Università
della
Svizzera
italiana

Faculty
of Informatics

Bachelor Thesis

May 31, 2020

Stack and Heap Diagrams

A Graphical Editor

Davide Ciulla

Abstract

Teaching students the fundamentals of programming is not an easy task at all, but luckily there are useful tools like Stack and Heap diagrams that can facilitate this process, by providing professors ways to easily assess the students understanding on a given topic. However, these diagrams are often drawn either on paper or with legacy software that must be installed on the computer. In this project, I developed a web-based stack and heap graphical editor, using modern front-end web technologies like React and Sass.

Advisor
Prof. Matthias Hauswirth

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

Contents

1 Introduction

In this first 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 the project with prof. Hauswirth, to the moment right before starting the pre-production phase (see **Section 3**).

1.1 Theoretical Background

Before continuing, I need to clarify some theoretical concepts that will be used many times throughout this report. In particular stack and heap, notional machines and clicker tools.

1.1.1 Stack and heap

In the world of computer science, stack and heap are two very important memory allocation areas. The stack is a LIFO (*Last In First Out*) data structure responsible for keeping track of function calls. Every time a function is called, a new stack frame is pushed onto the stack, containing its parameters, its local variables and the return address [1]. Stack frames only exist during the execution of a function, and are popped from the stack when the latter has completed its lifecycle.
Conversely, the heap does not have a LIFO structure and it is independent from the program execution, being a segment of memory dedicated to dynamic memory allocation. When we program and allocate memory, we create objects on the heap. These objects do not disappear automatically like stack frames, and in some programming languages (like C and C++) they must be manually deallocated by the programmer (bad heap management can cause memory leaks - a failure in a program to release discarded memory, causing impaired performance or failure. [2]).

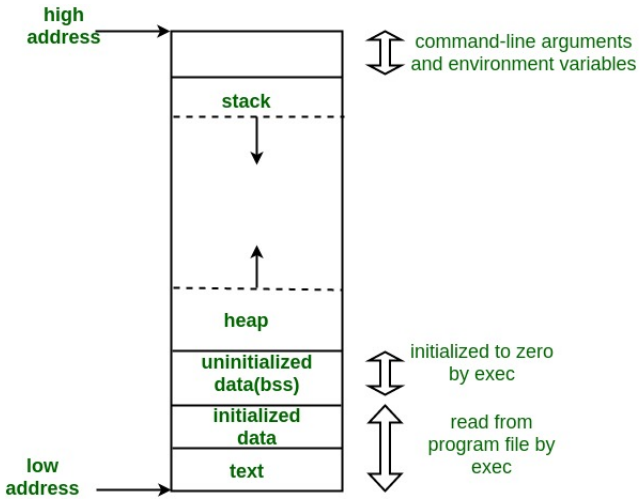

Figure 1. Generalized memory layout of a program

1.1.2 Notional machines

A notional machine can be described as a set of abstractions that define the structure and behavior of a computational device [4]. The goal of such a machine is not to be an exact replica of the real device, because by providing abstractions it is hopefully easier for students to understand the real machine [2].
Students build their mental models about the notional machine through teaching, learning methods and materials used in programming classes [5].

[1] https://medium.com/fhinkel/confused-about-stack-and-heap-2cf3e6adb771
[2] https://en.wikipedia.org/wiki/Memory_leak

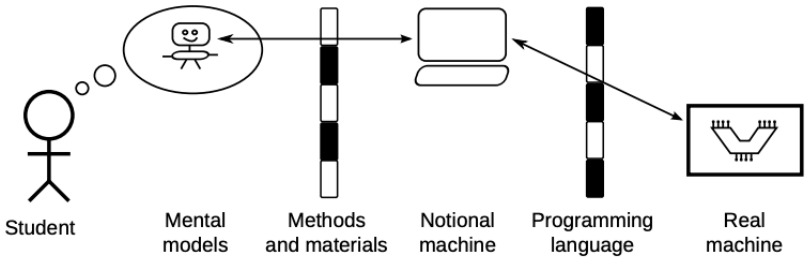

Figure 2. Process of creating mental models about notional machines.

1.1.3 Clicker tools

According to the Luce Group[3], "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." [4]

1.2 Motivation

As I am going to show in **Section 2** about state-of-the-art projects for programs visualizations, one can do a quick Google search and see that there are already tools in the real world that help students grasp important computer science topics, by showing the state of their code at each point of its execution.
However, the majority of them are automated, meaning that they take real code as input and automatically create diagrams as outputs. With such programs, **students cannot demonstrate their understanding**, and *that* is exactly why I am making this application.

1.3 Goal

The goal of this project is to create a web application inspired by the Informa clicker tool, that allows students 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 to 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. 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 JavaScript object containing the data representation of the diagram 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 Use cases

This application was developed with the two following use cases in mind:

1. Teacher provides some code and an arbitrary breakpoint, and the student must create a diagram mirroring the state of the program when the breakpoint is reached.

2. Teacher draws a diagram and student must be able to describe the current state of the program execution.

[3] `http://luce.inf.usi.ch/`
[4] `http://sape.inf.usi.ch/informa`

2 State of the Art

In this section I am going to showcase projects that I consider to be state-of-the-art in terms of visual representations of code. Not all of them are related to stack and heap diagrams, but they do still set a benchmark for graphical output and ease of learning.

2.1 Informa Clicker - Luce Group

Informa [5] is not only a 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 Luce Group.

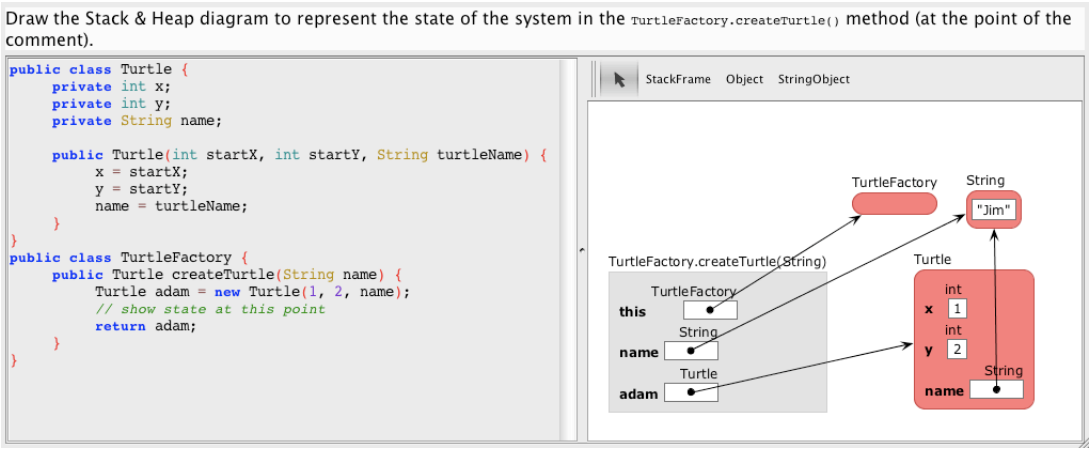

Figure 3. Stack and Heap Diagrams in the Informa clicker tool

2.2 BlueJ - King's College London

BlueJ [6] (see **Figure 4** and **Figure 5**) is free Java IDE (*Integrated Development Environment*) designed for beginners, developed by the King's College London and officially supported by Oracle, the company that owns the Java platform. The aspect that makes BlueJ stand out from the other IDEs is its powerful main view, which shows an interactive diagram of the different elements that compose your source code. It also renders arrows that are very useful, because they show how the different classes are interconnected. Objects are shows too, but they do not present any arrow. BlueJ is also used in the textbook *Objects First with Java: A Practical Introduction using BlueJ* by David J. Barnes & Michael Kölling [1], a popular choice for programming fundamentals classes that use Java as the first step into the world of object oriented programming languages.

2.3 Novis - Michael Berry & Michael Kölling

Novis [2] is a notional machine implementation, aimed at beginners and integrated into BlueJ, providing real-time visualisation of the notional machine.

2.4 JIVE

JIVE [7] stands for *Java Interactive Visualization Environment* and it is an interactive execution for the Eclipse IDE [8] that provides visualizations of Java programs.

[5] https://informa.inf.usi.ch/
[6] https://www.bluej.org/
[7] https://cse.buffalo.edu/jive/
[8] https://www.eclipse.org/

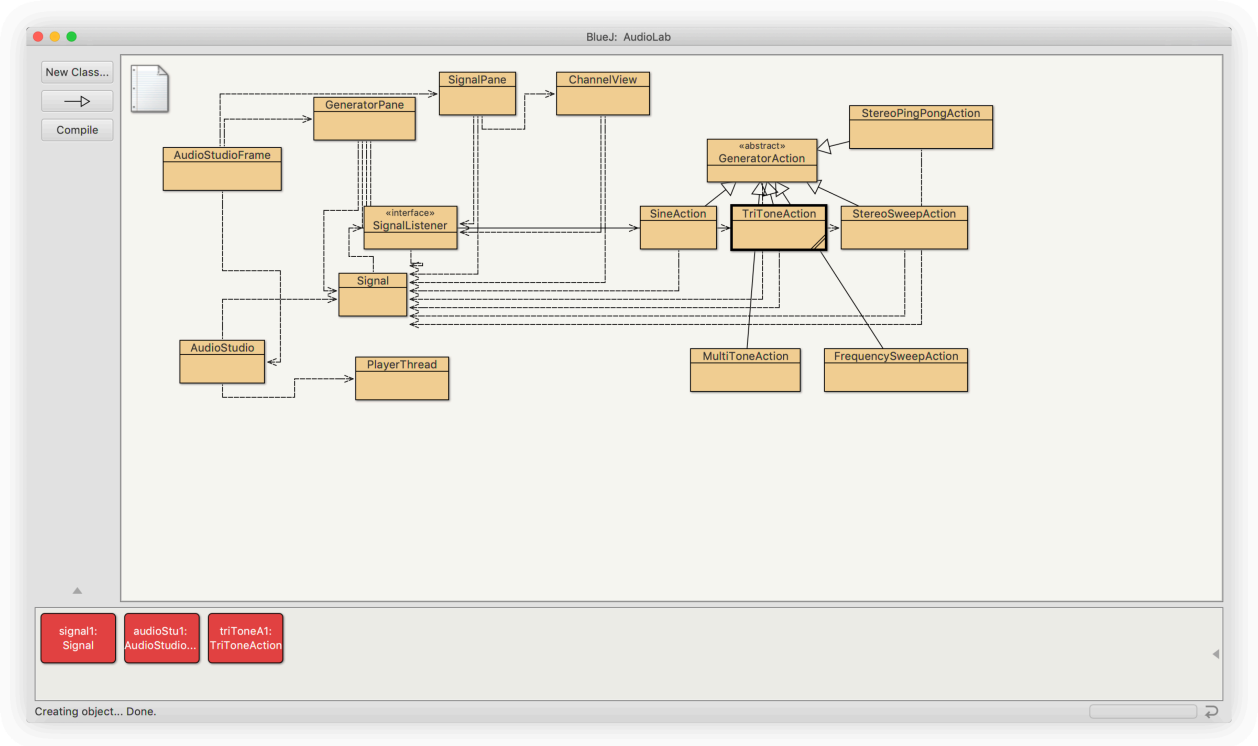

Figure 4. BlueJ and its main view showing the classes of a project and their interconnections, and the Objects Bench at the bottom, showing all the current classes instances.

2.5 ViLLE

ViLLE [9] is a multi-language program visualization tool developed by the University of Turku in Finland. Like other visualizers, its goal is to support the learning experience of beginner programmers, who might struggle to understand how to code. It has different use cases, for example it can be used to create and edit code snippets and to see in real time what happens during their execution.

2.6 Python Tutor

Python Tutor [10] is a web application for writing some code, see it visualized and executing step by step, and get live help from other users currently on the website.

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 always point to the center of the target block, avoiding arrows overlapping.
- The interface has a good layout, as all elements are properly positioned following a clear and organized structure.

[9] `https://ville.cs.utu.fi/old/`
[10] `http://www.pythontutor.com/`

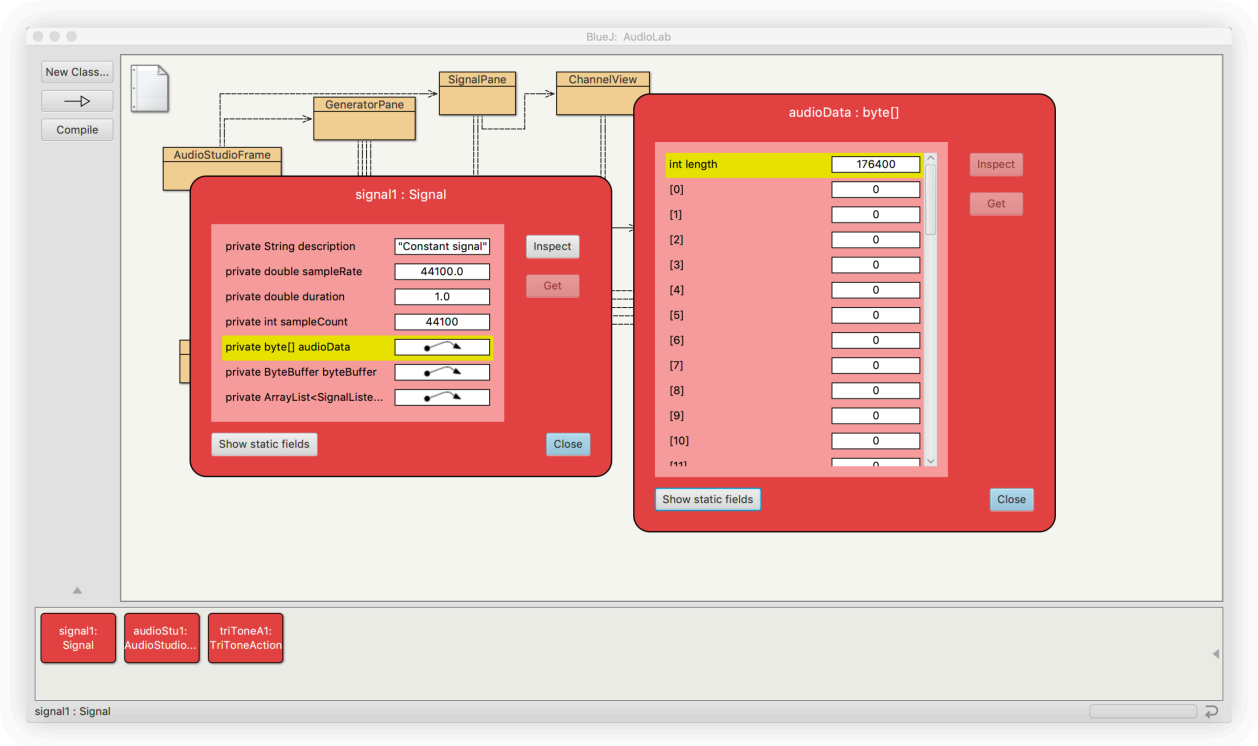

Figure 5. Detailed view of an object in BlueJ. As you can see, arrows are only displayed in the main view and for classes. In the object view, reference variable have an arrow icon, which can be double clicked to see what it is referring to.

UI - What does not work:

- Not enough overall white space [11].
- Colors can be improved, because they do not convey a lot of excitement being very pale. In this case, saturation be help create a more engaging experience.
- 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 thanks to its already praised layout, 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.
- Dragging the start of an arrow starting from a stack frame, makes it snap back to the border of a given block instead of the center of a variable.

As you can read from this analysis, the result is pretty clear: the Informa clicker does not have any problem in terms of user experience, but it lacks a proper user interface to better deliver what already works.

[11] `https://blog.bannersnack.com/white-space-in-graphic-design/`

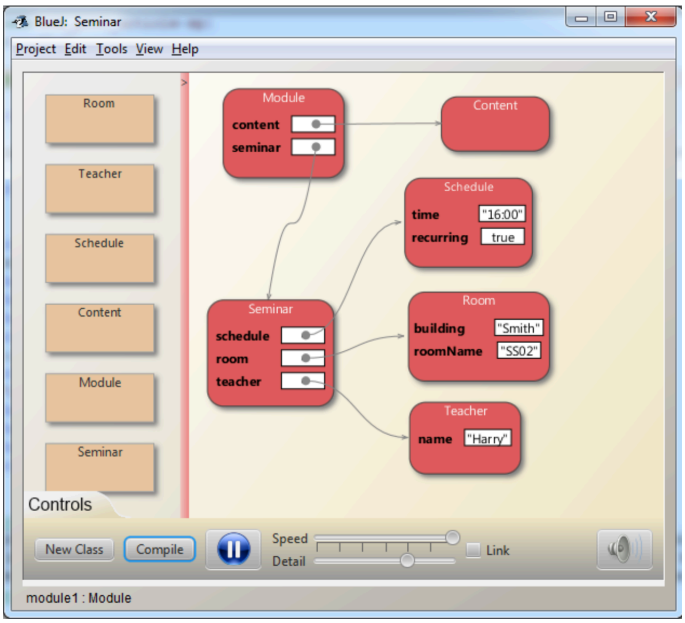

Figure 6. Novis.

3.2 Requirements

After running the UI/UX analysis and establishing that what had to be improved was the UI, prof. Hauswirth and I 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 [12].

3. Create arrows by clicking on a reference variable, holding that click and releasing it on a heap object.

4. Drag blocks around.

[12] A block is a general term for stack frames and heap objects.

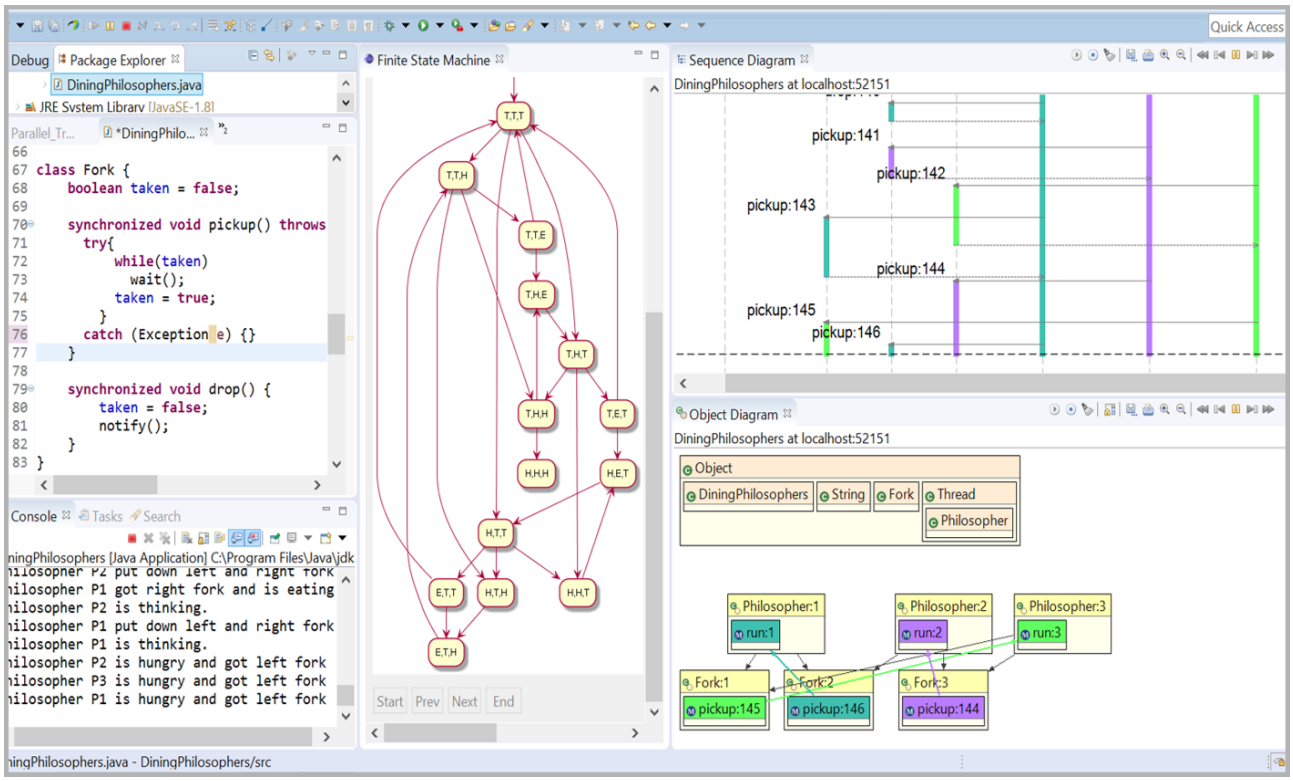

Figure 7. JIVE.

Plus something new:

1. A data structure must be updated with every state change within within the diagram (more on this is **Section 4.4**), and eventually be converted to a JSON file more offline editing and automating tests on Informa.
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 represents the current state of the diagram, can both be downloaded and uploaded into the web application. Uploading that file acts like loading a save state in a videogame, because the whole interface goes back as it was left at the time of downloading the JSON file.
2. Panning and zooming around in the interface.
3. Arrows as polygonal chains [13].

[13] https://en.wikipedia.org/wiki/Polygonal_chain

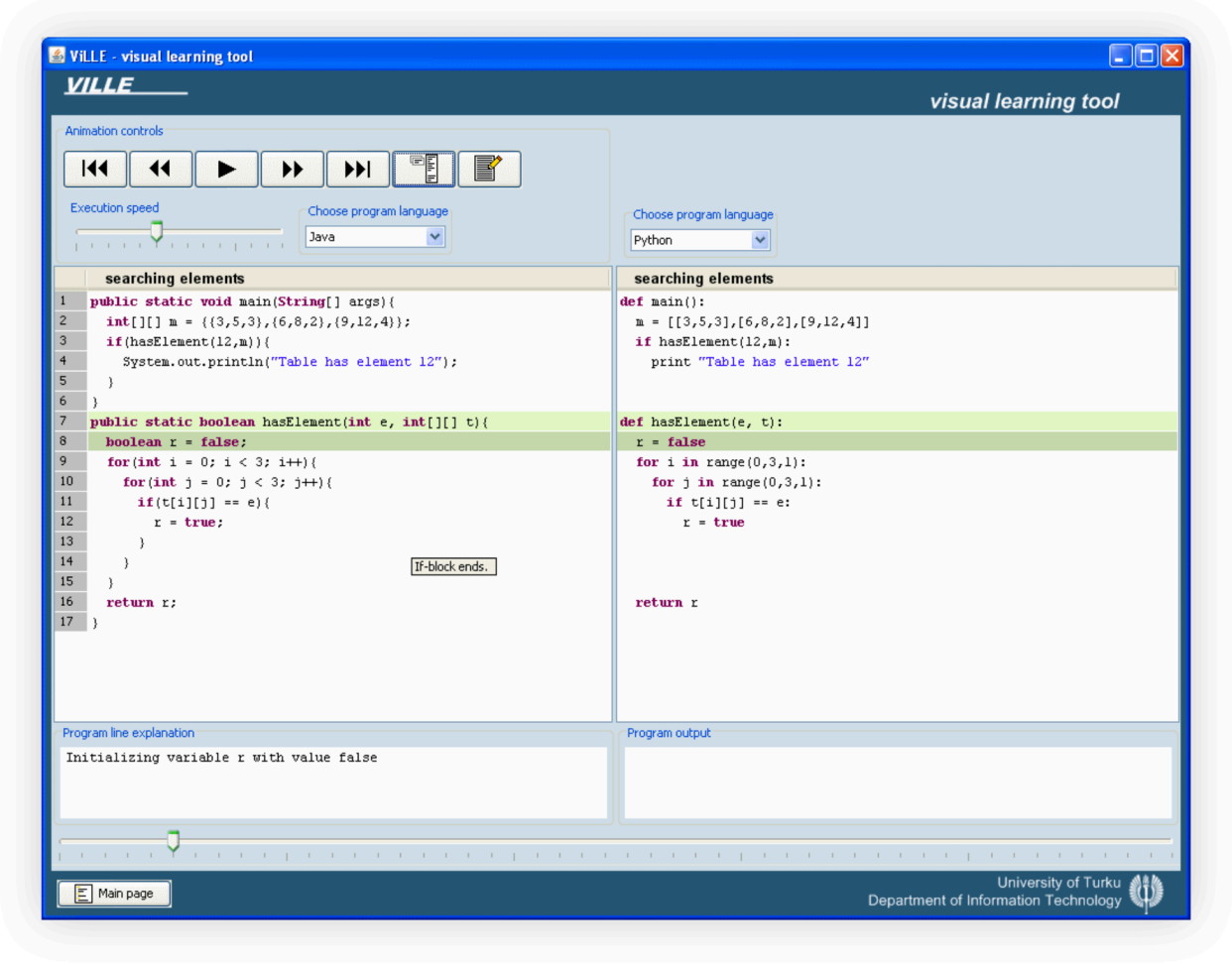

Figure 8. ViLLE.

4 Project design

In this section I am going to describe 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 with 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 [14] and ultimately creating the high-fidelity mockups [15] to show to prof. Hauswirth in order to get feedback.
The first step was to figure out how to divide the stack and heap areas, 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 7 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 a rectangular shape. The Informa clicker is very similar in this regard, since it has all variables in a similar looking element, so it

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

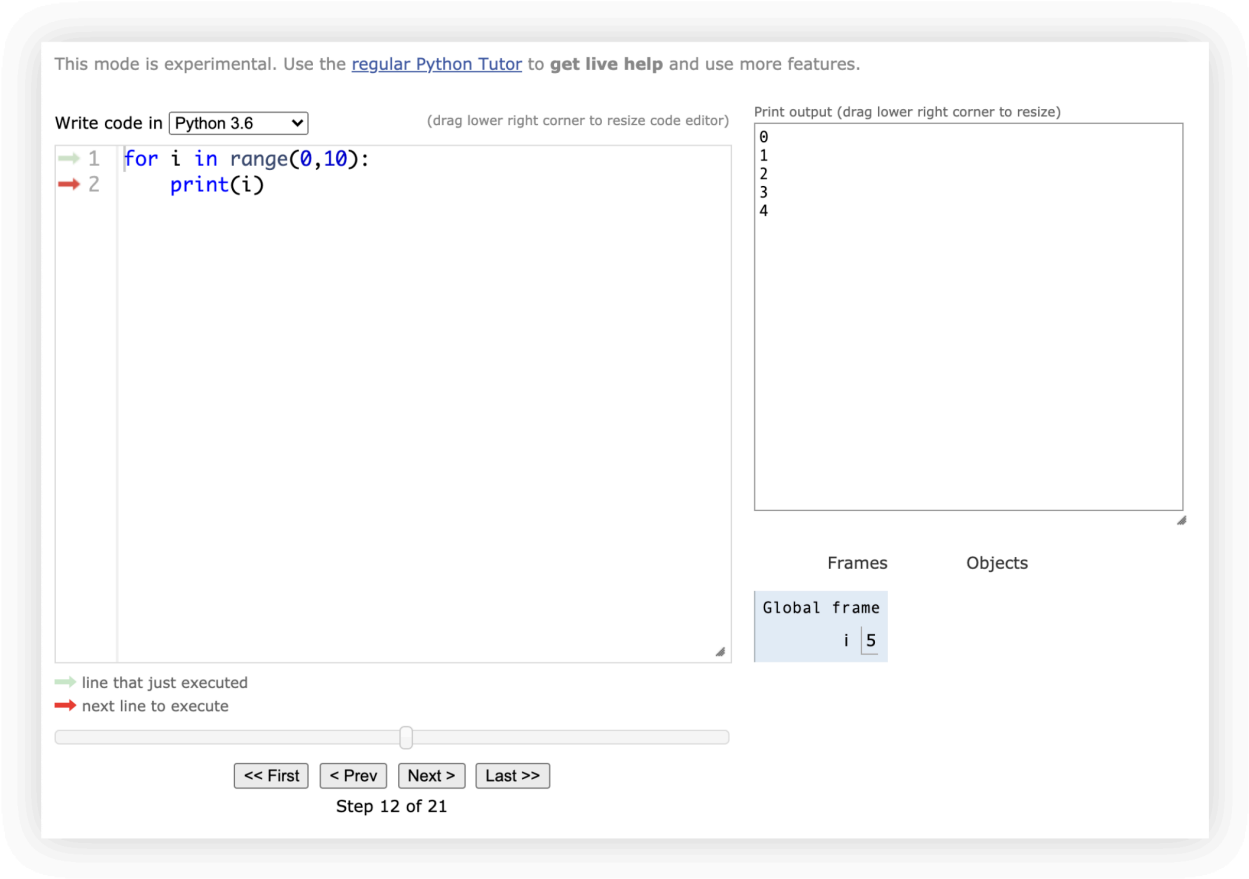

Figure 9. Python tutor.

is not a big change, but it certainly helps "glueing" the data and offers clearer separation between blocks.
As far as stack frames and heap objects, I tried to adopt a more schematic approach, and reserve one row for each detail (name, type and value) of every variable. 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.

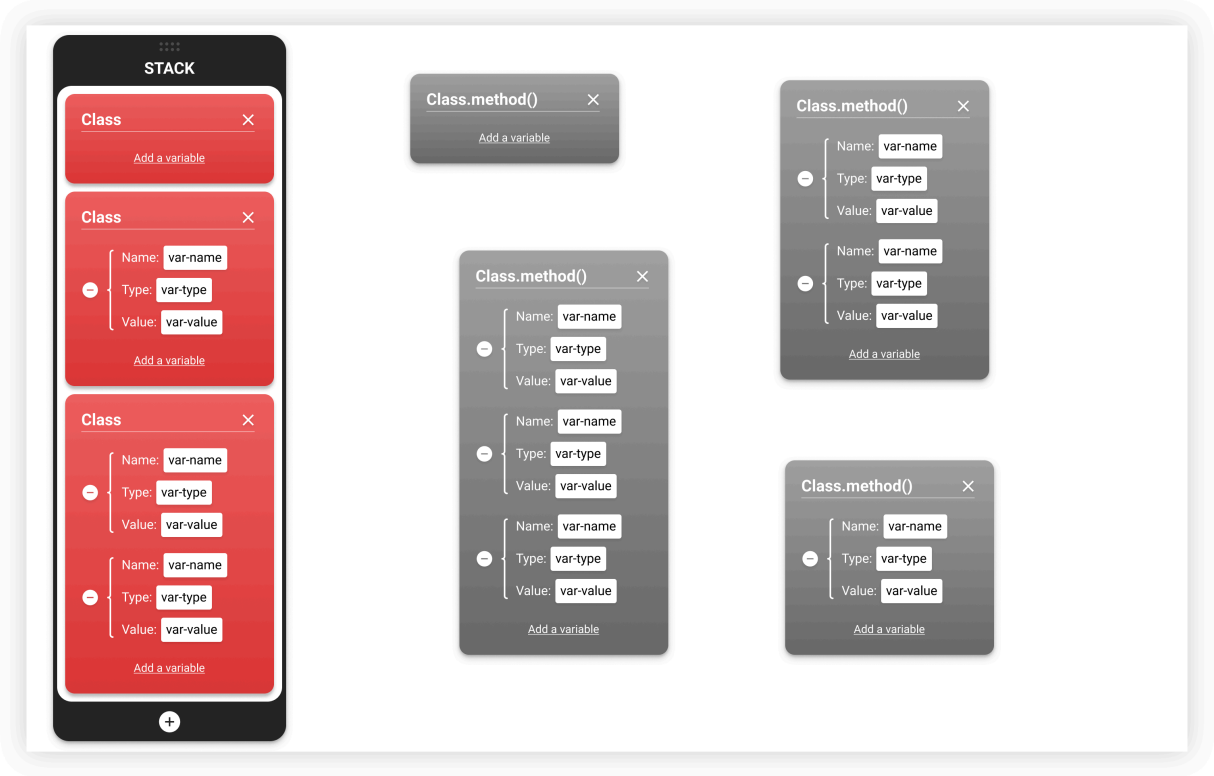

Figure 10. First version of the user interface, with a floating stack

Figure 8 shows an intermediate version of the design, which is substantially different from the previous one shown in **Figure 7**. 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.
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 9**. Compared to **Figure 8** 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.

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.

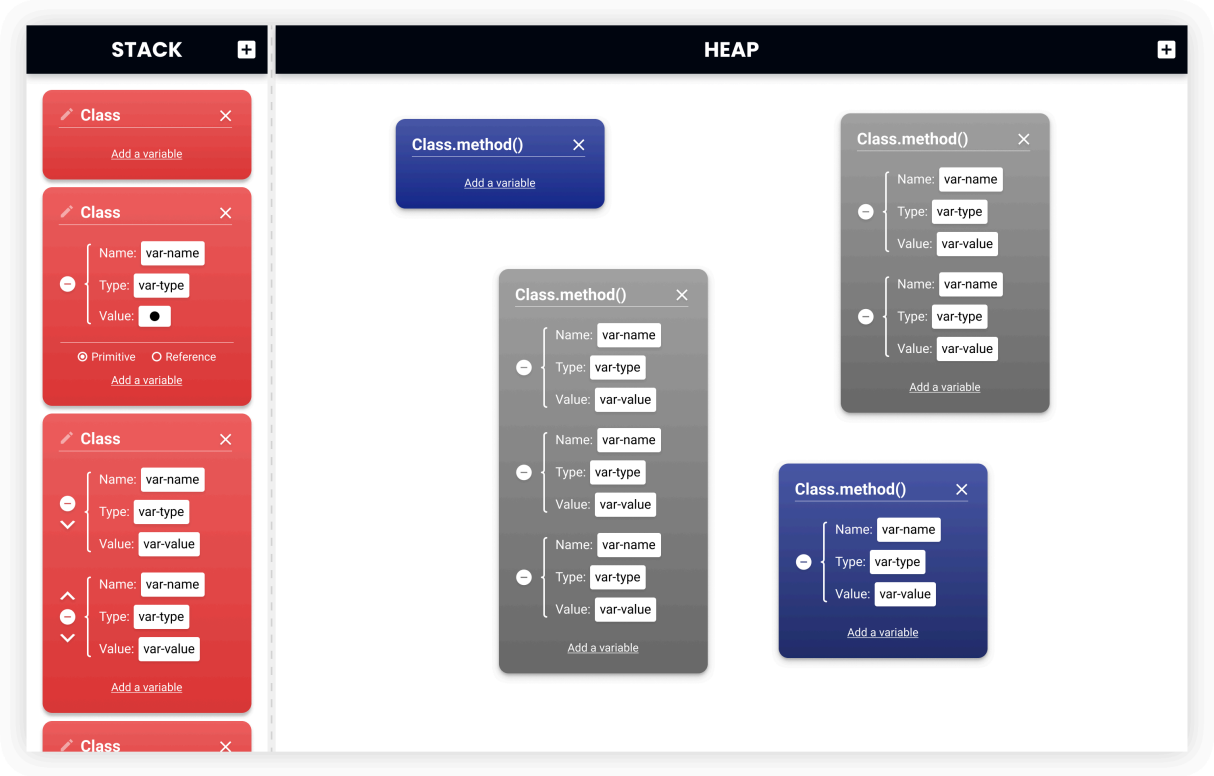

Figure 11. Intermediate version of the user interface with separate regions for stack and heap

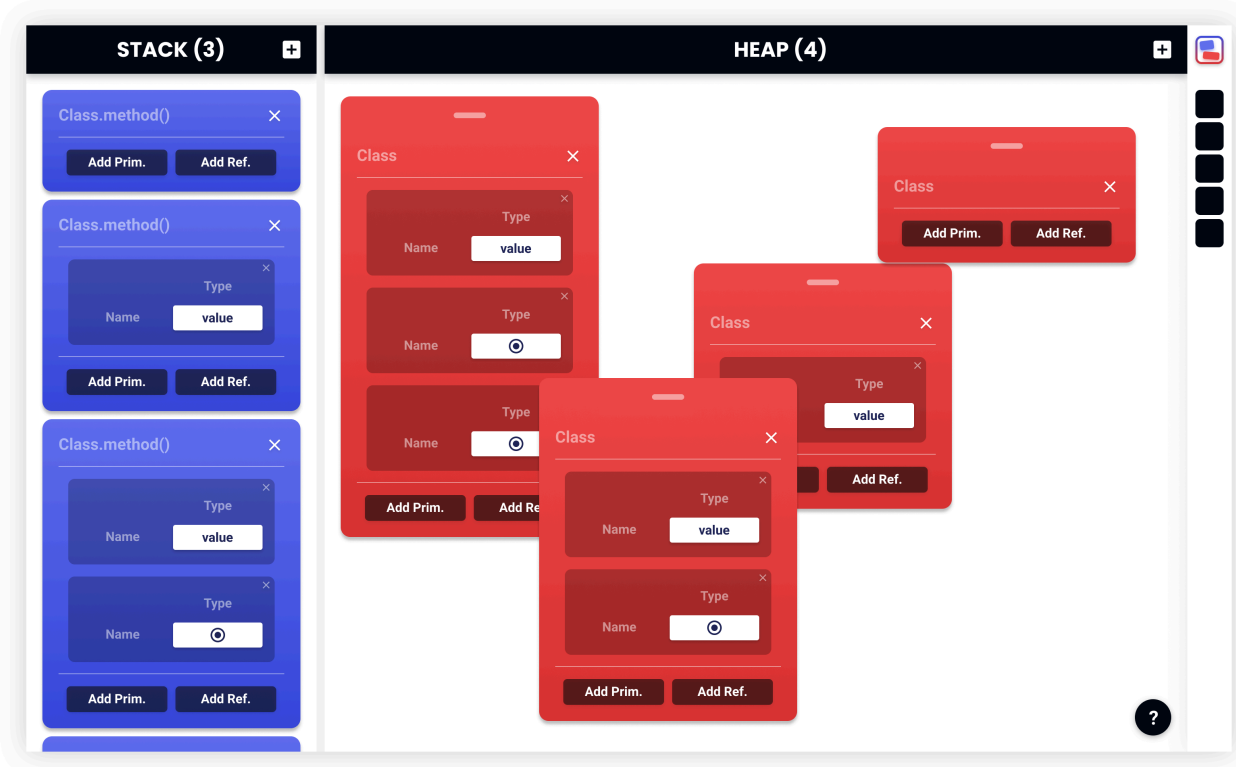

Figure 12. Final mockup of the user interface

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 10** shows the components tree on which I eventually settled.

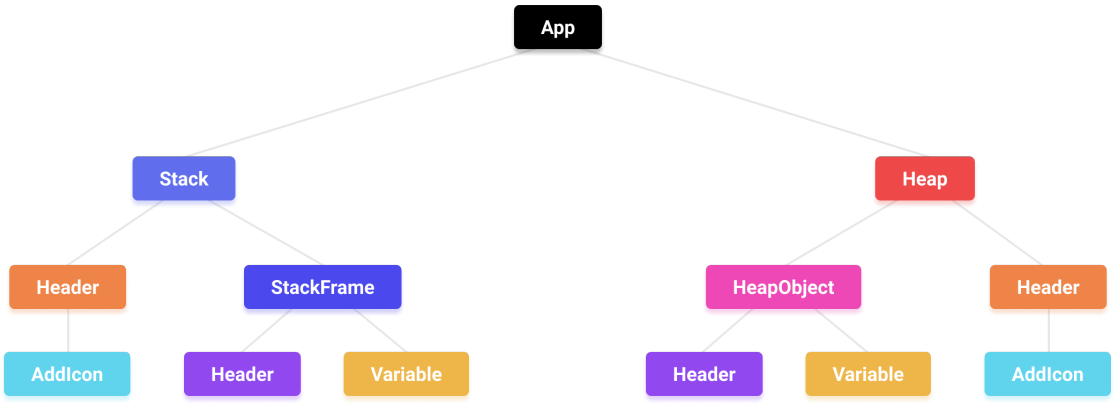

Figure 13. Components tree visualization

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 11**, 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 14. 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 handles the creation of new blocks.

4.3 UI functionalities

In this subsection I am going to walk you through every functionality that this stack and heap diagram editor offers to the user.

Stack

- Clicking on the plus button in the header, a new stack frame is created.
- Stack frames are always added at the top of the stack.
- When the stack contains more frames than the window can fit, the stack becomes scrollable.
- The header renders the number of frames currently populating the stack.

Heap

- Similarly to what happens in the stack, by clicking on the plus button, users can create new objects on the heap. However, the buttons works a bit differently: when it is yellow, the state is "on", the mouse cursor change and the user can create a new object by clicking on the heap; when it is white, the state is "off" and the objects creation is disabled.
- Objects on the heap can be dragged around by clicking on their handle at the top.
- In the header it is rendered the number of objects currently populating the heap.

Regions separator

- Users can drag the vertical separator between the stack and the heap to resize their width.

Stack frames and heap objects

- Users can type the name of the class method that generated a given stack frame.
- Users can type the name of the class that generated a given object on the heap.
- Users can create primitive and instance ("reference") variables.
- Users can fill the information of a variables, like the name, the type and the value.
- If a variable is of nature primitive, its value can be written using the keyboard.
- If a variable is of nature reference, an arrow is created by clicking on the circle icon within the value input field, holding the left mouse button and releasing it on a target object on the heap.

Toolbar

- Users can download a JSON file, representing the current state of the diagram. When pressing this button, a prompt appears to enter the name of this file.
- Users can upload a JSON file that was downloaded from the application. This file is parsed and it triggers the application to re-render with the state of the diagram at the time of downloading that file.
- Users can start a new project (all stack frames and heap objects are deleted).
- Users can scale up the interface. At every click, the base font-size (16px) is increased by 2px.
- Users can scale down the interface. At every click, the base font-size (16px) is decreased by 2px.

4.4 Storing global states

In React, data is usually shared via props from parent to children: they can receive functions as props and 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* [16], and it is generally seen as a situation to avoid if props needs to traverse more than 3 levels of components. In fact, it becomes difficult for the developer to clearly understand what is going on, ending up storing data in intermediate components that are never going to use it.

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

1. Composition
2. State management tools (e.g., *Redux* [17] and *Context* [18])

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* [19]. 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.

[16] https://newreactway.com/what-is-prop-drilling-in-react/
[17] https://react-redux.js.org/
[18] https://reactjs.org/docs/context.html
[19] https://reactjs.org/docs/hooks-intro.html

In particular I created the following contexts:

- *arrowsContext*: used to store data about arrows.
- *heapAddModeContext*: it allows communication between the plus icon in the heap header and the heap dragging area, effectively allowing the user to freely create objects on the mouse position.
- *heapDepthIndexContext*: it allows me to keep always the current dragged heap object above the others, by correctly setting their css z-index properties.
- *heapMousePositionContext*: in order to drag heap objects around, many different components needed to access the mouse position, so this context takes care of that.
- *resizableStackContext*: because of the layout of the application, resizing the stack causes many coordinates to be recomputed, involving components from different parts of the tree. This context allows all these parties to access the data they need.
- *stateContext*: this context stores all the data about the diagram and keeps a big JavaScript object updated with every state change happening in the application.

5 Implementation issues

During the development of this web application, I have faced a few implementation issues. In particular, I had problems of varying degrees with the following points.

5.1 Multiple coordinates conversions

At the beginning of the project, when I was building all the JSX and setting up all the CSS rules, I soon discovered that stack and heap needed different "treatments". The stack is built using flexbox with vertical direction, so the coordinates of each stack frame do not really matter, they just stack on top of each other. On the other hand, heap objects need precise coordinates in order to be placed on the screen. However, there is a twist: heap objects coordinates are *relative*, meaning that position $(0, 0)$ is the top left corner of the heap (header excluded), while mouse coordinates are absolute (= relative to the entire viewport).
Objects on the heap use relative coordinates because the stack is resizable: if an object on the heap had position absolute, it would stay in the same place while resizing the stack, not producing the desired effect of seeing everything shift; it would also go *behind* the stack, a very undesirable side effect.
Because of that, I often found myself converting coordinates from system to system.

5.2 Middle ground between dryness and modularity

In this web app, some components are incredibly similar in term of look but slightly differ in behavior. When using modular frameworks like React, such a situation is not as trivial as one might think. It is very easy to create one single big component that uses conditional rendering in order to avoid code repetition. In my case, that is exactly what happened when I was building the interface and I had just started to work on the drag feature of the objects on the heap.
I started having problems when I realized that stack frames and heap objects needed different data representations (i.e. spatial coordinates and z-index values because the current dragged object has to be on top of the other ones). At that point, I had to decouple stack/heap and stack frames/heap objects and find a good compromise between having components with identical elements inside and keep a relatively high level of modularity.

5.3 Arrows

Arrows were the toughest feature to implement. Partially because they are implemented using SVG [20] (*Scalable Vector Graphics*) and the syntax and some of its aspect are not very easy (like the path attribute), but also because of their highly dynamic nature in the context of the web application. In fact, in order to render them, I needed to have at least a rudimental system that would register all the information representing an arrow (start/end position, start region, variable and parent from where the connection was created, target object on the heap).
For that reason, during the first week of work on them, the only way I could have any sort of validation was by constantly printing values on the console in the browser, and by taking and analyzing screenshots. After a while I became very fast and it did not take me long to check on a photo editor if something was correct or not. However, when a bug appeared, the debugging process was always very slow and tedious, because it was difficult to pinpoint the source of the problem.

Another implementation issue I had when working with arrows, was about the way I was updating already drawn arrows. The web application keeps track of two things:

1. The new arrow that is being drawn on the screen.
2. All the arrows that have already been drawn and that need to "listen" for certain events and be updated accordingly (for example, an object that they are connected to, changes its position or gets deleted). When the new arrow is created (the mouse click is released), it gets added to the array if there is not already an arrow with the exact same data.

[20] `https://it.wikipedia.org/wiki/Scalable_Vector_Graphics`

When it came down to keep the arrows updated, I could follow two paths:

1. Listen for every event that led to the arrow coordinates being recomputed.
2. Rebuild the whole array of arrows every times the state of the diagram changes.

Technically, the first approach is more performant, because it only updates the arrows that do really change. However, in such an application it is really unlikely to have so many elements that the performance starts to suffer. Furthermore, using a single point of reference (the state of the diagram) is far easier than watching multiple ones. Hence, I opted for the second way as a solution for this issue [3].

6 Testing

6.1 Methodology

6.2 Results

6.3 Comments

7 Conclusions

7.1 Future work

References

[1] D. Barnes and M. Kolling. *Objects First with Java: A Practical Introduction Using BlueJ*. Pearson Education, 2016.

[2] M. Berry and M. KÃűlling. Novis: A notional machine implementation for teaching introductory programming. In *2016 International Conference on Learning and Teaching in Computing and Engineering (LaTICE)*, pages 54–59, 2016.

[3] B. Du Boulay. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

[4] M. Guzdial, S. Krishnamurthi, J. Sorva, and J. Vahrenhold. Notional Machines and Programming Language Semantics in Education (Dagstuhl Seminar 19281). *Dagstuhl Reports*, 9(7):1–23, 2019.

[5] J. Hidalgo-CÃ, G. MarÃn RaventÃ, and V. Lara-VillagrÃin. Understanding Notional Machines through Traditional Teaching with Conceptual Contraposition and Program Memory Tracing. *CLEI Electronic Journal*, 19:3 – 3, 08 2016.