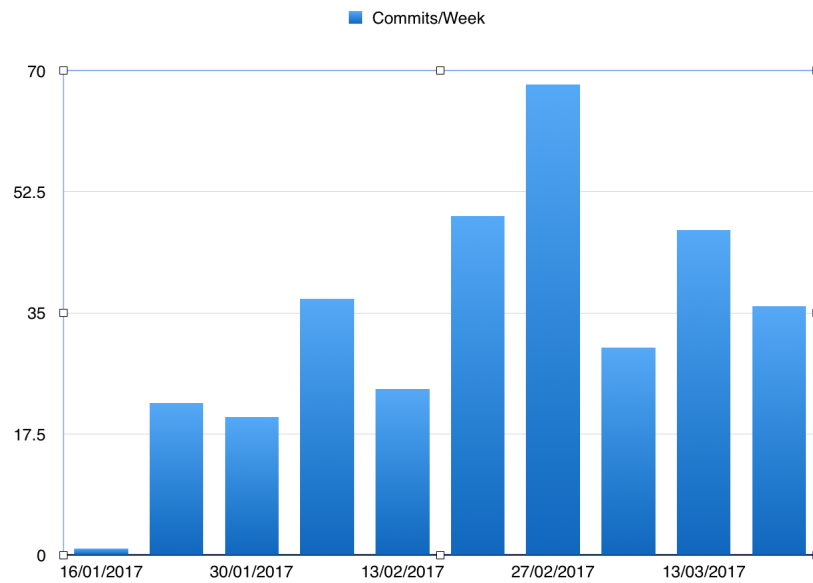

Figure 4.5: Commit frequency in the last 10 sprints.

This figure shows the commit frequency per week for the last 10 iterations. As it can be observed, apart from first week when the university semester just started, the number of commits was very high every sprint, thus following the practice of committing and testing the quality of our product often.

4.4   Issues & Bug Tracking

Issues and bugs occur all the time in every piece of software. Because the userbase was rather limited, I updated the database manually from what we (i.e. Dr. Norman and myself) found not working as expected and what the students marked as functionality not working properly or wanted features. This was a very useful practice to follow because it also promoted continuous feedback on changes from potential end users, thus implementing the Agile manifesto properly.

Talking about actual implementation, we have kept a custom database on GitHub in order to be always up-to-date with what is going on and spots where we lack quality and efficiency. We have made use of the custom labels GitHub already provides, among which the most used ones have been:

- bug - here we marked flaws in the system (or the students who have been providing constant feedback)

- enhancement - we added here the features most wanted by our users or by ourselves; this was refined from the requirements gatherings previously mentioned that ended up as multiple user stories eventually

- question - these were marked for research purposes - one example might be: can we change the shape of nodes dynamically as the algorithm animation makes progress?

As an example, the following figure will show how issues were continuously tracked throughout the software process. This screenshot was taken during one of the iterations.

Figure 4.6: Example of issue and bug tracking in the GitHub provided Issues page.

As it can be seen in the image, there is also a milestone tab assigned to each issue. Because the screenshot is from an earlier sprint, there were not milestones attached, but since January every issue had one so that we would plan effectively and reach all goals by the negotiated deadline.

Chapter 5

Design

This chapter will cover the design decisions taken throughout the software process. It will delve into why the main technologies were chosen eventually (e.g. Electron, Vis.Js) and why they proved to be the most effective to achieving our purposes.

5.1 Architecture

Because we would have a wide variety of users (students, experienced developers, university lecturers, enthusiasts) one needs to take into account a range of different perspectives. Therefore, architectural design and focus on simplicity is vital to the success of every software. We have tried to use UML diagrams in order to visualize the system as well as possible before proceeding with the implementation. Here is our final candidate for the component diagram.

5.2 Native Desktop App vs. Jar

5.3 Electron

5.4 Vis.js

5.5 Material Design

5.6 Compromises

Chapter 6

Implementation

6.1 Project Structure

6.2 JavaScript and Multi-Threading

6.3 JS Animation Engine

6.4 Extra Features

6.5 Lessons Learned

6.6 Issues Faced

Chapter 7

Testing

7.1 Unit Testing

7.2 Integration Testing

7.3 Prototype Evaluation

7.4 Results

Chapter 8

Conclusions

8.1   Open Source

8.2   Project Roadmap

8.3   Final Thoughts

8.4   Acknowledgements

Bibliography

[1] Kent Beck. Embracing change with extreme programming. *Computer*, 32(10):70–77, 1999.

[2] Grady Booch. *Object Oriented Design: With Applications.* Benjamin/Cummings, 1991.

[3] Bharat Choudhary and Shanu K Rakesh. An Approach using Agile Method for Software Development. *Innovation and Challenges in Cyber Security (ICICCS-INBUSH)*, 2016.

[4] CircleCI. CircleCI Website. `https://circleci.com`.

[5] Dai Clegg and Richard Barker. *Case method fast-track: a RAD approach.* Addison-Wesley Longman Publishing Co., Inc, 1994.

[6] Connextra. Connextra User Story 2001: ConnextraStoryCard. `https://en.wikipedia.org/wiki/User_story#cite_note-connextra-4`.

[7] Sarah A. Douglas Cristopher D. Hundhausen and John T. Stasko. A Meta-Study of Algorithm Visualization E¡ectiveness. *Journal of Visual Languages and Computing*, 13:259–290, 2002.

[8] Oxford Dictionaries. *Oxford English Dictionary*. OUP Oxford, 2012.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] Electron. Electron Website. `https://electron.atom.io`.

[11] David Galles. David Galles Algorithm Animator. `https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`.

[12] James Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Renaissance Software Consulting*, 2002.

[13] Steven et al. Halim. VisuAlgo Website. `http://visualgo.net`.

[14] Mike Mannion and Barry Keepence. SMART Requirements. *ACM SIGSOFT Software Engineering Notes*, 20(2):42–47, 1995.

[15] John Morris. A toolkit for algorithm animation. *Proceedings AEESEAP*, 2004.

[16] Susan Palmiter and Jay Elkerton. An evaluation of animated demonstrations of learning computer-based tasks. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, 1991.

[17] Brenda Parker and Ian Mitchell. Effective methods for learning: a study in visualization. *Journal of Computing Sciences in Colleges*, 22(2):176–182, 2006.

[18] Ian Sommerville. *Software Engineering.* Pearson, 2015.

[19] Matthias F Stallmann. Algorithm Animation with Galant. *IEEE Computer Graphics and Applications*, 37(1), 2017.

[20] Trello. Trello Website. `https://trello.com`.

[21] VisJS. VisJs Website. `http://visjs.org`.