Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

May 13, 2020

Stack and Heap Diagrams: A Graphical Editor

Davide Ciulla

Abstract

Teaching students the fundamentals of programming is not an easy task, and Stack and Heap diagrams are very useful because they allow to really understand what happens when writing code. However, they are often drawn either on paper or with legacy software that does not feel adequate in 2020. In this project I developed a web-based graphical editor for building this kind of diagram in real-time, using modern web technologies like React and Sass.

Advisor
Prof. Matthias Hauswirth

Advisor's approval (Prof. Matthias Hauswirth): Date:

Contents

1 Introduction **2**
1.1 Theoretical Background . . . 2
1.2 Problem . . . 2
1.3 Goal . . . 2
1.4 Approach . . . 2

2 State of the Art **3**
2.1 Informa Clicker - SAPE Group . . . 3
2.2 Objects First with Java - David J. Barnes and Michael Kölling . . . 3

3 Project analysis and requirements **4**
3.1 UI/UX analysis . . . 4
3.2 Requirements . . . 5
3.3 Extra features . . . 5

4 Project design **6**
4.1 UI/UX research . . . 6
4.2 Components tree . . . 8
4.3 UI functionalities . . . 9
4.4 Storing global states . . . 9

5 Implementation issues **11**

6 Testing **12**
6.1 Methodology . . . 12
6.2 Results . . . 12
6.3 Comments . . . 12

7 Conclusions **13**
7.1 Future work . . . 13

1 Introduction

In this section I am going to describe all the thoughts that went into this project at the very early stages, from the moment I first discussed with Prof. Hauswirth about the project, to the moment right before starting the "pre-production" phase (see **section 3**).

1.1 Theoretical Background

The stack and the

1.2 Problem

1.3 Goal

The goal of this project is to create a progressive web application that allows student to very easily create Stack and Heap diagrams to visualize the structure of their code and better understand what is actually happening under the hood. This kind of diagram can give very useful insights both to the student and the professor.
This application is planned to be integrated into Informa (prof. Hauswirth's web platform for his Programming Fundamentals 2 course), but it can also be used as a standalone application, because of the nature of progressive web apps. That way, students can not only use it in class during class exercises, but also for instance at home while studying on their own.
Furthermore, a JSON file is constantly kept up-to-date with every state change happening, which can be used for automatic testing and potentially instant feedback on the correctness of their diagrams.

1.4 Approach

Because of the integration on Informa, this application needs to be as scalable and modular as possible, and it is with this mindset that I have implemented every single feature.

2 State of the Art

In this section I am going to showcase projects that I consider to be the current state-of-the-art in terms of visual representations of stack and heap usage in programming.

2.1 Informa Clicker - SAPE Group

A clicker is a special purpose remote control used in traditional group response systems, where an instructor projects a multiple-choice question on the classroom projector and students use these tools to submit their answers by pressing one of the buttons. The advantage of using such systems is that the instructor has immediate feedback on the students understanding, by seeing their responses aggregated in the form of a histogram.[1].

Informa [2] is not only a really nice teaching platform, but it also has a clicker tool available for download, that students can use to solve interactive problems. One of the problems they can solve is showing the correct state of a Stack and Heap Diagram at a given point of a code execution.

This clicker tool was developed by professor Matthias Hauswirth, my bachelor thesis advisor, for the SAPE Research Group [3].

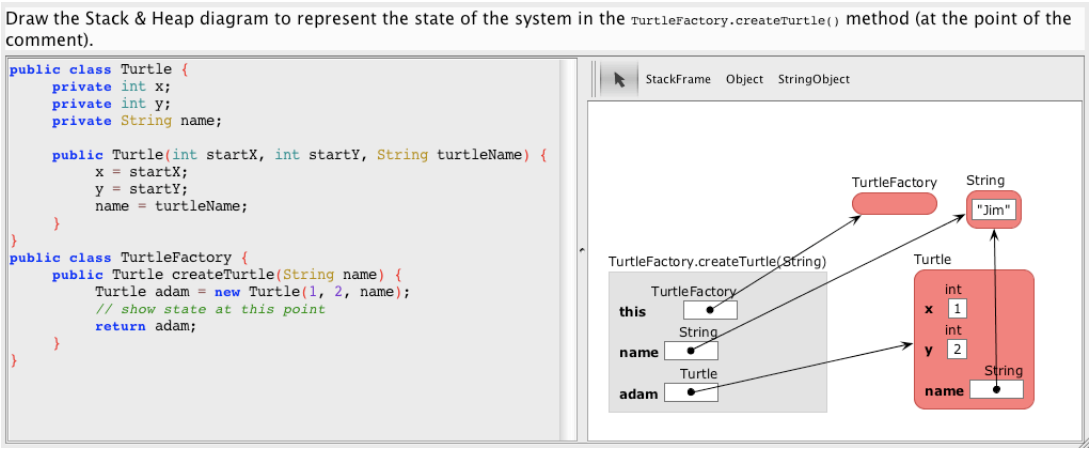

Figure 1. Stack and Heap Diagrams in the Informa clicker tool

2.2 Objects First with Java - David J. Barnes and Michael Kölling

Object First with Java is a popular textbook for people that have zero experience with object oriented programming and it is used in many school throughout the world. This book comes with a small *IDE* [4], which is very helpful for beginners because in its main view there is a class and object diagram, which shows you the classes you have written, their instances and all the connections that are present in your code.

[1] http://sape.inf.usi.ch/informa
[2] https://informa.inf.usi.ch/
[3] http://sape.inf.usi.ch/
[4] Integrated Development Environment

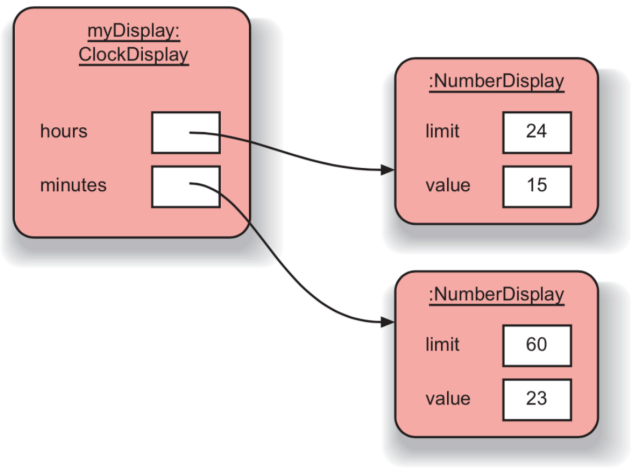

Figure 2. Object diagram in the textbook *Object First with Java*

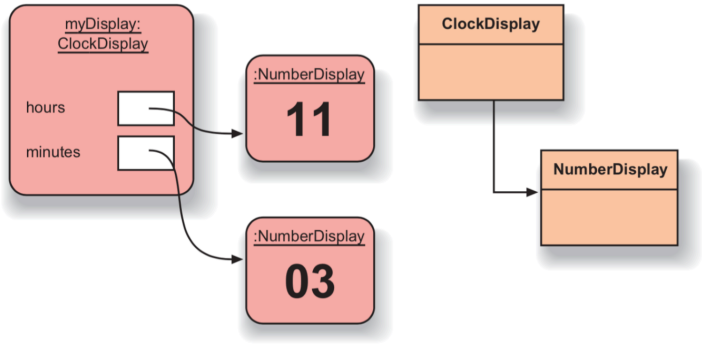

Figure 3. Class diagram in the textbook *Object First with Java*

3 Project analysis and requirements

In this section I am going to talk about the pre-production phase of the project, from the UI/UX analysis of the main Informa clicker tool, to the basic requirements that were decided as a baseline for the project.

3.1 UI/UX analysis

As already mentioned in **section 1.3**, the main goal of this project was to create an improved and web-based version of the already existing Stack and Heap Diagrams in the Informa clicker tool. In order to understand what worked and what did not work, I did an analysis of its user interface (UI) and user experience (UX) and here is what I found:

UI - What works:

- Arrows look nice and always point to the center of the target block, avoiding arrows overlapping.
- The interface looks tidy, there is not anything that looks out of place.

UI - What does not work:

- Not enough overall white space [5].
- Colors can be improved.
- All buttons that appear on mouseover events substantially clutter the interface and hide too many elements.
- Space dedicated to the diagram is relatively small compared to the whole screen.
- The buttons in the top toolbar are not styled as buttons.

UX - What works:

- The interface is generally very intuitive and I did not have any issues pursuing any task.

UX - What does not work:

- The buttons at the top of the diagram, by not looking like buttons, make their purpose unclear to the user.

As you can read from this analysis, the result is pretty clear: the Informa clicker feels great, but it lacks a proper user interface to better deliver what already works.

3.2 Requirements

After running the UI/UX analysis and establishing that what had to be improved was the UI, it made sense to set as minimum requirements all the features the clicker already has. In particular users can:

1. Freely create stack frames and heap objects by dragging and dropping them onto the interface.
2. Add and remove primitive and reference/instance variables from a given block [6]
3. Create arrows by clicking on a reference variable, holding that click and releasing it on a heap object.
4. Drag blocks around.

Plus something new:

1. A JSON file must be updated with every state change within within the diagram (more on this is **section 4.4**).
2. Clear division between stack and heap.

3.3 Extra features

Alongside the requirements, we have also agreed upon a set of possible bonus features that could be added if I would have finished the project early:

1. The JSON file which holds the diagram current state, can both be downloaded and uploaded into the web application. Uploading that file acts like a save state in a videogame, because the whole interface is reloaded as it was left at the time of downloading the JSON file.
2. Panning and zooming around in the interface.
3. Add a multi-step functionality to the arrows, which would allow to customize the path of a given arrow.

[5] `https://blog.bannersnack.com/white-space-in-graphic-design/`
[6] A block is a general term for stack frames and heap objects.

4 Project design

In this section I am going to walk you through the design process, not only coding-wise but also from the graphics perspective, elaborating all the decisions and solutions I had to make during this critical development phase. The design process can, in fact, either make or break the entire project. Making awful decisions at this stage, causes unavoidable problems down the line and it is the precise reason why I took maybe a bit more time than what I had initially planned, and only start coding when we were satisfied enough on the result.

4.1 UI/UX research

In **section 3**, we saw how the biggest problem of the Informa clicker tool was the user interface and because of that, the first week and a half were completely spent sketching ideas on paper, bringing the best ones in Figma [7] and ultimately creating the high-fidelity mockups [8] to show to prof. Hauswirth in order to get feedback.
The first step was to figure out how to divide the stack and heap, since one of the requirements was about having a clear distinction between the two. In my mind there were two possible ways to approach this: either using a floating or docked stack.

Figure 4 shows the first version of the user interface with a floating stack, and even though it is indeed separate from the heap and it might look somewhat cool, I did not like the clunkiness of having such an imposing object in the scene. However, one important solution was carried on until the end of the project, and that was the block data encapsulation, a fancy expression to indicate that every object has all its information contained within its rectangle shape. Compared to the Informa clicker, it is not a big change, but it certainly helps "glueing" the data and offer clearer separation between blocks.
As far as stack frames and heap objects, I tried to adopt a more schematic approach, and for each variable reserve one row for each detail (name, type and value). However, this idea was eventually dropped because it took up too much space, and the number of variables that were visible without causing the stack content to be scrollable, was extremely low.

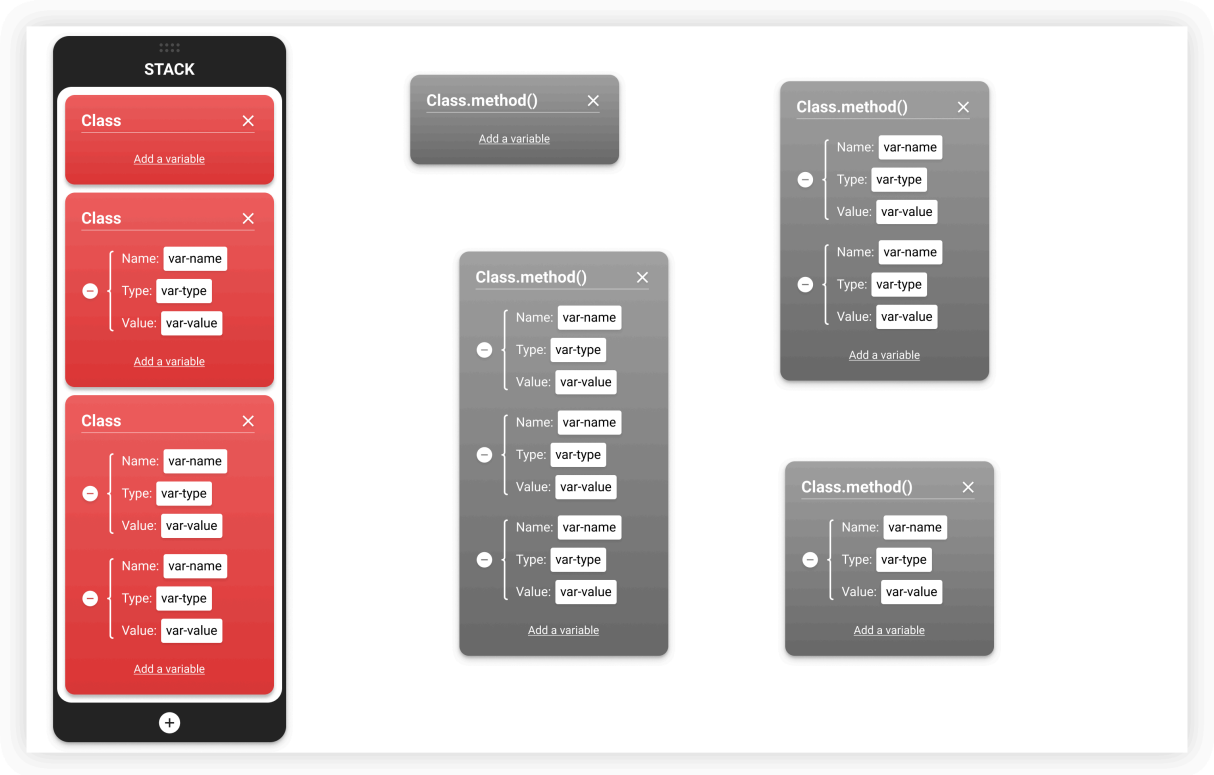

Figure 4. First version of the user interface, with a floating stack

[7] My go-to UI/UX design editor (`https://www.figma.com/`).
[8] Full size model of a design or device, used for product presentations or other purposes.

Figure 5 shows an intermediate version of the design, which is substantially different from the previous one shown in **Figure 4**. In fact, I tried to develop the idea of a docked stack. Actually, if compared to having it floating it might even have a few disadvantages, such as the impossibility of moving it around and occupying a fixed amount of space which will not be available to the heap. On the other hand, it is stylistically a lot nicer on the eyes and it definitely delivers the best separation between the two regions. I also started playing with colors, in order to bring out a bit more playfulness from a rather schematic UI.
Besides stack and heap being separated by a divider, each has a header which labels the region where the user is working and a plus button to add new stack frames and new heap objects.

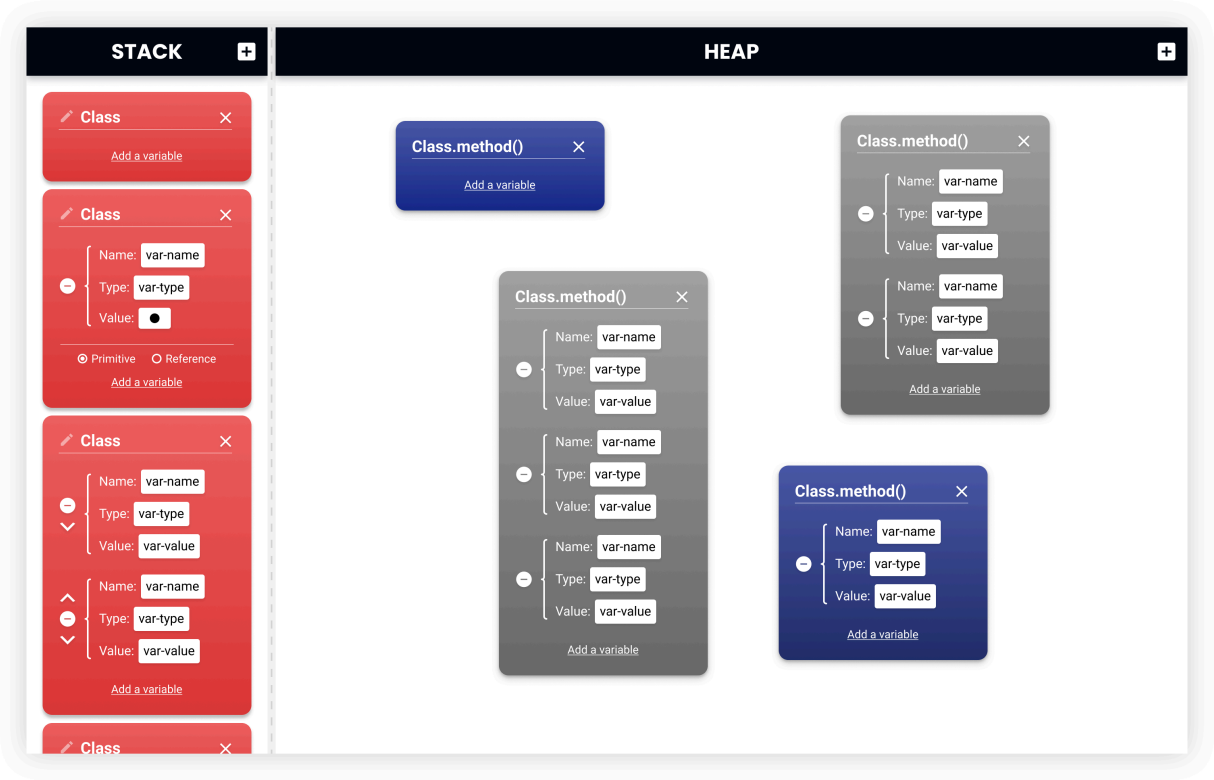

Figure 5. Intermediate version of the user interface with separate regions for stack and heap

After several revisions, we eventually found a design solution which was good enough to continue with the rest of the development phase and you can see that in **Figure 6**. Compared to **Figure 5** it presents some clear improvements, such as:

1. Redesigned variables. Each one is now represented as a separate box inside the enclosing region block.
2. Regions headers now display the number of blocks contained in it.
3. Better color schemes.
4. A toolbar is now docked on the right side of the window. It can be used to trigger some actions.
5. Heap objects now have a handle at the top, so that users can drag them around.
6.

However, at this stage we have also started to notice how blocks were really big as opposed to the ones present in the clicker tool. However, we decided to keep this design because ultimately the pros outnumbered the cons and this issue could have been addressed during development.

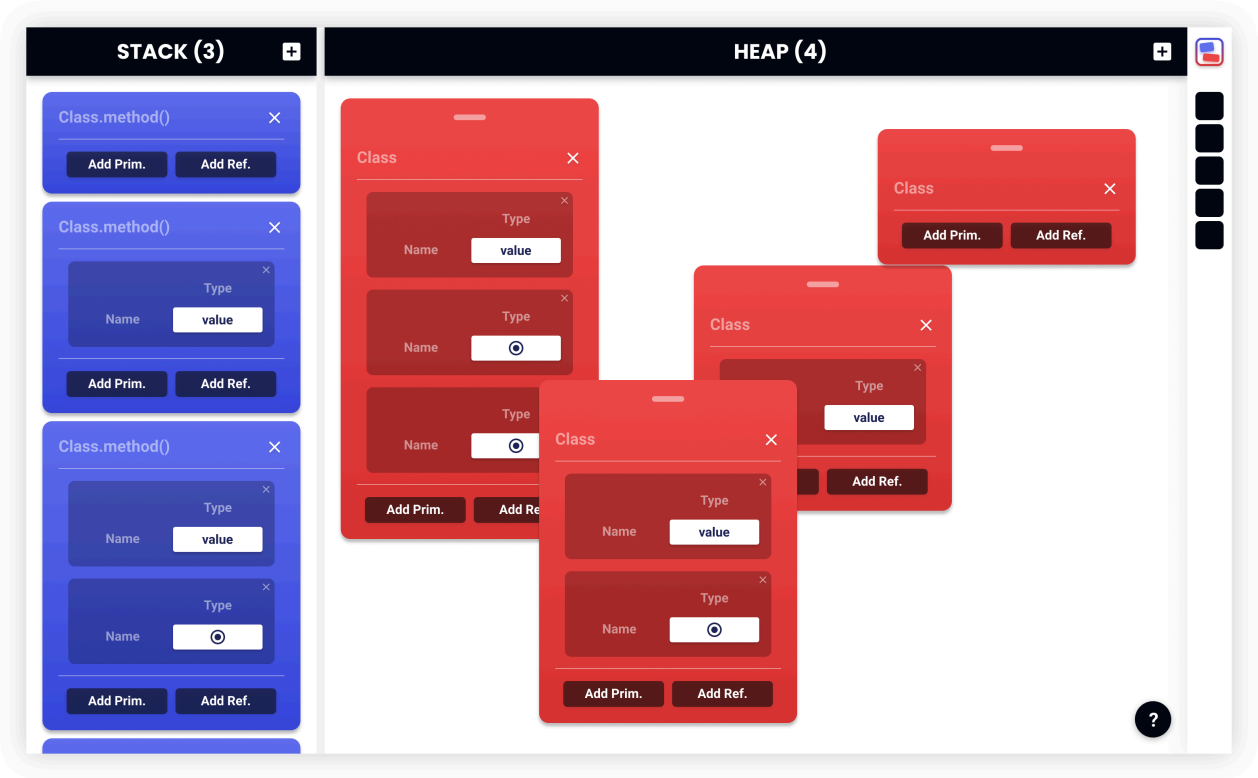

Figure 6. Final mockup of the user interface

4.2 Components tree

Although stack and heap are separate regions that expect different user interactions, they share important similarities. In fact, they both:

- Have a header, indicating the name of the area and the number of blocks that currently populate it.
- Have blocks, which look and function in the exact way, except their color schemes.

Because of these similarities, I have encountered some problems trying to extract the same features in common components in order to avoid as much code repetition as possible (see **section 5** for a more detailed explanation about this topic). **Figure 7** shows the components tree on which I eventually settled.

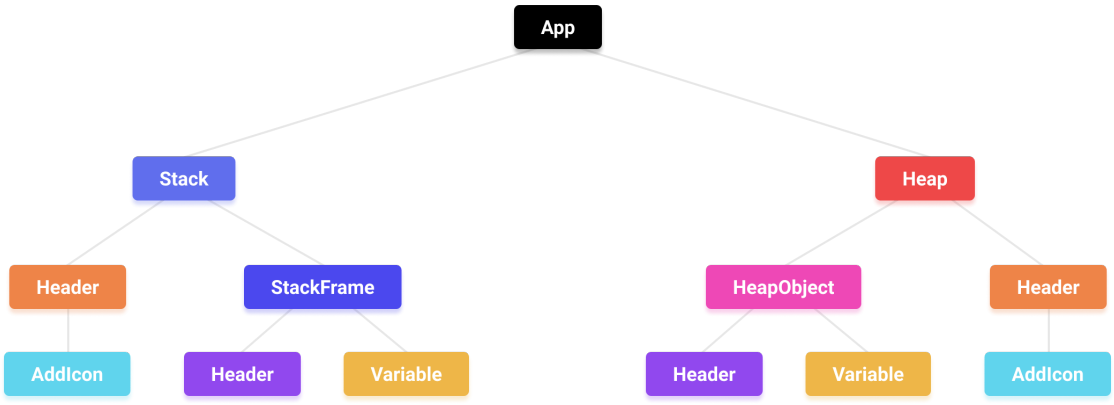

Figure 7. Components tree visualization

Starting from the top of the tree, **App** is the highest component and it is mainly used to define the global layout of the web application. Some top-level computations, such as resizing the stack width is done within this component.
Following the two branches, **Stack** and **Heap** are the components that represent the two areas of memory, and they keep and manage all the information related to their area within the application.
If we look at **Figure 7**, we can really see how stack and heap are really similar. In fact, the only difference in their subtrees is the type of building block they can store. The stack stores **StackFrames**, while heap stores **HeapObjects**. They look very similar, but the data they generate is different. Here is a snippet of the JSON file created by the web application:

Figure 8. Left: StackFrame, **Right**: HeapObject

As we can see, heap objects have more data due to the fact that you can drag them around (hence the *position*) and while doing so the one you are currently moving, goes on top of the other ones (hence the *depthIndex*).
Going back to the components tree, every block has a **Header**, where users can write either the name of the class method name or the class name, and one or more **Variables**.
Both the stack and the heap have a **Header**, which shows the region name and the number of frames/objects created. Headers at the region level contain a component called **AddIcon**, which that handles the creation of new blocks.

4.3 UI functionalities

4.4 Storing global states

In React, data is usually shared via props from parent to children: they can receive functions as props are send some data back to the parent using callbacks, but they cannot pass props back to the parent component.
Passing props through many levels is called *props drilling* [9], and it is generally seen as a situation to avoid if props needs to traverse more than 3 levels of componenets, because it becomes difficult for the developer to clearly understand what is going on, and by doing so you end up storing data in intermediate components, which is never going to be used, effectively cluttering the component

Depending on the task, there are generally two approach to solve (or at least improve) the situation:

1. Composition

2. State management tools (e.g., *Redux* [10] and *Context* [11])

Both approaches have pros and cons, but for global data transferring it makes more sense to use a state management tool, which is what I did. I used the native React Context API in conjunction with the relatively new *React Hooks* [12]. By doing so, I have been able to access data from any level of tree. However, I must also confess that Context, while being great at what it unlocks, introduces a bit of chaos when overused, which is sadly my case. This is maybe the aspect of this web application I would like to revisit the most in a few years, when I will be more experienced.

[9] https://newreactway.com/what-is-prop-drilling-in-react/
[10] https://react-redux.js.org/
[11] https://reactjs.org/docs/context.html
[12] https://reactjs.org/docs/hooks-intro.html

In particular I used the following contexts:

- *Context 1*:
- *Context 2*:
- *Context 3*:
- *Context 4*:
- *Context 5*:
- *Context 6*:

5 Implementation issues

6 Testing

6.1 Methodology

6.2 Results

6.3 Comments

7 Conclusions

7.1 Future work

References