JOHANNES KEPLER
UNIVERSITY LINZ

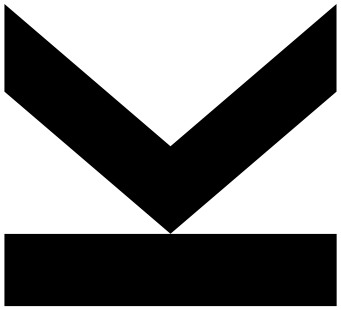
Author
Marks Osipovs
01428789

Submission
Institute of Systems Software

Thesis Supervisor
DI Dr. **Markus Weninger**

September 2022

New Exam Question Types for the Online Examination System Xaminer



Bachelor's Thesis

to attain the academic degree of

Bachelor of Science

in the Bachelor's Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

New Exam Question Types for the Online Exam System Xaminer

Student: Marks Osipovs
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: March 2022

Dipl.-Ing. Dr.
Markus Weninger, BSc
Institute for System Software

P +43-732-2468-4361
F +43-732-2468-4345
markus.weninger@jku.at

The exam system Xaminer is used by the Institute for System Software and other institutes to provide online exams for students. To improve the experience for both lecturers and students, the system is constantly being improved and extended. Especially (1) fill-in-the-blanks questions and (2) drag-and-drop questions are currently missing.

The goal of this thesis is to introduce these question types to the system. The student has to develop convenient editing features to allow lecturers to easily define fill-in-the-blanks and drag-and-drop questions. Fill-in-the-blank questions should be defined by the lecturer by providing a reference solution to a question and marking certain parts as this answer as “blank parts” which then have to be filled in by the exam taker.

Example:

```
class Student {
  // TODO define suitable fields
  ##BLANK_START##
  private final int age;
  private final String name;
  ##BLANK_END##
  …
}
```

Such a question should then be presented the exam taker with a text field that has to be filled out.

For drag-and-drop questions, the lecturer should be able to select a source image on which certain areas can be marked as “answer areas” with a respective reference solution. All available answers are then presented to the exam taker, who is then responsible for dragging all answer to their correct answer area.

Modalities:
The progress of the project should be discussed at least once per month with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisors. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.09.2022.

Abstract

Despite ongoing development, the area of online examination is still somewhat underutilized in the field of academic teaching. In this thesis, we help increase the usability of the online examination platform *Xaminer* by adding two new question types. Gap text questions should consist of a text with fillable blanks. Similarly, gap image questions should contain an image with blanks. Students have to fill the blanks during the exam to answer the questions.

First, we determine the system requirements and usability requirements for each question type. Then, we provide an in-depth insight into our implementation. Finally, we evaluate the user experience of the new question types by analyzing typical workflows and provide several ideas on how they can be improved in the future.

Kurzfassung

Trotz stetiger Verbesserungen ist das Gebiet der Online-Prüfungen im Bereich der akademischen Lehre unterrepräsentiert. In dieser Arbeit helfen wir die Benutzerfreundlichkeit der Online-Prüfungsplatform *Xaminer* zu verbessern, indem wir zwei neue Fragetypen hinzufügen. Lückentext-Fragen sollen aus einem Text mit ausfüllbaren Blankofeldern bestehen. Ebenso sollen Lückenbild-Fragen ein Bild mit Blankofeldern beinhalten. Studenten müssen die Blankofelder während der Prüfung ausfüllen, um die Fragen zu beantworten.

Zuerst bestimmen wir die System- und die Benutzerfreundlichkeitsanforderungen für jeden Fragetypen. Danach geben wir einen ausführlichen Einblick in unsere Implementierung. Schlussendlich werten wir die Benutzerfreundlichkeit der neuen Fragetypen aus, indem wir die typischen Arbeitsläufe analysieren, und bieten einige Verbesserungsvorschläge für die Zukunft an.

Table of Content

Contents

Abstract **i**

Kurzfassung **i**

1 Introduction **2**

2 Background **4**
2.1 System overview 4
2.2 Technology stack 6

3 Gap Text Questions **8**
3.1 Approach 8
3.2 Implementation 9
3.2.1 Block Editor 9
3.2.2 Gap tags 11
3.2.3 Block Visualizer 13

4 Gap Image Questions **16**
4.1 Approach 16
4.2 Implementation 17
4.2.1 Block Editor 17
4.2.2 Block Visualizer 20

5 Evaluation of the user experience **24**
5.1 Gap Text Questions 24
5.2 Gap Image Questions 27

6 Future Work **30**
6.1 Gap Text Questions 30
6.2 Gap Image Questions 31

7 Conclusion **32**

Literature **34**

1 Introduction

Academia has always been a driving force behind the development of the internet – it is no coincidence that the World Wide Web was created in 1989 specifically at CERN. It started as a document management system that would allow users from anywhere in the world to access and modify data. The rise of personal computers and smartphones combined with the increasing internet speeds have made the internet irreplaceable in many aspects of everyday life. And yet, to this day there are areas where this technology is still not widely adopted. Following the outbreak of the COVID-19 pandemic, one of such areas was revealed to be the field of examination in academic institutions. Handwritten exams are one excellent example for the lack of digitalization.

Several online platforms that allow students to take their exams over the internet already exist. The most prominent one in Austria is Moodle [9]. However, these platforms do not target specific topics, courses, or faculties and thus offer only general question types. Therefore, they are ill-suited for specialized fields of science, such as the field of computer science. For example, standard open-ended freewriting questions must be used for coding assignments on Moodle. They lack important quality-of-life programming features such as syntax highlighting or automatic code indentation, negatively impacting the students' ability to perform during the examination.

To address this, the Institute of Software Systems at the Johannes Kepler University Linz developed the *Xaminer* platform in early 2020. Using this Web application, lecturers can create exams online by writing their own questions digitally with the help of several question types. Students can take these exams online while being supervised via video over a conference call. Since its inception, several new features and improvements have been added to the platform as a part of an ongoing development process. Yet, Xaminer's functionality still remains limited in some areas. At the time of writing, it supports only a small set of standard question types, such as open-ended freewriting questions, as well as single-choice and multiple-choice questions. Additionally, due to the platform being developed and tested at a Computer Science institute, it supports code questions specifically designed for programming tasks that provide the much needed quality-of-life features.

In this thesis, we extend Xaminer's functionality by adding support for two new exam question types. A *Gap Text Question (GTQ)* allows exam authors to provide a text and mark parts of it as fillable blanks (called *gaps*). Students have to fill the gaps during an exam with correct answers. In a similar fashion, a *Gap Image Question (GIQ)* contains an image with fillable gaps, with additional answer options that are provided by the exam authors during exam creation. Students have to fill a gap by choosing the correct option.

Their inclusion enables Xaminer to be operated in a variety of new ways, allowing its usage to spread to new faculties and lectures.

2 Background

In this section, we discuss the status quo of the Xaminer system and its underlying technology stack[1].

2.1 System overview

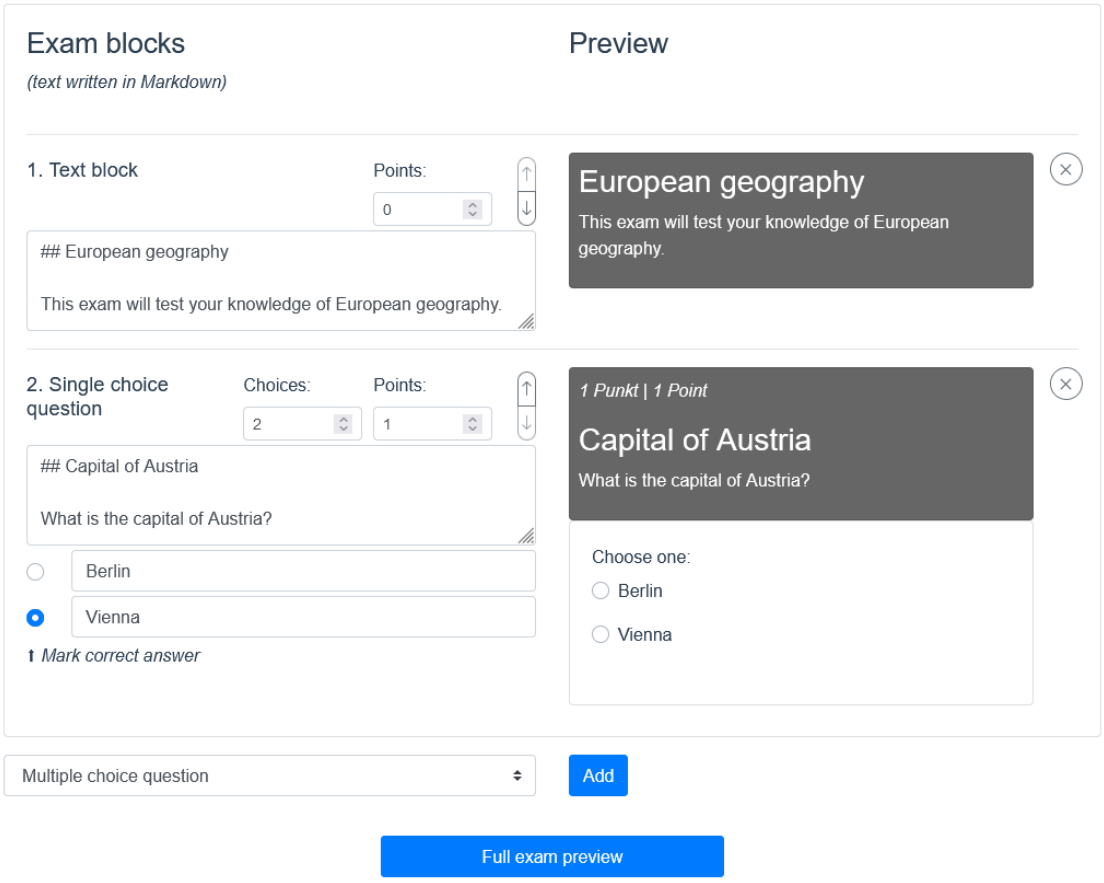


Figure 1: An exam with a text block and a single choice question block, as seen during exam creation.

First, we take a look at how an exam is created and processed by the current system. The Xaminer platform already provides a user interface to simplify exam creation. Authorized users have access to an administrative part of the Web application where they can create and edit their exams, as well as manage students and supervisors. Each exam consists of one or multiple *blocks* that can be deliberately added or removed by the exam author. Each block represents one question of a certain *question type*. Their contents and user interface vary from each other, as showcased in Figure 1. For example, we can use a Text block to display information to

[1]Set of technologies used to build and execute an application.

the students, but it cannot be interacted with. A Multiple-Choice Question block contains a description of the question and additionally includes a checkbox for each possible answer. Many blocks for the various tasks that can frequently be found in typical university exams are already available as a part of the system, i.e., blocks for open-ended freewriting tasks, for coding problems, for single-choice questions, for the multiple-choice questions, and for image questions.

When exam authors add a new block to the exam, it is split into two parts by Xaminer. On the left side of the Web page, one can find the *Block Editor*, where lecturers can modify a question by interacting with the user interface. The right side of the screen contains the *Block Visualizer*, which renders the question as it would be visible to the students during the exam. Additionally, a preview of the entire exam from the students' perspective can be viewed using the "Full exam preview" button (see bottom of Figure 1).

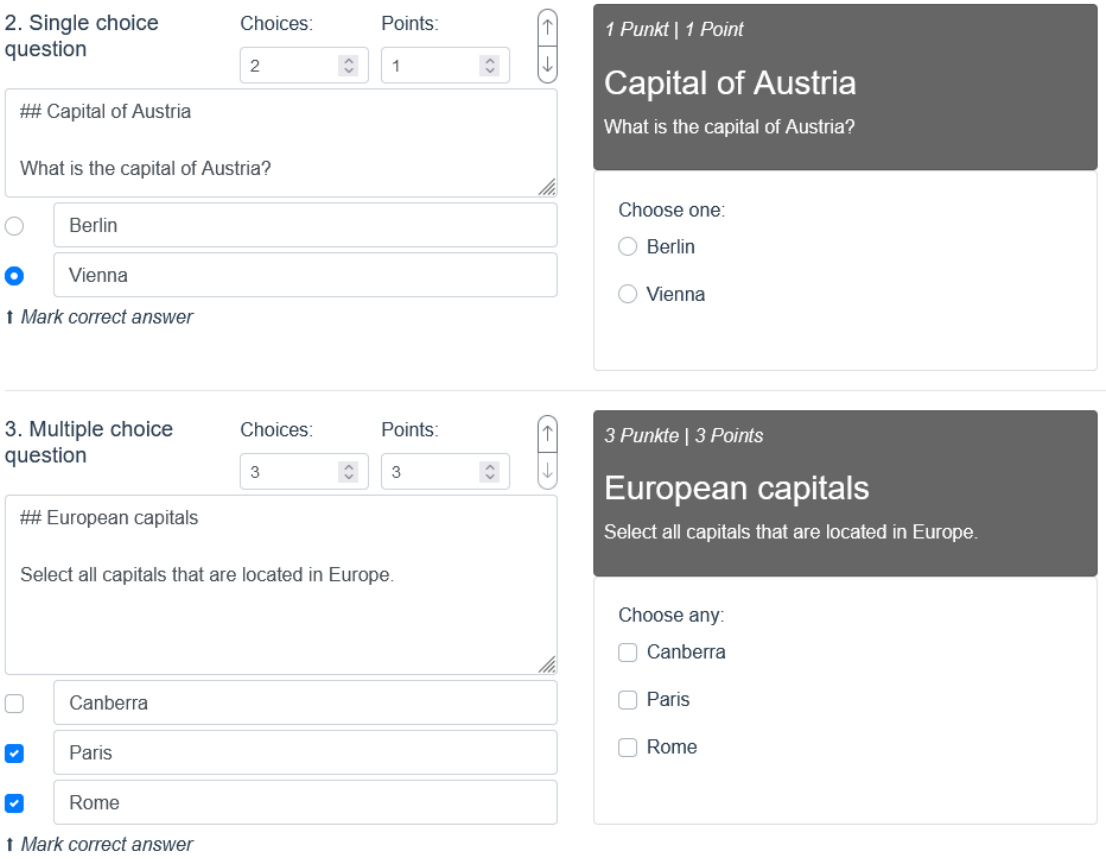


Figure 2: An exam with a single choice question block and a multiple choice question block. "Vienna" has been marked as the correct answer to the single choice question. "Paris" and "Rome" are marked as the correct answers to the multiple choice question.

Now, let us take a look at the implementation of the blocks. A base Block Editor consists of a single text field that is implemented by a `textarea` HTML element. Exam authors can use the text field to provide the question description. The Text block 1 in Figure 1 showcases this base Block Editor design. All other Block Editors are built upon it, thus, they also all include a

`textarea` HTML element. Additional standard user interface elements can be provided when required by the other question types, such as radioboxes for single-choice questions and checkboxes for multiple-choice questions. Furthermore, form inputs or sliders to modify various settings of a block can be included. Figure 2 shows an example usage of radioboxes and checkboxes in two different blocks.

The students are not authorized to access the Block Editors and only interact with the Block Visualizers during an exam. When they access an exam, the user interface of the visualizer is slightly altered. During exam creation, the question title and description are located above the area that the students can interact with. During the exam process, the area is now located to the left of it, as can be seen in Figure 3.

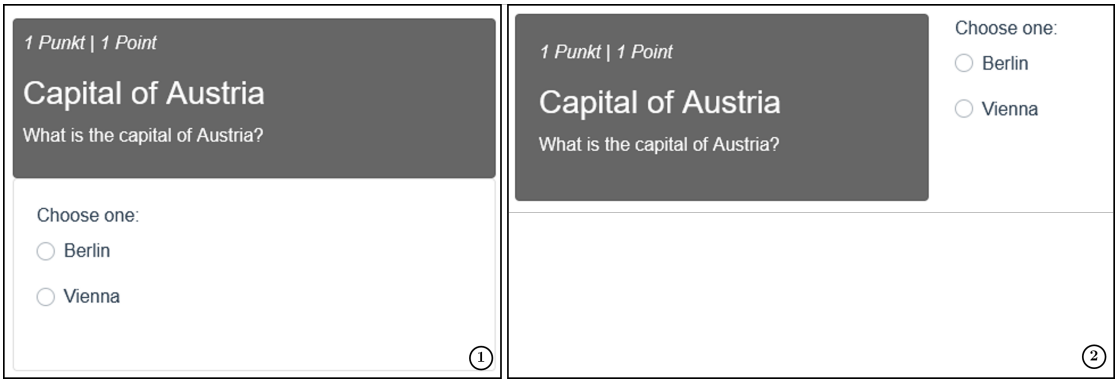


Figure 3: ① A single choice question rendered during the exam creation (exam author view). ② A single choice question rendered during the exam (student view).

2.2 Technology stack

The underlying technologies that power the application can be separated into two groups – the ones for the frontend and the ones for the backend[2]. For the frontend, Xaminer uses the TypeScript [7] programming language. Additionally, using Vue [13] as its model-view-viewmodel frontend framework allows us to keep the code base clean by splitting it into *Vue components*. A Vue component is a logical module that encapsulates custom UI elements and programming logic. It can be reused throughout the application. UI elements such as buttons, dropdowns, form inputs, sliders, or menus are provided by HTML and styled using the BootstrapVue [1] framework. The backend is powered by a Spring Boot [10] application written in the Kotlin [4] programming language and uses a MongoDB [8] database to store the exams, i.e., student submissions.

The Block Visualizer then accesses the configuration to render the exam. However, it would be very inefficient to communicate every change from a Block Editor to a Block Visualizer via the database. To solve this problem, the Block Visualizer fetches the configuration from the database exactly once, when the Vue component is loaded. From there on, the Block Visualizer listens for changes in the Block Editor, and updates itself accordingly, as illustrated in Figure 4.

[2] *Frontend* refers to parts of the app commonly exposed to and accessible by the user, as opposed to the *backend*, which is hidden and works in the background.

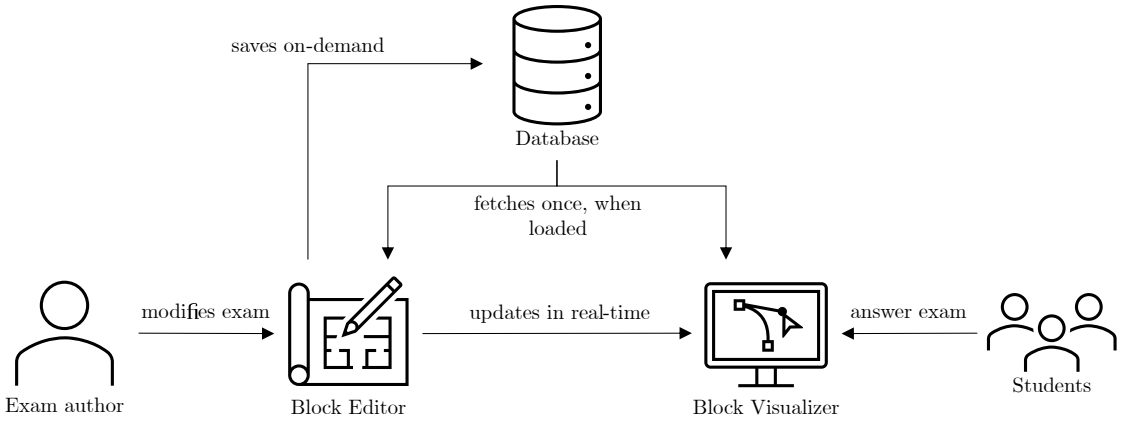


Figure 4: Block Editor - Block Visualizer - Database model.

To add Gap Text Questions and Gap Image Questions to the existing Xaminer platform, we need to create a new block for each question. The new Block Editors must provide additional user interface elements on top of the ones already present in Xaminer, to allow for quick and intuitive creation of exam questions. The new Block Visualizers must be reusable, i.e., we should be able to use them for both exam creation and the examination process itself. This will ensure that the exam questions appear and behave the same way for both lecturers and students, asserting a consistent experience. Therefore, the Block Visualizers must comply with our Block Editor - Block Visualizer - Database model (see Figure 4). Additionally, the visualizers should have the option to record students' answers and expose them to the rest of the app, to store the answers in the database as part of a submission.

3 Gap Text Questions

In this section, we discuss the design process and the implementation of Gap Text Questions.

3.1 Approach

In this section, we determine the system requirements and usability requirements for the Gap Text Questions and discuss our thought process when designing them.

Requirements definition. The first step to design the new Gap Text Question question type was to design the user interface of the Block Editor. We started by determining what would be required for the exam author to create the question with the desired functionality, and what UI elements we would have to include to make the process effortless.

A text field has to be present to specify a text that describes the assignment, just as with other Xaminer questions. Additionally, we require a separate text field to provide the text with the reference solution, which we refer to as the *template text*. The lecturers need a way to mark parts of the template text to be identified as gaps, i.e., those parts that later have to be filled in by the students. Two kinds of gaps are required:

1. *Inline Gaps* can be located in one line of text and can have text to the left and right of them.
2. *Multiline Gaps* that stretch several lines and can only have text above and below them.

A schematic showcasing how the Gap Text Question could appear to students during an exam can be seen in Figure 5.

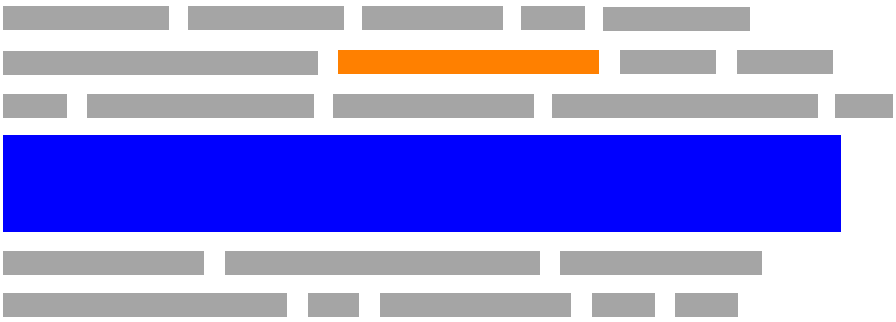


Figure 5: A mockup Gap Text Question. An Inline Gap and a Multiline Gap are specified in a paragraph of text.

A text field for the description is already included as a part of the base Block Editor. The most straightforward way to provide the template text was to provide another, separate `textarea` element. In order to mark the gaps of the template text, we looked at features of markup languages such as HTML [11] or Markdown [2]. They allow programmers to provide specific tags within the contents of a document. Information enclosed by these tags is treated differently from the rest of the text by the text processors. The exact approach varies based on the type of tag that encapsulates the information within. For example, text surrounded by two asterisks, such

as this `**bold text**`, is rendered by Markdown as **bold text**. As we can observe, the tags themselves are not visible to the end user. By utilizing this mechanic, we could specify where in our template text the gaps would start and end.

SVG as a mean of visualization. With the Block Editor layout figured out, the next problem we had to solve was the Block Visualizer. First, (1) it must be able to process and visualize the two kinds of gaps. Additionally, it would be ideal if the students are not limited in their workflows during the exam process. Everyday features that are expected of modern text editors should be allowed. Thus, students should be able to: (2) copy and paste text, (3) use keyboard shortcuts such as ctrl+V, ctrl+Z, and others to modify the text, and (4) right click a text selection to bring up a context menu that provides means of editing the text.

A straightforward solution to render the template text is to use *Scalable Vector Graphics (SVGs)*. Despite originally being designed for vector graphics, they allow for inclusion of other HTML elements. Some elements are supported natively, for example, the `text` element for the inclusion of text in graphics. Elements from other XML namespaces (such as HTML) must be included by importing them using a `foreignObject` element and wrapping them in it. In our case, we are using two elements from the (X)HTML namespace. `input` elements display a small inline box, i.e., a rectangular text field that is placed on the same line as the surrounding text. Using it, users can freely provide an input text, which makes it suitable to be used for Inline Gaps.

As `input` elements do not support multiple lines of text, we use `textarea` elements for the Multiline Gaps. Additionally, `textarea` elements can be resized arbitrarily, allowing us to accommodate inputs of any length.

SVGs use a combination of a coordinate system and dedicated width and height values for placement of elements and drawing of objects. This means that we can deliberately place and resize our `input` and `textarea` elements by moving and resizing either the elements themselves or their parent `foreignObject` element.

Finally, we determined that solutions provided by students might be longer than the solution that was intended by the exam author. It is thus necessary to include a mechanism to gracefully handle these overlong answers.

3.2 Implementation

In this section, we discuss the details of the implementation of Gap Text Questions and the layout of their user interface.

3.2.1 Block Editor

Implementing the Block Editor for Gap Text Questions was simplified using other Xaminer Block Editors as a template. An example showing the GTQ Block Editor ① and the GTQ Block Visualizer ② can be seen in Figure 6. The `textarea` element for the *question description* ③ is already included as a part of every base Block Editor. Additionally, we added a second `textarea` element to our Block Editor to provide the *template text* ④. Further controls for the scaling of the x-axis ⑤, the scaling of the y-axis ⑥, and for creating gaps in the template text ⑦ are implemented using BootstrapVue's standard user interface elements.

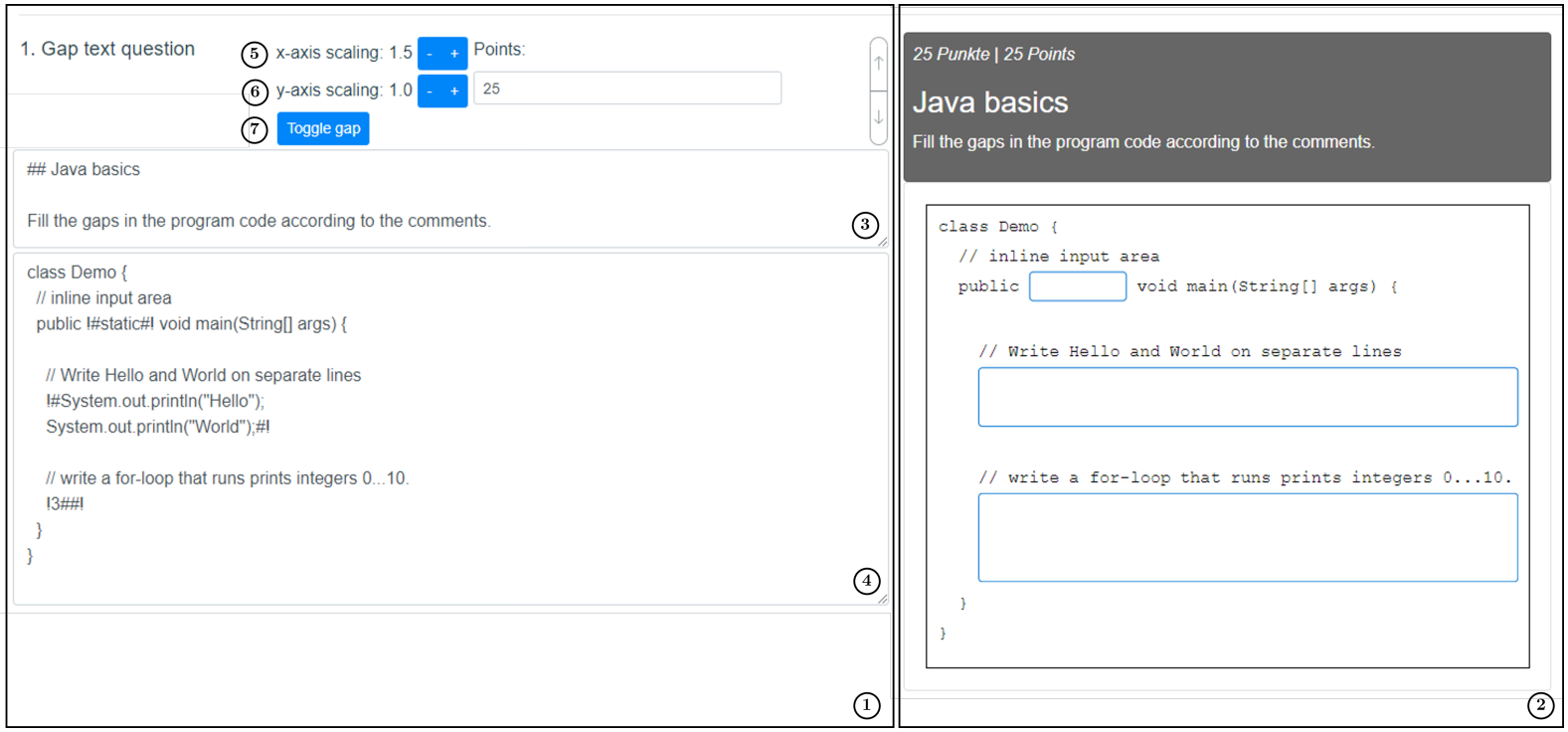


Figure 6: ① Block Editor and ② Block Visualizer of a Gap Text Question with the default template text. ③ Question description `textarea` element. ④ Template text `textarea` element. ⑤ x-axis-scaling factor controls. ⑥ y-axis-scaling factor controls. ⑦ Toggle gap button.

3.2.2 Gap tags

After that, we had to decide on a character sequence by which gaps could be marked within the template text. Xaminer is developed by the Institute of Software Systems and thus, at the moment of writing, is primarily used in the field of Computer Science. Therefore, we needed to find a character sequence for the opening gap that would not interfere with text or program code that could be commonly found in a programming task. For that goal, we analyzed the programming languages that are taught at the Johannes Kepler University of Linz. We settled on the character sequence !# to denote the beginning of a gap (*Opening Tag*) and the character sequence #! to signify its ending (*Closing Tag*). The correct solution for the gap is then placed between the two tags: `!#correct solution#!`.

To the best of our knowledge, these sequences are not used to denote inline comments or comment blocks in most of the common programming languages, as outlined in Table 1. Furthermore, the # character is not used for any mathematical or logical operations in the vast majority of the popular programming languages. Thus, it would be very unlikely to find it directly before or after an exclamation mark, which is used as either a mathematical operator for the factorial operation or as a logical operator for negation. However, several notable exceptions exist, which we discuss further in detail.

Programming language	**Single line comment**	**Block comment start**	**Block comment end**
ArangoDB Query Language	//	/*	*/
Assembly	#	Not available	Not available
Bash	#	Not available	Not available
C	Not available	/*	*/
C++	//	/*	*/
C#	//	/*	*/
Cypher	//	Not available	Not available
Haskell	–	{-	-}
Java	//	/*	*/
JavaScript	//	/*	*/
Kotlin	//	/*	*/
MATLAB	%	%{	}%
Prolog	% or %! or %%	/*	*/
Python	#	Not available	Not available
R	#	Not available	Not available
Rust	//	/*	*/
Scala	//	/*	*/
SQL	–	/*	*/
VHDL	–	Not available	Not available
Wolfram Language	Not available	(*	*)

Table 1: List of programming languages, query languages, and hardware description languages taught at JKU Linz with their respective comment indicators.

Assembly language and Python language. The character ! is unused in these two languages, meaning that the character sequence !# would result in a syntax error and thus does not appear in program code. Therefore, we can use !# as our Opening Tag. The Closing Tag #! is a valid character sequence and would denote a single line comment. However, inserting a whitespace character between the single line comment indicator # and the ! character solves the conflict.

Bash language and R language. The ! character is used for the negation operation in these two programming languages. However, the operation requires a variable to be specified immediately after the ! character. As # character cannot be used for variable names in the two languages, the character sequence !# can not appear in program code. Thus, we can use it as our Opening Tag. The Closing Tag conflict can be solved in the same way as the Closing Gap conflict with the Assembly language and the Python language.

Wolfram Language. Both sequences can be found in the Wolfram Language, which is used in the symbolic computational software *Mathematica*. There, the # character is a shorthand operator of the Slot expression[3] that is used to identify the first argument of a pure function. Arbitrary operations – including mathematical ones – can be performed upon it. Consider the code in Listing 1:

```
In[1]:=  Map[3!#&, {1, 2, 3, 4} ]
Out[1]:= {6, 12, 18, 24}

In[2]:=  Map[#!&, {1, 2, 3, 4} ]
Out[2]:= {1, 2, 6, 24}
```

Listing 1: Two Wolfram Language/Mathematica functions. In `In[1]`, we apply the function $3! * x$ to every element x of a list. In `In[2]`, we apply the factorial operation to every element of a list.

However, as every expression in Mathematica can be written using its full name instead of using just its symbol, we can rewrite both functions to avoid conflict. The expression `3!#&` can be altered to `3!*#&`, and the expression `#!&` can be altered to `Factorial[#]&`. Nonetheless, it will be up to exam authors to check their code for compatibility with Gap Text Questions.

Exam authors can provide the sequences !# and #! manually by typing the corresponding characters in the template text. Alternatively, they can select a portion of the text – the selection can span multiple lines – and press the "Toggle gap" button in the Block Editor toolbar (see ⑦ in Figure 6). The system will then subsequently insert the tags for the user at the beginning and the end of the selection. If the selected text passage is already a valid gap (i.e., it starts and ends with the correct tags), then the system will instead remove the existing tags instead of adding a new gap.

[3]Numbers, variables, functions, graphics and other objects are represented in Mathematica as *symbolic expressions*.

3.2.3 Block Visualizer

The next step is to render the template text in a Block Visualizer. Since its SVG visualization is quite different to existing Block Visualizers, we had to write several new helper functions. Further, we implemented it in its own dedicated Vue component called *Gap Text Question Visualizer (GTQ Visualizer)*. This allows us to reuse it to render the template text during (1) the exam creation, (2) the exam preview, and (3) the exam process.

Text processing. Since we are using SVGs with a coordinate system, we needed to create a visualizer which can read the template text word for word and create corresponding `text`, `input`, and `textarea` elements. Furthermore, the elements need to have correct x and y coordinates. In SVGs, the horizontal axis is called the x-axis, and the vertical axis is called the y-axis. After prototyping several designs, we settled on a solution that was inspired by parsers. Code parsers are designed to systematically iterate through program code and to convert it into a different representation. First, we implemented a text scanner similar to those used in LL-parsers[4]. It can iterate over the template text character by character and automatically detect four kinds of tokens: (1) a *Whitespace*, (2) a *Word* (represented in the SVG by a `text` SVG element), (3) an *InlineGap* (represented by an `input` HTML element) and (4) a *MultilineGap* (represented by a `textarea` HTML element). This allows us to view the entire template text as an array of tokens. Using Vue, we can then easily iterate over this array when creating the SVG code. This allows us to dynamically create the corresponding SVG elements for every token in the array and supply them with the correct coordinates, width and height.

Any whitespaces found in the template text are represented by Whitespace tokens. Code parsers usually skip whitespaces as they are useless for computers and only help to make program code easier to read. However, we must keep track of them in our task, as whitespaces can be used to denote the indentation of a line, and thus affect the coordinate system of the GTQ Visualizer. In our implementation, Whitespace tokens are not rendered by the GTQ Visualizer. Instead, as soon as the GTQ Visualizer reads such a token, it moves forward along the x-axis, indenting the next token by the width of one character.

A word is considered to be a sequence of characters that is connected together, i.e., not separated by whitespaces or gaps. They are represented by Word tokens.

InlineGap and MultilineGap tokens represent the Inline Gaps and Multiline Gaps, respectively. These tokens additionally have a unique identifier and a correct answer.

When reading the template text, as soon as the parser detects the beginning of a gap, it creates an InlineGap token and starts to keep track of the number of processed characters until it reads the ending sequence. If the parser finds a newline character while it is still processing an InlineGap token, it automatically converts the token to a MultilineGap token.

InlineGap, MultilineGap and SVG dimensions. Computing the length of the correct answer of an InlineGap allows us to automatically compute the width of the corresponding `input` element by multiplying the number of read characters with the character width. This way, we have enough space to accommodate the solution of this gap. The height of an InlineGap, as suggested by the name, is always strictly one line. It therefore corresponds to the default line height.

[4]Left-to-right, leftmost derivation top-down parsers.

On the contrary, `textarea` elements of MultilineGap tokens always span from their x-coordinate to the rightmost edge of the image, making them fill the entire width of the SVG minus their starting indentation. The height is calculated based on the number of lines of the MultilineGap token. Under normal circumstances, HTML `textarea` elements can be resized by the user using the mouse. As both the height and the width of the SVG in the Block Visualizer are fixed with respect to the dimensions computed from the template text, this behavior has been disabled.

The SVG is resized automatically based on the parameters of the provided template text, such as its number of lines and line length. Its height is computed by multiplying the number of read lines with the line height (22 pixels by default). The width is similarly dynamic. To compute it, we first determine the longest line of the template text by calculating the number of characters in a line while taking into the account the widths of the inline `input` elements. The width of the SVG then equals the length of the longest line multiplied by the default font size (16 pixels, which equals a monospace font size of 12pt).

Overlong answer handling. However, after implementing and evaluating this concept, we discovered a potential source of problems for students when solving the Gap Text Questions. As different students can provide different solutions to the same question, their lengths may vary from the length of the correct solution that was provided by the exam author. In the field of programming, a simple example of this is variable naming: the exam author might specify dimensions of a hypothetical object using the letters `w` and `h`, where a student might name the variables `width` and `height`. In such a case, the solution given by the student will be longer than the space provided by the `input` element.

To address this problem, we added an *x-axis-scaling* parameter to the question configuration. It affects the width of `input` elements of all InlineGap tokens in a particular block. Its default value is set to 1.5, meaning that the width of an `input` element will be 1.5 times the width of the specified solution, e.g., 15 characters instead of 10. As we have outlined before, the total width of the SVG depends on the width of the inputs in the longest line of the template text, therefore, it is directly affected by the x-axis-scaling factor. Exam authors can change the x-axis-scaling to a different value using the form input in a Block Editor's toolbar (see ⑤ in Figure 6). MultilineGap tokens and their corresponding `textarea` elements are not affected by the x-axis-scaling, as they already span the entire width of the SVG.

A similar issue is present for MultilineGap tokens, where a solution may span more lines than originally intended. Therefore, we introduced an additional *y-axis-scaling* parameter (see ⑥ in Figure 6) that affects the height of all `textarea` elements of a block. It functions in the same way as x-axis-scaling.

Simplified MultilineGap notation. Furthermore, it is sometimes easier for exam authors to specify a MultilineGap without providing a sample solution. In that case, we would have no line breaks, so the parser would treat the gap as an InlineGap token. The Visualizer would not render it either, as without a solution the width of the `input` element is 0. Therefore, we expanded the functionality of the opening gap tag by providing support for the special syntax $!x\#y\#!$. This sequence is automatically converted to a MultilineGap token with x number of lines. The correct answer y is optional and is skipped during parsing, not influencing its height.

Storing exam answers. Finally, the students' solutions must be stored in the database. Using Vue's component events, we added event handling after a value of either an `input` element or a `textarea` element has been modified by the student. The IDs of the elements, as well as their values are saved as a part of the current exam instance. They can later be stored in the database and are further made accessible for grading using the existing functionality of the Xaminer framework.

4 Gap Image Questions

In this section, we discuss the design process and the implementation of Gap Image Questions.

4.1 Approach

In this section, we determine the system requirements and usability requirements for the Gap Image Questions and discuss our thought process when designing them.

As with Gap Text Questions, determining the system requirements was imperative to create a functional and easy to understand user interface. After observing the common ways Gap Image Questions are used in classic paper exams, and evaluating how this functionality is implemented in other online platforms such as Moodle, we arrived at the following list of requirements:

1. Background image: the exam author should have the ability to upload an image to serve as the question's background.
2. "Gaps" or "blanks": there must be a way to specify where in the image the gaps are located, i.e., those locations the students have to assign their answers to.
3. "Answer tiles": it must be possible for the exam author to provide possible answers the students can choose from to assign to the gaps.
4. A system for assigning answer tiles to gaps. In offline paper exams, a common way to solve an assignment is to first pick out an answer from a list of possible answer tiles. Students then write the answer in a gap in the image, and cross out the corresponding tile. On a digital platform, a more fluid experience is expected.

An mockup of a Gap Image Question showcasing the first three requirements can be seen in Figure 7. It depicts a typical Computer Science exam question where students are tasked with filling missing values in a binary search tree. The background image depicts the mostly complete binary search tree with several existing values (denoted by ■). Students must select values from possible a list of possible answers (denoted by ■) and place them in the gaps (denoted by ■).

Following a discussion with our supervisor, we initially decided to reduce the complexity of the task. First, we determined that handling database access and modifying the database schema was not within the scope of this bachelor thesis. Therefore, we decided to host background images on dedicated servers and to access them by loading them via their URLs. This enables us to store the URL of the image as a text string, as opposed to having to store the image itself. Doing so, we are be able to reuse the existing database CRUD[5] operations. Second, we wanted to avoid the complexity of having to develop a full user interface with image drag and drop functionality. In this original approach, lecturers are able to place gaps on the background image by simply clicking on it. The positions of the gaps are final, as they are be immovable. The students filled the gaps by first clicking on an answer, and then clicking on a gap.

The immediately apparent way to implement the functionality of this approach was to once again utilize SVGs, as they are designed for graphics and images. However, while performing preliminary research and prototyping the user interface before the start of the implementation

[5]Create, read, update, delete.

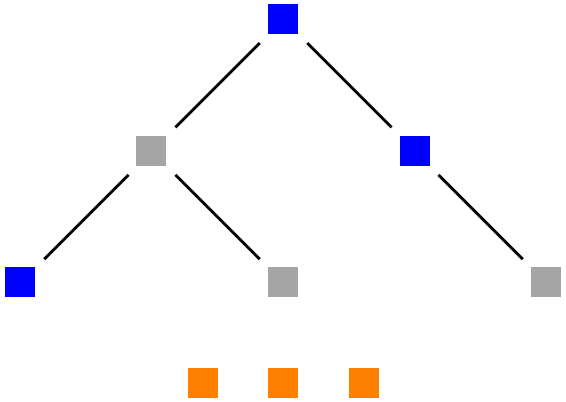


Figure 7: A mockup Gap Image Question. Three gaps are marked in the image. Three answers are located below the background image.

we discovered that it was proving to be impossible to design an intuitive user interface that was based on the mouse click workflow. The students must solve the exam in a limited time frame and are therefore subjected to stress. Thus, it is important to have a UI that immediately conveys its features and functionality to them without needlessly distracting students from the exam.

After prototyping and testing several UI mockups, we were not able to create a UI that functioned in accordance with our expectations. Therefore, we decided to look for an external library that (1) enabled an intuitive image editing workflow and (2) offered simplified event handling compared to the native SVG event handling. We settled on Konva [6], a state-of-the-art JavaScript library that specializes in 2D graphics for HTML5 using the `canvas` HTML element. It has several native features for image editing that are commonly found in desktop image editing software, e.g., layers and transformation boxes. Furthermore, it has full support for mouse and keyboard event handling. Finally, Konva supports Vue out-of-the-box and can thus be easily integrated into our technology stack.

Following that, we were able to enhance our original approach by developing a prototype user interface that was intuitive to use. Gap creation and answer location remained the same compared to our original approach. However, the gaps now can be interacted with by the exam authors, and answers are movable. This allows for an intuitive and fast workflow during exams.

4.2 Implementation

In this section, we discuss the details of the implementation of Gap Image Questions and the layout of their user interface.

4.2.1 Block Editor

As with other exam creation blocks within Xaminer, the user interface of this component is also split into two parts. The Block Editor can be seen in Figure 8. As always, the existing framework provides a base Block Editor ① with a text field for the question description ②. Additionally, we added a new button to load a background image ③. When clicked, it displays a popover

where exam authors can provide the background image URL ④. Due to the large number of modifications required for a Konva-powered image editor, we extracted the rest of the Block Editor functionality into a separate Vue component named *Gap Image Question Editor (GIQ Editor)* ⑤. It contains a Konva *stage* with two layers ⑥, as well as other UI controls to modify, insert, and remove the gaps (⑦ through ⑪). Their detailed functionality is explained later in this section. The GIQ Editor is only shown after a background image is successfully loaded from a URL. The image is then subsequently placed into its dedicated layer of the stage.

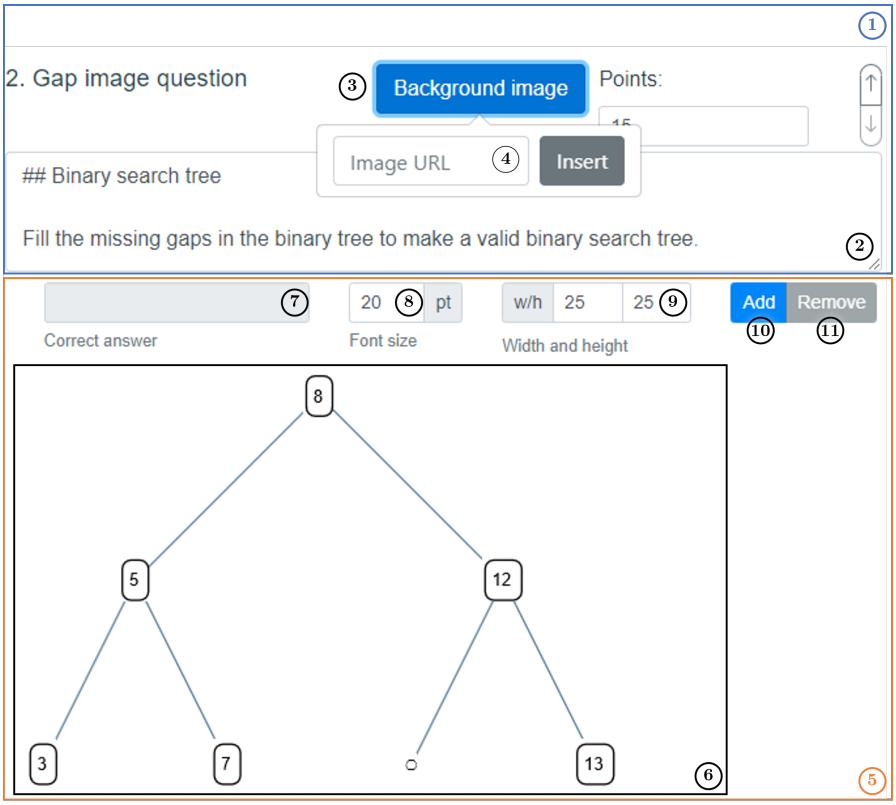


Figure 8: Block Editor of the Gap Image Question.

Gap representation. To represent the gaps, we introduced the concept of *slots*. An example question with two slots can be seen in Figure 9. One slot, with the assigned correct answer "5" is located in the left subtree of the binary tree. The second slot, to which no correct answer is assigned (as indicated by the "-" character), is the leftmost leaf of the binary tree.

A slot consists of a Konva *node* (called *slot node*) and a corresponding TypeScript object (called *slot object*). The nodes are Konva elements that natively support event handling and can be interacted with by the user, however, they only exist in the DOM[6]. A slot object contains the information that describes a slot node, such as its placement on the canvas grid via x and

[6] *Document Object Model*: a cross-platform and language-independent interface which operates on an HTML or an XML document by viewing it as a tree of objects.

y coordinates, as well as an optional correct answer that is associated with the slot. The slot objects are stored in a TypeScript array. When a slot node is updated by an exam author – for example, when new slots are added or the position of existing ones is changed – the corresponding object is updated. This allows us to store the configuration of the question in the database.

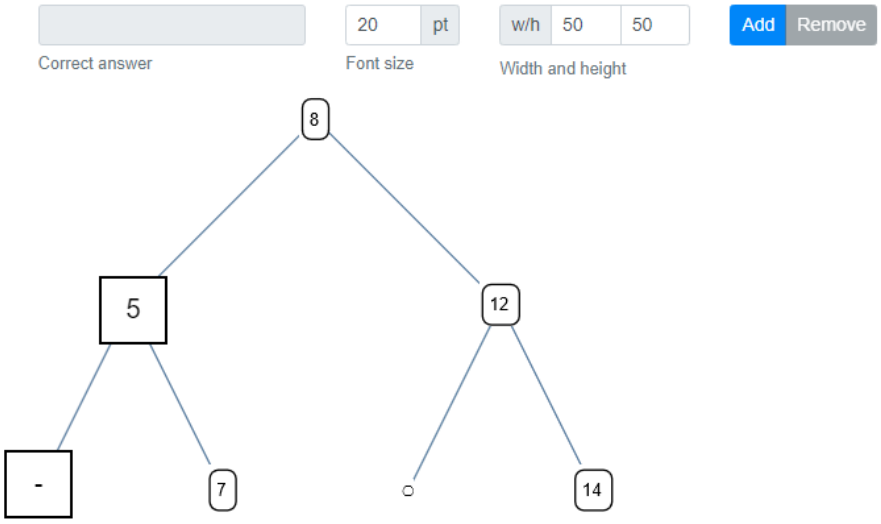


Figure 9: Gap Image Question Editor with a loaded background image and two slots.

A new slot node can be created at the $x = 20$ and $y = 20$ grid coordinates by pressing the "Add" button in the toolbar (see (10) in Figure 8). Alternatively, the exam author can click anywhere on the image to create a new slot node at the coordinates of the mouse click. A tooltip appears when the mouse is hovering over the "Add" button and provides a hint to this functionality. GIQ Editor creates all slots on the second layer of the image. The nodes can be moved freely within the canvas by dragging them with a mouse. Slots can be deleted by selecting them and either by clicking the "Remove" button (see (11) in Figure 8) or by pressing the ⌦ key.

Slot design. All slots of a block are uniform, i.e., they have the same height and width. This is done to prevent cheating, as students otherwise might be able to figure out the correct answers by just looking at the shapes of the slots. For the sake of a consistent design, the font size of all correct answers in the slot nodes is uniform as well. Exam authors can change these parameters using form inputs for the font size (see (8) in Figure 8) and for the width/height of slots (see (9) in Figure 8) in the GIQ Editor toolbar. Additionally, lecturers can alter the size of all slots by selecting any slot node with the mouse and transforming it by dragging the points on its frame. A selected node can be seen in Figure 10. While Konva supports the rotation of objects, we disabled this functionality in our implementation, as it is not required for our task. The slot settings are stored in the database as a part of the question's configuration, and are unique for each Gap Image Question block.

Additionally, the exam author can assign a correct answer to any slot. To add or modify it, they need to first select the slot node and then change the value in the respective form input in

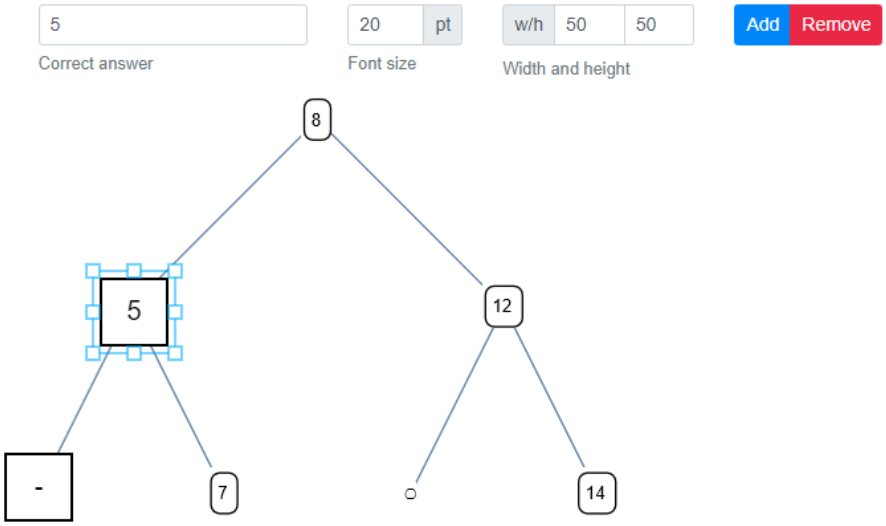


Figure 10: Gap Image Question with two slots. The slot with the correct answer "1" assigned to it is selected.

the toolbar (see ⑦ in Figure 8). By default, no correct answer is provided, which is represented by the GIQ Editor with a "-" character.

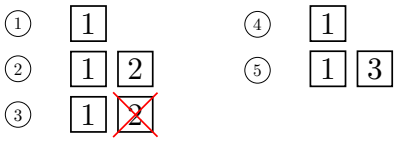


Figure 11: Slot ID behavior.

Slot IDs. Upon creation, each slot receives an integer ID which is unique within the exam block. It is never reused, even if the slot was deleted. This behaviour is visualized in the example in Figure 11: First, we create two new slots in steps ① and ②. Then, the second slot is deleted in step ③, leaving only the first slot (step ④). As seen in ⑤, if we then add another slot, it is assigned the ID 3. This is done to reduce the complexity of ID management, and we should never run out of new IDs for slots as there can be over $9 * 10^{24}$ of them per block (the largest possible integer value in TypeScript).

4.2.2 Block Visualizer

To visualize Gap Image Questions, we introduced a new Vue component named *Gap Image Question Visualizer (GIQ Visualizer)*, which can be seen in Figure 12. We can use it for both the exam edit view as well as running exams, as it is designed in compliance with our Editor-Database-Visualizer model (see Figure 4).

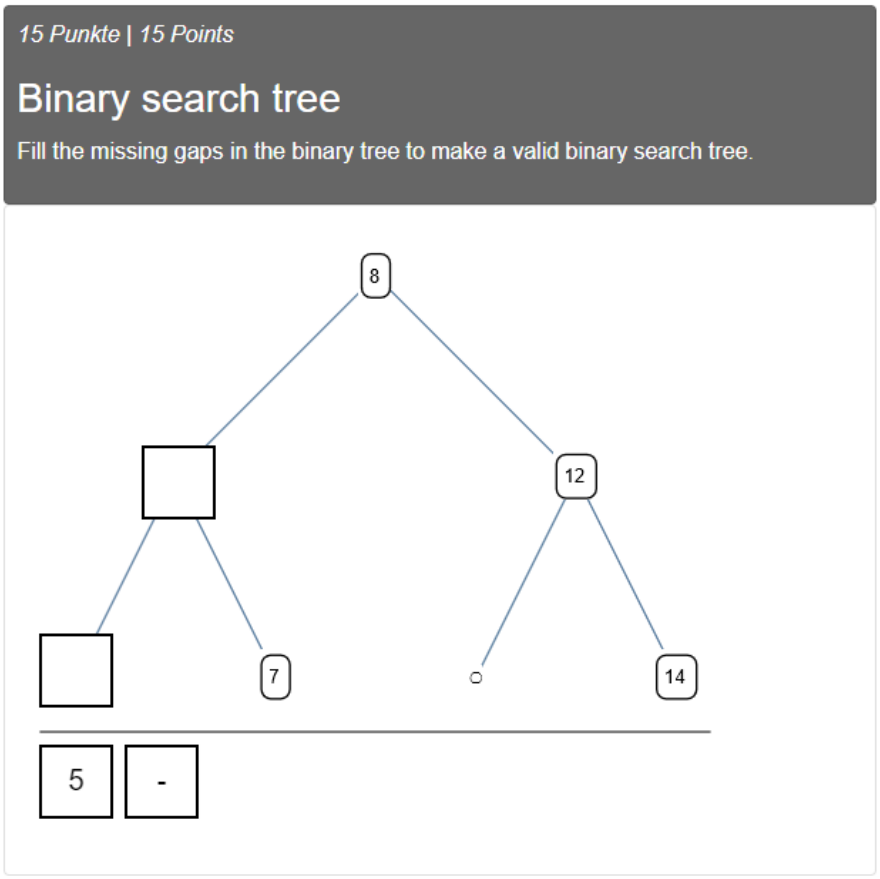


Figure 12: Gap Image Question Visualizer showing a Gap Image Question with two gaps that are represented by the two slots and the two tiles.

When the component is loaded, the GIQ Visualizer creates a new Konva stage with three layers. One layer is reserved for the background image, just as in the GIQ Editor. The second layer is reserved for a splitter line that separates the background image from the area where the answers are located. The third layer is reserved for the solutions of the questions and the slots. The display of the GIQ Visualizer follows the same logic as the display of the GIQ Editor: the component is only rendered after a background image is loaded from a URL.

Gap representation. When the component is rendered, the GIQ Visualizer fetches the list of slot objects and subsequently processes it. First, it creates one slot node for each slot object. The slot nodes in the GIQ Visualizer are functionally similar to the ones in the GIQ Editor. However, the correct answers associated with them are hidden from the students and the nodes are displayed empty instead, as shown in Figure 12.

Answer representation. Second, the GIQ Visualizer creates a *tile* for each slot object. A tile represents an answer to the Gap Image Question and is the only element of the question that students can interact with during an exam. The implementation of tiles is similar to the

implementation of slots: tiles are represented by a Konva node (called *tile node*) and a TypeScript object (called *tile object*) pair. Each tile has a unique ID that is created from the corresponding slot ID, and an answer that is created from the correct answer of the corresponding slot. We also reuse the slot configuration (width, height, and font size), and thus the look of tiles is the same as the look of the slots.

The tile nodes are placed in one or several rows below the image (see Figure 12). The number of rows and tile nodes per row is determined dynamically based on the width of the background image and the width of the tiles themselves. Most importantly, the order of the tiles is shuffled every time that the component is loaded. We use a custom implementation of the Durstenfeld shuffling [3], an improvement upon the Fisher-Yates shuffle algorithm that was popularized [12] by Donald E. Knuth in his 1969 book "The art of computer programming", as well as its subsequent editions [5]. The algorithm is given in Listing 2. This makes it impossible for students to guess the the correct answers by studying the order of the tile nodes.

```
shuffle<T> (array: T[]) {
  for (let i = 0; i < array.length; i++) {
    const j = Math.floor(Math.random() * (i + 1));

    [array[i], array[j]] = [array[j], array[i]]
  }

  return array
}
```

Listing 2: Our TypeScript implementation of the Durstenfeld shuffling algorithm.

User interaction and UX. Just as exam authors can drag a slot node during the exam creation, students can drag tile nodes to any point of the Konva canvas using the mouse. At the end of the tile node transformation, the new coordinates of the tile are updated in the corresponding tile object.

Furthermore, if at the end of the transformation more than 30% of the tile node's area overlap a slot node, we say that the tile *overlaps* a slot. In such a case, the GIQ Visualizer will automatically align the tile node perfectly with the slot node. To the student, the tile node appears as if it would "snap" into the slot node. This is essential for user experience, as it provides students with a clear feedback that their answer was registered by the system. If a tile node snaps to slot node, we say that the tile *attaches* itself to the slot or that the two are *attached*. The GIQ Visualizer keeps track of attachments of tiles to slots, as they represent the students' solution to the Gap Image Question.

To enable the "snapping" behavior, we first find the slot node that is closest to the tile node that was moved by the student. Then, we compute the overlap of the two nodes, and the system takes one of the following actions:

1. If the slot S and tile T overlap, and the slot S has no tile attached to it: The tile node T attaches itself to the slot S.

2. If the slot S and tile T_1 overlap, and the slot S has a tile T_2 attached to it: The GIQ Visualizer first detaches the tile T_2 from the slot S and returns it to its original position below the image. After that, the GIQ Visualizer attaches the tile T_1 to the slot S.

3. If the slot S and tile T do not overlap: The GIQ Visualizer returns the tile T to its original position below the background image.

After a tile is attached to a slot, an event that updates the block's answers is fired. The answers are stored as an array of TypeScript objects, where each answer object contains the slot ID, the tile ID, the answer represented by the tile and the correct answer assigned to the slot. In a similar fashion to Gap Text Questions, Xaminer stores the results in its database upon submission by the student.

5 Evaluation of the user experience

In this section, we review the new question types and evaluate their user interfaces based on a typical workflow.

5.1 Gap Text Questions

Question Creation. Exam authors can add new Gap Text Questions to an exam by selecting the respective entry from the list of options in the "block types" dropdown menu (see Figure 13).

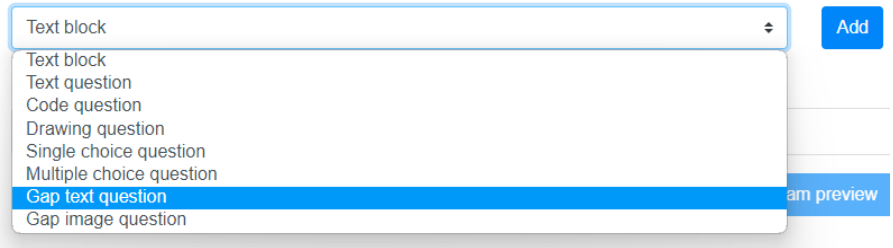


Figure 13: Dropdown menu with the list of all block types that can be added to an exam. Gap Text Question block is highlighted.

After the question type is selected, a Block Editor and a Block Visualizer appear on the screen, as can be seen in Figure 14. The first `textarea` element of the Block Editor is present in the base Block Editor, thus, exam authors will automatically be aware of its its purpose to provide the question description. Additionally, a second `textarea` element for the template text is displayed. A *default template text* is always provided by the platform when the question is added to an exam. This helps convey the purpose of the `textarea` element.

The default template text can be seen in detail in Listing 3. An Inline Gap, a Multiline Gap, and a Multiline Gap with the special $!x\#y\#!$ notation are present and provide a hands-on example on how exam authors can specify the gaps. Comments detailing further specification of the template text are also included.

We determined that there are two potential workflows to provide the template text. First, exam authors can start writing the text and can manually open and close the gap tags by typing the character sequences. This allows for a continuous workflow without requiring exam authors to lift their hands from the keyboard. Second, they can copy an already existing text and paste it into the template text `textarea` element. Exam authors can then insert the opening and the closing tags by selecting a text passage with the mouse and marking it as a gap by clicking the "Toggle gap" button. The second approach provides a workflow that solely relies on interactions with the mouse. Both workflows enable a smooth experience for exam authors, allowing them to edit the template text without switching the input devices. This helps to increase the productivity.

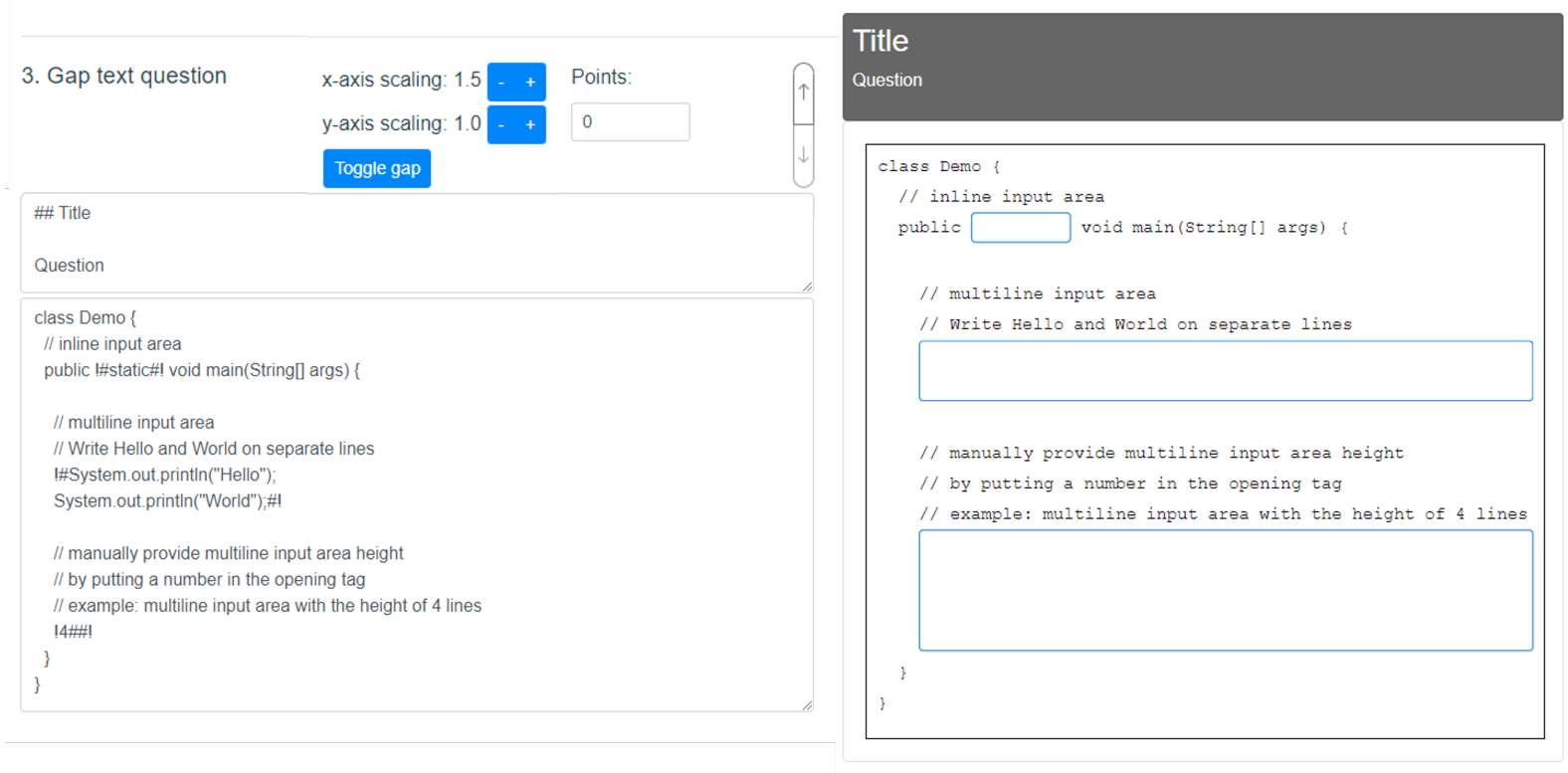


Figure 14: A Gap Text Question with the default template text.

```
class Demo {
  // inline input area
  public !#static#! void main(String[] args) {

    // multiline input area
    // Write Hello and World on separate lines
    !#System.out.println("Hello");
    System.out.println("World");#!

    // manually provide multiline input area height
    // by putting a number in the opening tag
    // example: multiline input area with the height of 4 lines
    !4##!
  }
}
```

Listing 3: Default template text.

However, the current approach is best suited strictly for programming tasks, and quickly reveals its limitations when it comes to freeform text. A standard text paragraph typically consists of a single line which wraps when the edge of the text editor is reached. This allows the text to remain within the bounds. In prose text, a manual line break is generally only present between paragraphs, in contrast to program code where each statement is typically written on a separate line. If we provide a long single-line text as a template text, the Gap Text Question Visualizer will create an SVG image that is potentially wider than the viewport[7], while having a height of one line.

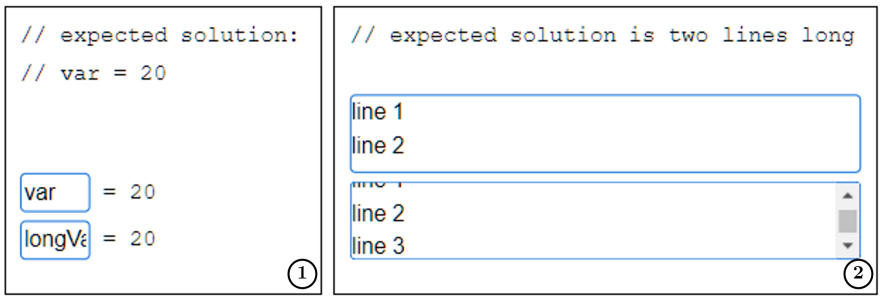


Figure 15: ① The first solution does not exceed the provided horizontal space in an InlineGap, the second one does. ② The first solution does not exceed the provided vertical space in a MultilineGap, the second one does.

Exam process. While taking an exam, the students can fill the gaps in any order. Students can expect that the space for their answers should always be sufficient, and a student answer exceeding the provided space could indicate a potentially wrong answer.

[7] Area of webpage visible to the user.

Should the length of their answer in an `input` element outgrow the space that was provided by the exam author, parts of it will automatically be hidden by the HTML element (see Figure 15). The students are then required to use the keyboard arrow keys to navigate and edit their input.

The same limitation applies to the `textarea` elements, albeit with regard to the text's height as opposed to its width. Should a student's solution exceed the length of the desired solution of the exam author, a scroll bar will appear on the side of the `textarea` element. The scroll bar and the arrow keys allow students to navigate their solutions (see Figure 15).

5.2 Gap Image Questions

Question Creation. The option to create a new Gap Image Question can be found among the other question options, similarly to Gap Text Questions (see Figure 16). When the Gap Image Question's Block Editor is shown, a part of its user interface is hidden until a background image is provided. At this point, the "Background image" button is the only visible part of the user interface aside from the question description `textarea` element. Its prominent color and the lack of other UI elements help to convey to the exam author that it must be interacted with in order to proceed with the question creation.

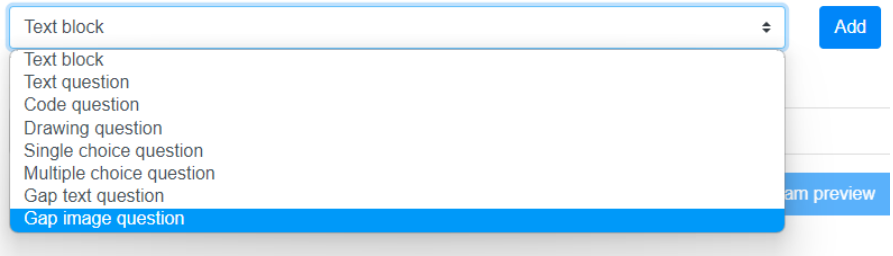


Figure 16: Dropdown menu with the list of all block types that can be added to an exam. Gap Image Question block is highlighted.

After the background image is loaded, the rest of the user interface becomes accessible (see Figure 17). Exam authors with previous experience of using an image editing software will find familiar functionality in the Gap Image Question Editor. They can select, move, and resize the slots using a standard user interface. The toolbar functionality is explained using labels and captions that describe each of the UI elements. Using a tooltip that appears when the mouse is hovering over the "Add" button, exam authors are made aware that they can add gaps to the background image by clicking anywhere in the canvas.

Exam process. The user interface that students interact with during the examination process is straightforward and familiar, as they only need drag & drop to place the tiles onto the slots. This workflow is intuitive and can be found in both other examination platforms as well as computer operating systems in general. This means that platform operators do not need extra trainings for the students.

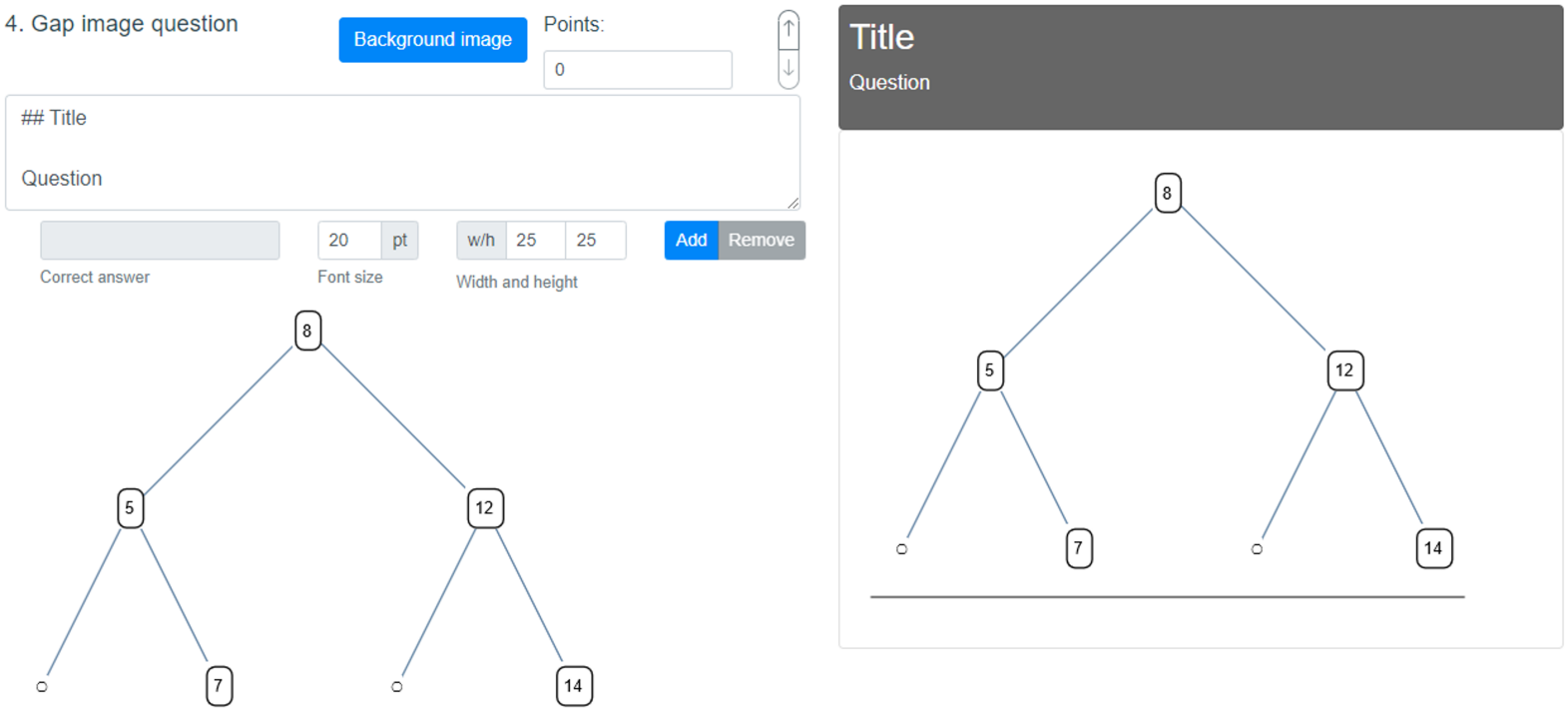


Figure 17: A Gap Image Question with a binary search tree loaded as a background image.

6 Future Work

In this section, we outline possible improvements to the new question types.

During the later stages of development and the evaluation of the questions, it became apparent that several improvements to the two question types can be pursued in future works.

6.1 Gap Text Questions

Handling long lines. Our current approach relies on exam authors to provide line breaks for the Gap Text Question Visualizer to display the text on a new line. Should they wish to provide a long text without a line break (e.g., a paragraph from a book), the created SVG image will likely exceed the width of the viewport. In order to avoid that, we could introduce a maximum SVG width setting. It would cause the GTQ Visualizer to move the text to a new line despite the parser not reading a line break. This would allow the Gap Text Questions to be more versatile. One good example for such lectures are language beginner courses, where paragraphs of prose text with some words replaced by gaps can be commonly found.

Code highlighting. Our text parser in the Gap Text Questions is built similarly to the parsers that are utilized in code compilers. It would thus be possible to implement their content-awareness features to enable syntax highlighting. Each word in the text is already viewed as a token of a certain type; therefore, it could be compared against an existing dictionary of keywords of a programming language – keywords such as "public" or "if" or "for". They could then subsequently be marked as such by introducing several new keyword tokens. The GTQ Visualizer then would be able to recognize a keyword tokens and highlight them according to a predefined color scheme.

Alternatively, code highlighting could be enabled using the functionality of the code highlighting system that is already used by Xaminer's Code Question question type. This system could further be used for simple syntax checking of keywords in gaps. Typos (for example, `pblic` instead of `public`) would not be recognized as valid keywords, and the lack of code highlighting for the keyword would hint at a typo.

Code highlighting could allow students to understand the provided code more quickly. It would also eliminate a potential source of unnecessary errors by making students aware of typos.

Custom font size and line height. Currently, we compute the dimensions of the SVG image using the default values for the font size and the line height. We could implement a setting to modify the two parameters. It would require the addition of two new control elements to the Gap Text Question Block Editor toolbar. Furthermore, we would need to store the parameters in the database as a part of the question's configuration. This would give exam authors more options to control the look of the question and slightly improve the accessibility of the user interface by allowing exam authors to make the text larger. However, as the Gap Text Question Visualizer relies upon native HTML elements such as `svg`, `text`, and `foreignObject`, the user interface already supports scaling to some degree using the Web browsers' zoom in/zoom out functionality.

6.2 Gap Image Questions

Uploading of background images. Currently, the background images must be hosted on a separate web server. In the future, it should be possible to store images directly in Xaminer's database as a part of an exam configuration. In order to do so, we would have to modify the "Background image" button to invoke the operating system's file dialog to select an image. Subsequently, we would have to upload it to the database. Furthermore, this would require the creation of several functions to handle database access.

Adding this feature would remove the need to rely on third-party servers. Furthermore, it would make the platform more reliable, as right now the Gap Image Questions component is vulnerable to the server that is hosting the image being inaccessible or the image file being deleted.

Quality-of-life improvements to slot creation. We have determined several possible improvements to slot design:

1. To position a slot, exam authors currently must select it with the mouse and drag it to the desired position. While this is the fastest and most intuitive approach, it is also imprecise, as it is impossible to create a pixel-perfect alignment of slots. We could remediate this by adding two more form inputs to the Gap Image Questions Editor toolbar, which would display the x and y coordinates of the current selection on the grid. Exam authors would then be able to select a slot and set its exact position by providing the exact values for the coordinates.

2. Additionally, it would be beneficial to include support for arrow keys as an alternative way to move slots.

3. Furthermore, we could improve the efficiency of exam authors by allowing them to select several slot nodes simultaneously, as they would be able to move them all at once.

4. Finally, a slot axis-snapping mechanism could be implemented. It would automatically snap the selected slot to the closest x or y coordinate that other slots reside at, in a way similar to how image editing software programs function. This would allow for faster and preciser slot placement in rows and columns, as exam authors would not need to select several slots and manually provide coordinates for each of them.

7 Conclusion

The introduction of two new question types greatly increases the versatility of the Xaminer platform. It also simplifies exam creation and students' workflow during an exam.

Gap Text Questions allow lecturers to save a lot of time when designing the exam, as they can provide a succinct template text instead of being required to write a long question description. At the same time, the new question type gives students more time during exams, as they are not required to read these long question descriptions. Finally, it removes a source of possible misunderstandings between the exam authors and students.

Gap Image Questions allow Xaminer to target new lectures and exam types. Exam authors can now create questions that could not easily be answered just with text.

Overall, our changes provide lecturers with more flexibility in terms of question design. At the same time, students are enabled to better demonstrate their knowledge during exams by the new intuitive question types.

List of Tables

1 List of programming languages, query languages, and hardware description languages taught at JKU Linz with their respective comment indicators. 11

List of Figures

1 An exam with a text block and a single choice question block, as seen during exam creation. 4
2 An exam with a single choice question block and a multiple choice question block. "Vienna" has been marked as the correct answer to the single choice question. "Paris" and "Rome" are marked as the correct answers to the multiple choice question. 5
3 ① A single choice question rendered during the exam creation (exam author view). ② A single choice question rendered during the exam (student view). 6
4 Block Editor - Block Visualizer - Database model. 7
5 A mockup Gap Text Question. An Inline Gap and a Multiline Gap are specified in a paragraph of text. 8
6 ① Block Editor and ② Block Visualizer of a Gap Text Question with the default template text. ③ Question description `textarea` element. ④ Template text `textarea` element. ⑤ x-axis-scaling factor controls. ⑥ y-axis-scaling factor controls. ⑦ Toggle gap button. 10
7 A mockup Gap Image Question. Three gaps are marked in the image. Three answers are located below the background image. 17
8 Block Editor of the Gap Image Question. 18
9 Gap Image Question Editor with a loaded background image and two slots. . . . 19
10 Gap Image Question with two slots. The slot with the correct answer "1" assigned to it is selected. 20
11 Slot ID behavior. 20
12 Gap Image Question Visualizer showing a Gap Image Question with two gaps that are represented by the two slots and the two tiles. 21
13 Dropdown menu with the list of all block types that can be added to an exam. Gap Text Question block is highlighted. 24
14 A Gap Text Question with the default template text. 25
15 ① The first solution does not exceed the provided horizontal space in an InlineGap, the second one does. ② The first solution does not exceed the provided vertical space in a MultilineGap, the second one does. 26
16 Dropdown menu with the list of all block types that can be added to an exam. Gap Image Question block is highlighted. 27
17 A Gap Image Question with a binary search tree loaded as a background image. 28

Listings

1 Two Wolfram Language/Mathematica functions. In `In[1]`, we apply the function $3! * x$ to every element x of a list. In `In[2]`, we apply the factorial operation to every element of a list. 12
2 Our TypeScript implementation of the Durstenfeld shuffling algorithm. 22
3 Default template text. 26

References

[1] BootstrapVue. https://bootstrap-vue.org/. Accessed: 2022-07-22.

[2] Daring Fireball. Daring fireball: Markdown. https://daringfireball.net/projects/markdown/, 17-Dec-2004. Accessed: 2022-07-22.

[3] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM*, 7(7):420, 1964.

[4] JetBrains s.r.o. Kotlin programming language. https://kotlinlang.org/. Accessed: 2022-07-22.

[5] D. E. Knuth. *The art of computer programming*. Addison Wesley series in computer science and information processing. Addison-Wesley, Reading, Mass., 2. ed., 3. print edition, 1977.

[6] Konva.js. Konva.js - JavaScript 2d canvas library. https://konvajs.org/, 2015. Accessed: 2022-07-22.

[7] Microsoft Corporation. TypeScript: JavaScript with syntax for types. https://www.typescriptlang.org/, 24-Jun-22. Accessed: 2022-07-22.

[8] MongoDB. MongoDB: The developer data platform | MongoDB. https://www.mongodb.com/, 11-Feb-2009. Accessed: 2022-07-22.

[9] Moodle HQ. Moodle - open-source learning platform | Moodle.org. https://moodle.org/. Accessed: 2022-07-22.

[10] VMware, Inc. Spring Boot | Spring.io. https://spring.io/projects/spring-boot/, 01-Apr-14. Accessed: 2022-09-27.

[11] Web Hypertext Application Technology Working Group. HTML standard. https://html.spec.whatwg.org/. Accessed: 2022-07-22.

[12] M. Yadav, V. Shokeen, and P. K. Singhal. Testing of Durstenfeld's algorithm based optimal random interleavers in OFDM-IDMA systems. In *2017 3rd International Conference on Advances in Computing,Communication & Automation (ICACCA) (Fall)*, pages 1–4. IEEE, 9/15/2017 - 9/16/2017.

[13] E. You. Vue.js - the progressive JavaScript framework | Vue.js. https://vuejs.org/. Accessed: 2022-07-22.