



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

# A Computer-Based Learning Environment Aimed for Pupils at the 3rd and 4th Grade Level

Bachelor Thesis

Florian Bütler  
flbuetle@ethz.ch

Chair of Information Technology and Education  
ETH Zürich

**Supervisors:**  
Prof. Juraj Hromkovič  
Regula Lacher

March 17, 2021

# Abstract

With the introduction of Lehrplan 21, Computer Science became an integral part of the Swiss education curriculum. To cover the new need for teaching material about Computer Science, the Education and Counselling Center for Computer Science Education at ETH Zürich has published the textbook series "einfach Informatik".

This thesis covers the implementation of a computer-based learning environment for pupils at the third and fourth grade level to teach concepts of Computer Science with exercises introduced in the textbook "einfach Informatik 3/4". The taught concepts are "representing information with symbols", "keeping information secret" and "learning from data".

# Acknowledgements

I would like to begin by expressing my gratitude to *Prof. Dr. Juraj Hromkovic* for giving me the opportunity to carry out my bachelor thesis on creating a computer-based learning environment for pupils at the third and fourth grade. It has helped me to learn many new things.

I am extremely thankful for the huge amount of feedback I received from *Regula Lacher, Jacqueline Staub, Giovanni Serafini* and *Urs Wildeisen*. This really helped me to create a satisfying result.

I am also grateful to my parents for their love and support throughout my life. I very much appreciate the opportunities and experiences they gave me that have made me who I am.

Many thanks to my proofreaders *Uchendu Nwachukwu* and *Lasse Meinen* for giving constructive criticism about this thesis.

# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgements</b>	<b>ii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Background . . . . .	1
1.2 Goals . . . . .	1
1.3 Outline . . . . .	2
<b>2 Architecture</b>	<b>3</b>
2.1 Introduction . . . . .	3
2.2 TypeScript . . . . .	3
2.3 Vue.js . . . . .	3
2.3.1 Components . . . . .	4
2.3.2 Communication Between Components . . . . .	4
2.3.3 Best Practices . . . . .	4
2.4 Basic Functionality . . . . .	5
2.4.1 Home Screen . . . . .	5
2.4.2 Game Mixin and Game Interface . . . . .	5
2.4.3 General Purpose Components . . . . .	6
<b>3 Concept - Representing Information with Symbols</b>	<b>10</b>
3.1 Similar Words . . . . .	10
3.1.1 Exercises . . . . .	10
3.1.2 Implementation . . . . .	12
3.2 Number Systems . . . . .	15
3.2.1 Exercises . . . . .	15
3.2.2 Implementation . . . . .	17

<b>4</b>	<b>Concept - Keeping Information Secret</b>	<b>21</b>
4.1	Cipher Texts from Reversed Letters . . . . .	21
4.1.1	Exercises . . . . .	21
4.1.2	Implementation . . . . .	22
4.2	Cipher Texts from New Characters . . . . .	25
4.2.1	Exercises . . . . .	25
4.2.2	Implementation . . . . .	26
<b>5</b>	<b>Concept - Learning from Data</b>	<b>29</b>
5.1	Row of Trees . . . . .	29
5.1.1	Exercises . . . . .	29
5.1.2	Implementation . . . . .	31
5.2	Tree Sudoku . . . . .	32
5.2.1	Exercises . . . . .	32
5.2.2	Implementation . . . . .	33
<b>6</b>	<b>Testing and Continuous Integration</b>	<b>38</b>
6.1	Testing . . . . .	38
6.1.1	Unit Testing . . . . .	38
6.1.2	Snapshot Testing . . . . .	40
6.1.3	End-to-End Testing . . . . .	40
6.2	Continuous Integration . . . . .	42
<b>7</b>	<b>Conclusion</b>	<b>43</b>
7.1	Obstacles . . . . .	43
7.2	Limitation . . . . .	44
7.3	Future Work . . . . .	44
	<b>Bibliography</b>	<b>45</b>

# Introduction

---

## 1.1 Motivation and Background

With the introduction of Lehrplan 21, Computer Science became an integral part of the Swiss education curriculum [1]. Pupils learn to understand the basic concepts of Computer Science and how to use them for problem-solving. These concepts include methods on how to process, evaluate and summarize data, how to securely communicate and how to develop solution strategies for simple problems of information processing [2]. The Education and Counselling Center for Computer Science Education at ETH Zürich [3] supports schools to teach these concepts among others by providing teaching materials and learning environments.

## 1.2 Goals

The main goal in this bachelor thesis is to implement tasks and riddles based on the textbook “einfach Informatik 3/4”, that will be published in spring 2021 [4], in a computer-based learning environment that teaches the following concepts:

- Representing information with symbols,
- Keeping information secret and
- Learning from data

for pupils in the second cycle, i.e the third and fourth grade of elementary school in German.

Along with solving tasks and riddles about the mentioned topics, reading, writing, counting and calculating skills are trained, too.

### 1.3 Outline

This report starts with the architecture of the computer-based learning environment and its core implementation. Next, it dedicates a chapter to each of the mentioned concepts. Each chapter first explains how the concept is taught by hands-on exercises, then it gives an in-depth technical insight on how these exercises are implemented. At this point it needs to be mentioned that the shown code might be simplified to highlight the interesting aspects, improve readability and to keep this report concise. Testing and Continuous Integration of the project is discussed before the report is rounded off with a conclusion of the project.

The source code, tests, this report and all other used scripts and documents can be found on the ETH GitLab at <https://gitlab.ethz.ch/flbuetle/bsc-thesis>.

# Architecture

---

## 2.1 Introduction

In this chapter the tools used for implementation and the basic architecture are explained. First, the used programming language and framework are introduced, then how the different parts of the learning environment work together and finally some frequently used mechanisms are explained.

## 2.2 TypeScript

JavaScript is a dynamically typed scripting language and was published in 1996. It is intended to be used in browsers to extend the possibilities of HTML and CSS. It can dynamically manipulate HTML and CSS, validate user data, send and receive data without reloading the page and much more. JavaScript used in browsers is run on client-side i.e. the workload is shifted from the provider of the web application to the client computer [5].

TypeScript extends JavaScript by providing type safety and concepts such as interfaces, methods signatures, interfaces, enumerations and tuples. It provides a way to describe what type a variable has and helps to catch errors before the code is run. The type safety is assured during compilation from TypeScript to JavaScript [6].

## 2.3 Vue.js

Vue.js is a reactive, client-side JavaScript web framework developed by Evan You and its community starting in 2014. It is an alternative to Angular and React and was developed to be a lightweight version of Angular. Vue.js is also based on reusable components, each having its own HTML, JavaScript and CSS. The reactivity system of Vue.js allows changes made to the application data to be automatically reflected in the browser.



When the development of this project started in November 2020, Vue.js version 3 was already published. However, Vue.js version 2 is used in this project, because the ecosystem has not caught up yet and many libraries only work with version 2 [7].

### 2.3.1 Components

Components are named reusable Vue.js instances and have the advantages that the structure, functionality and style of an element is implemented once and can then easily be used multiple times. Therefore, each component has a parent (except from the root component) and possibly multiple child components forming a tree structure.

Everything in Vue.js is a component, but not everything is a page. A page needs a route e.g. `/settings` and components with a route are called views.

Mixins can be used to reuse functionality over different Vue.js components. When a component uses a mixin, all functionalities of the mixin are mixed into the component [7].

### 2.3.2 Communication Between Components

Vue.js supports bi-directional communication between parent and child components. The communication takes place via properties from the parent to the child and via events from the child to the parent. Properties are custom attributes that pass data from the parent to the child component. Events are emitted by a child component, can carry data and a parent can listen and react upon receiving an event from a child component [7].

### 2.3.3 Best Practices

The following list represent the best practice that were applied in this project. To see examples for each best practice visit the Vue.js style guide [8].

- Properties should be as detailed as possible and at least have a type.
- Always use `key` with `v-for` to maintain the internal component state.
- Avoid `v-if` with `v-for`
- Only the top-level `App` component and layout components should have global styles. All other components should always have scoped styles i.e. the styles is only used within the component.
- Each component is in its own file.

- Filenames are in PascalCase.
- Components without any content should be self-closing e.g `<Component />` instead of `<Component><Component/>`.
- Components name casing in templates is PascalCase.
- Properties name casing is camelCase.
- Elements with multiple attributes should span multiple lines, with one attribute on each line.
- Component templates should only contain simple expressions. Complex expressions should be moved into computed properties or methods.
- Element attribute values should be quoted.
- Directive short hands are always used.
- Element attributes should be ordered consistently.
- Components should be ordered like `<templates>`, `<script>` and `<style>`.
- Element selectors should be avoided with `scoped`
- Properties and events should be used for parent-child communication.

## 2.4 Basic Functionality

### 2.4.1 Home Screen

The home screen is a view and is the first page seen when visiting the learning environment (figure 2.1). For each available exercise there is a card with an image to illustrate the exercise and its title. The images are inspired by those in the text book, but mostly needed to be manipulated to represent the task reasonably.

### 2.4.2 Game Mixin and Game Interface

#### Game Mixin

All exercises use the `GameMixin`, which contains the functionality for starting and evaluating an exercise. Additionally, a function to generate a random number exists to mock random number generation in tests. More on that in [Testing](#).

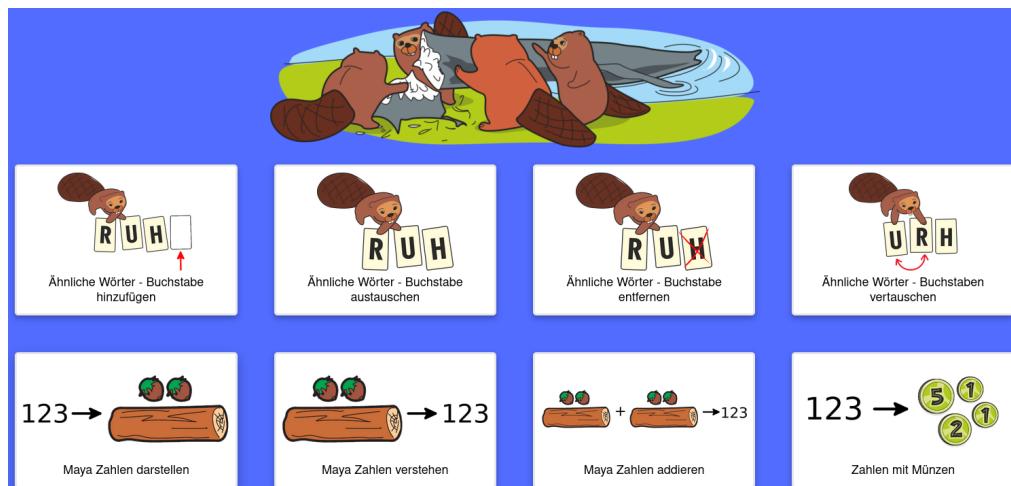


Figure 2.1: Homepage excerpt

## Game Interface

Each exercise implements the `GameInterface`. This ensures that specific functionalities, like starting and evaluating the exercise, are available and can be used in the `GameMixin`.

```

1 interface GameInterface {
2     isInitialized(): boolean;
3     start(): void;
4     isCorrect(): boolean;
5 }

```

Listing 2.1: GameInterface

### 2.4.3 General Purpose Components

The following presented components are components that are crucial or heavily reused. Most of them are part of the user interaction system since all exercises need some kind of user interaction elements.

## Game

The *game* component is the heart of the learning environment. This component defines the general structure of the exercises like the `Game Buttons`, `Tutorial Button` and shows the evaluation result of an exercise (figure 2.2). Every view uses the *game* component and it displays the exercise corresponding to the view.



Figure 2.2: Evaluation result: correct and incorrect

## Event Bus

Sometimes components are not in a direct parent-child relationship and still need to communicate. This can be achieved by passing properties and emitting events. Downside of this approach is the quick loss of a clear structure. To tackle this problem, one can use the event bus. The event bus is a Vue.js instance that allows to emit an event in one component and listen for that event on another component. To use the event bus, a component needs to import such an instance. The component can then emit or listen for events on this instance. In this project, the event bus is used between the [Game Buttons](#) and each exercise.

## Game Buttons

Every exercise needs a *check exercise* and a *next exercise* button (figure 2.3). The former is used to validate a given solution, the latter to load the next exercise.



Figure 2.3: Game buttons

## Undo Button

The *undo* button simply restores the current exercise's initial conditions so one can retry it.



Figure 2.4: Undo button

## Trashcan Button

The *trashcan* button (figure 2.5) is an area where elements can be dropped to remove them. For example, when pupils are asked to remove a letter from a word, they can either drag the letter to this area and drop it, or first click on the letter and then on the *trashcan* button to remove it.



Figure 2.5: Trashcan button

## Difficulty Level Buttons

Some exercises have multiple difficulty levels (figure 2.6). For those exercise the *difficulty level* buttons are used to change the difficulty level. This component gives the possibility to choose from up to three different difficulty levels indicated by an increasing amount of beavers on the button and a title that is shown on hover.

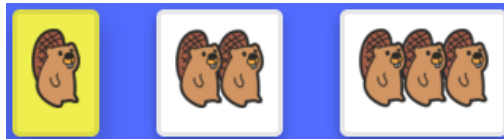


Figure 2.6: Difficulty level buttons from easy to hard

## Tutorial Button

To introduce an exercise to the pupils the title of the exercise and a short instruction is not enough. Therefore, for each exercise there is a detailed explanation and a tutorial video (figure 2.7 and figure 2.8). The tutorial video shows an example run first giving a wrong solution, then restarting the exercise and finally giving the correct solution.

The tutorial videos were created programmatically and recorded using a screen capture tool. The main reason for this is that the tutorial video needs to be redone whenever the user interface changes. With this, rerecording the tutorial video is much more convenient. Moving the mouse by hand introduces a jitter to the mouse movement, which might be irritating. The tutorial, on the other hand, should clearly show how to solve the exercise. Since this cannot be done by an average human being, the mouse movement is done programmatically.

Each graphical element part of the exercise has an ID and one can give a list of IDs that should be visited in a run. The tutorial video then shows a mouse visiting and interacting with these graphical elements, demonstrating how to solve the exercise.



Figure 2.7: Tutorial button

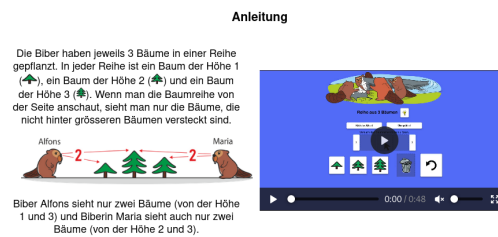


Figure 2.8: Tutorial of the row of trees exercise

# Concept - Representing Information with Symbols

---

## 3.1 Similar Words

Representing information with symbols is a fundamental concept of Computer Science as information should be displayed clearly and concisely. Words can be seen as a sequence of symbols, namely a sequence of letters. The examples given in this section are in German, since the final application is in German as well.

In order to transmit information, it needs to be encoded in a word and sent to the destination. The receiver needs to be able to decode and make sense of the transmitted information, even though the word might contain errors such as spelling mistakes. To achieve this, sender and receiver agree on a list of used words. Each pair of words from this list has a minimal editing distance.

The editing distance is the amount of operations that need to be done to transform a word in another by applying a sequence of operations. The operations are deleting, inserting and changing a letter [9].

**Example 3.1.** The editing distance between **BACH** and **FACH** is 1, since changing the first letter from B to F transforms the first word into the second.

If sender and receiver agree on a list of words with a minimal pairwise editing distance and to only transmit words from this list, then the receiver can still uniquely determine what word the sender has sent, even when an error has been made. The receiver calculates the minimal editing distance between the received word and each word of the list and chooses the word with the smallest editing distance. The receiver assumes that this was the word the sender originally sent.

### 3.1.1 Exercises

The purpose of the [Similar Words](#) exercises is to learn particular operations. An exercise is dedicated to one of the following operations: adding, changing and

removing a letter from a word. Additionally, an exercise about swapping adjacent letters in a word is included.

### **Adding a letter**

In the [Adding a letter](#) exercise, pupils are presented a word and the alphabet from A to Z. Pupils are supposed to choose a letter from the alphabet and add it to the word to form a new valid word.

**Example 3.2.** The word ARM is given. By adding the letter D before the first letter, the valid word DARM is formed.

### **Changing a letter**

In this exercise, pupils are again shown a word and the alphabet from A to Z. The pupils should select a letter from the alphabet, but this time, instead of adding it, the selected letter should replace a letter from the word itself to create a new valid word.

**Example 3.3.** The word BUCH is given. By changing the first letter from B to T the new valid word TUCH is formed.

### **Removing a letter**

In the [Removing a letter](#) exercise, pupils receive a word, from which they should remove a letter to form a new valid word.

**Example 3.4.** The word BAUCH is given. By removing the third letter the new valid word BACH is formed.

### **Swapping adjacent letters**

When typing on a keyboard, typing mistakes happen. This exercise is supposed to train the ability to spot a common mistake, where only two adjacent letters are swapped [4].

The pupils are presented a word with swapped adjacent letters, which they need to identify and swap back to restore the original word. This exercise has two difficulty levels. On the easy level only one pair of adjacent letters is swapped and on the medium level two pairs of adjacent letters are swapped.

**Example 3.5.** The word BCAH is given (easy level). By swapping the second and third letter the original word BACH is restored.



### 3.1.2 Implementation

#### Word List Generation

The foundation of these exercises is to have a list of words and for each word a list of similar words. A similar word is a word into which the original word can be transformed by either adding, changing or removing letters within a given number of operations, i.e. editing distance. In this implementation the editing distance used is one. This means only one operation can be applied, either adding, changing or removing a letter. This makes generating a list of words the easy part. Basically any list of words will do the job, but they should be understandable for children. So the first step is to collect a list of child-friendly nouns.

Not much more difficult, but much more expensive, is the computation of similar words. Given the list of child-friendly nouns and the list of allowed words, one can brute force a list of similar words by applying each transformation to every noun and checking whether the transformed word exists in the allowed word lists. The script to generate a list of similar words for each word and each operation is given in listing 3.1.

```

1 for word in children_words:
2     add = list()
3     for pos in range(len(word) + 1):
4         for letter in ALPHABET:
5             w = word[:pos] + letter + word[pos:]
6             if contains(allowed_words, w) and not w in add:
7                 add.append(w.upper())
8     similar_words["add"][1][word.upper()] = add
9
10    remove = list()
11    for pos in range(len(word)):
12        w = word[:pos] + word[pos + 1 :]
13        if contains(allowed_words, w) and not w in remove:
14            remove.append(w.upper())
15    similar_words["remove"][1][word.upper()] = remove
16
17    change = list()
18    for pos in range(len(word)):
19        for letter in ALPHABET:
20            w = word[:pos] + letter + word[pos + 1 :]
21            if contains(allowed_words, w) and not w in change:
22                change.append(w.upper())
23    similar_words["change"][1][word.upper()] = change

```

Listing 3.1: Algorithm to generate a list of similar words in Python

**Example 3.6.** Examples of similar words for adding, changing and removing a letter from a word

- Adding a letter - ARM: ARME, DARM, FARM, WARM

- [Changing a letter](#) - BUCH: AUCH, BACH, BUSH, EUCH, HUCH, SUCH, TUCH
- [Removing a letter](#) - BAUCH: AUCH, BACH, BUCH

Significantly more difficult is to generate a list of allowed words. Three different approaches were taken:

- Using a list of approximately ten thousand nouns
- Collecting a list from online dictionary of allowed words of the well known word game Scrabble [10]
- Generating a list from a spell checker

The first approach resulted in a word list with which exercises could be generated. However, the word list obviously didn't cover all possible words pupils could know. Therefore, the word list needed to be extended.

Hence the idea to create a list of words that are allowed in Scrabble came up. This seemed like a good idea until it showed that the online Scrabble dictionary does not include all valid words. Collecting words from the Scrabble dictionary resulted in approximately thirty thousand words, each between two and 8 letters long.

Finally, the last approach was taken by generating a list of allowed words from a spell checker. A well-known and open-source spell checker is Hunspell. However, Hunspell doesn't have a list of all allowed words, but rather generates them from two files: the dictionary file and the affix file. The dictionary file contains a list of root words and applicable rules for each root word. The affix file stores a list explaining these rules. Each rule defines an operation that can be applied to a root word. The spell checker evaluates a word as correct if it can construct the word by applying rules to a root word [11].

Luckily, there exists a tool called `unmunch` in the Hunspell ecosystem that generates a list of words from the dictionary file and the affix file as easy as shown in listing 3.2 [12]. In the end, a list of about one million allowed words was generated (without any word length limits). However, a drawback is that the word list does not contain any names from people, cities or any other instances.

```
1 $ unmunch German_de_CH.dic German_de_CH.aff > wordlist.txt
```

Listing 3.2: Bash command to unmunch a dictionary file and a affix file to a list of words

The actual implementation of these exercise types is straight forward. Each exercise type displays a random word from the word list and each letter of the word is on a card to allow easy user interaction. Two exercise types, namely adding and changing a letter, need an alphabet. The [Adding a letter](#) exercise is

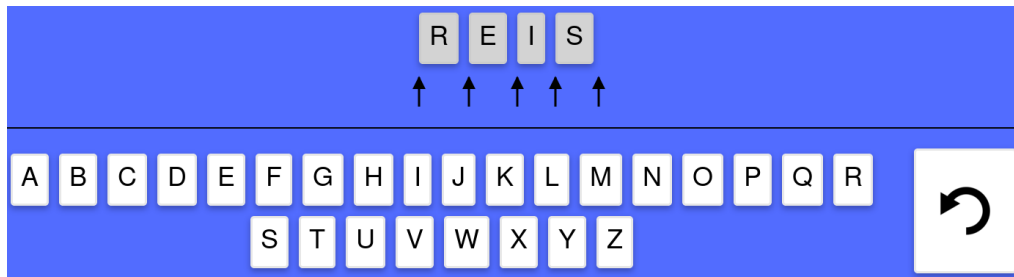


Figure 3.1: Adding a letter exercise

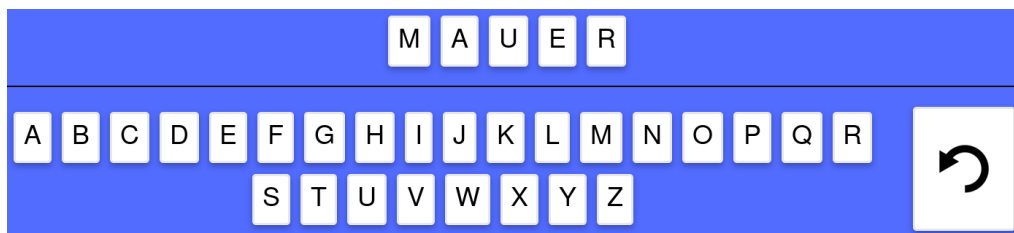


Figure 3.2: Changing a letter exercise

implemented in the `Add.vue` component (figure 3.1) and shows arrows between each letter and at the beginning and end of the word, where pupils can add a new letter selected from the alphabet. The [Changing a letter](#) exercise works similarly, but instead of having arrows, pupils can replace each letter themselves with a new letter selected from the alphabet (figure 3.2). In the [Removing a letter](#) exercise, pupils can move a letter to the trashcan to remove it (figure 3.3). In the [Swapping adjacent letters](#) exercise, they can click on the arrows to change the order of the letters (figure 3.4).

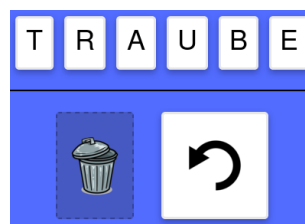


Figure 3.3: Removing a letter exercise

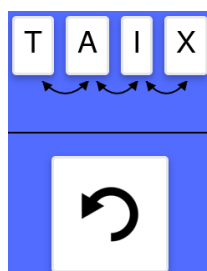


Figure 3.4: Swapping letters exercise

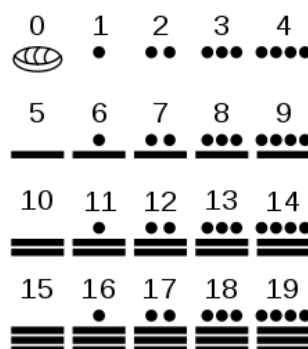


Figure 3.5: Representation of the Maya numbers from 0 to 19

## 3.2 Number Systems

### 3.2.1 Exercises

#### Representing Numbers like the Maya

Today, people are used to the decimal number system (base number 10), with 10 symbols. The Maya used a different number system with a base number of 20 i.e the vigesimal number system. It is believed that the Maya used their ten fingers and ten toes to count adding up to twenty. But instead of having a symbol for each number as in the decimal system, the Maya used three different symbols: zero (a turtle shell, belly side up), one (a dot) and five (a bar) [13]. The Mayan representation of the decimal numbers from 0 to 19 can be found in figure 3.5.

The following exercises are supposed to familiarize pupils with a new number system, the Mayan number system. First, converting numbers between the decimal number system and the Mayan numbers is trained and then topped off with an exercise about adding two Mayan numbers.



Figure 3.6: Representation of the decimal coins

### Representing Mayan numbers

A decimal number is presented to the pupils. They need to choose the correct amount of dots and bars representing the decimal number as shown in figure 3.5. For the purpose of simplicity, only decimal numbers between 1 and 19 are chosen.

### Understanding Mayan numbers

In this exercise the pupils learn to understand Mayan numbers. Again only decimal numbers between 1 and 19 are used. The pupils need to understand what Mayan number is shown and write down the decimal equivalent.

### Adding Mayan numbers

This exercise combines the previous two. The pupils need to understand the summands given in the Mayan number system, add them together and write down the sum in either the decimal number system or the Mayan number system. This exercise has two difficulty levels:

- **easy** - the pupils have to calculate the sum as a number
- **medium** - the pupils have to calculate the sum as a representation of numbers from the Mayan number system

### Representing Numbers with Coins

The following exercises introduce a new number system and train an already learned one: the binary number system and the decimal number system. Both number systems are practiced with coins and with numbers. The binary coins include the following numbers: 1, 2, 4, 8, 16, 32 and 64 (figure 3.7). The decimal coins include 1, 2, 5, 10, 20 and 50 (figure 3.6).

Overall, the same concepts are practiced for both number systems with the limitation that every binary coin can at most be used once.

### Conversion of a decimal number to its coin representation

This exercise includes two difficulty levels:



Figure 3.7: Representation of the binary coins

- Converting a decimal number to a coin representation, where the sum of all coins are equal to the decimal number
- Converting a decimal number to a coin representation, where the sum of all coins are equal to the decimal number **and** the amount of used coins is minimal.

### Conversion of a number given in its coin representation to a decimal number

A number given in its coin representation, either binary or decimal coins, has to be converted to a decimal number.

### Reducing the amount of used decimal coins while keeping the same sum

Similar to the previous exercise, a sum in its coin representation is given and pupils have to display the same sum but with fewer coins. This exercise only makes sense for decimal coins since binary coins are always minimal.

### 3.2.2 Implementation

The conversion exercises from the decimal number system to either the Mayan, decimal coins or binary coins all share the same logic implemented in the `To.vue` component (figure 3.8). The same goes for the inverse direction (`From.vue` component, figure 3.9).

However, adding Maya numbers (`Addition.vue`, figure 3.11) and reducing the amount of used coins (`Swap.vue`, figure 3.10) require their own logic and are therefore implemented in their own components.

The `To.vue` component is simple. It shows a a random number between one and a number system specific limit:

- **Mayan number system** - limit: 19
- **Decimal coins** - limit: 100
- **Binary coins** - limit: 100

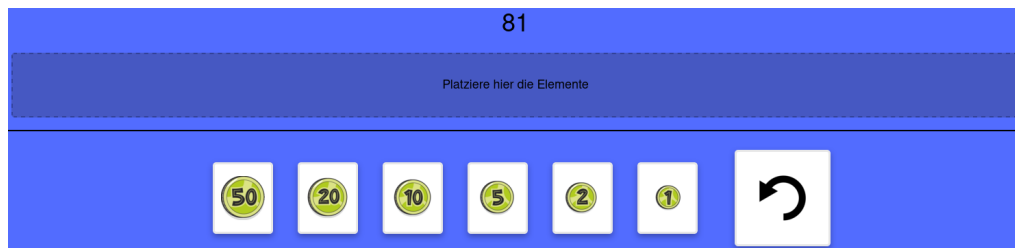


Figure 3.8: Conversion of a decimal number to its coin representation exercise

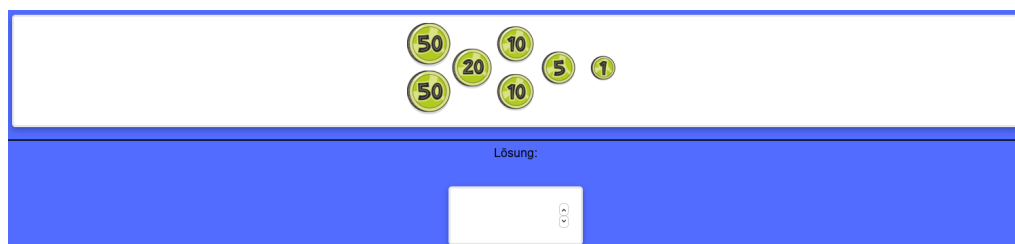


Figure 3.9: Conversion of a decimal number from its coin representation exercise

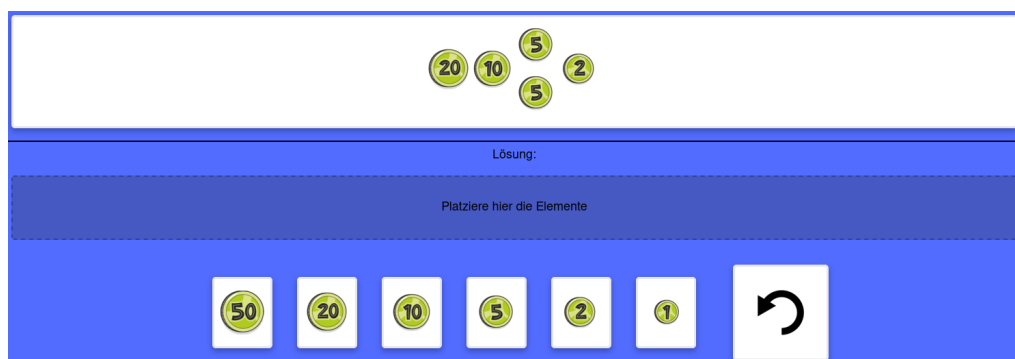


Figure 3.10: Reducing the amount of decimal coins in a given set of decimal coins exercise

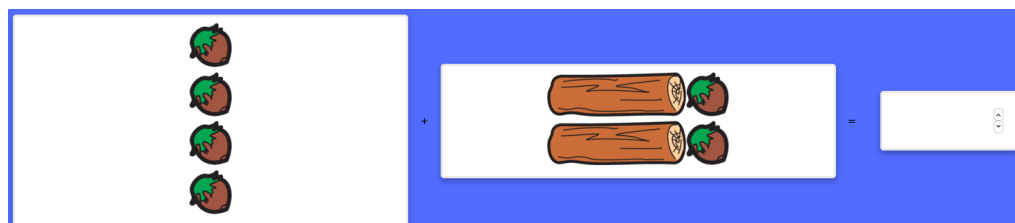


Figure 3.11: Adding Mayan numbers exercise (easy)

The pupils need to select items (either nuts, sticks or coins) with the correct value and place them such that they sum up to the shown random number.

In addition, there is another difficulty level for the conversion to decimal coins, where the aim is to represent the displayed random number with as few decimal coins as possible. The algorithm to calculate the minimal amount of items necessary for a certain sum is given in listing 3.3

```

1  calcMinimalAmount(type: numbersystemType, number: number): number[]
   {
2    const items = this.items(type);
3    let i = 0;
4    const minimalAmount = new Array<number>(items.length).fill(0);
5    while (number > 0 && i < items.length) {
6      const value = items[i].value;
7      if (number >= value) {
8        minimalAmount[i]++;
9        number -= value;
10     } else {
11       i++;
12     }
13   }
14   return minimalAmount;
15 }
```

Listing 3.3: Calculate minimal amount of items needed to reach a certain number

The implementation for the `From.vue` component works in a similar fashion. This time a random number of items is generated. The pupils need to add up these items' values.

Both of these components at some point need to sum the items' values. It would be convenient to have a map where each item type maps to the amount of selected or shown items and a map where each item type maps to its value. However, due to Vue.js constraints, this is not possible, as the Map datatype is not reactive.

The Array datatype on the other hand is reactive. Therefore an array is used, where the first element represents the highest item value, and the last element represents the item with the lowest value. This introduces some tight coupling between the array storing the amount of items and the array storing the value of each item. The implementation to sum items is shown in listing 3.4.

```

1  sumItems(type: numbersystemType, items: number[]): number {
2    if (items.length !== this.items(type).length) {
3      throw Error('array lengths do not match: ${items} ${this.items
         (type)}');
4    }
5    let sum = 0;
6    for (let i = 0; i < this.items(type).length; i++) {
7      sum += items[i] * this.items(type)[i].value;
8    }
9    return sum;
}
```



10 }

Listing 3.4: Sum up items

The `Swap.vue` component reuses some parts of the just mentioned components. It also generates a random amount of each item. The total amount of items is guaranteed to be reducible and pupils are asked to represent the same sum over all items but with fewer items. How such a configuration is achieved, is shown in listing 3.5.

```
1 do {  
2   this.generatedItems = this.generateItems(this.type);  
3 } while (  
4   this.sumItems(this.type, this.generatedItems) >= this.limit(this.  
    type) ||  
5   this.countItems(this.generatedItems) ===  
6     this.countItems(  
7       this.calcMinimalAmount(  
8         this.type,  
9         this.sumItems(this.type, this.generatedItems)  
10      )  
11    )  
12 );
```

Listing 3.5: Generate a reducible item configuration

In the `Addition.vue` component all previously seen concepts come together. Two summands are generated, each with a random amount of items. The sum is guaranteed to not overshoot the limit of this number system.

# Concept - Keeping Information Secret

---

Ciphers have been used for thousands of years [14]. They are used to keep information secret from people, that are not supposed to have knowledge of it. Non-encrypted information is called clear text. Once a clear text is encrypted, it is called a cipher text and only people who know the decryption procedure for the cipher text can read the originally encrypted information.

The exercises in this section introduce pupils to the concepts of cryptography. Simple nouns are used as clear text.

## 4.1 Cipher Texts from Reversed Letters

### 4.1.1 Exercises

The cipher used in these exercises is a simple mixture of letters. Both encryption and decryption are trained. In the decryption exercise, the pattern used for encryption is shown. The pupils need to understand the pattern and move the letters in the cipher text accordingly to obtain the clear text. The encryption exercise is set up the other way around. The pupils see the clear text and need to move letters according to the pattern to generate the cipher text. Both encryption and decryption exercises have two difficulty levels. They differ by the amount of tuples:

- **easy** - one tuple, i.e two swapped letters
- **medium** - two tuples, i.e in total four swapped letters if the the word has at least four letters, otherwise only two swapped letters

**Example 4.1.** The cipher text is **TNOAM** and the pattern is shown in figure 4.1 (medium level). By moving the letter in the cipher text according to the pattern, the clear text can be retrieved: **MONAT**.

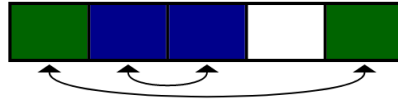


Figure 4.1: Pattern

### 4.1.2 Implementation

Both exercises have a similar implementation. Both need a drawn pattern that indicates the cipher. Fundamentally, the only difference is that, for the decryption exercise, the given text consists of reversed letters and the solution is compared to the original word, while the encryption exercise gives first the original word and the solution is compared to match the reversed letters (figures 4.2 and 4.3).

The pattern is created by generating an array of tuples. Each tuple indicates two swapped letters. These two letters are selected randomly and have to be distinct. The exact algorithm used is given in listing 4.1.

```

1  createPattern(text: string[], swapAmount: number): Array<[number,
    number]> {
2      const pattern = new Array<[number, number]>();
3      const letters = [...Array(text.length).keys()];
4      for (let j = 1; j <= swapAmount && 2 * j <= text.length; j++) {
5          const leftIndex = Math.floor(Math.random() * letters.length);
6          const left = letters[leftIndex];
7          letters.splice(leftIndex, 1);
8          const rightIndex = Math.floor(Math.random() * letters.length);
9          const right = letters[rightIndex];
10         letters.splice(rightIndex, 1);
11         pattern.push([Math.min(left, right), Math.max(left, right)]);
12     }
13     return pattern;
14 }

```

Listing 4.1: Algorithm to generate an array of distinct tuples of given size

The array of tuples needs to be graphically represented. For this purpose, the HTML `canvas` element is used. A `canvas` element is a two-dimensional area, where one can draw diagrams, edit images and create animations with the help of a scripting language like JavaScript or TypeScript. A `canvas` element has a height and width and is basically a coordinate system with the origin in the top left corner at (0, 0). Usually 1 unit corresponds to 1 pixel and all elements are drawn relative to the origin. Some examples of basic operations on a `canvas` element are drawing straight lines, curves and moving the pen without drawing. Additionally, one can choose to either fill the drawn structure with a color or only highlight its border [15]. These operations already meet the requirements to draw a pattern. The implementation is given in listing 4.1 and can be broken down to four steps:

- **drawing the grid** - each cell represents a letter of the word
- **colorizing the pairs** - the pairs representing the two letters that need to be swapped are colorized in the same color
- **drawing the arrow heads** - below each cell, that is part of a pair, an arrow head is drawn
- **drawing the arrow lines** - connecting the pairs with lines without the lines overlapping or unnecessarily crossing each other

Everything drawn outside of the `canvas` element border is invisible, hence special attention needs to be paid to the first few lines in listing 4.1 (L 2-5). Each drawn line has a width. When a line starting at the origin is drawn horizontally, the middle of the line is on the `canvas` element border, hence half of the line is outside of the `canvas` element border and hence not visible. Therefore, the rectangle is drawn with a distance of half a line width to the actual border of the `canvas` element, so the whole line width is visible instead of only half of it.

Creating the grid requires creating a rectangle, giving the shape first and then the lines creating the cells. To draw the lines, the pen needs to be moved to the start of the line, set down and moved to the end of the line (listing 4.1.2).

Drawing an arrow head includes more operations. The pen is first moved to the pointy end of the arrow, set down, moved to the two other corners and back to the pointy end. This time, instead of only highlighting just the border, the whole structure is filled black (listing 4.1.2).

```

1 drawPattern(cells: number, pairs: [number, number][[]]) {
2   const rectX = this.lineWidth / 2;
3   const rectY = this.lineWidth / 2;
4   const rectWidth = this.width - this.lineWidth;
5   const cellHeight = this.height / 2 - this.lineWidth;
6   const cellWidth = rectWidth / cells;
7
8   this.drawGrid(rectX, rectY, rectWidth, cellHeight, cells,
9     cellWidth);
10
11  // sort to have it easier to draw the lines connecting the boxes
12  // on the correct height
13  pairs.sort(([a, b], [c, d]) => Math.abs(a - b) - Math.abs(c - d))
14  ;
15  const arrowLevelY = this.calculateArrowLevelY(pairs);
16  for (let i = 0; i < pairs.length; i++) {
17    for (let j = 0; j < 2; j++) {
18      const pairIndex = pairs[i][j];
19
20      this.ctx.fillStyle = this.colors[i % this.colors.length];
21      this.fillRect(rectX, rectY, cellHeight, cellWidth, pairIndex);

```

```

20     const centerX = rectX + cellWidth * pairIndex + cellWidth /
21         2;
22     this.ctx.fillStyle = "black";
23     this.drawArrowHead(
24         centerX,
25         rectY + cellHeight + 5,
26         Math.min(30, cellWidth),
27         10
28     );
29
30     this.drawArrowLine(
31         cellHeight,
32         cellWidth,
33         rectY,
34         rectX,
35         arrowLevelY.get(JSON.stringify(pairs[i])),
36         cells,
37         pairs[i]
38     );
39 }
40 }

```

Listing 4.2: Implementation to draw the pattern on a canvas element

```

1  drawGrid(
2      rectX: number,
3      rectY: number,
4      rectWidth: number,
5      cellHeight: number,
6      cells: number,
7      cellWidth: number
8  ) {
9      // draw box
10     this.ctx.strokeRect(rectX, rectY, rectWidth, cellHeight);
11     // draw walls
12     for (let i = 1; i < cells; i++) {
13         this.ctx.beginPath();
14         this.ctx.moveTo(rectX + cellWidth * i, rectY);
15         this.ctx.lineTo(rectX + cellWidth * i, rectY + cellHeight);
16         this.ctx.stroke();
17     }
18 }

```

```

1  drawArrowHead(
2      headX: number,
3      headY: number,
4      headWidth: number,
5      headHeight: number
6  ) {
7      this.ctx.beginPath();
8      this.ctx.moveTo(headX, headY);
9      this.ctx.lineTo(headX - headWidth / 2, headY + headHeight);
10     this.ctx.lineTo(headX + headWidth / 2, headY + headHeight);

```

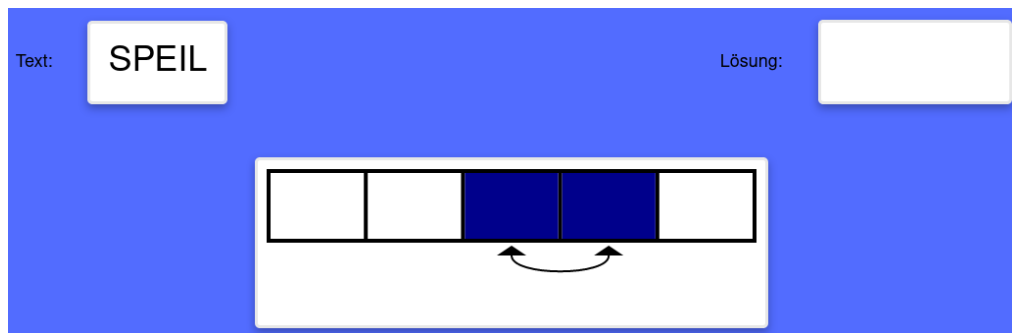


Figure 4.2: Pattern decryption exercise

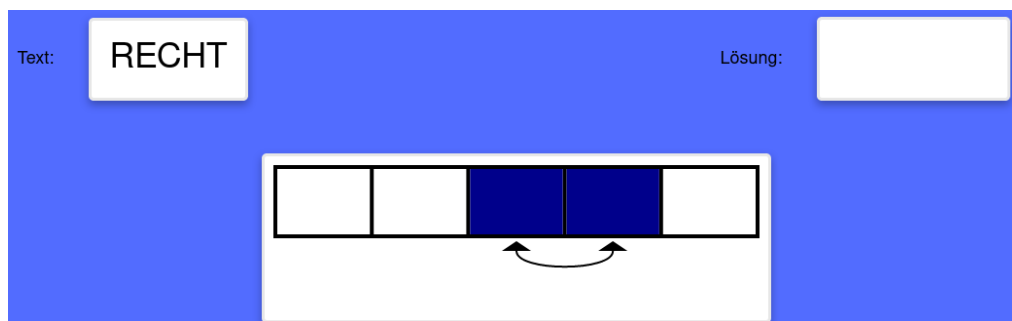


Figure 4.3: Pattern encryption exercise

```

11  this.ctx.lineTo(headX, headY);
12  this.ctx.fill();
13  }

```

**Example 4.2.** The pattern in figure 4.1 is generated by calling `drawPattern` with 5 as cells and `[[0, 4], [1, 2]]` as pairs.

```

1  drawPattern(5, [[0, 4], [1, 2]])

```

## 4.2 Cipher Texts from New Characters

### 4.2.1 Exercises

Sometimes, only moving symbols is not enough to keep information secret. A better way is to substitute symbols with new symbols. These symbols may be letters, numbers or completely new symbols, that are solely invented for the purpose of encrypting information.

In the following exercises the last approach is followed. Again both directions, encryption and decryption, are trained. But this time, instead of having a pat-

tern, there is a symbol table showing how the numbers or letters are encrypted. The decryption exercise has two difficulty levels:

- **easy** - use symbol table with numbers
- **medium** - use symbol table with letters

**Example 4.3.** The cipher text is shown in figure 4.4 and the symbol table in figure 4.5. By using the symbol table one can decrypt the cipher text to 52.



Figure 4.4: Cipher text of encrypted numbers

	.	:	:	:
□	0	1	2	3
○	4	5	6	7
△	8	9		

Figure 4.5: Symbol table for numbers

### 4.2.2 Implementation

The implementations of the encryption and decryption exercises using symbols are similar to the previous exercises using patterns (figures 4.8 and 4.9). The main difference is the use of a symbol table showing which alphanumerical letter is encrypted with which symbol. A symbol is composed of two shapes. The composition configuration is shown in the symbol table. In the decryption exercise a sequence of symbols is shown. Pupils have to use the symbol table to translate the sequence to its alphanumerical counterpart, decrypting the text. In the encryption exercise, it is the other way around. Pupils have to encrypt a text with the help of the symbol table.

The symbol table is implemented in the `SymbolTable.vue` component. It accepts a two-dimensional array representing the table content and generates this table with the help of `canvas` elements. Each cell in the table is a `canvas` element. Each shape consist of multiple elements, e.g the yellow shape in the top row of the symbol table for numbers (figure 4.5) consists of multiple dots. If there are three or fewer dots, the dots are drawn in a vertical line. For four points, they are drawn in grid style. Other shapes are more complex, like the yellow shapes in the top row of the symbol table for letters (figure 4.7), which



Figure 4.6: Cipher text of an encrypted text

	—	=	≡	▷	▷	▷	▷	▷	▷
□	A	B	C	D	E	F	G	H	I
□	J	K	L	M	N	O	P	Q	R
□	S	T	U	V	W	X	Y	Z	

Figure 4.7: Symbol table to encrypt and decrypt letters

consists of multiple arcs. The implementation for drawing these shapes takes as an argument the amount of arcs that need to be drawn. It first draws the straight vertical lines and then the specified amount of arcs on this line. This way the implementation can be used for all three arc shapes (listing 4.3) present in the symbol table (figure 4.7).

All other shapes are implemented in a similar way, so there is as little code as possible.


```

1 drawArcs(arcs: number) {
2   this.ctx.strokeStyle = "#ffa500";
3   if (arcs <= 0) {
4     return;
5   }
6   this.ctx.lineJoin = "bevel";
7   this.ctx.beginPath();
8   this.ctx.moveTo(this.lineWidth / 2, this.height - this.lineWidth
9     / 2);
10  this.ctx.lineTo(this.lineWidth / 2, this.lineWidth / 2);
11  const diffY = (this.height - this.lineWidth) / (2 * arcs);
12  for (let i = 1; i <= arcs; i++) {
13    this.ctx.quadraticCurveTo(
14      this.width - this.lineWidth / 2,
15      (2 * i - 1) * diffY,
16      this.lineWidth / 2,
17      2 * i * diffY
18    );
19  }
20  this.ctx.stroke();

```

Listing 4.3: Implementation of drawing a variable amount of arc symbols



Text:  Lösung:









				
	0	1	2	3
	4	5	6	7
	8	9		

Figure 4.8: Symbol decryption exercise

Text: TANZ  













									
	A	B	C	D	E	F	G	H	I
	J	K	L	M	N	O	P	Q	R
	S	T	U	V	W	X	Y	Z	

Figure 4.9: Symbol encryption exercise

# Concept - Learning from Data

---

The following exercises are logic-based, combinatorial puzzles. The idea is that pupils need to obtain the solution from a partially completed grid.

## 5.1 Row of Trees

### 5.1.1 Exercises

A Row of Trees is a one-dimensional grid with three or four elements. In every row there's exactly one tree of each height, going from one to three (figure 5.1), respectively four (figure 5.2). At both ends of the row, the amount of trees visible from that end of the row is given. These indications will be referred to as view slots from now on.

**Example 5.1.** An example of a Row of Trees with size three is shown in figure 5.3

Pupils are given an empty row and two view slots for both ends of the row. They need to place exactly one tree of every height such that the row is complete and the aforementioned rules are met.

**Example 5.2.** Consider a row of size three. If both view slots have the value 2, then one possible solution to the puzzle is shown in figure 5.4.



Figure 5.1: Trees from height one to three

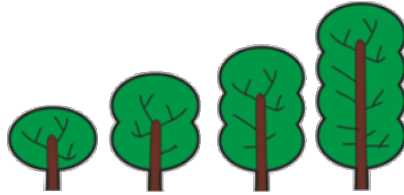


Figure 5.2: Trees from height one to four

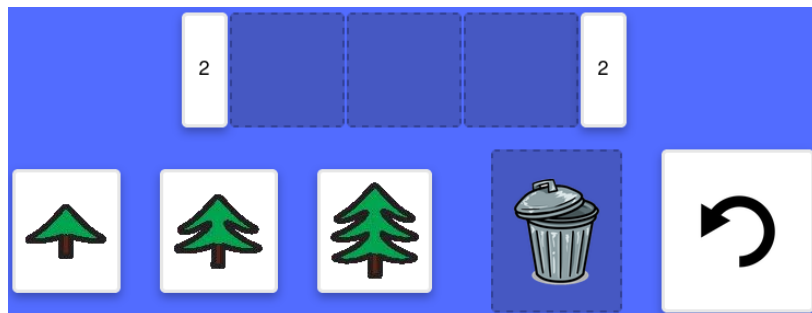


Figure 5.3: Row of tree exercise

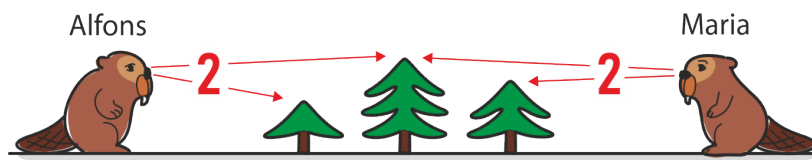


Figure 5.4: Visualization of the puzzle solution from the tree row example

### 5.1.2 Implementation

The `Row.vue` component is implemented to work with any row size. However, since we only have images of trees for rows of size three or four, it makes sense to only use it for those row sizes.

Every number between one and the row size is used as values, i.e. for a row size of three, the numbers 1, 2 and 3 are used. Every exercise case is a permutation of these values. Hence, the approach is to generate an array with these values, shuffle the array and calculate the view slots for both ends of the row. Listing 5.1 shows this approach and returns the view slot values for both ends as a tuple.

```

1 generate(): [number, number] {
2   const values = Array.from({ length: this.size }, (_, i) => i + 1)
3   ;
4   this.shuffle(values);
5   return [
6     this.getVisibleTrees(values),
7     this.getVisibleTrees(values.slice().reverse()),
8   ];
9 }

```

Listing 5.1: Algorithm to generate a row of trees exercise instance of `this.size`

Interesting functions used in listing 5.1 are `shuffle` and `getVisibleTrees`.

Apparently, no built-in function for shuffling arrays exists in TypeScript. Therefore, this functionality needs to be implemented. This is not as easy as it seems, since naive approaches may result in distinct permutations appearing with different probabilities. An implementation of a function that shuffles an array, where all permutations are approximately equal, is given in listing 5.2 [16].

`getVisibleTrees`, given in listing 5.3, calculates the amount of trees visible from one end of the row. To calculate the view slot of the other side, the same function can be used, as we only need to reverse the array as shown in listing 5.1.

```

1 shuffle(arr: number[]): void {
2   for (let i = arr.length; i >= 0; i--) {
3     const randomIndex = Math.floor(Math.random() * i);
4
5     const temp = arr[i];
6     arr[i] = arr[randomIndex];
7     arr[randomIndex] = temp;
8   }
9 }

```

Listing 5.2: Algorithm to shuffle an array

```

1 getVisibleTrees(values: number[]): number {
2   let min = 0;
3   let visible = 0;
4   for (let i = 0; i < values.length; i++) {

```

```

5     if (values[i] > min) {
6         visible++;
7         min = values[i];
8     }
9 }
10 return visible;
11 }

```

Listing 5.3: Algorithm to calculate the amount of visible tree from one end

One may have spotted the use of the custom type `row`. This custom type is the internal representation of a row. Its definition is given in listing 5.4. A `row` is an array of the custom type `rowField`, where the field `value` saves which tree was placed on this field.

```

1 type rowField = {
2     id: number;
3     value: number;
4 };
5 type row = rowField[];

```

Listing 5.4: Definition of the custom row and rowField type

To determine whether a given solution is correct or incorrect, the amount of trees visible is again calculated for both ends and compared to the view slots. If both match, then the solution is evaluated as correct (listing 5.1).

```

1 isCorrect(): boolean {
2     const row = this.values.map((field) => field.value);
3     const visibleLeft = this.getVisibleTrees(row);
4     const visibleRight = this.getVisibleTrees(row.slice().reverse()
5     );
6     return !(visibleLeft !== this.leftView || visibleRight !== this
7     .rightView);
8 }

```

Listing 5.5: Correctness check of a row of trees exercise

## 5.2 Tree Sudoku

### 5.2.1 Exercises

Tree Sudoku is similar to the well known traditional Sudoku with the difference that trees of different heights are placed instead of numbers, and for both ends of every row and column the amount of visible trees is stated in view slots. The puzzle follows the same rules as the row of trees and is either of size 3x3 or 4x4.

The Tree Sudoku exercise has three different difficulty levels. The difference between the levels is the amount of initially given information for a Tree Sudoku instance:

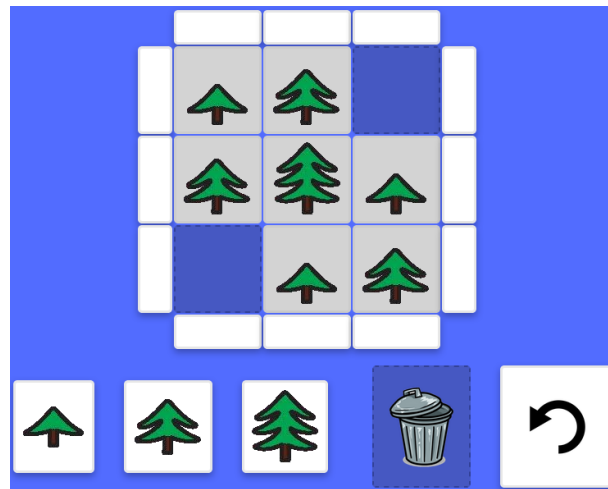


Figure 5.5: Tree Sudoku exercise

- **easy** - no views are given and at least 75% of the trees are placed
- **medium** - some views are given and at least 50% of the trees are placed
- **hard** - some views are given and only the minimal amount of information to solve the Tree Sudoku instance is given

**Example 5.3.** An example of a 3x3 Tree Sudoku is shown in figure 5.5 (level easy)

### 5.2.2 Implementation

The `Sudoku.vue` component is implemented such that it works with different sizes. In general, a similar approach for the Tree Sudoku exercise is taken as for the [Row of Trees](#) exercise. Again, some custom types are used: `sudoku` and `sudokuField` (listing 5.6). The major difference between these two exercises is that the Tree Sudoku is two-dimensional. Therefore a two-dimensional array is used for the internal representation of the Tree Sudoku grid. `sudokuField` introduces a new field `locked`. This field indicates that this Tree Sudoku field is part of the initial generation and when the same Tree Sudoku should be restarted, only non-locked fields are erased.

```

1 type sudokuField = {
2   id: number;
3   value: number;
4   locked: boolean;
5 };
6

```

```
7 type sudoku = sudokuField [] [];
```

Listing 5.6: Definition of the custom sudoku and sudokuField type

The generation of an instance of the Tree Sudoku exercise is a bit more complicated, since the additional constraints of a Sudoku have to be considered. The approach taken here is to first generate an empty Tree Sudoku. In each round, a random tree is placed in a random field or a random value is placed in a random view slot until a solvable configuration is found.

The most basic operation on a Tree Sudoku is to decide whether it is valid or invalid. A Tree Sudoku is valid if every tree height appears exactly once in each row and column and the views match the placed trees.

The same implementation is used to validate a Tree Sudoku during exercise generation and when pupils want to check their solution. Empty fields are allowed during exercise generation, but not in a pupil's solution. Therefore, the validity check takes an additional argument to distinguish between those two cases: `complete`.

For both ends of each row and column the amount of visible trees is calculated. If these values match the view slots the instance is evaluated as valid. The algorithm to check the validity can be found in listing 5.7.

```
1 isValid(values: sudoku, views: number [] [], complete: boolean):
  boolean {
2   for (let i = 0; i < values.length; i++) {
3     const rowSeen = new Set<number>();
4     const colSeen = new Set<number>();
5     for (let j = 0; j < values[i].length; j++) {
6       const traversal: [number, Set<number>][] = [
7         [values[i][j].value, rowSeen],
8         [values[j][i].value, colSeen],
9       ];
10      for (let k = 0; k < traversal.length; k++) {
11        const [el, seen] = traversal[k];
12        if (seen.has(el)) {
13          return false;
14        } else if (el === 0) {
15          if (complete) {
16            return false;
17          }
18          continue;
19        } else {
20          seen.add(el);
21        }
22      }
23    }
24
25    if (rowSeen.size === this.size) {
26      const row = values[i].map((el) => el.value);
27      const visible = this.getVisibleTrees(row);
```

```

28     const visibleRev = this.getVisibleTrees(row.slice().reverse()
29     );
30     if (
31       (views[1][i] !== 0 && views[1][i] !== visible) ||
32       (views[2][i] !== 0 && views[2][i] !== visibleRev)
33     ) {
34       return false;
35     }
36     if (colSeen.size === this.size) {
37       const col: number[] = [];
38       for (let k = 0; k < values.length; k++) {
39         col[k] = values[k][i].value;
40       }
41       const visible = this.getVisibleTrees(col);
42       const visibleRev = this.getVisibleTrees(col.slice().reverse()
43       );
44       if (
45         (views[0][i] !== 0 && views[0][i] !== visible) ||
46         (views[3][i] !== 0 && views[3][i] !== visibleRev)
47       ) {
48         return false;
49       }
50     }
51     return true;
52 }

```

Listing 5.7: Validation algorithm for a Tree Sudoku instance

A Tree Sudoku generator needs a Tree Sudoku solver to check whether an instance is actually solvable. Solving a Tree Sudoku means that one needs to repeatedly place trees in fields such that the Tree Sudoku remains valid. The Tree Sudoku generator requires the number of possible solutions for a Tree Sudoku as a stopping criterion. Hence, the solving algorithm shown in listing 5.8 additionally computes and returns the number of possible solutions.

```

1 solve(values: sudoku, views: number[][]): number {
2   const [complete, emptyValueSlotRow, emptyValueSlotCol] = this.
    findEmptySlot(
3     values.map((row) => row.map((col) => col.value))
4   );
5   if (complete) {
6     this.valuesSolution = JSON.parse(JSON.stringify(values)) as
        sudoku; // deep copy
7     return 1;
8   }
9
10  let solutions = 0;
11  for (let i = 1; i <= this.size; i++) {
12    values[emptyValueSlotRow][emptyValueSlotCol].value = i;
13    if (this.isValid(values, views, false)) {
14      solutions += this.solve(values, views);

```



```

15     if (solutions > 1) {
16         break;
17     }
18 }
19 }
20 values[emptyValueSlotRow][emptyValueSlotCol].value = 0;
21 return solutions;
22 }

```

Listing 5.8: Solving algorithm for a Tree Sudoku instance

In order to generate a Tree Sudoku, an empty field and an empty end of a row or column need to first be found. Then, with a probability of 0.5, either the empty field or the empty end of a row or column is picked. Next, a random number is put in the picked field. After that, the validity of the Tree Sudoku is checked. If it is valid, the amount of possible solutions for the Tree Sudoku is calculated. If no solution is found, a different number has to be tried. If multiple solutions are found, we start all over again by choosing a random empty field or empty end of a row or column. If the Tree Sudoku has exactly one solution, the Tree Sudoku is uniquely solvable and the algorithm terminates. An implementation of the described algorithm can be found in listing 5.9.

```

1 generate(values: sudoku, views: number[][]): [sudoku, number[][]] {
2     const [
3         valuesComplete,
4         emptyValueSlotRow,
5         emptyValueSlotCol,
6     ] = this.findEmptySlot(values.map((row) => row.map((col) => col.
7         value)));
8     if (valuesComplete) {
9         return [values, views];
10    }
11    const [
12        viewsComplete,
13        emptyViewSlotRow,
14        emptyViewSlotCol,
15    ] = this.findEmptySlot(views);
16    if (viewsComplete) {
17        return [values, views];
18    }
19    const numbers: number[] = [];
20    for (let i = 0; i < this.size; i++) {
21        numbers[i] = i + 1;
22    }
23    this.shuffle(numbers);
24    for (let i = 0; i < numbers.length; i++) {
25        if (
26            this.randomNumber(1) <= 0.5 ||
27            this.LevelsWithNoViews.includes(this.currentDifficultyLevel)
28        ) {
29            values[emptyValueSlotRow][emptyValueSlotCol].value = numbers[
30                i];
31        }
32    }
33    return [values, views];
34 }

```

```

30     values[emptyValueSlotRow][emptyValueSlotCol].locked = true;
31   } else {
32     views[emptyViewSlotRow][emptyViewSlotCol] = numbers[i];
33   }
34   if (this.isValid(values, views, false)) {
35     const solutions = this.solve(values, views);
36     if (solutions === 0) {
37       values[emptyValueSlotRow][emptyValueSlotCol].value = 0;
38       values[emptyValueSlotRow][emptyValueSlotCol].locked = false
39       ;
40       views[emptyViewSlotRow][emptyViewSlotCol] = 0;
41       continue;
42     } else if (solutions === 1) {
43       return [values, views];
44     } else {
45       return this.generate(values, views);
46     }
47   } else {
48     values[emptyValueSlotRow][emptyValueSlotCol].value = 0;
49     values[emptyValueSlotRow][emptyValueSlotCol].locked = false;
50     views[emptyViewSlotRow][emptyViewSlotCol] = 0;
51   }
52   return [values, views];
53 }

```

Listing 5.9: Generation algorithm for a Tree Sudoku instance

To fit the requirements of each difficulty level, the generated Tree Sudoku is solved field by field until the difficulty-level-specific coverage percentage is reached.

To check a pupil's solution, the algorithm from listing 5.7 can be reused by disallowing empty fields.

# Testing and Continuous Integration

---

Implemented functionality should always be tested. Not only to ensure that it does what it is supposed to do, but also to avoid regression. Nothing is worse than introducing a bug by implementing a new feature without noticing it. Therefore, unit, snapshot and end-to-end tests are used to avoid regression in this project.

Moreover, all tests are run on each commit to the GitLab repository.

## 6.1 Testing

For unit and snapshot testing Jest is used. Jest is a JavaScript testing framework focusing on simplicity and is maintained by Facebook. It requires zero configuration, runs tests in isolation and can therefore be parallelized [17]. For end-to-end testing Nightwatch is used. Nightwatch is a Node.js powered end-to-end testing framework for web applications. It supports testing in Chrome, Firefox and Edge, has the concept of page objects to easily abstract the content of a page and allows extending the framework with custom commands [18].

### 6.1.1 Unit Testing

Unit tests are used to verify the correctness of individual functions and apply the following concept: the tester has an input to a function and knows what the function should output for this input. The tester then compares the actual output of the function and compares it to the expected output. If those are not the same the test is evaluated as failed, and succeeds otherwise.

There are two main categories of unit tests: black-box and white-box testing.

## Black-box and White-box Testing

Black-box testing is a testing method where the tester does not know how a function is implemented, but knows its specification and creates tests based on this knowledge. White-box testing on the other hand is a testing method where the tester does know the implementation of the function. Usually, both kinds of software testing should be used, but since there are no software testers (testers applying black-box testing) involved in this project, only white-box testing is applied [19].

## Test Plan

General purpose components are tested on their respective functionality, i.e. whether they react correctly to different user inputs. The tests for a component representing an exercise check whether:

- Initial conditions hold
- Various exercise-specific user inputs are handled correctly
- Restarting the exercise works as expected
- Starting the next example cleans up everything from previous exercise
- The correct answer is accepted
- Incorrect answers are rejected

## Code Coverage

As mentioned before, unit tests usually test at the function level. To catch as many possible code paths and edge cases, the same function is tested multiple times with different inputs.

A metric to judge the usefulness of a test suite is code coverage. Usually, a coverage report includes:

- **Statement coverage** - The percentage of statements that have been executed
- **Branch coverage** - The percentage of branches of the control structures (e.g. if and loop statements) that have been executed
- **Function coverage** - The percentage of functions that have been called
- **Line coverage** - The percentage of lines of the source code that have been executed

High coverage percentage does not imply that it is a good test suite. Some critical paths can still be untested. It is generally accepted that a code coverage of 80% is a desirable goal. Going above 80% of code coverage is usually costly and doesn't provide much additional benefit [20].

The code coverage for the whole project and each component is given in table 6.1. Most of the components are well tested with a code coverage of above 80% in most categories. Some rows in table 6.1 indicate bad code coverage. Usually those components with bad code coverage have fewer than forty lines and are therefore not tested thoroughly.

## Non-determinism

A problem that has to be tackled for testing is how to handle non-determinism. Some exercises are based on random behaviour to achieve a rich user experience. However, if random behaviour is used, the actual output may not be the same as the expected output and the test is incorrectly evaluated as failed. To avoid this issue, one can mock the function that is used to generate random numbers. Mock functions allow to overwrite the actual implementation of the function by intercepting calls to this function and return values configured by the test suite [17]. For this reason, there is a function to generate random numbers in [Game Mixin](#).

### 6.1.2 Snapshot Testing

Snapshot tests ensure correctness of the user interface and show an alert when it changes unexpectedly. More precisely, snapshot tests render the template of a component to HTML, take a snapshot and compare the snapshot with the previously saved reference snapshot. If they don't match, the test fails. Otherwise the test succeeds. If the changes made to the source code were intended to change the UI, the snapshot has to be updated. This requires an initial snapshot that is known to be correct. Hence, snapshots should be committed to the repository as well [17].

Every component in this project is snapshot tested and is therefore safe from unintended UI changes.

### 6.1.3 End-to-End Testing

End-to-end tests are used to test the entire application flow. The main goal is to test it from a user's perspective by simulating the interactions of a use case [21].

Nightwatch allows to run end-to-end tests for Chrome, Firefox and Edge to ensure the application works across browser-borders. The end-to-end tests for

Table 6.1: Test coverage of all components

File	% Statements	% Branches	% Functions	% Lines
All files	86.71	73.11	86.73	86.56
App.vue	100	100	100	100
GameMixins.vue	55.56	100	27.27	55.56
Home.vue	100	100	100	100
ciphertexts/PatternEncryption.vue	79.41	50	81.82	78.13
ciphertexts/PatternDecryption.vue	78.79	50	80	77.42
layout/Footer.vue	100	100	100	100
layout/Header.vue	100	100	100	100
numbersystems/From.vue	100	100	100	100
numbersystems/ItemDropzone.vue	100	100	100	100
numbersystems/ItemGroup.vue	100	100	100	100
numbersystems/NumbersystemsMixin.vue	75.41	62.5	84.62	77.59
numbersystems/Swap.vue	88.89	83.33	88.89	91.18
numbersystems/To.vue	90.24	45.45	90	92.11
shared/Buttonmenu.vue	66.67	100	0	66.67
shared/Difficulty.vue	87.5	62.5	100	87.5
shared/ItemSelection.vue	100	100	100	100
shared/Modal.vue	100	100	100	100
shared/Trashcan.vue	100	100	100	100
shared/Tutorial.vue	100	100	100	100
shared/Undo.vue	100	100	100	100
trees/Row.vue	89.04	65.71	100	88.89
trees/Sudoku.vue	91.12	76.09	94.12	90.26
trees/TreesMixin.vue	100	100	100	100
words/Add.vue	98.33	100	95.24	98.28
words/Change.vue	91.43	83.33	93.33	91.18
words/Remove.vue	100	100	100	100
words/Swap.vue	61.22	87.5	76	59.78

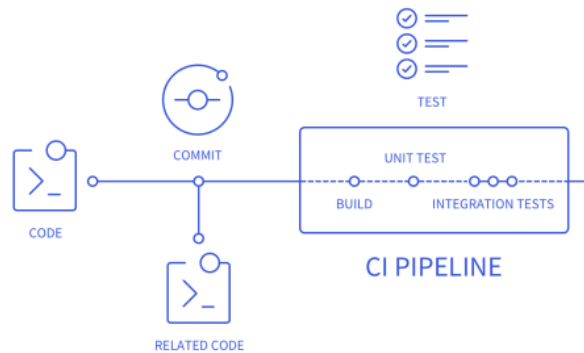


Figure 6.1: Pipeline visualization [22]

this project make use of page objects. A page object is an abstraction of a page represented as an object. Its goal is to simplify the end-to-end test [18].

Many exercises are based on non-determinism, which a user cannot influence. Thus, the end-to-end tests aren't able to, either. Therefore, the end-to-end tests in this project cover all parts of an exercise one needs to see to solve it.

## 6.2 Continuous Integration

This project make use of the GitLab Continuous Integration (CI). When pushing a commit to the project's GitLab repository, the CI pipeline is run (figure 6.1). The CI pipeline is specified in the `.gitlab-ci.yml` file in the project's root folder, which specifies which jobs need to be executed. Usually, these jobs build, test and validate changes made to the source code and allow to easily catch bugs and errors.

The CI pipeline of this project consists of two parts:

- **Unit and Snapshot tests** - runs the unit and snapshot tests as described in [Unit Testing](#) and [Snapshot Testing](#)
- **End-to-end tests** - runs the end-to-end tests as described in [End-to-End Testing](#)

# Conclusion

---

The main goal of this bachelor thesis was to implement tasks and riddles based on the textbook “einfach Informatik 3/4” in a computer-based learning environment for pupils in the second cycle. The concepts covered in this thesis are:

- representing information with symbols,
- keeping information secret and
- learning from data

## 7.1 Obstacles

One of the first obstacles I encountered was finding an adequate structure for the code. Vue.js is much less opinionated than, for example, Angular. Therefore, many possible ways to structure an application exist. On the one hand, I can enjoy the freedom that Vue.js gives. Most of the time, any application structure will work. On the other hand, this may backfire later down the road, when the application grows without any real structure having been defined. One thing I really missed in this project is feedback by other people about the structure and logic of my code. Code reviews by experienced people could point me in the right direction, thereby avoiding nightmares later on.

Another thing that turned out to be a harder problem than expected, was the generation of allowed words in the [Similar Words](#) exercises. As already showed, different approaches were tested until a satisfying result was achieved. The problem here was that the application needs to know all possible words pupils can think of, as having an exercise evaluated as incorrect, even though the word exists, is just demotivating. In the end, having a list of about one millions allowed words will hopefully cover this.



## 7.2 Limitation

An interesting aspect of this thesis was implementing exercises teaching different concepts. However, since not all exercises in the textbook “einfach Informatik 3/4” were implemented, I consider this a major drawback. Some exercises, like [Similar Words](#), are preparatory exercises for more advanced exercises, such as communication with damaged messages. I think it would improve the overall learning experience of pupils if a whole topic is covered, not only a fraction of it.

## 7.3 Future Work

The future work is directly derived from its limitations.

A big improvement would be to cover the remaining exercises of a concept that were not covered in this learning environment. Specifically, those that are built on exercises already implemented in this learning environment.

Another improvement and possible next step would be to integrate the exercises covered in this learning environment into a fully fleshed out learning environment.

# Bibliography

- [1] Lehrplan 21. Accessed on 01.02.2021. [Online]. Available: <https://lehrplan21.ch>
- [2] Medien und informatik. Accessed on 01.02.2021. [Online]. Available: <https://www.regionalkonferenzen.ch/medien-und-informatik>
- [3] Ausbildungs- und beratungszentrum für informatikunterricht der eth zürich. Accessed on 19.02.2021. [Online]. Available: <https://abz.inf.ethz.ch>
- [4] Klett und balmer verlag- einfach informatik 3/4. Accessed on 19.02.2021. [Online]. Available: <https://www.klett.ch/lehrwerke/einfach-informatik-3-4>
- [5] Javascript. Accessed on 23.02.2021. [Online]. Available: <https://developer.mozilla.org/de/docs/Web/JavaScript>
- [6] Typescript. Accessed on 11.02.2021. [Online]. Available: <https://www.typescriptlang.org/>
- [7] Vue.js. Accessed on 11.02.2021. [Online]. Available: <https://vuejs.org/>
- [8] Vue.js style guide. Accessed on 15.02.2021. [Online]. Available: <https://vuejs.org/v2/style-guide/>
- [9] T. Ottmann and P. Widmayer, *Algorithmen und Datenstrukturen*. Spektrum, 2012.
- [10] Scrabble. Accessed on 18.02.2021. [Online]. Available: <https://en.wikipedia.org/wiki/Scrabble>
- [11] hunspell - linux man pages. Accessed on 18.02.2021. [Online]. Available: <https://www.systutorials.com/docs/linux/man/4-hunspell/>
- [12] Hunspell github. Accessed on 18.02.2021. [Online]. Available: <https://github.com/hunspell/hunspell>
- [13] S. A. Kallen, *The Mayans*. Lucent Books, 2001.
- [14] A brief history of cryptography. Accessed on 11.02.2021. [Online]. Available: [http://www.cypher.com.au/crypto\\_history.htm](http://www.cypher.com.au/crypto_history.htm)
- [15] Mdn web docs. Accessed on 17.02.2021. [Online]. Available: <https://developer.mozilla.org>

- [16] Shuffle an array. Accessed on 17.02.2021. [Online]. Available: <https://javascript.info/task/shuffle>
- [17] Jest. Accessed on 16.02.2021. [Online]. Available: <https://jestjs.io/>
- [18] Nightwatch.js. Accessed on 16.02.2021. [Online]. Available: <https://nightwatchjs.org/>
- [19] Differences between black box testing vs white box testing. Accessed on 16.02.2021. [Online]. Available: <https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/>
- [20] An introduction to code coverage. Accessed on 16.02.2021. [Online]. Available: <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>
- [21] What is end-to-end (e2e) testing? all you need to know. Accessed on 16.02.2021. [Online]. Available: <https://www.katalon.com/resources-center/blog/end-to-end-e2e-testing/>
- [22] How to create a ci cd pipeline in gitlab using api. Accessed on 19.02.2021. [Online]. Available: <https://cloudaffaire.com>



Eidgenössische Technische Hochschule Zürich  
Swiss Federal Institute of Technology Zurich

## Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

**Titel der Arbeit** (in Druckschrift):

A Computer-Based Learning Environment Aimed for Pupils at the 3rd and 4th Grade Level

**Verfasst von** (in Druckschrift):

*Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.*

**Name(n):**

Bütler

**Vorname(n):**

Florian

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge“ beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

**Ort, Datum**

25.02.2021

**Unterschrift(en)**

*Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.*