MASTER'S THESIS

BLOCK HIGHLIGHTING TO IMPROVE CODE COMPREHENSION AMONG CS STUDENTS

van Battel, S.

Award date:
2022

Block Highlighting to Improve Code Comprehension Among CS Students

by

Sam Van Battel

in partial fulfillment of the requirements for the degree of

Master of Science

in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering

to be defended publicly on Wednesday February 02, 2022 at 10:00 AM.

Course code:	IM9906	
Thesis committee:	dr. ir. Fenia Aivaloglou (chairman),	Open University
	dr. ir. Alaaeddin Swidan (supervisor),	Open University

Open Universiteit
de beste! www.ou.nl

Contents

SUMMARY

In the past decade there has been a rise in the popularity of visual programming IDEs. These IDEs use custom programming languages as an introductory system into the world of programming and algorithmic thinking. Research has shown that these IDEs can be effective towards those purposes. We want to explore which advantages of visual programming IDEs are most effective. Transferring those advantages to a well-known text-based IDE, CS students can reap the benefits of both systems. We hypothesize that the visualization of codeblocks by these visual IDEs provides students with a deeper understanding of program structure.

Our research is divided into two parts. First, we use a design and creation approach to explore the visual elements that are available in visual IDEs and transfer them into a popular IDE. Then, we perform a quasi-experimental study with a group of twenty-eight high-school students, where students participate in a five-week course that teaches them the base principles of programming and algorithmic thinking. Students are divided into two groups. One group is provided with a popular text-based IDE that does not use codeblock visualizations. The other is provided with the same text-based IDE, but this time enhanced with codeblock visualizations. We use a content assessment based on the Commutative Assessment by Weintrop to evaluate basic code comprehension for each student. Statistical analysis is performed to evaluate the relevance of our results.

We conclude that our tool negatively impacts student progression during the course. Our small sample size and environmental conditions due to, among others, Covid-19 regulations opens the door towards future research.

1

INTRODUCTION

In the past decade there has been a rise in the popularity of visual programming IDEs, e.g. Snap (2020), Blockly (2019), Pencil.cc (2015), Scratch (2013). Research has shown that these IDEs can be effective as a learning tool in Computer Science education [34, 36, 37, 1]. The work by Weintrop and Alrubaye make use of a side-by-side hybrid environment in order to highlight the advantages of block-based IDEs and text IDEs together [36, 1]. However, these side-by-side hybrid environments lack important features that other major (text-based) IDEs offer, e.g. advanced syntax-highlighting, intellisence and support for multiple programming languages (to name a few). Additionally, most programming languages do not have a visual programming translation. The few that do can only provide this for a limited subset of features for that programming language. Developing and maintaining a visual programming alternative for every major programming language might not be feasible.

Many modern programming languages use codeblocks to structure code. These codeblocks add functionality (e.g. functions, classes, selection, iterations) and can be nested. In order to visualize these codeblocks and their nesting levels, programmers have used whitespace since the early beginnings of software development [6]. This type of codeblock visualization is known as indentation. Indentation has become such a staple of programming that some programming languages forgo curly brackets entirely in favor of indentation. These languages, also known as Off-Side Rule languages, enforce indentation as a way to delimit the start and end of a block of code (e.g. Python, Coffeescript). Most other block structured programming languages provide at least a style-guide[1], where the type and amount of white-spaces are specified for proper indentation [28]. Even when this is absent, many software development companies opt to provide their own style guide in the form of coding guidelines. These coding guidelines provide specific guidelines for, among other things, white-space indentation [20]. It is clear that indentation is an important part of programming and, as a consequence, so too is codeblock visualization.

The aim of this thesis is to create an extension to a popular IDE that adds codeblock

[1] Java code conventions: https://www.oracle.com/java/technologies/javase/codeconventions-contents.html

visualization, and then perform an experimental study to investigate if the tool provides similar results to previous work by Weintrop and Alrubaye, where students using a visual or hybrid IDE outperform students using a text-based IDE [34, 36, 37, 1]. We hypothesize that the visualization of codeblocks by these visual IDEs provide students with a deeper understanding of program structure.

1.1. Objectives and Research Questions

The goal of this paper is two-fold. First, we will explore the benefits of using a set of integrated visual elements in an IDE that highlights syntax and scope. Second, we will measure how this visualization will affect code comprehension among high school students.

There is a growing number of visual programming languages that are used to teach novice programmers. Several research papers have concluded that using these visual, block-based languages have a positive effect on student performance and comprehension, both as a separate way to learn programming as well as in a hybrid mode alongside text-based programming. With these research questions we focus on the visual characteristics of these visual programming languages and see if a similar effect can be observed when these characteristics are transferred to text-based languages.

In order to achieve this goal, we ask two research questions:

$RQ1$ How can the main visual characteristics of a visual IDE be implemented into a text-based IDE?

$RQ2$ How is code comprehension affected among students from the integration of block-scope visualization in a text-based IDE?

RQ1 involves multiple goals:

1. To distinguish the visual properties of codeblock highlighting in visual IDEs.
2. To analyze these properties, so we can mock-up one or more solutions to integrate these properties into a text-based IDE as visual elements.
3. Lastly, to develop a working solution/tool where a popular IDE is expanded to include those visual elements. The tool can subsequently be used to answer RQ2.

RQ2 has the following null hypothesis associated with it, which we formulate as follows:

$H1_0$ Adding codeblock visualizations to a text-based IDE does not impact code comprehension for students.

The alternative hypothesis that we will evaluate is the following:

$H1_1$ Adding codeblock visualizations to a text-based IDE impacts code comprehension for students.

In order to answer RQ2 and its hypotheses, there are two important elements that need further definition:

- How do we define code comprehension? Our definition of code comprehension will be based on previous research. Not only is creating and researching our own definition beyond the scope of this thesis, but this allows us to better compare our results to those research papers.
- How will block-scope be visualized in a student's IDE? By answering the first research question, we define how codeblock will be visualized in a student's IDE. This is why RQ1 needs to be answered before exploring RQ2: the first research question is answered before progressing to the second one.

2

Background and Related Work

The development of online IDEs has served as a major stepping stone in bringing programming capabilities to many devices. These IDEs provide several benefits, especially for education, as maintenance is low and device support is very wide [14, 38, 11]. With the advent of visual IDEs, many of which are developed as an online IDE as welle.g. Blockly, Scratch, Snap, along with many educational agencies around the world pushing for the need for a stronger IT curriculum [16, 19], the question on how to teach children programming has grown all the more important.

Initial research has mainly focused on the efficacy of visual IDEs versus text-based IDEs [36]. Initial findings point towards several advantages of using visual IDEs over text-based IDEs when it comes to teaching novice students. Further research has shown that hybrid environments improve code comprehension even more among students [1]. This is important to note, because it suggests that there are elements in both approaches that contribute to improved code comprehension among students.

Identifying which elements impact code comprehension for each approach may allow us to improve current coding practices in education, by merging these methods into popular programming IDEs. This links to other research-domains, such as color theory, code highlighting methods and text readability, all of which have been researched for several decades.

We are discussing these different domains into three different sections:

- 2.1 Text-based and Block-based IDEs, where we discuss the efficacy of block-based and text-based IDEs, along with the research methods used so far in this domain.
- 2.2 Defining Code Comprehension where we define code comprehension based on the works of Weintrop [35].
- 2.3 Highlighting Methods for Syntax and Blocks where we discuss the different code highlighting methods (mainly syntax highlighting and codeblock highlighting), their differences and evolution since their inception.

- 2.4 Color Theory for Highlighting Text where we define readability of text in terms of color and contrast.

2.1. Text-based and Block-based IDEs

Within the study of visual programming versus text-based IDEs for the purpose of programming education, few studies have been conducted [34, 36, 37, 1]. These studies focus primarily on the merits of either environment separately (be it visual programming or text-based) or a hybrid (side-by-side) environment. These studies have shown that improvements can be found with students using visual programming environments over those using a purely text-based environment [36]. A larger, more significant improvement can be found within groups of students that learn programming when using a hybrid environment [37]. Both studies used the Commutative Assessment method by Weintrop [35] in order to measure code comprehension among students.

When Weintrop presented his paper to an audience of computer science teachers[1], his results often matched the expectations of the teachers, although some teachers noticed issues that need to be part of further research. For example, one teacher remarked that during these studies, a full-featured visual programming environment was presented to the students. However, the text-based environment was very much slimmed down. Many of the comforts programmers take for granted, such as syntax highlighting and elaborate intellisence, were very limited or not available in the text-based and hybrid environments.

Because of the results that are presented by Weintrop and Alrubaye, we hypothesize that the visualization of block-scope by these visual IDEs provides students with a deeper understanding of program structure. We can support this hypothesis based on previous research. Asenov et al [4] researched how visually enhanced code can improve code comprehension. In this study, participants were presented with different levels of visual variety, starting with basic syntax highlighting and indentation, and adding further visual elements to distinguish each element of code, codeblocks being one such element. It is notable that when answering questions, the response times of participants would go down, while correctness in answering those questions remained the same. While they used a limited number of participants and questions in a controlled setting, these results do support the hypothesis that visual elements can improve code comprehension.

2.2. Defining Code Comprehension

In order to measure code comprehension among students, it is important to define exactly what it is we will measure. For the purpose of this research, we define code comprehension by following the commutative assessment method by Weintrop [35], which in turn is based on the 2013 CS Curriculum [26] and the FCS1 assessment by Tew and Guzdial [30, 31].

[1] Dr David Weintrop speaking at the Raspberry Pi Foundation: https://www.youtube.com/watch?v=6M7Vvf7ZrbU

The FCS1 assessment is a generalized approach to test and measure student comprehension in introductory computing concepts. It was developed as a language-independent assessment instrument that shows how a student can solve problems, read and analyze code, and understand introductory computing concepts. In making the FCS1 assessment, Tew and Guzdial reviewed the contents of 12 introductory computer science textbooks along with other published curricula to establish a list of ten core CS1 concepts.

Weintrop and Wilensky selected the five primary core concepts of this curriculum and included an additional two categories, based on their review of the CS2013 Curriculum. This adds up to a total of seven concepts which can be used to measure and define code comprehension, by presenting a student with practical questions and exercises. These seven concepts are:

1. programming fundamentals: How can I direct a computer to save and adjust values using a programming language?
2. selection statements: How can I use conditional logic to adjust the outcome of a program?
3. definite loops: How can I use the previous concepts (programming fundamentals and conditional logic) to execute code a specific number of times?
4. indefinite loops: How can I use the previous concepts (programming fundamentals and conditional logic) to execute code an indefinite number of times?
5. function parameters: How can I break down code into smaller, more manageable chunks?
6. program comprehension: How can I predict the outcome of a program?
7. algorithms: How can I break down a problem into smaller steps by using natural language?

2.3. Highlighting Methods for Syntax and Blocks

Apart from indentation, modern IDEs also provide another important visualization technique: syntax highlighting, by which each word is highlighted differently in order to improve readability (and thus comprehension) of the code [8, 7].

This is done through the use of color, font-weight, font-style and text-decoration[2]. Of these properties, color seems to be the most effective tool [29]. Although there are some motivations against using proper syntax highlighting (such as color blindness, printout issues and novelty), they are easily debunked [24].

Work by Hannebauer et al. suggest that syntax highlighting has few benefits towards code comprehension among novice programmers [13], but further research is needed to

[2]These terms were borrowed from the CSS language

Figure 2.1: Syntax highlighting and indentation of the same HTML file in two different code editors (VS Code and Notepad++). Notice how syntax highlighting differs between editors, but indentation does not.

say this with any certainty. Within the world of software development, syntax highlighting has become fully integrated into the work routine.

We discuss two major differences between the current method of scope highlighting (i.e. indentation) and syntax highlighting, mainly on how a programmer can influence the experience of a peer and how these methods have evolved during the past few decades.

The Consequences of Code Highlighting Preferences

The first difference is how one programmer can influence scope and syntax highlighting for their peers. Syntax highlighting is based on local settings and parsing methods[3]. When opening a document, the code is tokenized and parsed according to your own, personal settings[4]. Changing these settings has no influence on other users of the same codebase.

In contrast, indentation is saved within the document structure itself. Whether a programmer switches between spaces and tabs, changes the tab size, or even forgoes on indentation altogether, these changes determine how their peers will experience the scope visualization of the document (see Figure 2.1).

The Evolution of Code Highlighting Methods

The second difference is the advancement of highlighting techniques in the past decades. Syntax highlighting has advanced much and now provides the ability to tweak many parameters. During the same time indentation remained the main - if not the only - option for scope highlighting. As mentioned before, syntax highlighting can be adjusted in a multitude of ways, including (but not limited to) color, font-weight, font-style and text-decoration. This is in stark contrast to code block highlighting, where the only actual choice remains: how much and which white-spaces do I use to move a line of code to the right?

It is here where many visual programming IDEs provide a major improvement over conventional IDEs (see Figure 2.2). Code blocks are distinguished from each other using shape, color and, of course, indentation. Surprisingly, a majority of visual programming IDEs drop some syntax highlighting features. E.g. most visual programming IDEs provide syntax highlighting through the use of statement blocks, but do not provide any syntax highlighting for

[3]VS Code highlighting settings: https://code.visualstudio.com/docs/getstarted/themes
[4]VS Code highlighting methods: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide

value types.

```
for (let i = 0; i < 5; ++i) {
    if (5 + 2 === 7) {
        console.log("Hello World!");
    }
}
```

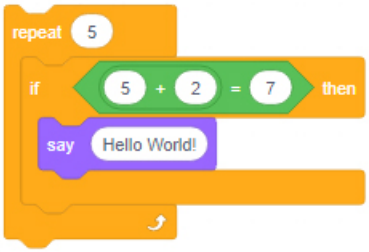

Figure 2.2: Visual IDEs such as Scratch (right) distinguish code blocks more visually compared to text-based IDEs.

2.4. Color Theory for Highlighting Text

There are few usability standards for user interfaces when it comes to readability and contrast. The most extensive standard, WCAG 2[5], is developed for the accessibility of content on the web, but can easily be applied to application development. Technically Visual Studio Code and Atom (two popular, light-weight text-based IDEs) can even be seen as web apps, as they both use the electron framework, which is a framework for building desktop applications using JavaScript, HTML, and CSS by embedding Chromium and Node.js[6].

WCAG 2 contains many different success criteria. The two that are interesting for the purpose of readability and color contrast are:

- Success Criterion 1.4.3 Contrast (Minimum)
- Success Criterion 1.4.6 Contrast (Enhanced)

Both success criteria use a contrast ratio formula based on the relative luminescence of the background and foreground colors. The formula provides us with a number between 1 and 21 (commonly written as a ratio between 1:1 and 21:1), where a minimum of 3:1 is needed for 1.4.3 and a minimum of 4.5:1 is needed for 1.4.6.

A contrast ratio of 3:1 is the minimum level recommended by ISO-9241-3 and ANSI-HFES-100-1988 for standard text and vision [32]. Criterion 1.4.6 is needed for people who are impeded by moderately low vision and do not make use of any assistive technology to artificially enhance contrast [33].

2.5. Similar Extensions

While researching the possibilities of Visual Studio Code as a base for our research tool, we came across several tools that implement some type of codeblock parsing. These exten-

[5] https://www.w3.org/TR/WCAG21/
[6] https://www.electronjs.org

sions heavily inspired us and provided a lot of practical examples to the API documentation.

- bracket pair colorizer: This extension allows matching brackets to be identified with colours. The user can define which characters to match, and which colours to use[7]. It even provides a visualization of the current codeblock with a 'C'-like shape. However, the type of block is of no consequence to the coloring, it only works on bracket-based languages and the visualization is only provided for the codeblock where the cursor is positioned at.
  We experimented with the code of this tool, and managed to change it such that every codeblock gets highlighted. However, implementing a way to identify which codeblock-type each codeblock is took a lot of effort and it will still only work for bracket-based languages.

- indent rainbow: This extension colorizes the indentation in front of your text alternating four different colors on each step. Some may find it helpful in writing code for Nim or Python[8]. This was a great example of indentation parsing for each line of code, although it does not provide any way to identify the current codeblock and empty lines are ignored.

[7] https://marketplace.visualstudio.com/items?itemName=CoenraadS.bracket-pair-colorizer
[8] https://marketplace.visualstudio.com/items?itemName=oderwat.indent-rainbow

3

CODE BLOCK HIGHLIGHTING TOOL

Previous research in the domain of measuring the efficacy of visual IDEs and text-based IDEs has mainly consisted of an experimental research approach, where a custom tool is developed in order to allow students to experience either method separately [1]. Developing such a tool is the first step to performing the experiment. Similarly to those previous experiments, we want to provide students with a programming IDE where codeblock visualization can be turned on or off, depending on which group of students.

It is our hypothesis that the visualization of codeblocks by visual IDEs provides students with a deeper understanding of program structure. Therefor, we want to create a text-based IDE that provides similar visualizations as visual IDEs to users. In order to create such a tool (one that can be used during the experiment) we first need to define which characteristics of codeblock visualizations are common across multiple visual IDEs. Then we mock-up an idea for transferring those characteristics into a text-based IDE.

Previous experiments have either created an IDE from scratch or, most often, adjusted an existing IDE. For the purpose of our experiment, we pursue the option to adjust an existing IDE. This allows students to experience the advantages of the (text-based) IDE in its current state, such as syntax-highlighting and auto-formatting. We do this through the use of the design and creation approach.

In the following sections we will explain this approach:

- 3.1 Design and Creation Approach for Codeblock Highlighting Methods where we explain the process of the design and creation approach.
- 3.2 Designing a Codeblock Highlighting Tool where we define the characteristics of codeblock visualization elements in visual IDEs and how we would go about transferring those characteristics to a text-based IDE.
- 3.3 Creating a Codeblock Highlighting Tool where we take you through the selection process of selecting an IDE, and the development process of developing the tool within the constraints of that IDE.

3.1. Design and Creation Approach for Codeblock Highlighting Methods

The first research question will be answered by using a design and creation approach [21]. This approach is well suited as it involves developing a tool to support a new hypothesis that has not been previously tested. It involves parts from other domains, such as color theory, layout and spacing, when very little of these theories have previously been applied to measure code comprehension. Answering this question will include designing and developing a working prototype.

Developing such a prototype requires more insight into the different visualization techniques that are used in text-based and visual IDEs. Exploring related research on text-, code- and shape-visualization techniques allows us to make substantiated choices. Additionally, constructing IDE/visualization tuples will show which visualizations are used by which IDEs. We will also evaluate each IDE on the following topics:

- Extensibility: Many schools restrict or disrupt some software installation procedures. The ease of extensibility should be considered in order to ease the installation process for the students.
- Programming language compatibility: visual IDEs may restrict teachers and students to specific programming languages or even subsets thereof. We should therefore consider the compatibility of the IDE with multiple major programming languages.

3.2. Designing a Codeblock Highlighting Tool

Research has shown that visual IDEs can be an effective learning tool for computer science students. Most often this research has focused on Snap [15], Blocky [18, 27], Pencil.cc [5] and Scratch [22, 17]. By analysing how these four visual IDEs incorporate their visual characteristics, and then selecting which ones are useful towards code comprehension, we can better understand which ones should be used in our research tool and which ones do not.

3.2.1. Characteristics of Blocks in Visual IDEs

Looking at these visual IDEs shown in Figure 3.1, we separate two main characteristics:

1. Shape topology
2. Shape color

Although typography and text color differs as well between these visual IDEs, it is mostly automatically adjusted to contrasting colors (black/white) in respect to the background-color.

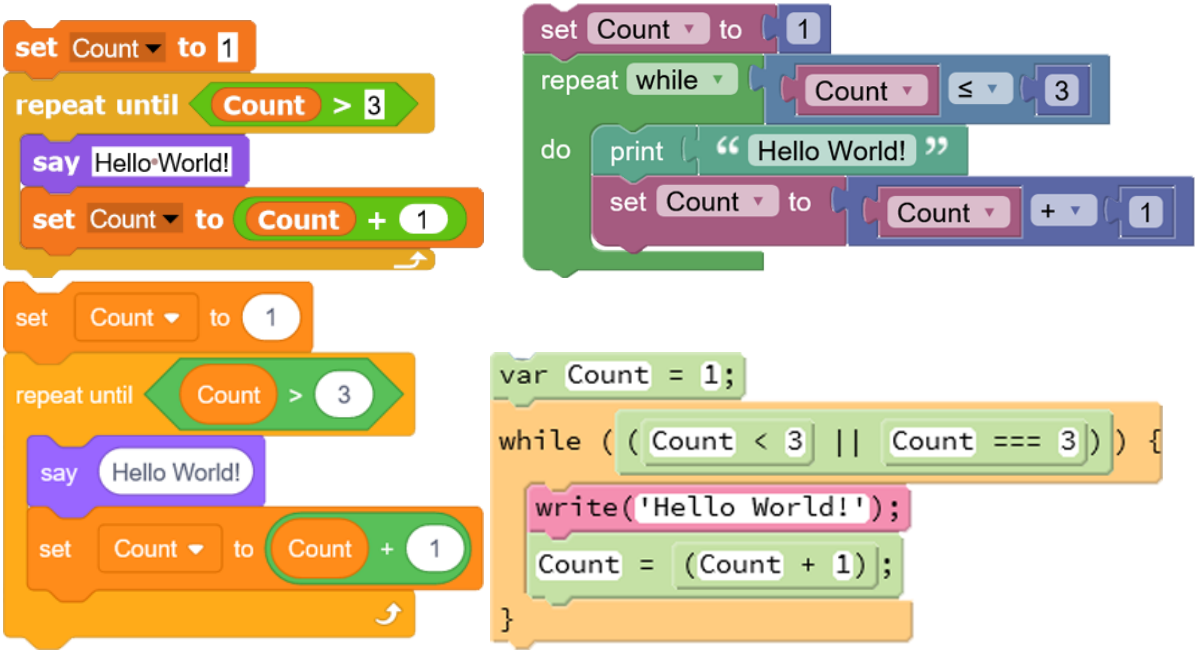

Figure 3.1: The four most popular Visual IDEs that are often used in research: Snap (topleft), Blockly (topright), Scratch (bottomleft) and Pencil.cc (bottomright). They use shape topology and shape color to distinguish between different blocks, such as operations, statements and codeblocks.

By looking more closely into these characteristics, we can establish the specific features we need to implement in our testing method.

SHAPE TOPOLOGY

Codeblocks, shown in every visual IDE as a 'C'-shape, look very similar in each of these visual IDEs. However, there are a few differences which contribute to usability and readability [10]. The topology for codeblocks in Figure 3.2 always has a connector on the inside-top, but some environments also have a connector on the inside bottom (e.g. Scratch, Pencil.cc) whereas others do not (e.g. Blockly, Snap). The usefulness of the bottom connector is a point of ongoing discussion [10]. The top connector is, on the other hand, unanimously implemented across all visual IDEs. These connectors will fit the shape of statements, conveying to the user that certain statements can or cannot be added to these 'C'-shaped codeblocks.

It is important to note here that these connectors are implemented *because* they are used in a visual programming language. Users drag and drop statements instead of typing instructions. They help visual IDE users to find and select the correct type of statement-blocks that will fit their codeblocks, but do not improve upon the readability of program-structure. Because we want to research if visualizing codeblocks in text-based IDEs improves code comprehension, we will not implement this feature in our testing method.

The 'C'-shape itself is very interesting. It clearly shows the start and end of a codeblock, with the beginning and end elongated. The thickness of the left side varies across different visual IDEs, with Blockly using a wider spacing and other IDEs a smaller spacing. There is

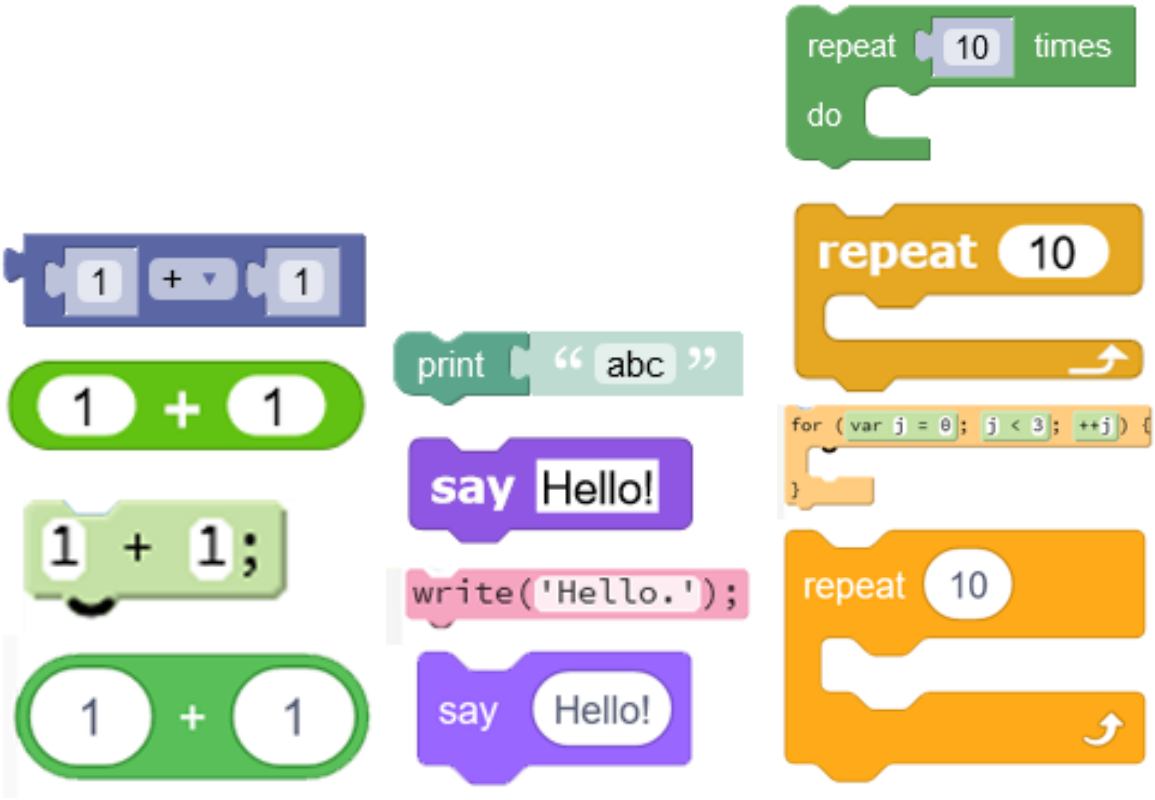

Figure 3.2: Topology for inline items, such as operators (left) and statements (center) is show here as items that can be horizontally or vertically daisy-chained. Topology for codeblocks (right) are shown as a C shape with a connector on the inside-top and inside-bottom.

no clear indication why Blockly chose this amount of spacing, and there are several implementations of Blockly where they use a smaller spacing on the left[1].

The thickness of the left side corresponds to indentation size in text-based programming. When using spacing instead of tabs, there is often a choice in indentation size between two to four spaces (and sometimes even larger). Previous research on indentation size has shown that there is little difference between smaller and larger indentation sizes for small code snippets [6], where 'small' refers to code snippets that do not require scrolling to read the entire snippet. As our research focuses on an introductory programming course for high school students, we will seldomly present students with code snippets larger than that.

Shape Color

Shape color is used to differentiate between operations, statements and codeblocks. We are mostly interested in the visualization of codeblocks. Figure 3.3 shows that most visual IDEs do not differentiate between different types of codeblocks, with the only exception being Blockly and Pencil.cc. Pencil.cc divides codeblocks into two major categories:

[1] https://blockly.games/pond-duck

- Control: these codeblocks are selective or iterative. They consist of the blocks ‘for’, ‘while’, ‘if’ and ‘forever’.
- Operators: these codeblocks group statements together. They consist of the block ‘function’.

Blockly divides codeblocks into three major categories:

- Logic: these codeblocks are selective. They consist of the blocks ‘if’, ‘else’ and ‘elif’.
- Loops: these codeblocks are iterative. They consist of the blocks ‘repeat’, ‘count’ and ‘foreach’.
- Functions: these codeblocks group statements together. They consist of the blocks ‘function’ and ‘function return’.

In both cases, each category gets a distinctive color that is consistent across all blocks of that category.

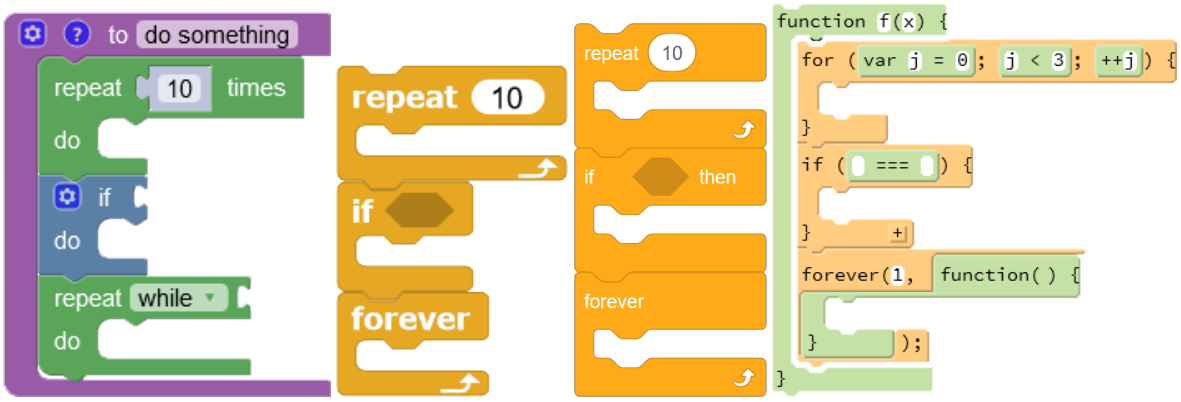

Figure 3.3: Most visual IDEs do not change colors between different types of codeblocks. Only Blockly (left) and Pencil.cc (right) differentiates between different categories of codeblocks.

It is not clear why some of these visual IDEs choose to divide these blocks into different categories and some do not. The choice of colors is also not documented, as far as we can tell. Blockly, which allows developers to use their framework to implement custom visual programming languages, provides several tips for creating a block language [23]. There, they suggest using color in order to reinforce similarities between blocks, but do not offer any advice on color choice or categories.

Because there is little information on the choice of colors within visual IDEs, we must turn to other media that try to convey textual information with the use of shape and color. One of those media is web development, where the use of colors (in order to convey information to a user) is much more researched and well-defined.
The Web Content Accessibility Guidelines offer a multitude of guidelines that allows developers and designers to convey information to their users as clearly as possible. Specifically,

guidelines *1.4.3 Contrast (Minimum)* and *1.4.6 Contrast (Enhanced)* provide clear directions to improve the readability of text through the use of color contrast.

WCAG2.1 Chapter 1 contains information and techniques on the information and user interface components must be presentable to users in ways they can perceive [9]. As we want to show users (the students) specific information (the code they have written) in a way they can perceive through a user interface component (a codeblock visualization), this standard may very well apply to our research tool.

Section '1.4.3 Contrast (Minimum)'[9] sets a baseline contrast for the readability of text in most situations. Except for two exceptions (large text and incidental text, both of which are not applicable to our study) all text should have a contrast ratio of at least 4.5:1. The contrast ratio is calculated through the following formula:

$$(L1 + 0.05) / (L2 + 0.05)$$

where:

- L1 is the relative luminance of the lighter of the colors, and
- L2 is the relative luminance of the darker of the colors.

This formula is based on several different aspects:

- The ISO-9241-3 and ANSI-HFES-100-1988 standards for the formula (L1/L2)
- The typical viewing flare from IEC-4WD and the proposal for a standard default color space on the internet [2]. This is in turn an answer to the ANSI/HFS 100-1988 standard, which calls for a contribution from ambient light in the calculation for L1 and L2.

The ISO-9241-3 and ANSI-HFES-100-1988 standards recommend a contrast ratio of 3:1 or higher for standard text and vision. In case the user suffers from lower visual acuity, any color deficiencies or loss of contrast sensitivity, this contrast ratio should be adjusted to 4.5:1 or higher [3].

The 4.5:1 contrast ratio can be used as compensation for users with a visual impairment comparable to 20/40 vision, which in itself is comparable to the vision of a typical 80 year old person [12]. For our purposes, the contrast ratio 3:1 is therefor sufficient.

3.2.2. Transferring Characteristics to Text-based IDEs

We translate the characteristics concerns into requirements. Every requirement is given a priority, based on a simplified version of the MoSCoW-method[2]. This simplified version is based on experiences in other software companies that use this same simplified method for appointing priorities to requirements before they are translated into tasks.

[2] https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation

- **B**lockers: That which must be included. without this the system will not work as intended.
- **M**ust-haves: Requirements *required* by a stakeholder that are not a Blocker.
- **P**epper&Salt: Requirements *suggested* by a stakeholder that are not a Must-have.

It is important to maintain a distinction between Blockers and Must-haves. Blockers contain all core functionality of the software. Changes to these requirements should be scarce. When all blockers are implemented a functional proof-of-concept can be showcased. Meanwhile, Must-haves are important to the stakeholders but have no impact on the core functionality of the system. They are much more prone to changes. These requirements should be implemented with changeability and maintainability in mind.

color requirements	Priority	
maintain a contrast ratio of at least 3:1	**M**	A contrast ratio of at least 3:1 is important to maintain readability of the text an UI.
maintain a contrast ratio of at least 4.5:1	**P**	A contrast ratio of 4.5:1 and greater is preferred, in case students with visual impairment want to use the research tool.
Categorize codeblocks in logical groups	**M**	Using categories to distinguish between different types of codeblocks can help users to more easily read and understand their code.
Distinguish different codeblock categories	**M**	A clear color scheme that is specific to its codeblock category, so users can easily see what types of codeblocks they have used in different situations.
usability requirements	Priority	
Frequent updates of visualizations	**B**	Visualizations should be updated frequently, so that the user can use these visualizations to read their code more easily.
Live updates of visualizations	**M**	Live visualization updates are preferred to frequent (timed) updates, because code structure might change dramatically between updates when timed updates are used.
topology requirements	Priority	
Distinguish different codeblocks	**B**	A clear start and end, so users can easily see where each codeblock starts and ends.

Provide each codeblock with a 'C' shaped visualization	**M**	The 'C' shape is a consistent marker for codeblocks in each visual IDE. It provides a clear sense of start and end of a codeblock.
Provide the 'C' shape visualization with an inside-top connector	/	These connectors are implemented in visual IDEs to visualize where certain statement-blocks can be dragged and dropped. Because we want to research if visualizing code-blocks in text-based IDEs improves code comprehension, we will not implement this feature.
Provide the 'C' shape visualization with an inside-bottom connector	/	These connectors are implemented in visual IDEs to visualize where certain statement-blocks can be dragged and dropped. Because we want to research if visualizing code-blocks in text-based IDEs improves code comprehension, we will not implement this feature.
Provide each codeblock with indentation-like padding to the left	**B**	Left-side indentation is a consistent marker for codeblock-nesting in each visual IDE, as well as it is required by many programming languages and/or companies.

Table 3.1: Requirements for developing a codeblock highlighting extension.

To further guide us, we create a mockup based on all Blocker and Must-have requirements. To do this, we made a screenshot of a popular code editor and used an image editor to imagine a non-intrusive way to visualize codeblocks (see Figure 3.4).

- Each codeblock is equipped with a darker 'C' shape
- Each codeblock is equipped with a lighter background-color. (e.g. the function 'submitRegisterForm')
- Different types of codeblocks are colored differently. (e.g. the function 'userRegistered' and the nested selection block)
- similar types of codeblocks are colored identically (e.g. the if-block and the else-block)
- Each codeblock is indented to visualize nesting-level.
- text- and background-color has a contrast-ratio of at least 3:1.

```
let form = document.querySelector("#RegisterForm");
form.addEventListener("submit", submitRegisterForm);

let request = new XMLHttpRequest();

function submitRegisterForm(submitEvent) {
    submitEvent.preventDefault();

    let email = form.elements["registerEmail"].value;
    let name = form.elements["registerName"].value;
    let password = form.elements["registerPassword"].value;

    console.log(password);

    request.open("POST", "register.php");
    request.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    request.addEventListener("load", userRegistered);
    request.send("name=" + name + "&email=" + email + "&password=" + password);
}

function userRegistered() {
    let response = JSON.parse(request.response);

    if (response.registerSucceeded == true) {
        document.querySelector("#registerSucces").style.display = "block";
    }
    else {
        document.querySelector("#registerFail").style.display = "block";
    }
    console.log(response);
}
```

Figure 3.4: A mockup of codeblock visualization, overlayed on top of the VS Code editor.

3.3. Creating a Codeblock Highlighting Tool

In order to teach students the basics of programming, we need to pick a programming language. Because we offer a course of five weeks, the language needs to be comprehensible enough in order to allow novice-programmers to understand the basic concepts that we will teach them within that time frame. It is preferred to use a language with minimal tooling requirements, ideally just an IDE and compiler/interpreter.

The codeblock-visualization tool also needs to be compatible with the language. We can fit the language to the tool, or fit the tool to the language, but it is primarily important they are compatible with each other.

Because the experiment will take place in a school-environment, we need to consult the local ICT-team and look at what types of configurations are possible with or without administrative privileges. Using a popular IDE as a development environment is preferred, because it will increase the trustworthiness of the tool for outside parties.

3.3.1. IDE Selection Process and Parameters

To find out which IDEs are most popular across the world, we use PYPL[3]. PYPL is a website that maintains rankings for the most popular programming languages, IDEs, ODEs and databases. The website creates this ranked list based on the amount of times a download page is searched on google for an IDE. The more a download page is searched, the more popular an IDE is assumed to be.

rank	IDE	Share
1	Visual Studio	27.37
2	Eclipse	15.94
~~3~~	~~Android Studio~~	~~11.87~~
4	Visual Studio Code	10.03
5	pyCharm	7.68
~~6~~	~~IntelliJ~~	~~6.86~~
7	NetBeans	5.58
8	Sublime Text	3.79
9	Atom	3.51
10	Xcode	3.42

Table 3.2: Top 10 most popular IDEs according to PYPL.

In consultation with the school's ICT team, we came to a list of properties that are important to consider when choosing an IDE as a basis for our research tool. By sorting this list based on priority, we can use this list as a priority list.

1. Extension support (required): for our research needs, the IDE needs to be extensible with an API that allows for visual extensions.

2. Visual extension capabilities (required): the IDE's extension support should be able to visually adjust the workspace with shapes and colors.

3. Python support (required): it also needs to support Python as a programming language, as this will be the program language of choice to teach the curriculum.

4. Price (lower is better): there is very little budget for this project, so price is a factor in order to install and maintain such a software package.

5. Size (lower is better): the download and installation size should not be too large, as it needs to be installed on many machines as quickly as possible. Install size is also a good indicator for extensiveness of the IDE. Because the experiment has novice programmers as its subjects, an extensive IDE may be too overwhelming.

6. Multi-platform support (more is better): the school utilizes a combination of Apple and Windows machines. Also, students may have different machines at home than

[3]https://pypl.github.io/PYPL.html

they have access to at school. The IDE should therefor preferably work on multiple systems.

7. Support for other programming languages (preferred yes): In order to facilitate future ICT lessons, a wider range of supported programming languages is preferred by the school. This may also allow us to test the research tool for other programming languages if needed.

Going through the documentation and support pages of each IDE listed in table 3.2, we can create a more detailed overview of these properties in table 3.3. Each IDE was checked for Python support, install size, cross-platform support, support for other languages, price, extensibility and capabilities for visualization extensions.

Table 3.2 shows the ten most popular IDEs as of April 2021 according to PYPL. Two of these IDEs do not have Python support, and are therefore crossed out. The official documentation of every other IDE lists python as a supported language, or references a maintained plugin/extension that allows support for python.

IDE	Share	python (Y/N)	size	Mac Win Linux (M/W/L)	Other Languages	price	extensibility (Y/N)	visual extensibility
VS Code	10.03%	Y	+	M W L	Y	free	Y	+++
Atom	3.51%	Y	+	M W L	Y	free	Y	+++
Sublime Text	3.79%	Y	+	M W L	Y	paid	Y	+++
NetBeans	5.58%	Y	++	M W L	Y	free	Y	++
pyCharm	7.69%	Y	+	M W L	N	paid	Y	+++
Visual Studio	27.37%	Y	+++	M W	Y	free	Y	+++
Eclipse	15.94%	Y	+++	M W L	Y	free	Y	++
Xcode	3.42%	Y	+++	M	Y	free	Y	++

Table 3.3: Top 10 most popular IDEs, excluding the ones without Python support, sorted by fitness. IDE name, Share (higher is better), size (lower is better), Apple/Windows/Linux support (more is better), support for other languages (preference for yes), price (lower is better), extensibility (required), visual extension API (higher is better)

Using this prioritization, we can sort table 3.3 to reach a ranked list of preferred IDEs. We remove the rank column from table 3.2, because a) there is little use for it and b) it is a PYPL ranking based on the share percentage, which is still included.

By using a low-cost, easy to install and multi-platform IDE, our proposed research-tool will be able to run on most IT infrastructures, including systems at schools and at home.

From table 3.3 there are two IDEs that stand head and shoulders above the other options: VS Code and Atom. They both have an extensible extension API with documentation, are lightweight, can be kept minimalistic, have Python language support, are free to

use, are easy to install and come with several installation options. Having these two options means that we can start developing our research tool in the first option, and in case of failure have a second option to start and try again.

We start with extending VS Code because it has the larger market share. There are already many visual extensions available in the VS Code extension market, much of which is available as open-source projects on Github. This means we can learn from many different sources to work out and test a working prototype as quickly as possible.

It is important to note that Visual Studio Code has several limitations. It does not provide a public API for its tokenization engine, and only minimal access to its token parsing engine. This will limit the parsing methods we can apply to detect and visualize codeblocks.

3.3.2. Extension Development for Visual Studio Code

Table 3.1 lists all requirements for the development of the extension. One of the Blocker-requirements needs us to distinguish between different codeblocks with a clear start and end, so users can easily see where each codeblock starts and ends. In order to implement this feature, we need to first detect codeblocks in a document, and subsequently mark them in a visual and distinctive way. Codeblock detection can be done with three different methods:

- Abstract Syntax Tree derivation
- Token parsing
- Indentation parsing

Abstract Syntax Tree Derivation

Visual IDEs allow a user to work directly on the Abstract Syntax Tree of the program, so using the AST to back our visualization methods allows us to closely match the visualization techniques used in visual IDEs. Some text-based IDEs parse each file for known programming languages and build a complete or partial Abstract Syntax Tree. It may be possible that these ASTs are exposed to the extension framework in the form of a public API.

This method has some downsides. Not all IDEs build an AST for each file, as this can be a computationally expensive task. Even if an IDE creates one, they may not expose it to its extension development. Lastly, the AST will be very framework-dependant, making portability of the tool less possible.

Token Parsing

A second method is the use of token parsing. By tokenizing the code we can recursively match brackets, revealing each codeblock for block-based programming languages that

use brackets to denote the start and end of each block (e.g. C, C#, C++, Java, Javascript, PHP). A major upside of this method is that it is much easier to find the specific block-type.

However, this method has its downsides as well. To include Off-Side Rule languages (e.g. Python, Coffeescript) codeblocks need to be parsed differently, because bracket-pair matching is not an option. We will also need language-specific token-parsing methods, as block-type definitions can differ across languages, e.g. functions have no defined return-type in Javascript while they do in Java. Many IDEs include token-grammars for the most popular programming languages, but some may not expose those grammars to their extension frameworks. These token-grammars may also differ from IDE to IDE, making this method less transferable between different IDEs. These downsides may be countered by using a standardized grammar, although there are few IDEs that try to adhere to such standards. Microsoft has developed a Language Server Protocol for its VS Code editor, citing it as a standard that can be used to develop parsing methods cross-IDE 3.5. It works as a middleman between a language server (running in their own process) and a code editor, providing tooling such as code completion, diagnostics, formatting, etc. This way, any LSP-compliant code editor can use these language servers and vice versa, greatly reducing the amount of work is needed to implement specific language features.

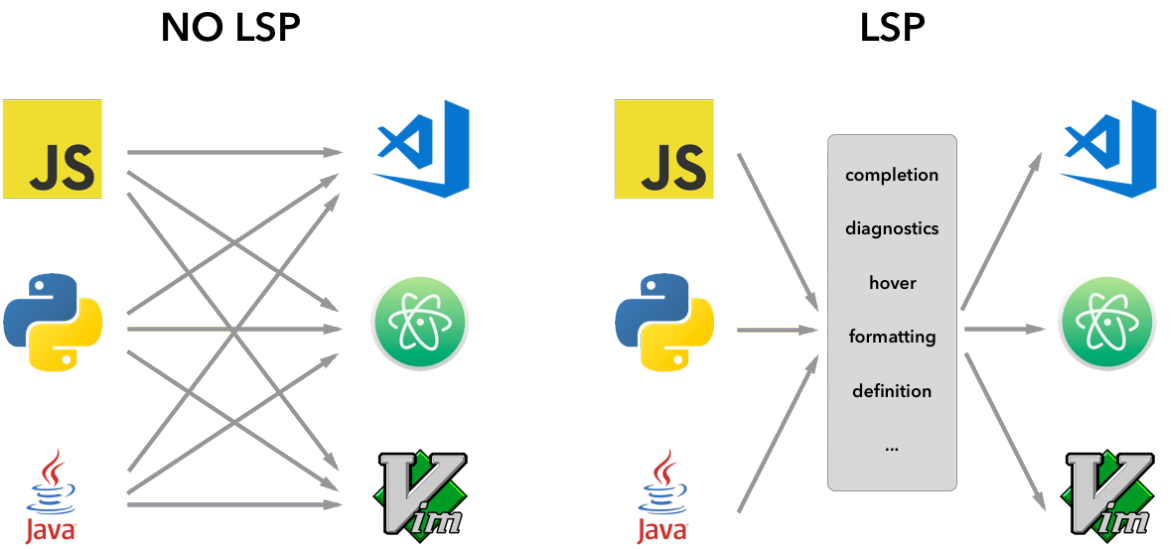

Figure 3.5: Instead of developing language tooling for each language in each editor, the LSP can greatly reduce the amount of work that is needed to implement specific language tools.

Indentation Parsing

As indentation is the accepted method for visualizing codeblocks, we may visualize the indentation in text-files. There are some upsides to this method: it is fast because it needs very little tokenization, it builds upon a known visualization technique that has been tried and tested in the community and may be easy to implement, it is a universal visualization technique across all block-based programming languages and, as it is purely character-based parsing, it will be supported by almost any IDE, making it easy to port between different environments.

However, there are some downsides to this method. Firstly, indentation is not always

correctly applied, especially with novice programmers. This can be remedied by turning on an auto-formatter within the IDE. Most text-based IDEs now include this functionality, many even provide options to auto-format 'on save' or 'on input'. Second, indentation contains no information about the type of block. This means we might be able to quickly find the position of the block without tokenizing the code, but still need those tokens to figure out the type of block. Third, this way we still save (part of) the visualization within the file.

IMPLEMENTATION

Because Visual Studio Code does not provide a public API for its tokenization engine, and only minimal access to its token parsing engine, we are by default limited to indentation parsing. In order to implement an indentation parser, we will need to assume that indentation for each line of code is correct. We can use the built-in auto-indentation to continuously format the code written by the user[4]. The auto-indenter works for multiple programming languages. Python support in VS Code is provided through a different extension, which includes indentation-rules specific to the Python programming language.

By utilizing the built-in auto-indenter, a program loop can be defined as in Figure 3.6. The extension waits for any user input in the IDE. When the user provides this input (code), the extension gets triggered. It will then execute three operations. First it will indent the code correctly according to the built-in indentation-rules. Then it parses that indentation and looks for any changes in the tree structure. Lastly, it updates the codeblock visualizations where necessary.

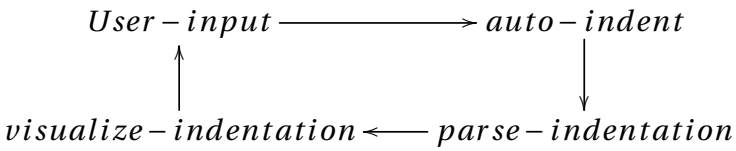

Figure 3.6: The program loop: A user provides input (code) to the IDE, after which the extension auto-indents this input. This triggers the indentation parser, which in turn triggers the indentation visualizer. Finally, we wait for new user input.

In order to trigger auto-indentation each time a user provides some user input, we configure VS Code in such a way that the document gets formatted each time the user types or saves. Snippet 3.1 shows how to activate these options for the currect workspace in VS Code. In order to watch for user input, we opted to use the built-in autosave feature, as shown in Snippet 3.2. Saving a document triggers an 'onWillSaveTextDocument'-event, which we can use to call our indentation-parser and -decorator function as seen in Snippet 3.3.

```
1 vscode.workspace.getConfiguration().update('editor.formatOnType', true, vscode.ConfigurationTarget.
      Workspace);
2 vscode.workspace.getConfiguration().update('editor.formatOnSave', true, vscode.ConfigurationTarget.
      Workspace);
```

Snippet 3.1: Activating auto-indentation (called 'auto-format') configuration options in VS Code

[4] https://code.visualstudio.com/

```
vscode.workspace.getConfiguration().update('files.autoSave', 'afterDelay', vscode.
    ConfigurationTarget.Workspace);
vscode.workspace.getConfiguration().update('files.autoSaveDelay', 500, vscode.ConfigurationTarget.
    Workspace);
```

Snippet 3.2: Activating auto-save configuration options in VS Code

```
vscode.workspace.onWillSaveTextDocument(event => {
    const openEditor = vscode.window.visibleTextEditors.filter(
        editor => editor.document.uri === event.document.uri
    )[0];
    decorateIndent(openEditor);
});
```

Snippet 3.3: When the 'onWillSaveTextDocument'-event is triggered, call the indentation function.

Parsing the indentation and providing some visualization method proved to be more difficult than turning some configuration options on or off. Because this is not a built-in feature of VS Code, it required a custom approach. We started by building a list of codeblock objects, where each block is defined by its range (start and end position in the document), type (e.g. class, function, ...) and indentation level. This is created using a simple class, such as in Snippet 3.4.

```
class CodeBlock {
    constructor(range: vscode.Range, type: CodeBlockType, indentLevel: number) {
        this.range = range;
        this.type = type;
        this.indentLevel = indentLevel;
    }
}
```

Snippet 3.4: A simple ES6 class datastructure, representing a codeblock by range, type and indentation level.

Creating a list for each codeblock requires us to parse each line of code for its indentation level. Indentation level shows us where a codeblock starts and ends, especially in Off-Side Rule languages such as python. We currently implemented an iterative approach. A recursive approach would also be possible, but as our priority is a working prototype, we did not explore this option further. Visualizing each codeblock can then be done by using the decoration API of VS Code. This API provides a stylesheet-like method to define how text in an editor is presented, e.g. Snippet 3.5.

```
backgroundColor: `rgba(${color}, ${opacity})`,
border: `2px solid rgba(${color}, ${Math.max(opacity * 2, 0.5)})`,
fontWeight: 'bold',
before: {
    backgroundColor: `rgba(${color}, ${Math.max(opacity * 2, 0.5)})`,
    width: '4px',
    height: '67%',
    contentText: ' ',
    margin: `0 0 0 0`
}
```

Snippet 3.5: The decoration API uses a stylesheet-like approach to visualizing code.

Using these methods, we have implemented every Blocker requirement:

- Distinguish different codeblocks: using the decoration API we can now visualize each codeblock separately from each other codeblock.
- Provide each codeblock with indentation-like padding to the left: classic indentation, using the auto-format configuration options in VS Code, allows us to maintain indentation as expected.
- Frequent updates of visualizations: VS Code provides us with options to format and visualize the codeblocks on user input and document saves.

We can immediately implement several other Must-have requirements as well:

- Live updates of visualizations: VS Code provides us with options to format and visualize the codeblocks on user input and document saves.
- maintain a contrast ratio of at least 3:1: Tweaking the colors used by in the decorator stylesheets and using a contrast checker tool[5], we can quickly adjust the color ratio of the codeblocks.
- provide each codeblock with a 'C' shaped visualization: We now know the start- and end-position of each codeblock. This provides us with enough information to visualize the start and end of each codeblock differently.

In order to implement codeblock categorization, we categorize codeblocks similarly to Pencil.cc:

- Control: these codeblocks are selective or iterative in nature. They consist of the blocks 'for', 'while', 'if', 'else', ...
- Functional: these codeblocks group statements together. They consist of the block 'method', 'constructor' and 'function'.

Each document in VS Code contains a Document Symbol Provider, which provides access to all named symbols in the document that can be reached by the 'go-to-symbol'-feature. Each symbol is provided with a SymbolKind, an enum value containing the type of symbol it represents. By iterating over the entire list of named symbols in the document, and comparing the range of each symbol to the range of each codeblock we collected earlier, we can see if a codeblock matches a specific symbol.

The Document Symbol Provides doesn't provide us with any information about codeblocks in our control-category. However, every codeblock from the functional-category is

[5] The contrast checker tool from WebAim, which is referred to in the WCAG documentation.

Symbolkind	codeblock category
Array	not a codeblock
Boolean	not a codeblock
Class	uncategorised
Constant	not a codeblock
Constructor	Functional
Enum	not a codeblock
EnumMember	not a codeblock
Event	not a codeblock
Field	not a codeblock
File	not a codeblock
Function	Functional
Interface	uncategorised
Key	not a codeblock
Method	Functional
Module	not a codeblock
Namespace	uncategorised
Null	not a codeblock
Number	not a codeblock
Object	not a codeblock
Operator	not a codeblock
Package	not a codeblock
Property	not a codeblock
String	not a codeblock
Struct	uncategorised
TypeParameter	not a codeblock
Variable	not a codeblock

Table 3.4: SymbolKind enum from the VS Code extension API vs. the codeblock categories we defined.

represented in the list of Symbols, as seen in table 3.4. There is even information about a third possible category - a structural or object-oriented category, containing classes, structs and namespaces. Every other codeblock that is not detected by the Document Symbol Provider can thus be categorized as a control-codeblock, as can be seen in example 3.7. This is the reason why we cannot use the Document Symbol Provider as a way to detect codeblocks, as none of the codeblocks in the control-category would get detected.

After applying all these methods we have created a tool that is capable of handling multiple languages that looks similar to the mockup we created in Figure 3.4. However, there are still some differences as can be seen in Figure 3.8. The 'C' shape is now a vertically mirrored 'L' shape for Python. The decoration API in VS Code only allows to decorate text, which is a problem for Python, as there is no text to mark the end of a codeblock.

```
switch (symbol.kind) {
    case vscode.SymbolKind.Interface:
    case vscode.SymbolKind.Object:
    case vscode.SymbolKind.Class:
        refineRange(symbol.range, CodeBlockType.class);
        break;
    case vscode.SymbolKind.Function:
    case vscode.SymbolKind.Method:
    case vscode.SymbolKind.Constructor:
        refineRange(symbol.range, CodeBlockType.function);
        break;
    default:
        break;
}
```

Figure 3.7: Categorizing each codeblock according to its symbolkind.

Figure 3.8: The prototype working in Python (left) and Javascript (right), with a similar visualization as the mockup in Figure 3.4

Hurdles and Limitations

The tool we present as a working prototype is stable for small programs on a novice-programmer level, but is not well-suited for more complex structures where the header of a block con-

sists of multiple lines of code (e.g. a complex condition spread out over multiple lines of code for a selection block as in Figure 3.9).

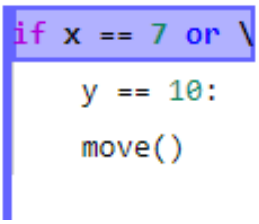

Figure 3.9: A bug in the prototype, where a multiline condition breaks the visualization partially.

The Document Symbol Provider is loaded asynchronously, and there is currently no event that signals when it is ready. This means that the extension will sometimes not recognize all blocks for the first few seconds (or before the first edit of the document), rendering them all in the same color as can be seen in Figure 3.10.

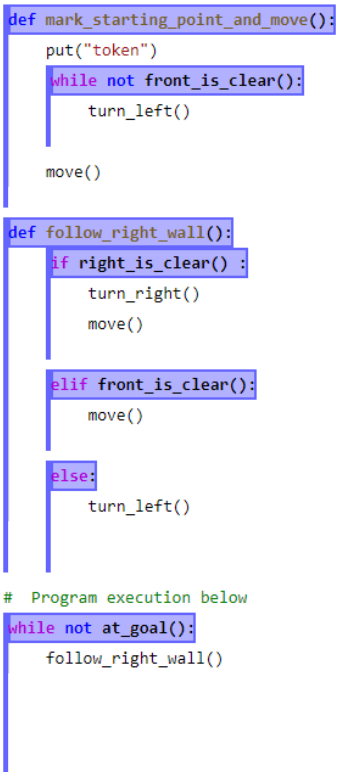

Figure 3.10: A bug in the prototype. On initialization, the extension will sometimes not recognize all blocks, rendering them all in the same color.

Both issues may be solved by using a more extensive public API or a custom build of the editor, along with more extensive visualization APIs. The Atom editor would be a very good choice to explore these possibilities, as its codebase is more public.

We do not consider these bugs as critical, as every Blocker requirement is still fully implemented. The only Must-have requirement that is not fully implemented is requirement 'provide each codeblock with a C-shaped visualization', where the first half of the C-shape is

fully functional in every language and the bottom half is missing in Off-side Rule languages, still providing ample difference between multiple codeblocks.

4

Experimental Evaluation

With a working prototype for the visualization tool we can construct an experiment to measure the efficacy of codeblock visualization as implemented by the tool. Similar to previous research in the domain of measuring the efficacy of visual IDEs and text-based IDEs, we will use an experimental research approach [35, 1, 36]. In the following sections we explain this experimental research approach in more detail:

- 4.1 Experimental Research Approach, where we explain the methodology used in the experiment and a detailed explanation of the curriculum used during the experiment.
- 4.2 Data Collection where we elaborate on the participants and the testing methods used to collect the results.
- 4.3 Data Analysis where we explain which statistical methods were used to analyze the results and our reasoning behind those choices.
- 4.4 Results of the Experiment where we show and interpret the result of the statistical analysis that was conducted.

4.1. Experimental Research Approach

Previous research from Weintrop and Wilensky and Alrubaye has used a quasi-experimental setup with two separate class groups that follow the same introductory programming course [1, 37, 35]. These studies have followed the first few weeks of the course, meaning that most students have little to no background in programming. Each group is presented with a different tool for writing code (e.g. a block-based IDE vs a text-based IDE). On the first day of the course, students are presented with a content assessment.

The content assessments in the studies of both Weintrop and Alrubaye were based on the Commutative Assessment [35], a multiple-choice test that contains thirty questions

which cover the concepts that students will encounter during the introductory programming course. The study of Weintrop used the questionnaire directly [37], while the study of Alrubaye used a similar questionnaire based on the Commutative Assessment with fewer questions [1]. Questions in the Commutative Assessment are either content, algorithm or comprehension questions. Content and comprehension are each setup in a similar fashion: it shows a short program, followed by five multiple-choice answers and asks the question: "What will be the output of the program?" (for content questions) or "What does this program do?" (for comprehension questions). Algorithm questions are different, as they are written in plain text and ask students to identify the order of steps in an algorithm or identify potential missing steps.

The study of Weintrop presented these questions in multiple formfactors, from block-based to text-based or hybrid presentations. This allows students to 'read' code in a way that is most comfortable for them. The study by Alrubaye approached this differently: as its goal was to measure the performance of different learning tools in order to start programming using a text-based IDE, the questions are presented in a text-based fashion.

We will also use an experimental research approach, similar to Weintrop and Wilensky and Alrubaye [1, 36]. We use this to measure the difference in performance between students using visual elements to indicate codeblocks and students using a traditional text-based IDE. This is similar to how Alrubaye conducted an experiment to measure the difference in performance between students using a hybrid environment and a block-based IDE. Because we want to see how visual elements contribute to the learning performance of students when using a text-based IDE, where the goal is to read and write code in a text-based IDE, we opt to present students with code in a text-based fashion. Code-snippets are created using the same IDE as the one students will work with.

4.1.1. Course of the Experiment

The experiment consists of multiple phases: a grouping phase, a learning phase and a testing phase. It is important to account for the time it takes to perform each phase, especially the learning phase. The three phases of the experiment are based on research performed by Weintrop [36] and Alrubaye [1].

Grouping Phase

During the grouping phase, students were divided into one of two groups: the text-based group and the block-visualization group. Each student was presented with the correct software installation. During this phase, we also conducted a content assessment in order to measure prior knowledge. This content assessment is based on the Commutative Assessment [35], a multiple choice test containing thirty questions that cover the concepts that students will encounter during the learning phase. The test was shortened (due to time constraints) to nine questions that cover the same topics.

Learning Phase

In the learning phase, the students followed a curriculum explaining all basic concepts that are tested during the testing phase. Each concept was explained with both visual and text- or number-based activities. This phase had a five-week duration, where students had two hours of in-class lessons and the rest of the week to practice the subjects that were taught during class, with a possibility to ask questions over the Learning Management System (LMS).

Testing Phase

During the testing phase, students will again perform a similar content assessment. We used a questionnaire that was used in the research by Alrubaye, which in turn is based on the commutative assessment by Weintrop [1, 35], but translated it into Dutch. Framework- and language-specific questions were modified in order to match the curriculum, although we tried to match the questions as closely as possible. It contains a mix of content, algorithm and comprehension questions that each covers one or more specific topics.

4.1.2. Curriculum

The curriculum was divided into five lesson-plans, where each lesson contained one or more concepts from the curriculum. As the lessons were being taught in Dutch, the lesson plans were also drawn up in Dutch. A new concept was introduced each week and previous concepts were repeated within the context of the new concept.

Each lesson plan contained a similar structure:

- Data about the teacher, school, class (group) and time/location of the lesson
- General information about the lesson, including subject, main goal, initial situation, connection to government-mandated learning goals and necessary materials (Figure 4.1).
- A step-by-step guide that determines how a subject is taught, including the specified content of the subject content, guidelines on what to say and do and a time limit for each piece of content (Figure 4.2).

A major part of the first lesson was devoted towards installing the software and tools necessary in order to write code and run Python programs. The following four weeks (and the second part of the first lesson) were used to teach the multiple concepts as described in the Curriculum.

In order to measure code comprehension among students, it is important to define exactly what it is we will measure. For the purpose of this research, we define code comprehension by following the commutative assessment method by Weintrop [35], which in turn is based on the 2013 CS Curriculum [26] and the work of Tew and Guzdial [30, 31]. In

lesonderwerp	Leren programmeren in Python
hoofddoelstelling	De leerlingen leren de basisbeginselen van progammeren in python
didactische beginsituatie	materieel: leerlingen hebben eigen laptop leerlinggebonden: Leerlingen brengen steeds laptop mee leerstofgebonden: Geen
situering in het leerplan	koepel/net: KathOndVla leerplancode: D/2011/7841/045 Deelcompetentie 11.1 - Inzien wat een algoritme is. Het verschil tussen een algoritme en een programma kennen Deelcompetentie 11.2 - Een probleemstelling omzetten in een werkend programma. De verschillende controlestructuren kennen en gebruiken
leerboek en/of benodigd materiaal	Geen leerboek Laptop + software (visual studio code & python interpreter)

Figure 4.1: The general information of the lesson plan for the first week.

leerinhoud	lesontwikkeling	organisatie / media / tijd
Intro		
	"Dag iedereen. Zoals jullie voor de vakantie hebben vernomen van meneer Laveren en meneer Van Battel, ga ik jullie de komende 5 weken leren programmeren. Zoals eerder ook uitgelegd door meneer Van Battel doen we voor en na de 5 weken een kleine meting om te bekijken hoe goed jullie vooruit gaan in het interpreteren van programmacode. Daarom starten we vandaag met de zogeheten nulmeting, om te bekijken wat jullie voorkennis is. Jullie krijgen zo dadelijk allemaal een link naar een test, probeer die allemaal zo goed mogelijk in te vullen, jullie krijgen daar 20 minuten de tijd voor." Nog niet iedereen heeft toestemming gegeven, zij die nog geen toestemming hebben gegeven, gelieve dat zeker in orde te brengen.	25 minuten
Wat heb je allemaal nodig om in python te kunnen programmeren?		
*Leerlingen **begrijpen** dat ze bepaalde software en een code interpreter moeten installeren om te kunnen programmeren ➔ **TOEPASSEN***		
	Vooraleer we kunnen beginnen met python code schrijven moeten we ervoor zorgen dat jullie de nodige software hebben om te kunnen programmeren. Ga naar http://www.piustien.net/python daar vinden jullie het installatiebestand voor Visual Studio Code, dat is de software waarin we de code gaan schrijven. Download deze en installeer dit programma op je computer. Nadat Visual Studio Code geïnstalleerd is, installeer je de python interpreter van dezelfde locatie, dat is software die ervoor zorgt dat je computer python code kan interpreteren, daarom dus een interpreter. Als dat klaar is start je Visual Studio Code op en klik je op de link om de Python plugin voor Visual Studio Code te installeren.	25 minuten
Wat is Python?		
*Leerlingen **begrijpen** dat Python een programmeertaal is en kunnen analyseren voor welke situaties python een geschikte programmeertaal is*		
	Python is een programmeertaal ontwikkeld in de jaren 90 in Amsterdam. De naam is gekozen owv de favoriete komische televisieprogramma van de uitvinder, Monty Python's Flying Circus.	15 minuten

Figure 4.2: Specific step-by-step guide of the lesson plan for the first week.

making the FCS1 assessment, Tew and Guzdial reviewed the contents of 12 introductory computer science textbooks along with other published curricula to establish a list of ten core CS1 concepts. Weintrop selected the five primary core concepts of this curriculum and included an additional two categories, based on his review of the CS2013 Curriculum. This adds up to a total of seven concepts which can be used to measure and define code comprehension, by presenting a student with practical questions and exercises. These seven concepts are:

1. programming fundamentals: How can we direct a computer to save and adjust values using a programming language?

2. selection statements: How can we use conditional logic to adjust the outcome of a program?

3. definite loops: How can we use the previous concepts (programming fundamentals and conditional logic) to execute code a specific number of times?

4. indefinite loops: How can we use the previous concepts (programming fundamentals and conditional logic) to execute code an indefinite number of times?

5. function parameters: How can we break down code into smaller, more manageable chunks?

6. program comprehension: How can I predict the outcome of a program?

7. algorithms: How can we break down a problem into smaller steps by using natural language?

In the following subsections each of these concepts is explained in more detail.

PROGRAMMING FUNDAMENTALS

How can we direct a computer to save and adjust values using a programming language?

During the first week, students get taught what variables, datatypes and values are and apply these concepts in practical applications. This is done by first explaining these concepts to students, using analogies to lessen the abstract nature of these concepts and asking questions to determine what current knowledge students have. This current knowledge can come from many different fields of study, and can help students more accurately understand abstract concepts through the use of different analogies.

After explaining what variables, datatypes and values are, students are shown a practical application of these concepts. First within the concept of a larger program (a game of 'hangman'), where students are show what effect these concepts have on the program. Later they practice these concepts one by one with isolated examples to guide them, so students can focus on these new concepts without the distraction of untaught concepts.

Finally, students get the chance to practice these concepts with a series of exercises. A selection of good and bad solutions are used to explain good and bad practices and common misconceptions. Here, students learn the importance of statement order and how it can affect how (or if) a program functions.

The second week goes more into the concepts of arithmetic expressions and operations. This lesson starts with a repetition of the previous concepts (variables, datatypes and values) and explains the arithmetic operators that can be applied to those values. Each operator is explained within the context of specific datatypes and linked to the current knowledge of students on arithmetic. The same lesson structure was used for explaining through analogies, teaching by example and learning through exercises.

In week four, we elaborate further on different ways to save multiple values using lists, dictionaries, tuples and sets.

SELECTION STATEMENTS

How can we use conditional logic to adjust the outcome of a program?

During the second week, after the students learn about arithmetic expressions and operations, they learn about logical expressions and operations as well. These concepts get introduced along with the larger concept of selection statements during the second week. Selection statements are explained with the three selection codeblocks in Python:

- if
- elif
- else

Each block is explained through multiple examples and exercises that apply the new knowledge. Within these blocks, the concepts of programming fundamentals are repeated. Next, combinations of these blocks are shown through even more examples, along with an explanation of the logical consequences that block choice and conditional order can have on these structures. Students also practice the nesting of similar codeblocks.

This is the first codeblock students get to experience. It is the first time they will apply logic to their programs and alter the flow of the program based on the initial input. It is also the first time our research tool gets activated for the group that has installed it.

DEFINITE LOOPS

How can we use the previous concepts of programming fundamentals and conditional logic to execute code a specific number of times?

During the third week, the concept of loops is taught to the students. Using the 'while' codeblock we show students how they can use conditional logic to determine how often a 'while' codeblock will iterate over the statements contained within the block. Then the concept of a loop counter is shown, where a variable is created to count the number of iterations.

After this, we introduce the concept of iterable variables by looking more closely at the string datatype. Together with the students, we reason how a variable of this datatype can be broken down into smaller bits (each character). We use this as a gateway to talk about iterator variables: variables that store a portion of the iterable object during the execution of a loop. In case of a 'string' variable, the iterator stores a character.

Armed with the knowledge of loops, conditional logic, iterables and iterators, the students are introduced to the 'for' codeblock, which is, in Python, a codeblock that specifically defines an iterator over an iterable object.

INDEFINITE LOOPS

How can we use the previous concepts of programming fundamentals and conditional logic to execute code an indefinite number of times?

During the fourth week, students are shown how different codeblocks can be nested. Along with the students, we create a program that repeatedly captures user input through the 'input' function, saves that input into a list, and only breaks from that loop if the user enters a specific command. This shows students how a loop does not always have a predetermined number of iterations, and can in fact be indefinite.

FUNCTIONS AND PARAMETERS

How can we break down code into smaller, more manageable chunks?

Throughout the complete course, we put each example in a clearly named function. We first introduce this as an easy way to name each exercise and keep our code a bit more structured in multiple codeblocks. Later on, we show how we can call those previous exercises by simply calling their name, just so we don't have to write previous code over and over again. By the end of the course, students are using functions as a practical way to structure their code and make their code more readable.

From the start of the course we use the 'input' function to capture user input, the 'output' function to show information on screen, and the 'int' function to selectively cast user input (which is character-based) to integers. Each of these functions accepts at least one argument.

By week four we start to combine the two concepts, where our 'exercises' start to accept one or more arguments. Students are taught how this increases the reusability of the codeblocks that we now call 'functions'.

PROGRAM COMPREHENSION

How can I predict the outcome of a program?

Students are taught through the use of examples. Bigger examples are analyzed with the whole group, smaller examples as a way to guide them through similar exercises. Through this method, we teach the students how to read code and predict the output of a program based on possible inputs.

ALGORITHMS

How can we break down a problem into smaller steps by using natural language?

For each programming concept we create a specific 'human-readable' problem statement that can be solved through the use of a student's current programming knowledge.

Students first try to solve these problems on their own. Often this gives us multiple possible answers to the same problem. Students then solve the problem in groups, step by step, breaking the problem down to multiple smaller problems until we can concretely translate a given problem statement into a specific programming statement. We guide them through the process, until we come to a complete solution. We do this multiple times, in order to cover multiple possible solutions to the problem.

4.2. Data Collection

4.2.1. Student Participants

We approached a school in Antwerp (Belgium) to conduct the experiment. Within the school, there are several class groups which have an algorithmic thinking and programming part in their curriculum. class groups consist of students from multiple genders and ethnic and social backgrounds. Most students originate from the surrounding urban areas. These class groups can be divided into three experience categories, based on the amount of programming each student was offered during their curriculum prior to the experiment:

1. no prior programming experience
2. little (0.5 years) prior programming experience
3. some (1.5 years) prior programming experience

These three class groups amount to a total sum of 28 students, of which 1 student could not complete the experiment and was therefore not included in the results. Every student was informed about the experiment beforehand in two different ways. First, we provided a verbal info session where students could freely ask questions about the experiment, its results and data handling. Second, we provided students and their parents with a letter containing a detailed explanation about the experiment. The letter was digitally sent through the schools Learning Management System. This way, we could actively track which parents and students did or did not read the letter. The digital letter included a consent-form in compliance with the ethics procedure from the Open University. Using these consent-forms, we collected consent for each student, either from their parents or directly from the students themselves[1].

4.2.2. Pre-Test

Students were assembled in a controlled and familiar classroom environment. Each class group was presented with an introduction to the course, where both the goals of the course and the goals of the experiment were clearly laid out. Students were informed on both accounts beforehand, which allowed this introduction to act more as a reminder.

[1] students 16 years of age or older are allowed to give consent themselves

After the introduction, students are presented with a digital form that contains the questions from the first content assessment. The test is limited in size, and will allow us to establish a baseline for a student's comprehension of the different subjects.

After the pre-test, each of the three class groups was further divided into a control-group that will use the default visualization of the IDE and a group that will use the visualization extension on top of the IDE. Due to Covid-19 restrictions, class groups were already divided up into two, randomly distributed, equally sized groups. We simply used these Covid-19 groups as either the control-group or the visualization-group.

In order to make sure that the results of each group are separated clearly, a copy of the digital questionnaire is made for each group. In order to access the questionnaire, students have to login using the same credentials that are used to access the schools Learning Management System (Smartschool/Office365). This way we can verify each result to originate from a specific student.

4.2.3. Post-Test

Conducting the post-test was done in a very similar fashion to the pre-test. Students were assembled in the same classroom. We explain that the test will be similar to the one they filled in at the beginning of the course, and that every question will cover one or more topics from the past five weeks.

The same digital questionnaire tool was used to collect the results, where students have to login using the same credentials that are used to access the schools Learning Management System (Smartschool/Office365). This way we can verify each result to originate from a specific student. Identical to the pre-test, a copy of the digital questionnaire is made for each group.

4.3. Data Analysis

We used a Shapiro-Wilk normality test to check the normal distribution of our datasets. Where these datasets showed such a normal distribution, we made use of parametric tests to further analyze the data.

We did two regression analyses to check for a correlation between programming experience and post-test scores on the one hand, and between programming experience and student progress on the other.

A two-way Anova test was performed between post-test scores, amount of programming experience and the use of the tool. This was done to see if our tool has any effect on the final score of the students.

A second two-way Anova test was performed between student progress, amount of programming experience and the use of the tool. This was done to see if our tool has any effect

on the progression score of the students.

4.4. Results of the Experiment

We conducted an experimental study to measure the difference in performance between students using visual elements to indicate codeblocks within a text-based IDE and students using a traditional text-based IDE. We hypothesized that code comprehension among students is impacted by the use of codeblock visualizations within a text-based editor.

Students were presented with two tests: one at the start of the course and one at the end. This way we collected two different datasets. The first dataset shows how each student performed on the test at the end of the course, while the second dataset shows how much progress each student made between the first and second dataset. Statistical analysis was applied to both datasets, allowing us to answer our hypotheses.

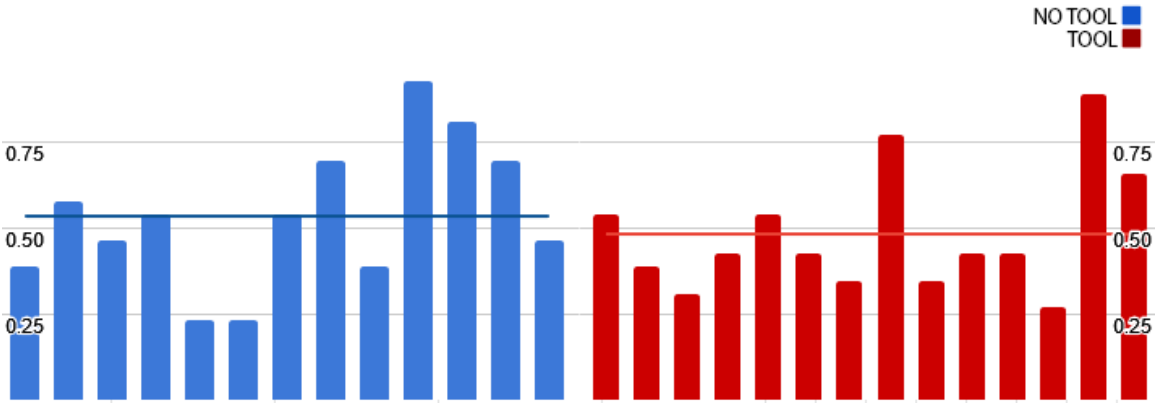

Figure 4.3: Post-test scores for students. The group that did not use the tool is shown in blue (left), the group that did use the tool is shown in red (right). The score is normalized to a scale of 0 - 1. The lines show the average score for each group.

In Figure 4.3 we show the normalized result of each student. Figure 4.4 shows the progress of each student between the initial pre-test that was taken during the grouping phase and the final post-test that was taken during the testing phase. We did a Shapiro-Wilk normality test to check the normal distribution of these datasets. Both datasets show a normal distribution ($W = 0.94076$, p-value = 0.1272 and $W = 0.97993$, p-value = 0.8607), allowing us to make use of parametric tests to further analyze the data.

There were 3 different student groups that participated in the experiment. Each of these groups had a different level of programming experience (no experience, 0.5 years and 1.5 years). Each group was divided into a control-group that used the default visualization of the IDE and a group that uses the visualization extension on top of the IDE. We did two regression analyses to check for a correlation between programming experience and post-test scores on the one hand, and between programming experience and student progress on the other. As expected, students with more programming experience have a significantly higher score on the post-test (p-value of 0.019), as can be seen in Figure 4.5. However, there is no significant effect on student progress (p-value of 0.698).

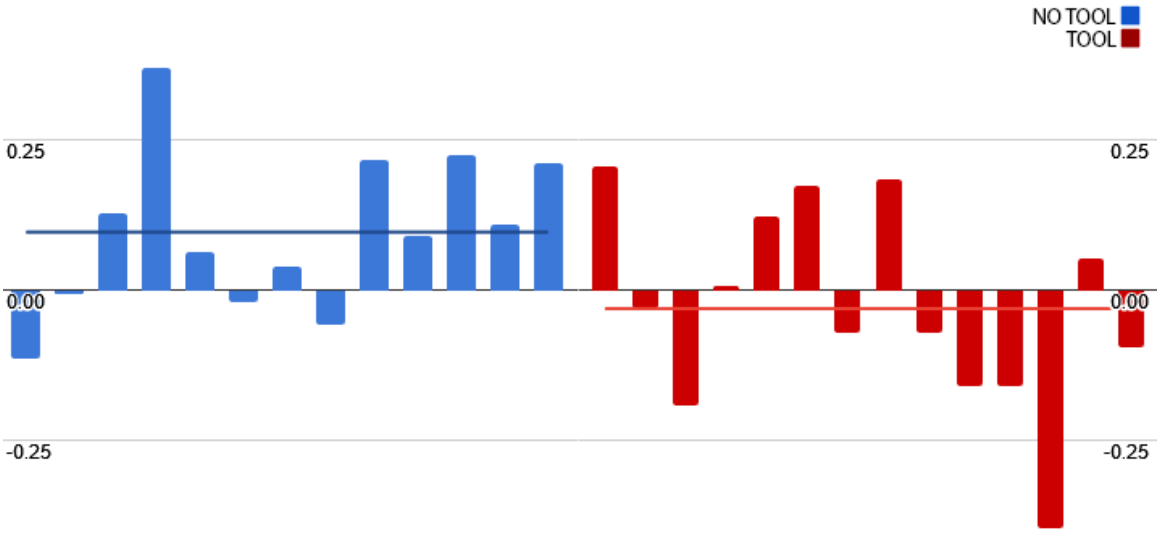

Figure 4.4: Relative scores, showing how much students improved their result between the pre-test and the post-test. The group that did not use the tool is shown in blue (left), the group that did use the tool is shown in red (right). The lines show the average progress for each group.

A two-way Anova test was performed between post-test scores, amount of programming experience and the use of the tool. Although students with more experience had a higher score on the post-test, there was no significant difference between using the tool or not on the average result of each group (p-value of 0.52311). We did not observe an influence of our tool on a students' final score, as seen in Figure 4.3.

Another two-way Anova test was performed between student progress, amount of programming experience and the use of the tool. Here, we observed a significant negative influence of our tool (p-value of 0.04099). Students that did use the tool had a worse progress score than those who did. This can be clearly seen in Figure 4.4.

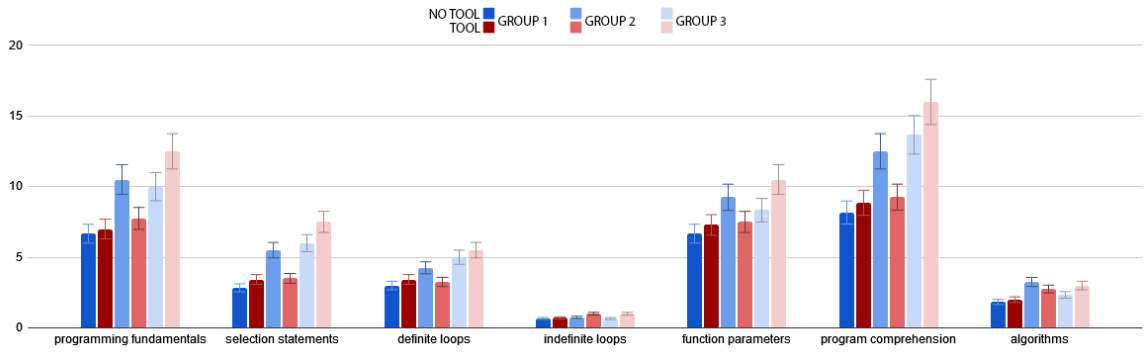

Figure 4.5: Average scores on the post-test for each group, broken down into categories. Blue groups did not use the tool, while red groups did use the tool. Lighter bars have more programming experience, while darker bars have less experience.

In short, we observed no influence of our tool on a students' final score. We did observe a significant negative influence of our tool on student progress. This means that students

that did use the tool had a worse progress score than those who did not. This rejects our null hypothesis. It also confirms our alternative hypothesis, although not in the way we expected.

5

Discussion

Research by Weintrop and Wilensky has shown that Visual programming languages can be used to increase code comprehension among novice high school students [36]. Their results show an inverse effect when compared to our results, as students that use a visual IDE performed better than students that use a text-based IDE, most significantly on code comprehension questions. An explanation for this could be due to limitations in respect to the tools and programming language used. It should be noted that both the visual IDE and the text-based IDE have multiple features not implemented, which can skew the results. One of those shortcomings is a limited form of syntax-highlighting and autoformat, and no autocomplete within the text-based IDE. The visual IDE had a clunkier user interface when compared to other visual IDEs such as Scratch or Blockly.

5.1. An Unexpected Result

Research by Alrubaye et al. has shown that hybrid forms of programming, where students can switch between block-based and text-based versions of their code, show more promising results than in pure visual IDEs [1]. These results are more significant than the previous research by Weintrop and Wilensky [36]. It shows that text-based IDEs have significant advantages as well. At first glance, this seems to directly contradict our results.

However, combined with our results, a possible conclusion may be that visual IDEs provide other advantages that are more significant than the visualization of code in a block-like fashion. These other advantages may include the listing of possible statements and codeblocks, easy access to code snippets for these codeblocks, the clear organization of such statements and codeblocks into categories or the drag and drop interface.

It is possible as well that some of the decisions we made during development had a more negative impact than anticipated. Many decisions were made during the development of the tool and the design of the experiment that could have an effect on the outcome of the experiment. The experiment was conducted using a specific programming

language and IDE, while the tools' design still had a lot of variables such as variations on color scheme or shape topology.

5.1.1. Choice of Programming Language

Python is an off-side rule language, meaning that indentation dictates code structure. This allowed our visualizations to directly indicate errors in code structure, instead of merely suggesting them. However, this means that Python has no clear end-of-codeblock indicator. This broke our visualization partially, because we had no bottom bar for the 'C'-shape. Students that did not use the tool were forced to focus on indentation early on because of this, possibly lessening the effect of the codeblock visualizations.

In the end, we did pick Python for some very obvious reasons:

- Its syntax is simple, consistent and readable for the fundamentals
- Its learning curve is beginner-friendly for the fundamentals
- Its setup is very straightforward

The idea that Python should be simple, consistent and readable is found directly within its documentation, specifically a document called 'the Zen of Python'. This document lists nineteen principles that are meant to guide developers when writing Python[1]. It includes maxims such as (1) beautiful is better than ugly, (2) sparse is better than dense and (3) if the implementation is hard to explain, it's a bad idea. In future work we propose that other languages such as Java, C# or Javascript should be considered as they may alliviate some of the downsides of using an off-side rule language.

5.1.2. Choice of IDE

Several factors were included when deciding which IDE should be used during the experiment. Because the experiment was done in a real-life environment, many factors were included that had nothing to do with the initial research question (e.g. cost, installation size, language support). Because we had to include these unrelated factors, it may be possible we did not use the optimal IDE as our primary choice for the experiment. Other IDEs that were not considered might provide a more elaborate visualization API.

5.1.3. Color Scheme Variations

Color theory is an entire research field of its own. It explains how colors are perceived, how colors can mix, match or contrast with each other and, maybe most importantly, what messages colors communicate. In our research we focussed on a very small part of this:

[1] The Zen of Python

contrast. Our current color scheme focussed entirely on readability in accordance to the WCAG2 standard.

Expanding our color scheme with slight variations for similar, but not identical, types of codeblocks may have further improved readability. By not doing this we may even have decreased the readability, because similar types of codeblocks now look visually identical (e.g. an 'if' block and 'while' block are now visually the same type of codeblock).

5.1.4. Shape Topology Variations

As we mentioned above, the use of an off-side rule language, where codeblocks have no clear end-of-block indicator, made it technically a lot more difficult to implement a complete 'C'-shape. We decided that we had a compelling visualization of each codeblock, even without the bottom bar for the 'C'-shape. However, this is a clear deviation from each of the researched visual IDEs.

Within visual IDEs there is a clear difference in the left-side padding of the 'C'-shape, where Blockly even allows for different sizes for different implementations. This is similar to how some programmers prefer 2-space indentation versus 4-space indentation. However, research has shown that the amount of spaces does not contribute to the readability of code, so it is unlikely that this was a contributing factor for the negative effect of the tool [6].

Line-height was also increased for students that used the tool, in order to create more space for the top and bottom visualizations of the 'C'-shape. This inherently meant that students that did use the tool had larger line spacings between statements. This is an indirect effect of the implementation of the tool itself, not the concept of codeblock visualization. However, previous research on text comprehension and readability has shown that larger line-heights should increase comprehension and have little effect on readability [25]. If anything, this should have increased code comprehension for students that did use the tool. However, further research is needed to see if the results of Rello et al. are applicable to reading code as well.

5.2. Threats to Validity

5.2.1. Internal Validity

The experiment was conducted in the middle of the Covid-19 crisis. As a consequence, a ventilation mandate was active, where doors and windows had to be open at all times. With most days averaging on a maximum temperature of 18°C or lower[2], weather conditions might have had an influence on the concentration of students. Because the doors of every classroom were open throughout the school, there was a lot of noise coming into the classroom which may have distracted participants. Students were also obligated to wear masks.

[2] Meteoblue - weather archive Belgium

Although students did not complain about discomfort during the experiment specifically, this might have impacted performance.

The tests were conducted during a normal school day, where students also went to different classes before and after the test. Other tests from other classes were sometimes planned on the same day, which may have impacted student mood and concentration.

In the first test, an error was found in one of the code snippets. This meant that none of the multiple choice answers were completely correct, although one answer was partially correct.

It is a possibility that students did not understand the translated questions well. We did involve several teachers to read the questions and evaluate them on clarity, but due to the subject of the course and the limited amount of CS teachers present in the school, there were few adjustments made to the wording of the questions.

5.2.2. External Validity

The number of students that were part of the experiment was small, especially compared to the research done by Weintrop in [36]. All of the participants were high-school students that were taught by the same teacher. This means that the limited results from this experiment are hard to transfer to other settings, such as the effectiveness of codeblock visualizations on experienced programmers. This limits the generalizability of the results.

5.2.3. Statistical Conclusion Validity

Although our results are statistically valid for our test subjects, we cannot extrapolate these results towards a larger population. It is entirely possible that our small sample size contains an accidental offset that have skewed the results. The data shown in Figure 4.5 shows some hints towards such an offset, as one group performed differently from the two others. A larger sample size would be able to give a clearer image for the larger population. Future work should especially focus on the results where we did not find any significance.

6

CONCLUSION

The goal of this thesis is to find out if codeblock visualizations within a text-based IDE can increase code comprehension among high school students, similar to the results of using visual IDEs. To that end we have analyzed multiple, popular visual IDEs in order to create a tool that provides codeblock visualizations within a text-based IDE. We then used the tool in a quasi-experimental study where high school students were randomly divided into two groups. One group was provided with a popular text-based IDE that did not use codeblock visualizations, while the other was provided with the same text-based IDE, but this time enhanced with codeblock visualizations. Both groups followed the same curriculum during a period of five weeks. Both groups were given the same test at the start and the end of the five-week period. We find that students that did use the tool performed similarly to the ones that did not on their final test. However, students that did not use the tool, progressed more between the initial and final test.

There are multiple possibilities for future work. First and foremost, our findings should be replicated on a larger scale. Secondly, we showed just one way in which codeblocks can be visualized. Different programming languages, color schemes, shape topologies, in- or exclusion of syntax highlighting can all have impacted these results. Thirdly, there are more differences between visual IDEs and text-based IDEs than just the visualizations, although these are the most straightforward. For example, the drag-and-drop interface for different programming structures that also provides students with an easy lookup system for different codeblocks, operators and statements.

7

ACKNOWLEDGEMENTS

Writing this thesis was not easy, and I have received a great deal of help and support from a number of people.

First and foremost, I would like to thank my supervisors, professors Alaaeddin Swidan and Fenia Aivaloglou, for their support and patience. Alaaeddin, Without your guidance and understanding I would not have made it this far. Fenia, your enthusiasm and feedback at the end were invaluable.

I would like to thank my colleagues and principals at Pius X highschool for their help and support. This would not have been possible without your help.

I would also like to thank my fellow students, especially Steven, Bob and Dennis, who I shared many moments of joy and suffering with throughout my Masters.

In addition, I would like to thank my partner, Anemoon, and my family for their endless patience and understanding. Anemoon, you have been my bedrock throughout the past three years. This work is as much an accomplishment by you.

Special thanks to my sister, Evi, for being my personal spell checker.

Finally, this would not have been possible without the help of my friends, the Uncles, who not only gave me a listening ear whenever I needed it, but many happy distractions as well.

BIBLIOGRAPHY

[1] H. Alrubaye, S. Ludi, and M. W. Mkaouer. Comparison of block-based and hybrid-based environments in transferring programming skills to text-based environments. in Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, 2019. 2, 3, 5, 6, 11, 31, 32, 33, 43

[2] M. Anderson, R. Motta, S. Chandrasekar, and M. Stokes. Proposal for a standard default color space for the internet—srgb. In Color and imaging conference, 4th Color and Imaging Conference Final Program and Proceedings, pages 238–245. Society for Imaging Science and Technology, 1996. 16

[3] A. Arditi. Monocular and binocular letter contrast sensitivity and letter acuity in a diverse ophthalmologic practice. Supplement to Optometry and Vision Science, 81:287, 2004. 16

[4] D. Asenov, O. Hilliges, and P. Müller. The effect of richer visualizations on code comprehension. In Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems, pages 5040–5045, 2016. 6

[5] D. Bau and D. A. Bau. A preview of pencil code: A tool for developing mastery of programming. In Proceedings of the 2nd Workshop on Programming for Mobile & Touch, pages 21–24, 2014. 12

[6] J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel. Indentation: simply a matter of style or support for program comprehension? In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 154–164. IEEE, 2019. 2, 14, 45

[7] T. Beelders and J.-P. du Plessis. The influence of syntax highlighting on scanning and reading behaviour for source code. In Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists, pages 1–10, 2016. 7

[8] T. R. Beelders and J.-P. L. du Plessis. Syntax highlighting as an influencing factor when reading and comprehending source code. Journal of Eye Movement Research, 9(1), 2016. 7

[9] B. Caldwell, M. Cooper, L. G. Reid, G. Vanderheiden, W. Chisholm, J. Slatin, and J. White. Web content accessibility guidelines (wcag) 2.0. WWW Consortium (W3C), 290, 2008. 16

[10] N. Fraser. Ten things we've learned from blockly. In 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), pages 49–50. IEEE, 2015. 13

[11] L. M. Gadhikar, L. Mohan, M. Chaudhari, P. Sawant, and Y. Bhusara. Browser based ide to code in the cloud. In New Paradigms in Internet Computing, pages 59–69. Springer, 2013. 5

[12] N. S. Gittings and J. L. Fozard. Age related changes in visual acuity. Experimental gerontology, 21(4-5):423–433, 1986. 16

[13] C. Hannebauer, M. Hesenius, and V. Gruhn. Does syntax highlighting help programming novices? Empirical Software Engineering, 23(5):2795–2828, 2018. 7

[14] J. Jenkins, E. Brannock, and S. Dekhane. Javawide: innovation in an online ide: tutorial presentation. Journal of Computing Sciences in Colleges, 25(5):102–104, 2010. 5

[15] K. M. Kahn, R. Megasari, E. Piantari, and E. Junaeti. Ai programming by children using snap! block programming in a developing country. 2018. 12

[16] R. B. Kozma. Comparative analysis of policies for ict in education. In International handbook of information technology in primary and secondary education, pages 1083–1096. Springer, 2008. 5

[17] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. ACM Transactions on Computing Education (TOCE), 10(4):1–15, 2010. 12

[18] A. Marron, G. Weiss, and G. Wiener. A decentralized approach for programming interactive applications with javascript and blockly. In Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions, pages 59–70. Association for Computing Machinery, 2012. 12

[19] J. Moonen. Evolution of it and related educational policies in international organizations. In International handbook of information technology in primary and secondary education, pages 1071–1081. Springer, 2008. 5

[20] F. Nogatz, P. Körner, and S. Krings. Prolog coding guidelines: Status and tool support. EPTCS 306, pages 8–21, 2019. 2

[21] B. J. Oates. Researching information systems and computing. Sage, 2005. 12

[22] I. Ouahbi, F. Kaddari, H. Darhmaoui, A. Elachqar, and S. Lahmine. Learning basic programming concepts by creating games with scratch programming environment. Procedia-Social and Behavioral Sciences, 191:1479–1482, 2015. 12

[23] E. Pasternak, R. Fenichel, and A. N. Marshall. Tips for creating a block language with blockly. In 2017 IEEE Blocks and Beyond Workshop (B&B), pages 21–24. IEEE, 2017. 15

[24] M. Patrignani. Why should anyone use colours? or, syntax highlighting beyond code snippets. CoRR, abs/2001.11334, 2020. 7

[25] L. Rello, M. Pielot, and M.-C. Marcos. Make it big! the effect of font size and line spacing on online readability. In Proceedings of the 2016 CHI conference on Human Factors in Computing Systems, pages 3637–3648, 2016. 45

[26] M. Sahami and S. Roach. Acm/ieee-cs joint task force on computing curricula (2013). Computer science curricula, 2013. 6, 33

[27] M. Seraj, E.-S. Katterfeldt, K. Bub, S. Autexier, and R. Drechsler. Scratch and google blockly: How girls' programming skills and attitudes are influenced. In Proceedings of the 19th Koli Calling International Conference on Computing Education Research, pages 1–10, 2019. 12

[28] D. Spinellis. Code reading: the open source perspective. Addison-Wesley Professional, 2003. 2

[29] R. Tapp and R. Kazman. Determining the usefulness of colour and fonts in a programming task. In Proceedings 1994 IEEE 3rd Workshop on Program Comprehension-WPC'94, pages 154–161. IEEE, 1994. 7

[30] A. E. Tew and M. Guzdial. Developing a validated assessment of fundamental cs1 concepts. In Proceedings of the 41st ACM technical symposium on Computer science education, pages 97–101, 2010. 6, 33

[31] A. E. Tew and M. Guzdial. The fcs1: a language independent assessment of cs1 knowledge. In Proceedings of the 42nd ACM technical symposium on Computer science education, pages 111–116, 2011. 6, 33

[32] w3c. Web content accessibility guidelines 2.0, w3c world wide web consortium recommendation, criterion 1.4.3. https://www.w3.org/TR/WCAG21/#contrast-minimum, 2018. Accessed: 2021-03-02. 9

[33] w3c. Web content accessibility guidelines 2.0, w3c world wide web consortium recommendation, criterion 1.4.6. https://www.w3.org/TR/WCAG21/#contrast-enhanced, 2018. Accessed: 2021-03-02. 9

[34] D. Weintrop and U. Wilensky. To block or not to block, that is the question: students' perceptions of blocks-based programming. In Proceedings of the 14th international conference on interaction design and children, pages 199–208, 2015. 2, 3, 6

[35] D. Weintrop and U. Wilensky. Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In Proceedings of the Eleventh Annual International Conference on International Computing Education Research, pages 101–110, 2015. 5, 6, 31, 32, 33

[36] D. Weintrop and U. Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. ACM Transactions on Computing Education (TOCE), 18(1):1–25, 2017. 2, 3, 5, 6, 31, 32, 43, 46

[37] D. Weintrop and U. Wilensky. How block-based, text-based, and hybrid block-/text modalities shape novice programming practices. International Journal of Child-Computer Interaction, 17:83–92, 2018. 2, 3, 6, 31, 32

[38] L. Wu, G. Liang, S. Kui, and Q. Wang. Ceclipse: An online ide for programing in the cloud. In 2011 IEEE World Congress on Services, pages 45–52. IEEE, 2011. 5

PRE-TEST QUESTIONS

WELKE WAARDEN HEBBEN X EN Y AAN HET EINDE VAN DIT PROGRAMMA?

```
x = 7
y = 4
y = x - 1
x = x + 3
```

- x=7, y=4
- x=6, y=10
- x=10, y=6
- x=4, y=7
- X=X+3, y=X+1
- x=6, y=8
- Ik weet het niet

WELKE WAARDEN HEBBEN X EN Y AAN HET EINDE VAN DIT PROGRAMMA?

```
x = 4
y = 6
x = x + y
y = x - y
x = x - y
```

- x=4, y=5
- x=6, y=4
- x=5, y=6
- x=10, y=7
- X=6, y=10
- x=10, y=6
- Ik weet het niet

Welke waarden hebben X en Y aan het einde van dit programma?

(met 'and' wordt er het woord 'en' bedoelt, met 'or' wordt er het woord 'of' bedoelt)

```
x = True and False
y = False or True
```

- x=false, y=false
- x=false, y=true
- x=true, y=false
- X=true, y=true
- Ik weet het niet

Wat is de output van het programma?

```
x = 0
if x > 0:
    print('getal is positief')
else:
    print('getal is negatief')
```

- getal is positief
- getal is negatief
- niets
- Ik weet het niet

Wat is de output van het programma?

```
x = 8
y = x - 9
if x >= y:
    print('x is groter dan y')
else:
    print('y is groter dan x')
```

- x is groter dan y
- y is groter dan x
- niets
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 1
while x < 6:
    print(x)
    x = x + 1
```

- 1 2 3 4 5 6
- 1 2 3 4 5
- 2 3 4 5 6
- 2 3 4 5
- niets
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 1
while x < 6:
    print(x)
    if x > 3:
        break;
    x = x + 1
```

- 1 2 3 4 5 6
- 1 2 3 4
- 1 2 3
- 1 2
- niets
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
fruit = ['appel', 'banaan', 'druif']
for stuk in fruit:
    print(stuk)
```

- appel

- appel banaan druif
- druif banaan appel
- niets
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
kleuren = ['rode', 'groene']
fruit = ['appel', 'banaan', 'druif ']
for stuk in fruit:
    for kleur in kleuren:
        print(kleur, stuk)
```

- rode appel groene banaan druif
- rode appel groene appel rode banaan groene banaan rode druif groene druif
- appel banaan druif rode groene
- appel rode banaan groene druif
- appel rode appel groene banaan rode banaan groene druif rode druif groene
- niets
- Ik weet het niet

WANNEER JE EEN INGEWIKKELD PROBLEEM TEGENKOMT, WELKE STAPPEN GEBRUIK JE OM DAT PROBLEEM OP TE LOSSEN?

DEZE VRAAG GAAT NIET OVER DE VRAGEN HIERVOOR. MET 'PROBLEEM' BEDOELEN WE IETS (EENDER WAT!) WAAR JE ZELF EEN OPLOSSING VOOR MOET ZOEKEN.

(open question, no multiple choice)

PRE-TEST QUESTIONS

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 7
y = 4
y = x -1
x = x + 3
print("x is gelijk aan", x, "en y is gelijk aan", y)
```

- x is gelijk aan 7; y is gelijk aan 4
- x is gelijk aan 10; y is gelijk aan 6
- x is gelijk aan 4; y is gelijk an 7
- x is gelijk aan "x+3": y is gelijk aan "x-1"
- x is gelijk aan 6; y is gelijk aan 8
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
som = 0
for x in [o, 1, 2, 3]:
    som = som + X
    print("som : ", som)
```

- "som: 3"
- "som: 6"
- "som: 0" "som: 1" "som: 3"
- "som: 0" "som: 1" "som: 3" "som: 6"
- Ik weet het niet

WAT DOET HET VOLGENDE PROGRAMMA?

```
x = 0
som = 0
while x <= 200:
    if x > 10 and x < 50:
        som= som + X
    x = x + 1
    print("som:" . som)
```

- Het berekent de som van getallen tussen 1 en 200
- Het berekent de som van getallen groter dan 10
- Het berekent de som van getallen kleiner dan 50
- Het berekent de som van getallen tussen 10 en 50
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 0
som = 0
while x <= 10:
    if x > 8:
    som= som + x
    x = x + 1
print("som : ", som)
```

- "som: 9"
- "som: 19"
- "som: 9" "som: 19"
- Het programma werkt niet
- Ik weet het niet

SCHRIJF EEN PROGRAMMA IN PYTHON WAARMEE JE DE VOLGENDE OUTPUT ZOU KRIJGEN IN DE TERMINAL.

```
1 : 0
2 : 4
3 : 8
4 : 12
```

(open question, no multiple choice)

STEL: JE WORDT GEVRAAGD OM EEN PROGRAMMA TE SCHRIJVEN WAARBIJ GEBRUIKERS EEN ZIN KUNNEN INVOEREN. HET PROGRAMMA TOONT DAARNA HOEVEEL KEER DE LETTER 'E' IN DE ZIN VOORKOMT.
WELKE VAN DEZE OPTIES HEEFT EEN PROGRAMMEERTAAL MINSTENS NODIG, ZODAT JIJ DIT PROGRAMMA KAN SCHRIJVEN?

JE KAN MEERDERE OPTIES AANDUIDEN

- De invoer van een gebruiker opslaan
- Tekst weergeven op het scherm
- Twee letters met elkaar vergelijken om te zien of ze gelijk zijn
- Letters omzetten naar cijfers en omgekeerd
- Data aanmaken en wijzigen terwijl het programma wordt uitgevoerd

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 0
while x < 5:
    print(x)
```

- 0 1 2 3 4
- 5
- Het programma werkt niet
- Het programma zit vast in een oneindige lus
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
for x in [1, 2, 3]:
    print("Appel")
    print("Appelsien")
print ("Appelsien")
```

- Appel Appelsien Appel Appelsien Appel Appel Appelsien
- Appelsien Appel Appelsien Appel Appelsien Appel Appelsien
- Appel Appelsien Appel Appelsien Appel Appelsien Appelsien
- Appelsien Appel Appelsien Appel Appelsien Appel Appel
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 0
while x <= 10:
    if x > 5:
        print(x)
    x = x + 1
```

- 0 1 2 3
- 5 6 7 8
- 6 7 8 9
- 6 7 8 9 10
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 4
if x > 10:
    x = 10
else:
    if x < 5:
        x = 5
print (x)
```

- 4
- 10
- 5
- Niets
- Error
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = "ik"
z = "games"
boodschap = x + "speel" + z
print (boodschap)
```

- ik speel games

- speel games ik
- ikspeelgames
- games speel ik
- Error
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 7
if x > 0:
    print("getal is positief")
else:
    print("getal is negatief")
```

- Het programma kijkt na of je een getal kunt delen door 0
- Het programma kijkt na of een getal positief of negatief is
- Het programma zorgt ervoor dat je niet kunt delen door 0

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
x = 7
if x > 0:
    print("getal is positief")
else:
    print("getal is negatief")
```

- getal is positief
- getal is negatief
- niets
- Ik weet het niet

ALS X GELIJK ZOU ZIJN AAN -12, WAT IS DAN DE OUTPUT VAN HET PROGRAMMA?

```
x = 7
if x > 0:
    print("getal is positief")
else:
    print("getal is negatief")
```

- getal is positief
- getal is negatief
- niets
- Ik weet het niet

0 IS NIET POSITIEF, EN OOK NIET NEGATIEF. MAAR DIT PROGRAMMA TOONT BIJ 0 DAT HET GETAL NEGATIEF IS.
SCHRIJF DE CODE OPNIEUW ZODAT BIJ (X = 0) DE OUTPUT ZAL ZIJN "HET GETAL HEEFT GEEN TEKEN". ZORG ERVOOR DAT ALLE ANDERE CODE BLIJFT WERKEN.

```
x = 7
if x > 0:
    print("getal is positief")
else:
    print("getal is negatief")
```

(open question, no multiple choice)

WELKE WAARDES HEBBEN DE VARIABELEN X, Y EN Z AAN HET EINDE VAN HET PROGRAMMA?

```
x = "ja"
y = x
z = x
x = "nee"
y = "misschien"
z = x
```

- x is gelijk aan "ja, nee": y is gelijk aan "ja, misschien"; z is gelijk aan "ja, nee"
- x is gelijk aan "nee"; y is gelijk aan "misschien"; z is gelijk aan "ja"
- x is gelijk aan "ja"; y is gelijk aan "misschien; z is gelijk aan "nee"
- x is gelijk aan "nee"; y is gelijk aan "misschien"; z is gelijk aan "nee"
- x is gelijk aan "nee": y is gelijk aan "misschien"; z is gelijk aan "*"
- Ik weet het niet

WAT IS DE OUTPUT VAN HET PROGRAMMA?

```
c = 5
while c > 0:
    c = c- 2
    print(c)
```

- 3
- 5 3 1
- 5 4 3 2 1 0
- 3 1 -1
- Ik weet het niet

WERKT DIT PROGRAMMA?

```
print (boodschap)
boodschap = "Hallo!"
```

- Ja
- Nee
- Ik weet het niet

INDIEN JE "JA" ANTWOOORDDE:
WAT IS DE OUTPUT VAN HET PROGRAMMA?

(open question, no multiple choice)

INDIEN JE "NEE" ANTWOOORDDE:
PAS HET SCRIPT ZO AAN DAT HET NU WEL WERKT?

(open question, no multiple choice)

SCHRIJF EEN PROGRAMMA DAT TWEE GETALLEN VERGELIJKT MET ELKAAR. HET GROOTSTE GETAL WORDT DAN GETOOND OP HET SCHERM.

BIJVOORBEELD: ALS GETAL1 = 5 EN GETAL2 = 7, WORDT GETAL2 GETOOND OP HET SCHERM.

(open question, no multiple choice)

WERKT DIT PROGRAMMA? WAAROM WEL/NIET?

```
x = 8
y = x - 9
if y > 0:
    z = 8
    nieuwePlaats = y + z
    print("punt s bevindt zich op x, y")
else:
    nieuwePlaats = y + z
    print("punt s bevindt zich op x, -y")
```

(open question, no multiple choice)