Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

An Online Learning Environment for the Introduction to Object Orientation in Python

Bachelor Thesis

Kai Zheng

kzheng@ethz.ch

Chair of Information Technology and Education
ETH Zürich

Supervisors:
Prof. Dr. Juraj Hromkovič
Regula Lacher

March 5, 2024

Acknowledgements

Thank you Prof. Dr. Juraj Hromkovič and Regula Lacher for the delightful supervision of this thesis. You have helped me with feedback and many suggestions improving the learning environment.

Also, thank you Giovanni Serafini, Andre Macejko, my family, friends and all the volunteers testing and giving feedback on the environment.

Abstract

Object orientation is a fundamental concept in computer science education. This Bachelor thesis presents the development of an interactive online learning environment, designed to accompany gymnasium level students and beyond looking to understand and practice object orientation in Python.

By leveraging available web technologies, exciting design and pedagogical ideas are discussed and how they are implemented.

Contents

CHAPTER 1

Introduction

1.1 Background

The ABZ [1] at ETH Zurich is dedicated to the development of high-quality educational resources in computer science, designed to cater to a broad spectrum of learners within the Swiss educational system. Collaborating closely with the Chair of Information Technology and Education, the ABZ has published a range of books covering diverse topics in the field. These publications target students from early primary school through to gymnasium level. Complementing these texts, the ABZ also provides interactive learning environments, reinforcing the concepts introduced in the books.

1.2 Motivation

Object orientation is a central topic in computer science. As students take their first steps in programming, it is the moment they are introduced to how variables actually store their values in the computer. This is known to be a rather difficult topic to begin with, hence highlighting the need for qualitative educational resources.

Online learning environments have several advantages. They make learning more interactive and engaging by turning theory into practice. Students can see concepts in action and experiment with them in a playful way. In the context of object orientation, an interactive platform is especially useful. It allows students to write and edit code, see how their programs run, and understand how the computer handles memory.

1.3 Goals

The main objective of this thesis is to plan, design and implement an interactive learning environment to teach students the basics of object orientation in Python; specifically, values and reference types in Python and how they are implemented in the computer. It should enable students to form initial intuitions about what objects are and how they are implemented in the computer. In particular, the aim is to convey the concepts pedagogically in an understandable and appealing way.

The result of this thesis is a well documented, reliable platform that can be used as an accompaniment for lessons about object orientation in German speaking countries.

1.4 Related Work

There exist multiple other learning environments that are designed in cooperation with ABZ. They cover various topics in computer science.

CHAPTER 2

Object Orientation in Python

Python, a high-level programming language, is known for its simplicity and readability, making it a good choice for beginners in programming. One key concept is that everything in Python is an object, categorized as either mutable or immutable. The following explanation highlights the fundamentals, while abstracting away any optimizations and other advanced features of Python.

2.1 Immutable Objects

Immutable objects cannot be altered once created. Operations that appear to modify an immutable object actually result in the creation of a new object, with the reference being updated to this new object.

In Python, immutable objects include strings, tuples, and numbers.

Listing 2.1: Example immutable objects

```
x = 1  # x -> 1
y = -1 # x -> 1; y -> -1
y = x  # x -> 1; y -> 1  # x and y reference objects with the same value
```

2.2 Mutable Objects

Mutable objects can be changed after their creation. Modifications to these objects are made directly, altering the content without affecting the identity of the object. This implies that if multiple references point to the same mutable object, a change in one will be reflected in the others.

In Python, mutable objects include lists, dictionaries, and sets,

Listing 2.2: Example mutable objects

```
x = [1, 2] # x -> [1, 2]
y = [-1]   # x -> [1, 2]; y -> [-1]
y = x      # x, y -> [1, 2]  # x and y reference the same object
```

2.3 Value Types and Reference Types

Following that Python's approach to data types is rather unique, the terms *value types* and *reference types* are used instead, drawing on familiar terminology from other programming languages for educational clarity and effectiveness.

This is because everything in Python is an object and hence per definition *reference types*, but what makes some objects behave like *value types* is their immutable nature. Concretely speaking, any change to immutable objects result in the creation of a new object. This behaviour mimics value types, where variables directly hold the data and any change results in a different instance of the data.

CHAPTER 3

The Learning Environment

3.1 Learning Objective

The primary learning objective of the environment is to deepen students understanding of *value types* and *reference types* within the context of object orientation. As this requires students to understand how the computer handles memory, they will need to engage with diverse programs and accompanying memory graphs.

3.2 Idea

The interaction with the learning environment is intended to feel like a game which is done by implementing a typical game structure. This means, presenting students with a variety of levels to solve, alongside opportunities for practice and skill refinement.

The platform's structure is divided into three *stages* (Figure 3.1), each containing a flexible number of *levels*. Beginning with *stage value types*, then, progressing to *stage reference types*, and finishing up with *stage value types reference types*. This structured progression fosters a gradual and effective learning journey.

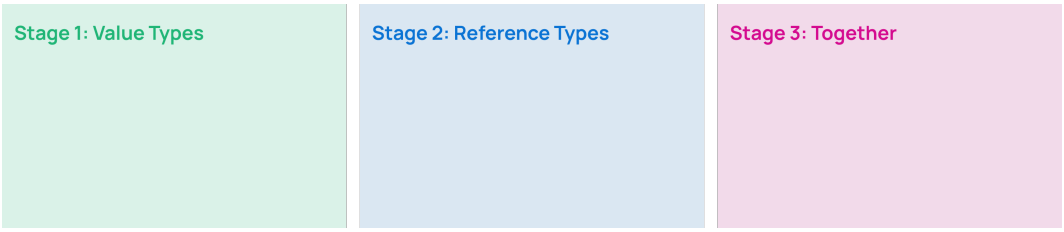

Figure 3.1: Stages, low fidelity prototype

Furthermore, each *level* falls into one of three *categories*: *code-the-memory*, *memory-to-code*, and *coding-challenge* (Figure 3.2). In *code-the-memory*, students are tasked with creating a program that accurately reflects a given memory. Conversely, *memory-to-code* requires them to fill the memory graph from a given program. The third *category*, *coding-challenge*, is a special one; as students work on a programming task, the corresponding memory graph is dynamically generated and displayed in real-time alongside their program. I like to call it "learning by doing". These *categories* allow students to engage comprehensively with the material as well as ensuring exposure from multiple angles.

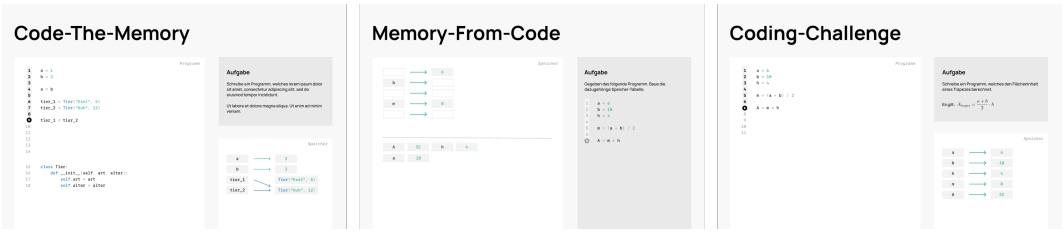

Figure 3.2: Categories, low fidelity prototype

3.3 Demonstration

Upon start, the first screen visible to the student is a loading animation (Figure 3.3). This first screen will disappear after completing loading and initialization of the required data, afterwards transitioning to either the *tutorial page*, if it is the student's first time visiting the environment, or the *stage page*, containing all the *levels*. Inside the *stage page*, students can then navigate to the specific *levels*. This section briefly summarizes the features and ideas of the *tutorial page*, *stage page* and *level page*.

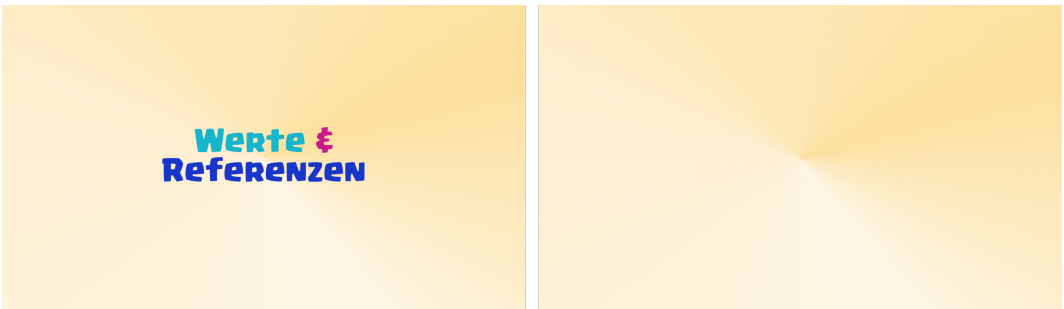

Figure 3.3: Starting page

3.3.1 Tutorial and Stage Page

The *tutorial page* (Figure 3.4) consists of two parts. An introduction and summary of value and reference types in Python and a first challenge level task helping to get to know the user interface.

The *stage page* (Figure 3.5) contains all the *levels* and works as the hub of the environment. It contains buttons to the respective *levels*, which float across the screen and are colored based on the *levels stage* and progress. The castles at the bottom correspond to the respective *stage* and are unlocked during the student's journey.

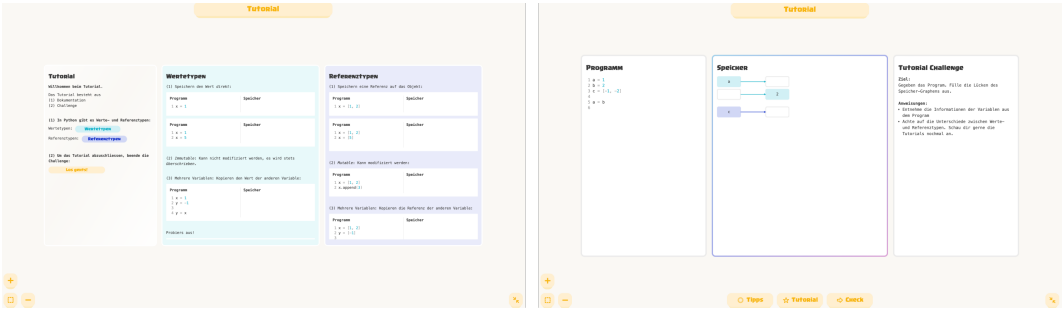

Figure 3.4: Tutorial page

Figure 3.5: Stage page: 0%; 0% opening level; 33%; 100% progress

3.3.2 Level Page

Once starting the *level*, a short gradient loading page is shown, before transitioning to the *level page*. *Levels* of all three *stages* have the same interface, differentiating only in color and castle. There are two main layers, the overlay, where information and controls are displayed and the actual contents of the *level*, which are placed inside an interactive *level canvas* containing custom, dynamic nodes (Figure 3.6).

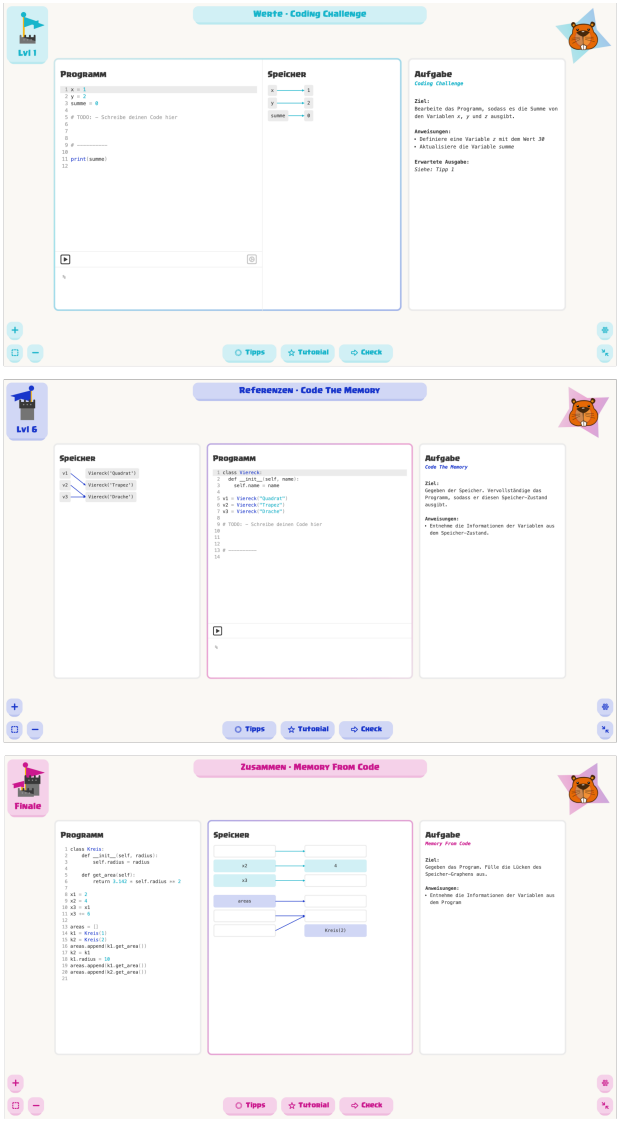

Figure 3.6: Level page: stage value types, coding-challenge; stage reference types, code-the-memory; stage reference types, memory-from-code

This *level canvas* marks the design concept of the platform. Inspired by windows in operating systems and canvas applications such as Figma [2], it gives many dimensions of freedom, allowing students to have full control over their workspace. Inside the *level canvas* are *level nodes* which contain different functionalities such as the *coding IDE*, the environment containing the code editor and memory graph, or the task description. As it is designed to work dynamically, students can insert further *level nodes*, for example one containing the tutorial.

At the top are information about the *stage*, *category* and *level*. At the bottom are controls, separated into three groups, ranging from *level canvas* controls to *level* controls to navigation.

Upon completion, a popup appears and the colors become golden (Figure 3.7).

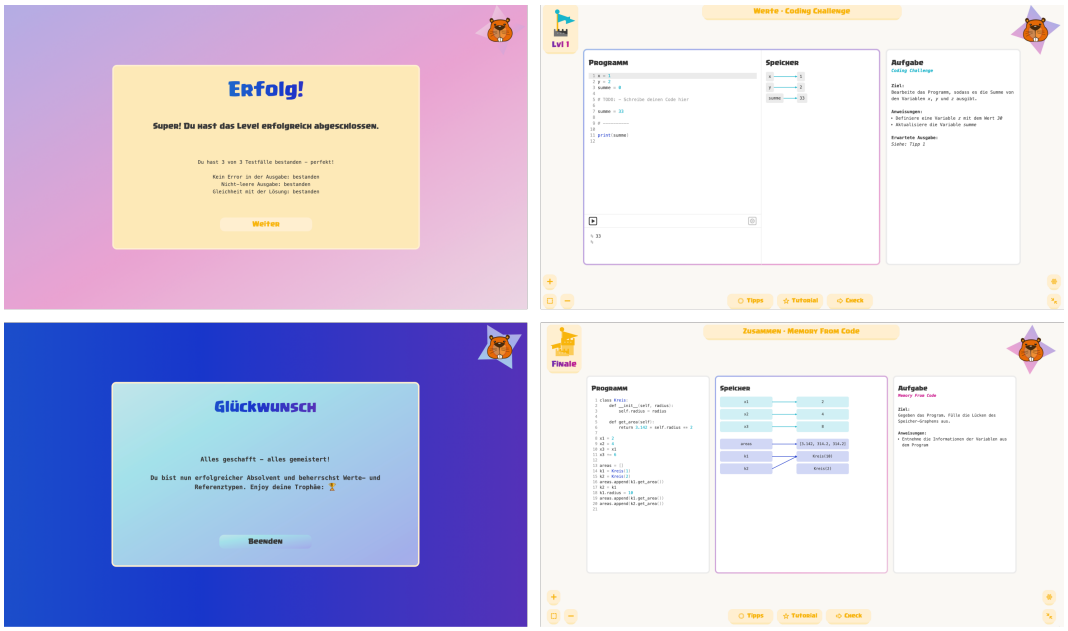

Figure 3.7: Level page: level completion success; stage completion success

3.4 Pedagogical Thoughts

3.4.1 Memory Graph

There are multiple options when thinking about how to display the memory graph from a given program. Following that everything in Python is an object, I chose to show *value*

types as overriding (Figure 3.8). Although this does not reflect Python's memory management, it provides clear visualization and reflects the behaviour of Python's memory, which is what students need to understand.

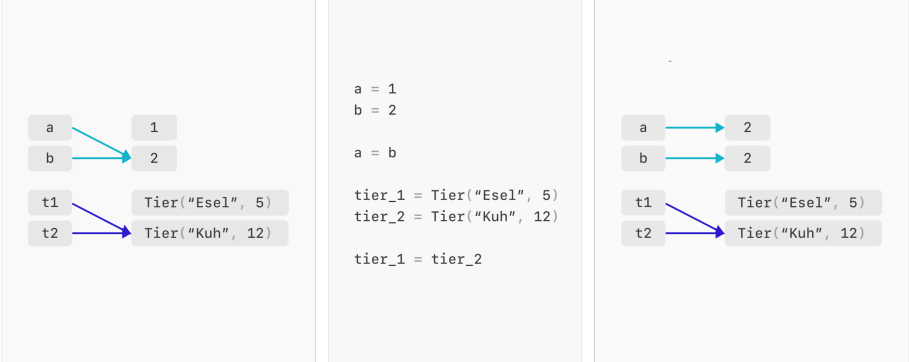

Figure 3.8: Python; program; simplified

3.4.2 Tips

For students requiring assistance while solving a *level*, the environment offers strategically designed tips. These tips are less about direct instruction and more about providing tools or additional functionalities that guide understanding. This approach follows the teaching philosophy: "Tell me and I will forget, show me and I may remember; involve me and I will understand." [3].

By leveraging the *level canvas* design, tips are added as *level nodes* to the *level canvas*. Depending on the *category*, there are different *tip level nodes*. For example, *category code-the-memory*, comes with a tip that gives another memory, one synchronized to the program the student is writing. This enables students to see their current progress towards programming the expected memory graph (Figure 3.9).

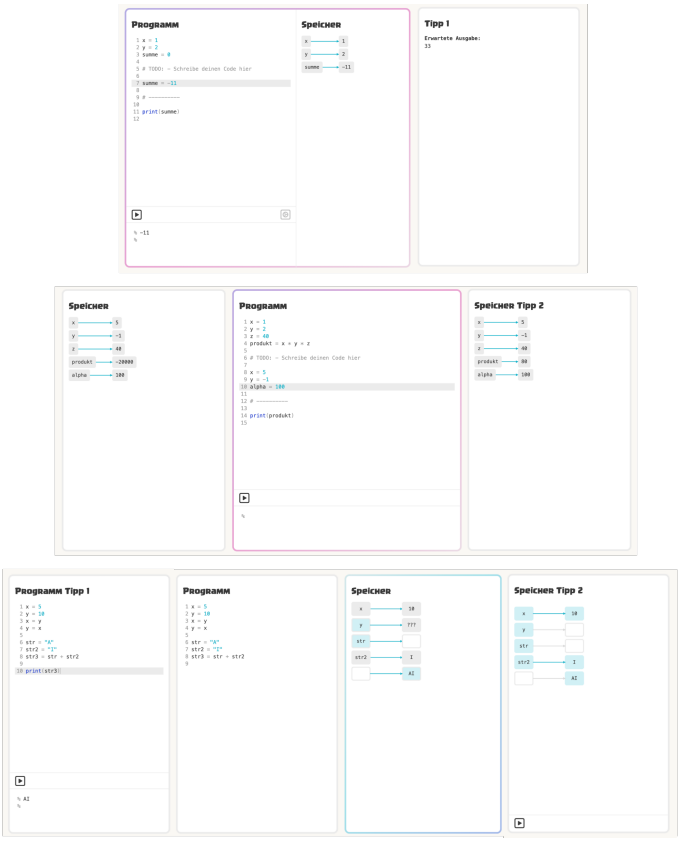

Figure 3.9: Tips: coding-challenge; code-the-memory; memory-from-code

3.4.3 Pace

Apart from *levels*, the environment also provides practice opportunities with a blank *code IDE*. For this reason, *levels* are not meant to be solved in one go and a sudden increase in difficulty is possible, allowing students to follow their own pace (Figure 3.10).

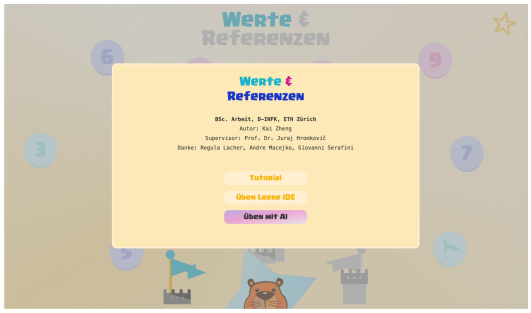

Figure 3.10: Stage page: credits and practice popup

3.5 Design

The environment is designed to facilitate an intuitive, active and enjoyable learning experience. Providing as much freedom as possible and combining joyful with focus and prioritization. This section highlights key design considerations.

3.5.1 User Experience

Inspired by the renowned language learning application Duolingo [4], to me achieving a perfect balance between entertainment, joy and focus on learning, the environment aims to provide a similar feeling. Students should perceive it as not just a learning platform, but as a dynamic and engaging space, which is alive and gives freedoms to explore (Figure 3.11, 3.12).

Figure 3.11: Infinite animations

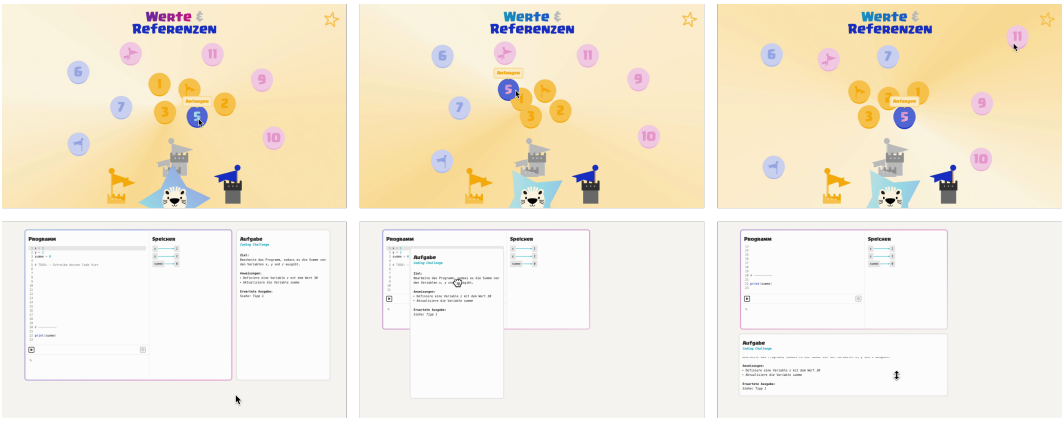

Figure 3.12: Freedom dimensions

3.5.2 User Interface

The user interface enhances the user experience by aiming to achieve an ideal balance between functionality and simplicity. It should always be clear what the focus is, elements of the same type have the same design structure, elements of same functionality are grouped together and important components are highlighted (Figure 3.13). Colors add to a clear visual distinction between different states (Figure 3.14).

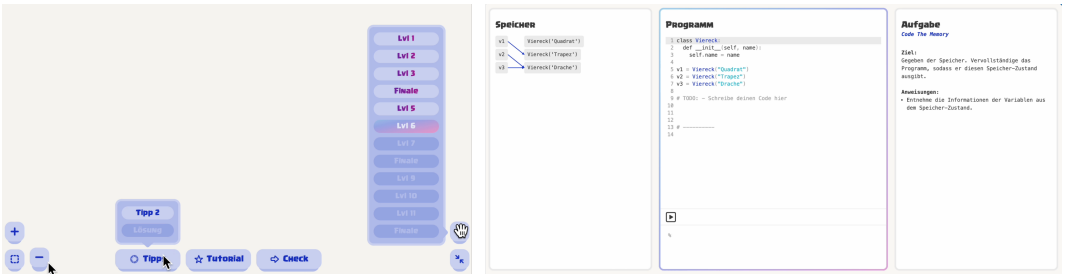

Figure 3.13: Level canvas buttons, level related buttons, navigation buttons; gradient highlight for the active code IDE

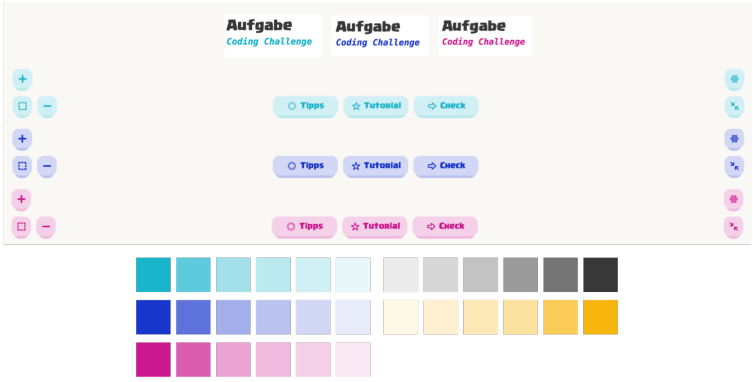

Figure 3.14: Colors

CHAPTER 4

Implementation

4.1 Overview

The implementation of the website includes multiple technologies. Foremost, it is split between a front end and back ends. The front end is built using React [5] with TypeScript [6], TailwindCSS [7] and Zustand [8]. User progress data are persisted in the users local storage. On the back end, the technology stack is a Python, Flask [9] and Docker [10] combination (Figure 4.1).

I chose React, TypeScript and Flask for their widespread popularity and extensive documentation, making them reliable and robust choices. Additionally, Flask is a Python back end, which allows for seamless compilation of Python code. TailwindCSS is a modern CSS library enabling CSS directly inside HTML elements and Zustand enables easy and straightforward global state management, a great alternative to Redux [11] or React Context-API [12]. Lastly, Docker adds a layer of security by sand-boxing code execution.

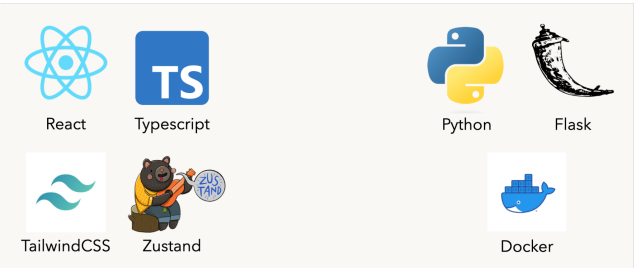

Figure 4.1: Technology stack: front end, back end

In addition to the primary technologies, various libraries were integrated into the project, chosen based on robust documentation and consistent maintenance, ensuring the en-

vironment's long-term reliability. These libraries include ReactFlow [13] for the implementation of the *level canvas*, CodeMirror [14] for the code editor inside *code IDEs*, JointJS [15] to display the memory graph inside *code IDEs*, and D3 [16] for the interactive floating *level buttons*.

The development process was supported by tools like Visual Studio Code [17] for writing code, GitKraken [18] for version control, and GitHub [19] for source code management. The website's visual aspects, such as icons and graphics, were created using a range of design tools. Procreate [20] was used for initial sketches, Figma [2] for prototyping and interface design, and Adobe Photoshop [21] for final edits. Finally, AI technologies including OpenAI GPT-4 [22], Google Gemini [23], GitHub Copilot [24], OpenAI DALL·E 3 [25], Midjourney [26] and Adobe Firefly [27] were utilized during the development as well as designing process.

4.1.1 Architecture

The architecture of the front end is structured into three layers, each capturing a specific page in the user interface. On the server side, three distinct back ends support the platform's operations (Figure 4.2).

- **Root.tsx:** This component is responsible for routing of different pages. It initializes each page with the necessary data fetched from the back end.
- **Stage.tsx:** This component corresponds to the *stage page* discussed earlier. It serves as the interactive hub of the environment, containing all *level buttons*.
- **Level.tsx:** This component corresponds to the *level page* discussed earlier. It implements the *level canvas* containing *level nodes* with the respective functionalities such as the *code IDE* or text descriptions.

- **data.py:** This module houses all environment specific data, initializing the learning environment, individual *levels* and evaluating student's solutions.
- **code_ide.py:** This module is responsible for the *code IDE*. It compiles student's code and generates the respective memory graph.
- **user:** This module manages user data, synchronized with already existing clouds by ABZ.

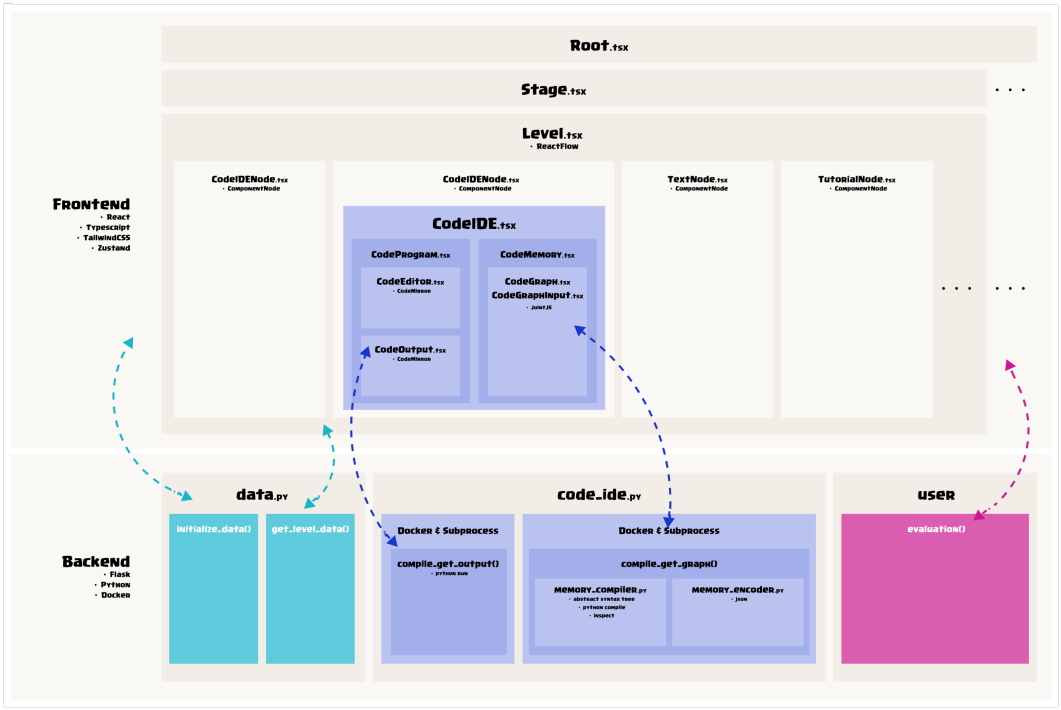

Figure 4.2: Overview architecture

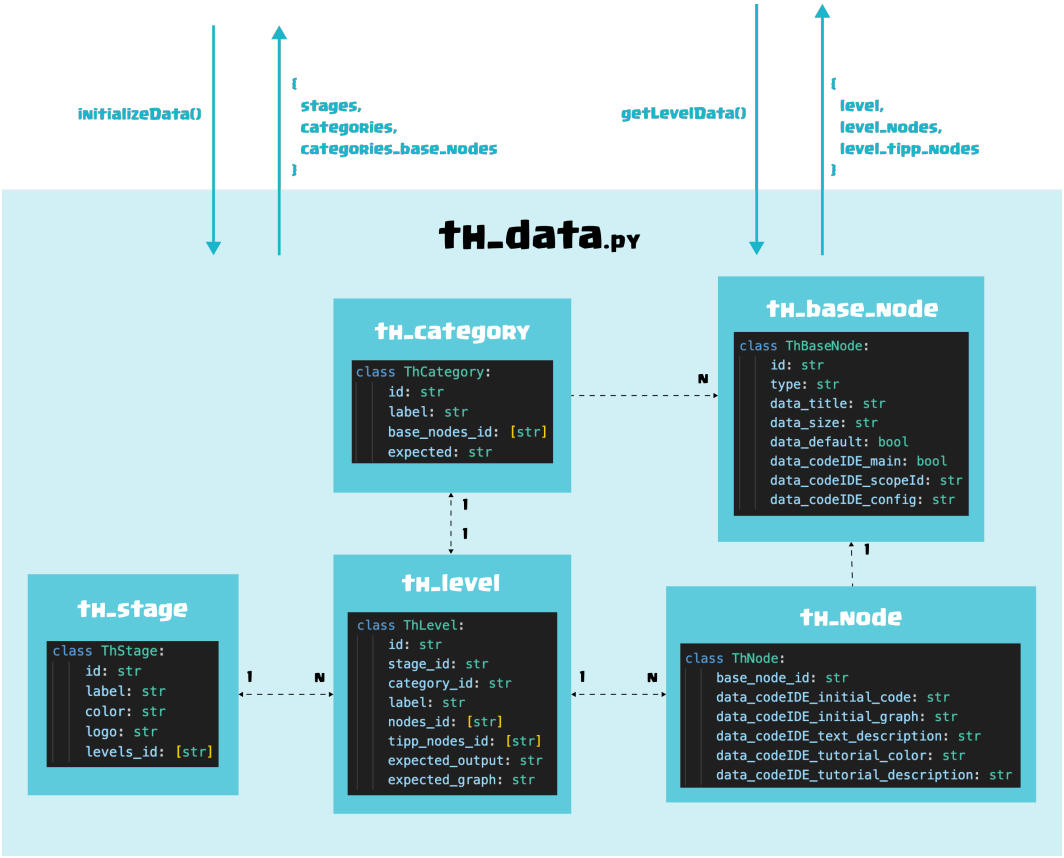

Figure 4.3: Overview level page

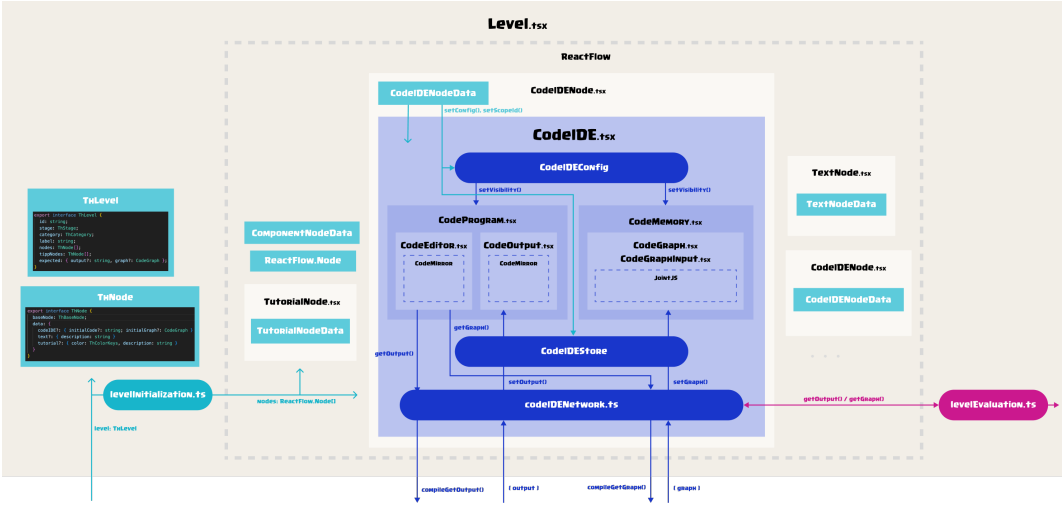

Figure 4.4: Overview th_data.py

4.2 Memory Graph Generation

Let's delve into the *code_ide.py* back end to understand how it generates a memory graph from Python code (Figure 4.5). Equivalent to the code compilation process, when a front end request is received, *code_ide.py* follows the principle of running a sub-process within a Docker sandbox. This approach is primarily for security reasons, ensuring that code execution is contained and controlled. Future enhancements such as limiting the number of requests can be considered to further bolster security.

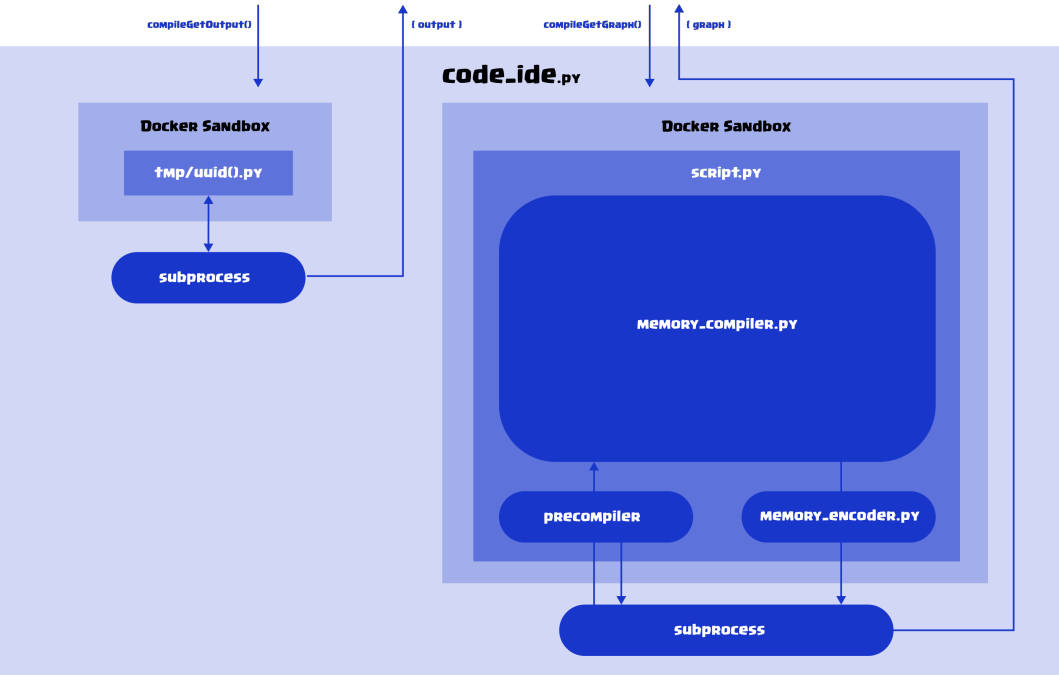

Figure 4.5: Overview codeide.py

In this process, the precompiler reduces complexity. It exits before *memory_compiler.py* executes if it encounters any errors. This step also involves removing print statements and normalizing the code. Subsequently, *memory_encoder.py* translates the output of *memory_compiler.py* into a JSON string (Figure 4.6).

Figure 4.6: Pipeline memory graph generation

Listing 4.1: script.py

```
if __name__ == "__main__":
    code = sys.argv[1]

    # Precompiling
    try:
        validate_code(code)
        code = remove_print_statements(code)
    except ValueError as error:
        print(error, file=sys.stderr)
        sys.exit(1)

    # Compiling (Graph generating algorithm)
    compiler = MemoryCompiler()
    compiler.generate_memory_graph_for(code)
    memory_graph = compiler.get_memory_graph()
```

```
    # Encoding
    encoder = MemoryEncoderManager()
    memory_graph_json = encoder.encode_graph(memory_graph)

    # Output
    print(memory_graph_json)
```

4.2.1 Compiler

Looking closer into *memory_compiler.py*, it begins with the normalized and error free Python code and passes it through different stages of inspection. Key is the Inspect [**?**] library with which it is possible to distinguish immutable and mutable objects as well as filtering out non objects such as functions and classes. The abstract syntax tree is used in order to track for deallocated objects and enables tracking the history for every line. This algorithm could be further optimized but suffices for the small programs (Figure 4.7).

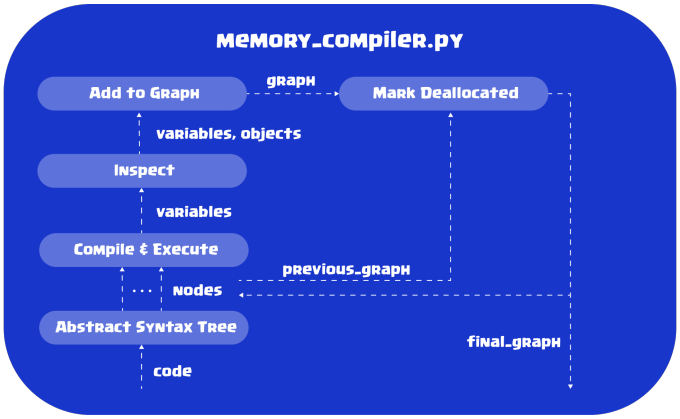

Figure 4.7: Overview memory_compiler.py

Listing 4.2: MemoryCompiler

```
class MemoryCompiler:
    ...

    def _analyze(self, code, code_variables, code_objects):
        tree = ast.parse(code)
```

```
        # (1)
        for node in ast.iter_child_nodes(tree):
            graph = MemoryGraph()
            previous_graph = self.__graphs[-1] if self.__graphs else
                MemoryGraph()

            # (2)
            self._execute(node, code_variables)

            # (3)
            for var_name, var_value in code_variables.items():
                if not (inspect.isclass(var_value) or
                    inspect.isfunction(var_value) or
                    inspect.ismodule(var_value)):
                    # (4)
                    graph.add_variable(var_name, var_value, code_objects)

            # (5)
            if previous_graph:
                graph.mark_deallocated(previous_graph)

            self.__graphs.append(graph)

    def generate_memory_graph_for(self, code):
        ...

        self._analyze(code, code_variables, code_objects)

        ...
```

4.3 CodeIDE

The CodeIDE is the React component which implements the *levels nodes code IDE*, interacting with the *code_ide.py* back end (Figure 4.8). It revolves around two principal logics: the *CodeIDEConfig*, which sets the environment's configurations and the Zustand's *CodeIDEStore*, managing state and data, essentially storing the current and initial code and graphs while updating the component's state based on back end requests. Multiple individual *CodeIDE* components are managed by unique *CodeIDEStores*, which are

identified by their *scopeId*. This is a crucial and exciting feature which allows the Zustand *CodeIDEStore*, by default globally accessible, to be limited to only one or any number of components.

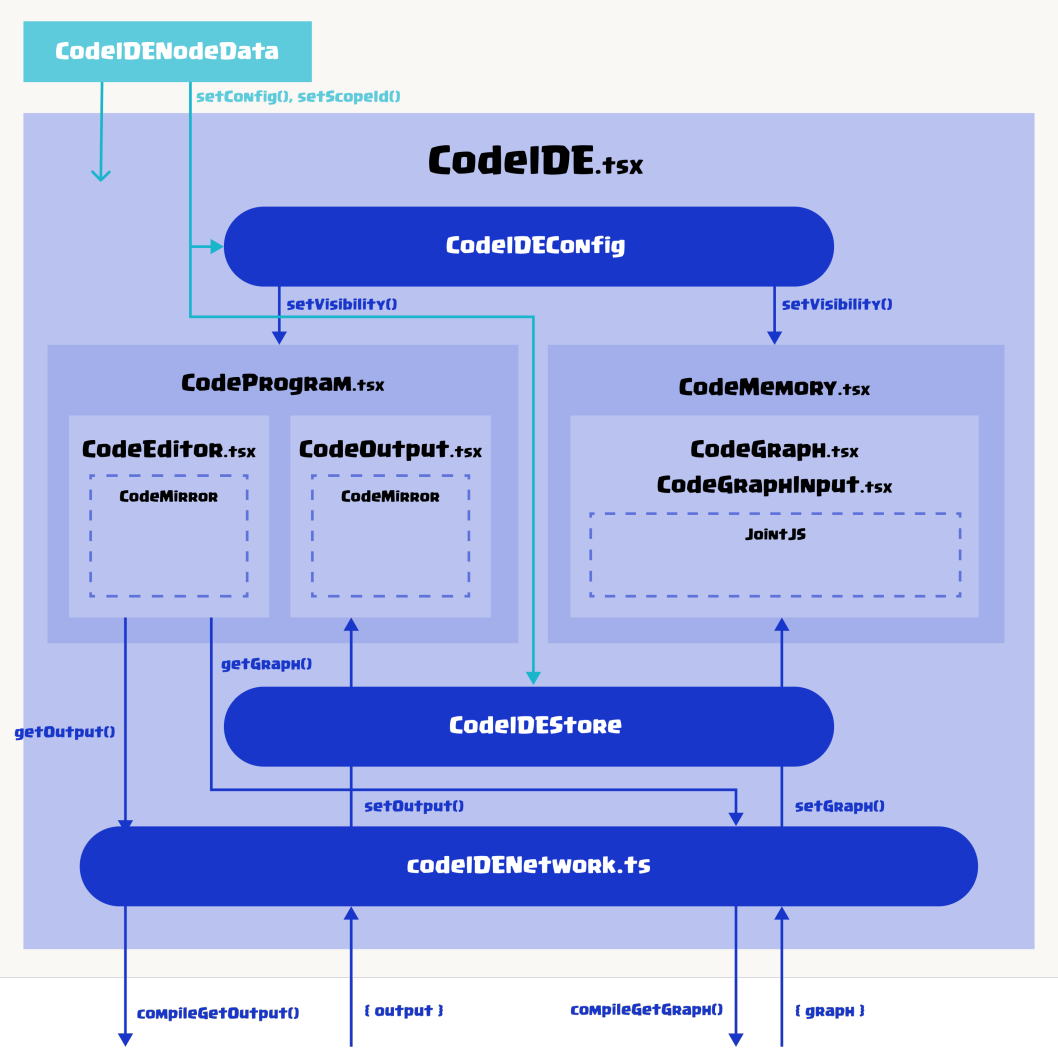

Figure 4.8: Overview CodeIDE.tsx

Listing 4.3: CodeIDEConfig

```
export default interface CodeIDEConfig {
  type: "program" | "graph" | "program+graph";
  mode: "read" | "write";
  runnable: boolean;
```

```
}
```

Listing 4.4: CodeIDEStore

```
export type CodeIDEStore = {
  code: string;
  codeOutput: string;
  initialCode: string;

  graph: CodeGraph;
  graphOutput: { nodeIds: Set<string>, edgeIds: Set<string> }
  initialGraph: CodeGraph;

  ...
};

export const useCodeIDEStore = (scopeId: string) => {
  if (!storeMap.has(scopeId)) {
    storeMap.set(scopeId, createCodeIDEStore(scopeId));
  }
  return storeMap.get(scopeId);
};
```

Listing 4.5: CodeIDE component

```
const CodeIDE: React.FC<CodeIDEProps> = ({
  height,
  scopeId,
  config,
  initialCode,
  initialGraph,
}) => {
  let codeIDEComponent;

  switch (config.type) {
    case "program+graph":
      codeIDEComponent = ...; break;
    case "program":
      codeIDEComponent = <CodeProgram height={height} scopeId={scopeId}
          config={config} />; break;
    case "graph":
```

```
      codeIDEComponent = <CodeMemory height={height} scopeId={scopeId}
          config={config} />; break;
    default: ...
  }

  ...

  return (<div>{codeIDEComponent}</div>);
}
```

CHAPTER 5

Conclusion

5.1 Reflection

Creating the environment was an exciting journey. Going full circle conceptualizing, designing, implementing and testing told me a lot. I started out with close to no previous experience in web development, coming from an iOS engineering background. This forced me to start developing while still taking online courses and understanding core concepts, presenting a steep learning curve. While often resulting in a trial-and-error process, it proved to be immensely educational.

As the project progressed, I continuously refined the design and pedagogical frameworks, shaped by new ideas and feedback from peers. Regrettably, my travels to Beijing meant that I did not get as much feedback as I should have, which I realized towards the end of the development process, missing improvements and changes. In hindsight, this marks my foremost learning.

All in all, today, I can confidently say that I feel well engineering websites. In the end, I am the most happy to imagine the future value of this environment, hopefully teaching and inspiring many generations to come.

5.2 Outlook

The learning environment is fully operational, but here are a few additional intriguing ideas and areas for further exploration:

- *Level canvas*: While the current freedom dimensions are beneficial, improvements in zooming and node-dragging are necessary, as currently, it is only possible at

specific locations. Moreover, incorporating different preset layouts could improve the user experience.

- Own Pace: The platform's goal is to encourage practising between solving *levels*, which is currently only possible with the empty *code IDE* page. Involving the use of AI to generate challenges including various prompt possibilities could bring a noticeable improvement.

- Personalization: Presently, all *levels* are preset and the platform is only available in German. Including language options and allowing for custom *levels* would allow teachers and users to tailor the environment to meet their specific educational needs and preferences.

To conclude, I propose one more feature, "peer inspiration". This feature enables students within the same group, such as a class, to view each other's progress (Figure 5.1). What a fascinating way to foster motivation, engagement and a feeling of community.

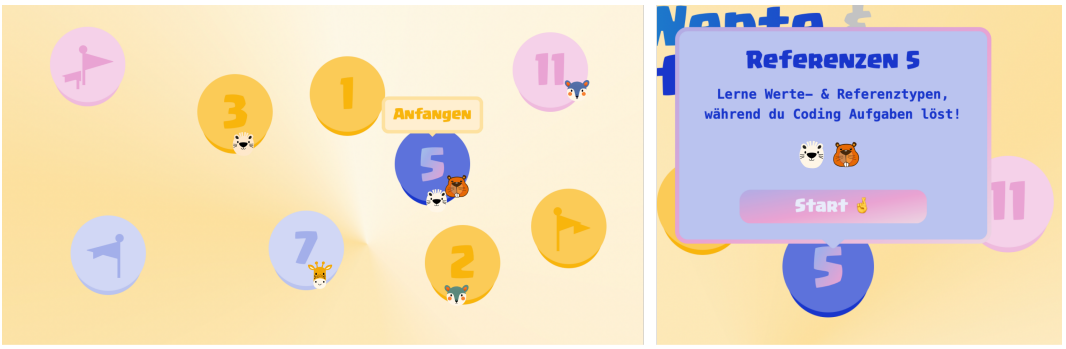

Figure 5.1: Peer inspiration

Bibliography

[1] Ausbildungs- und Beratungszentrum für Informatikunterricht der ETH Zürich, "Informatikunterricht in zeiten der digitalisierung," https://abz.inf.ethz.ch/, 2023.

[2] "Figma: Vector graphics editor and prototyping tool for digital design and collaboration," https://figma.com/, 2024.

[3] "Tell me and I forget; Teach me and I may remember; Involve me and I learn," https://quoteinvestigator.com/2019/02/27/tell/, 2019.

[4] "Duolingo: Popular language-learning platform offering courses in various languages through interactive online exercises," https://duolingo.com/, 2024.

[5] "React: JavaScript framework for building user interfaces, particularly in web development," https://react.dev/, 2024.

[6] "TypeScript: Typed superset of JavaScript that compiles to plain JavaScript," https://typescriptlang.org/, 2024.

[7] "TailwindCSS: Utility-first CSS framework," https://tailwindcss.com/, 2024.

[8] "Zustand: Alternative state-management solution for React," https://docs.pmnd.rs/zustand/, 2024.

[9] "Flask: Micro web framework written in Python," https://flask.palletsprojects.com/, 2024.

[10] "Docker: Platform for developing, shipping, and running applications in containers," https://docker.com/, 2024.

[11] "Redux: State container for JavaScript apps, often used with React," https://redux.js.org/, 2024.

[12] "React Context API: State management tool for React apps, by React," https://react.dev/reference/react/useContext/, 2024.

[13] "ReactFlow: Library for building node-based editors and applications in React," https://reactflow.dev/, 2024.

[14] "CodeMirror: Code editor implemented in JavaScript for the browser." https://codemirror.net/, 2024.

[15] "JointJS: JavaScript diagramming library used to create static diagrams or diagramming tools," https://resources.jointjs.com/, 2024.

[16] "D3: JavaScript library for producing dynamic, interactive data visualizations in web browsers," https://d3js.org/, 2024.

[17] "Visual Studio Code: Source-code editor made by Microsoft for Windows, Linux, and macOS," https://code.visualstudio.com/, 2024.

[18] "GitKraken: Git client designed to improve version control," https://gitkraken.com/, 2024.

[19] "GitHub: Platform for hosting and collaborating on software development projects," https://github.com/, 2024.

[20] "Procreate: Raster graphics editor application for digital painting developed for iOS and iPadOS," https://procreate.com/, 2024.

[21] "Adobe Photoshop: Raster graphics editor," https://www.adobe.com/de/products/photoshop.html/, 2024.

[22] "OpenAI GPT-4: Generative AI model," https://openai.com/product/gpt-4/, 2024.

[23] "Google Gemini: Generative AI model," http://gemini.google.com/, 2024.

[24] "GitHub Copilot: AI pair programmer, year = 2024, howpublished = https://github.com/features/copilot/."

[25] "OpenAI DALLE 3: AI model generating images and art," https://openai.com/dall-e-3/, 2024.

[26] "Midjourney: AI model generating images and art," https://midjourney.com/, 2024.

[27] "Adobe Firefly: AI model generating images and art," https://firefly.adobe.com/, 2023.