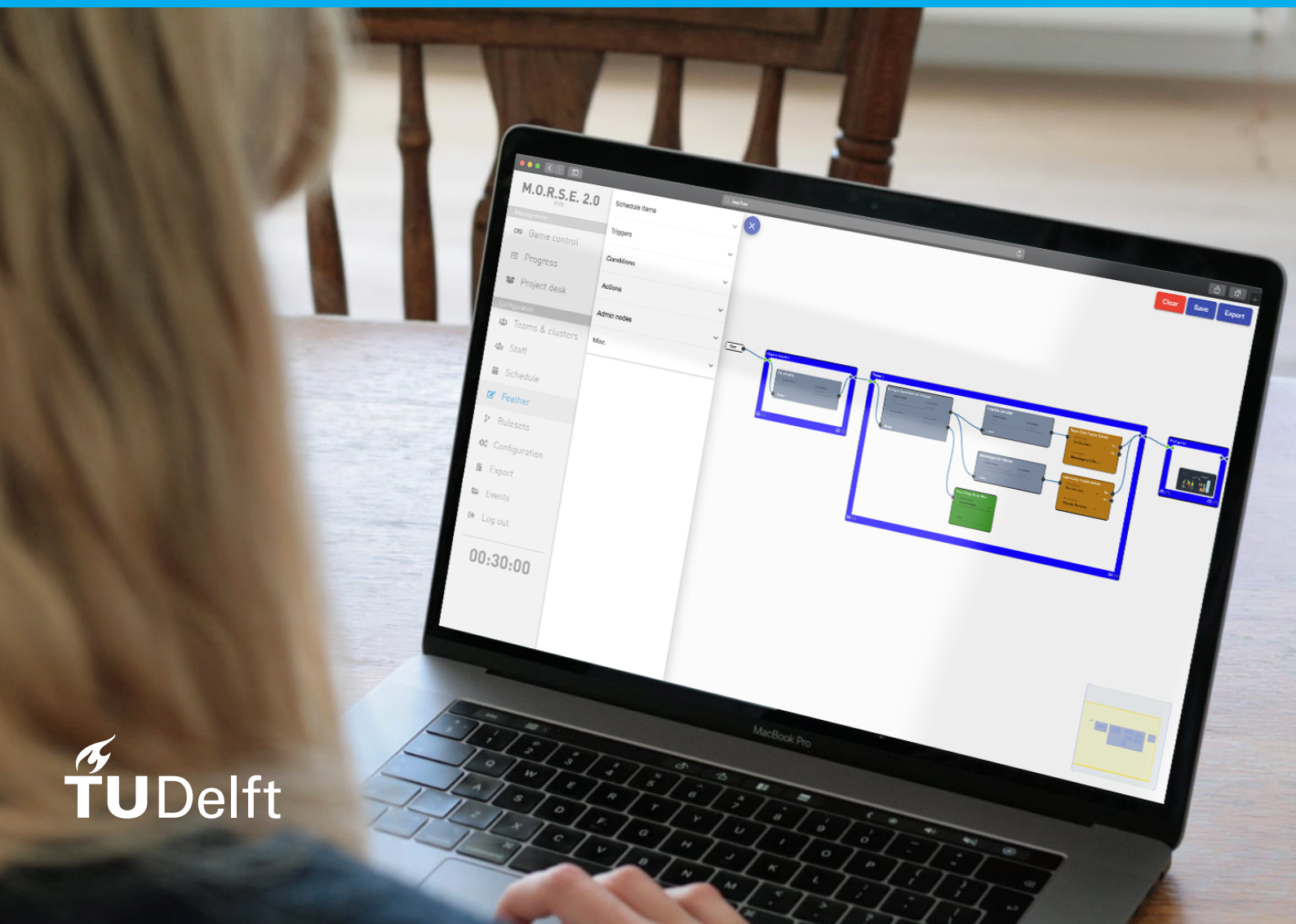
FEATHER: Visual Editor for Escape Rooms

The Software behind Escape Room Games

Eben Rogers
Siert Sebus
Wouter Polet
Yana Angelova
Yoshi van den Akker



TU Delft

FEATHER: Visual Editor for Escape Rooms

The Software behind Escape Room Games

by

Eben Rogers
Siert Sebus
Wouter Polet
Yana Angelova
Yoshi van den Akker

to obtain the degree of Bachelor of Science
at the Delft University of Technology

Project duration:	April 20, 2020 – July 1, 2020	
Thesis committee:	Dr. H. Wang,	TU Delft, coordinator
	Ir. O. W. Visser,	TU Delft, coordinator
	Ir. T. A. R. Overklift Vaupel Klein,	TU Delft, coach
	J. W. Manenschijn BSc,	Raccoon Serious Games, client

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Preface

This is the Bachelor Thesis created by Eben Rogers, Siert Sebus, Wouter Polet, Yana Angelova, and Yoshi van den Akker documenting their Bachelor End Project (shortened to BEP). The BEP solves a problem of Raccoon Serious Games with respect to designing digitally based escape events using their M.O.R.S.E. 2.0 system. As the solution to this problem, the team presents a visual editor to give a better overview of the underlying logical rules of such events and to make it easier to configure these rules.

We would like to thank our supervisor Thomas Overklift for guiding us during this project and providing feedback whenever possible.

We would also like to thank Jan-Willem Manenschijn for giving us the opportunity to work on this product and everyone at Raccoon Serious Games for the warm welcome we received despite the unusual situation.

Eben Rogers
Siert Sebus
Wouter Polet
Yana Angelova
Yoshi van den Akker
Delft, June 2020

Contents

Summary

Raccoon Serious Games develops and hosts educational activities such as escape room events and serious games. They create both physically- and digitally-based escape rooms across many different scales. These events consist of a variety of puzzles and tasks the player(s) have to solve in order to finish or 'escape' the event.

For their digitally hosted events, the Massive Online Reactive Serious Escape 2.0 (MORSE) system is used for creation and configuration of the needed underlying rules of the event. The system uses the 'If This Then That' (IFTTT) principle for creating rules, where a trigger activated by the player/game can initiate a check about the state of the game which then results in an action by the game. In MORSE the user (usually the game host) can choose from the multiple types of triggers, conditions, and actions to create logical statements in the IFTTT format. These statements together form the rules of the game.

This system, although a good improvement over the previously hard-coded procedure, has proven unintuitive to program for most of the employees at Raccoon Serious Games. The IFTTT format used is unwieldy to work with for the designers, who have little to no programming background. Furthermore this existing system provides no overview of the rules system making it challenging to visualise the whole game and its dynamics.

To solve the unintuitive nature of MORSE, our team designed and developed Feather: A graph-based visual editing tool that is integrated into MORSE. It can generate rule and ruleset logic needed for the client's escape events. It uses visual components and presents the user with a graph of the whole game during the design process. The editor can be used together with all other, earlier existing, features for creating rulesets of the MORSE system. This tool has most of the functionality the current system has, with the possibility of easily extending it with new components.

The product was built as an addition to MORSE over the course of 10 weeks. In the initial part of the project a thorough research was performed on the needs of the client as well as useful resources or libraries and design practices for domain specific visual languages. The second part of the project was devoted to the design and implementation of the tool. Throughout the duration of the project a number of user tests were conducted with the employees at Raccoon Serious Games to assess the understanding and usability of the product.

1 Introduction

Serious games are fun activities with an educational twist. In recent years these types of activities have gained popularity for use in team building activities. These events can be hosted both in the physical world as well as online. For physical events digital devices can be used to allow for higher scalability.

Raccoon Serious Games (RSG), the client of this project, provides different types of serious games with different scales and domains. For the creation of their digitally based escape events they have developed the Massive Online Reactive Serious Escape 2.0 (MORSE) system. This system employs a rule-based configuration in which the 'If This Then That' (IFTTT) style is used. Unfortunately they have found the system's usability to be below their expectations, as it requires too much technical background knowledge to. This Bachelor End Project (BEP), as part of the Bachelor of Computer Science and Engineering at Technical University Delft, addresses these issues.

The system designed and developed in for project is an extension of the existing MORSE system. It allows the user to visualise the capabilities of the old system through a visual drag and drop editor, implementing most of the functionalities present in the currently used one as well as extending them further.

This report will outline the process and product developed during the 10 weeks of this BEP. In chapter 2, the main problem is discussed as well as its underlying challenges and the proposed solutions used for this project. Chapter 3 describes the design and design principles used for the creation of the new system. Following that, the implementation details are presented in chapter 4. The system linking the visual editor with rules in MORSE is be discussed in chapter 5. In chapter 6 the main quality assurance techniques, that were used throughout the project to ensure the quality of the product developed, are discussed. Chapter 7 gives proper insight into the process during the 10 weeks, the difficulties that were encountered, as well as a reflection on the experience as a whole. Finally chapter 8 and chapter 9 conclude this thesis and provide discussion on maintainability, ethical implications and future improvements of the project.

2

Problem Definition and Analysis

This chapter will discuss the general problem our client, Raccoon Serious Games, was facing before the start of this project and the main challenges that were tackled in the process of finding a solution. Furthermore, the proposed solution will be explained as an introduction to the design and implementation, which will be discussed in the following chapters.

Previously the creation of digitally based escape rooms heavily depended on one of the co-founders of RSG, who has a computer science background. This was due to that fact that the existing MORSE system was designed under the assumption that the user can use the IFTTT structure easily to create rulesets, without any knowledge of programming. However, this design was found not to be user-friendly for the designers in the company. For this reason they did not use it, and only the aforementioned co-founder could configure the escape rooms. This was a hindrance to the workflow of our client, which is why this project was started.

To solve this problem we propose the following question, that will be answered throughout this thesis: **"How can we design and implement an integrated User Interface (UI) that facilitates the configuration of events in MORSE, which is usable by users without a computer science background or similar, while maintaining all of the functionality that the current editor has?"**. From the start it was clear that the new UI would have to represent the event in a more visual manner. This, in combination with the main question, poses the following sub-questions:

- How do we integrate the new UI in MORSE, such that it can be used as another page in the admin view of MORSE?
- How do we make sure that the visual representation can be used to configure the escape events of MORSE?
- How do we preserve all the functionality that the current editor of MORSE offers?
- How do we create a UI that fits the workflow of RSG employees and that is usable by users without a technical background?

The task of this project is to address all questions posed above and find the best solution possible that fits the client's needs.

Before moving on to the challenges and proposed solutions, we will give a small introduction of MORSE to give a clearer introduction to the context of the project. This introduction only describes the parts that are relevant and needed to understand the problem and proposed solution of this project.

MORSE is used to run the escape events that RSG creates. In such an event multiple teams try to solve their puzzles at the same time. Each team has a tablet that is used to display certain screens, which contain either text, puzzles, or images. These screens are called "schedule items", which have their own editor where they can be added, removed, and configured. Between these schedule items there is logic that should be configured to perform the right actions, based on certain occurances during the event. Examples of this are moving teams to another screen when they answer a puzzle correctly or displaying a textual hint when there is one hour left on the game timer. Configuring this logic is done in the "Ruleset editor". Each rule follows the IFTTT structure, consisting of a number of triggers, conditions, and actions.

2.1. Challenges

The creation of such a product comes with its challenges. In this section the main obstacles will be discussed as well as the measures taken to overcome them during the project. We identify three main challenges that we had to tackle and give importance to for the successful completion of the project. These will be explained in more detail further in the rest of this section.

2.1.1. Integration with MORSE

One crucial aspect of the solution is its integration in MORSE. MORSE is the software system that provides all the functionality needed to run the large scale escape events offered by RSG. This project does not change any of the functionalities the system currently offers, but only the way in which the escape events are configured. The solution should become part of the existing system, rather than being an additional program that communicates with MORSE. This integration limits the choice of frameworks and languages: MORSE is created in Typescript with the Meteor and Angular frameworks. Therefore the product has to be written in Typescript using, or at least remaining compatible with, these same frameworks.

Additionally, the changes that can be made to MORSE are limited, as another group of bachelor students worked on it in the same time period. At the end, the implementations of both BEP groups should be compatible with each other and work together. While this limits the ability to change the current system, it also introduces new opportunities. The other BEP group will be extending the functionality of MORSE, by adding new configurable schedule items and rules. Of course, the logic behind these new schedule items has to be configured. Therefore, nearing the end of the project support for the other group's work can be added to our product.

2.1.2. Visual Representation and Rules

A visual representation differs greatly from a textual representation. In MORSE, the configuration is performed using a large amount of separate IFTTT-style rules. These rules start with a trigger, something that happens in the escape event, optionally followed by a number of conditions, which all have to be true for the rule to actually execute, and finally the rules end with a number of actions. These actions make something happen in the escape event. This IFTTT structure can be amazingly powerful and expressive.

The first challenge following from the textual configuration is maintaining the configurability of the escape event. Ideally the visual configuration would be as powerful as the textual one. With a configuration as modular as the IFTTT structure, the expressiveness poses a challenge that fundamentally influences the design of the visual editor.

The second challenge is a flipped version of the first one. The visual representation has to be turned into the rules that can be used by MORSE. While these rules offer a lot of functionality there are certain constraints they have to follow. In a visual graph, the customisability is typically higher than when certain defined parts of rules are added together. Having these extra options is great, as it also makes configuring an event easier, but they also introduce new risks. With so much freedom, it is easy to create invalid configurations that cannot be turned into sensible rules. In these situations the visual editor should limit the user in what they create.

Finding the balance between offering enough possibilities and limiting the user to maintain valid configurations is crucial and is something that was carefully considered during the research and design phase in section A.4. Limiting the user too much will hurt the expressiveness of the visual editor, making it less powerful than the current configuration. However, limiting the user too little will allow for invalid configurations that cannot be used by the system.

2.1.3. Usability

One of the big issues with the existing system is its usability. The tool was not well-designed for people who have little to no background knowledge in programming. This limitation led to a dependency on one specific person in the company for the configuration of the rules of an escape event. Furthermore, with the current design, the ruleset configuration of bigger events becomes a long list of rules, that are both hard to read and maintain. To resolve this, the main purpose of this project is aimed at the usability of the system by all employees. One of the main tasks was to create a product that will best fit for the designer's needs, ease their workflow, and allow them to be independent in the creation and configuration of escape rooms. The goal was to make the system easy to use, maintain, and understand. When talking about 'easier to use' we refer to the system's ability to visualise the workflow and the overall satisfaction of the client compared to the old system.

2.2. Proposed solutions

Although the IFTTT structure is very powerful, it is hard to visualize the bigger picture. To make the editing process as intuitive as possible, we proposed to build a visual editor.

At the start of the project, a rough project plan was created, which is located in Appendix D. Then a research phase of two weeks was conducted to get an idea on how to solve the problem that the client was facing. From this research phase, from which the report can be found in Appendix A, a more concrete plan for the editor was formed.

In this editor we want to keep the current triggers, conditions and actions, but connecting them with a drag and drop interface. Furthermore, to create an easier overview, we wanted to group the logic per-stage.

To be able to use the visual configuration, the editor should tie into the current MORSE system. Once the user has connected the triggers, conditions and actions, our system will construct rules from the graph the user created. These rules are put in the database directly, after which the escape room is ready to be used.

With this method the functionalities of the original MORSE system are left intact, as it is still usable next to the visual editor; Feather is built on top of MORSE. If the user desires, they can configure custom rules besides the ones configured in the visual editor.

At the end of the research phase a number of requirements for the proposed solution have been specified and approved by the client. We will not copy the entire requirement list here, but it can be viewed in section A.7 and section A.8. The lists account for both the functional and non-functional aspects of the product. With those specification in place, it was ensured that all of the above mentioned challenges and problem questions have been taken into account and resolved during the implementation of the product.

A notable distinction made in the requirement list are the general features for the editor and the “expressiveness” features. The first type refers to the features that the UI of the editor has. The second type refers to what can be configured using the UI. This distinction helps to keep track of the goal of maintaining the functionality of the UI.

The requirements are created following the MoSCoW method, providing a prioritisation. First, the highest priority features are described in the Must Haves. These are features that are needed for a minimal viable product. The must have contains basic features of the visual UI and the most commonly used, essential elements that should be configurable. In this version of the editor only the logic of the event can be defined.

Second the Should Haves are presented. These are features with medium priority and as the name suggest should be implemented. For the UI, some quality of life improvements are described, like being able to select multiple nodes at once and perform the same action on all the selected nodes. The should-have requirements also add the configuring of all other mainly used triggers, conditions, and actions that were not present in the must-haves.

Third, the Could Haves are written out. These are the features with the lowest priority and thus it is nice to have in the resulting system. These features should only be implemented when extra time is left, but are by no means necessary. Finally, the Won’t Haves specify the feature that will not be considered for the implementation.

Taking into account the MoSCoW requirements and the main challenges, to consider this project successful, the team should at least implement all must have requirements and most, but not necessarily all, should haves requirements. Furthermore, all the non-functional requirements, described in section A.8, should be adhered to.

3

Design

Our design goal was to create a visual language for defining escape rooms that *had* to be able to compile to the existing MORSE system for compatibility. We had several additional requirements on top of this base one that were more abstract and were related to functionality:

1. Using this new editor should be more intuitive than using the original ruleset editor that existed in MORSE, especially for users without a technical background.

2. The user should be able to define escape rooms to the same extent in which that was possible to do in the old system.

3.1. Initial Design Process

For our full research findings from the start of the project, see Appendix A. During out design process we found several existing examples of visual programming languages that we could look to for design principles:

1. **Pipes** was a great example of a flow-based data processing programming language that lent itself nicely to programming actions on a data stream *Flowgorithm* (2020). The concept of connecting nodes to each other and by doing so defining interaction between or a flow through these nodes is central in Pipes. This concept has been an important inspiration to our design.

2. **Kodu** was a visual programming language designed by Microsoft for children which had a very nice accompanying paper discussing all of the findings during the design process. One of the key principles extracted from this paper was the 'only show the user what they need' principle (e.g. don't overwhelm the user with UI elements) MacLaurin (2011).

3. **Scratch** is a programming language developed at MIT whose popularity has soared recently due to more intuitively laying out programming concepts. Its block-based programming visualisation did not quite speak to us due the fact that we wanted the user to be able to build a 'flow' of an escape room in the same way in which they might sketch one before programming it. However, it does have the flexibility and power that we ended up seeking to replicate.

4. **Flowgorithm** is very similar but yet less popular example to Scratch. Its power also spoke to use but its way of representing the code using a flowchart gave us a very good example of what we could work towards when making our own editor.

Furthermore we also took a look at the existing development methodology used by RSG to help guide us towards a design that fit nicely into their existing way of thinking about escape rooms. Some of our initial designs either too closely followed the existing system or deviated too far from it as can be seen in figure A.4 and figure A.5 respectively. Figure A.4 has three types of nodes, each corresponding to a different section in the MORSE system. Meanwhile, figure A.5 has no elements like that whatsoever.

As we iterated from one design to another the languages became a little bit less about making the design as user-friendly as possible and more towards something more realistic and implementable that more closely

represented the existing system. We primarily did this to scale the project more to the timeline that we had to implement everything (six weeks).

In the design that was set out to be implemented, a middle ground between the different approaches was found. The configuration of an escape room in MORSE was divided into two main parts: First, the schedule items, corresponding to the screens and puzzles that the teams would see, were created. Then, to add the logic to these schedule items, the rulesets would be added in the ruleset editor. During our research we noticed that the design process of escape rooms at RSG had these two elements intertwined. The proposed visual editor would therefore merge these two processes: the schedule items can be configured similar to the schedule items editor, but also be connected with each other to define how a player moves between them. Additionally, for each screen, rules can be defined in the visual editor, thereby replacing the ruleset editor. This design is visualised in figure 3.1 and further illustrated with an example in figure 3.2.

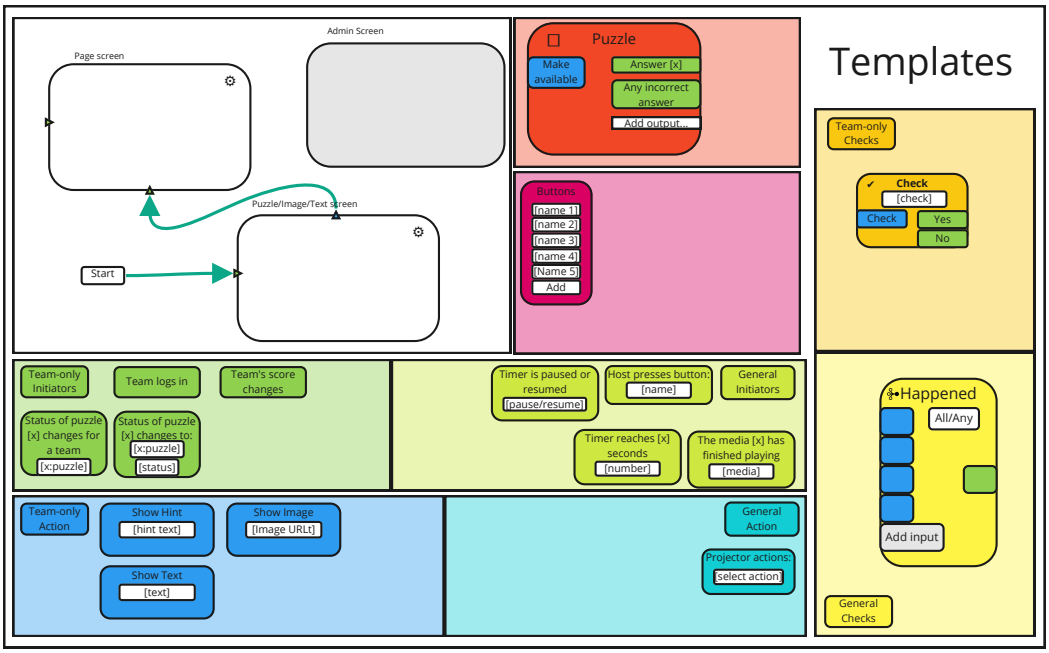


Figure 3.1: Initial design of the visual modelling language.

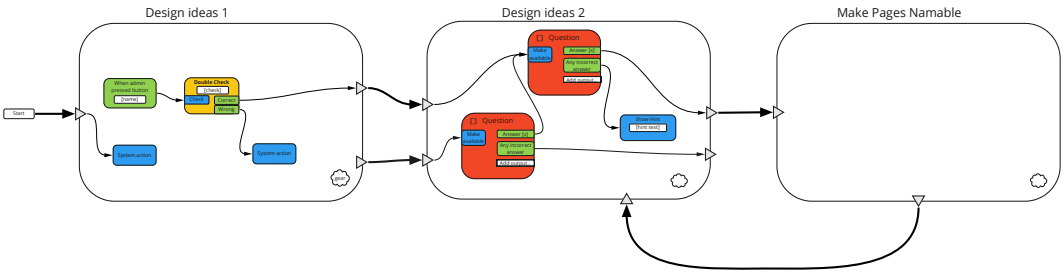


Figure 3.2: Example of a small configuration with the visual language.

Every big node in this visual language corresponds to a schedule item. These can be created and configured directly in the view. To switch to the configuration of the logic for a screen, a gear icon can be clicked. The screens are connected with each other over various so-called ports on their edges. The connections are represented by big arrows, to clearly visualise the flow of, i.e. the way a player progresses through, the escape room. A screen can have multiple incoming and/or outgoing ports, allowing for various ways of transitioning from one screen to another. These ports can be placed freely over the edges of the screen.

In each screen, its logic can be configured. This is done by connecting various nodes together. As can be noticed in figure 3.1 some of these nodes have fields, to define additional needed parameters. This can for example be a puzzle that has to be selected, or a number. These nodes and the way they are connected

follow the existing trigger, condition, action structure. A trigger can be followed by any number of actions. Between these actions, the conditions can be placed to stop the action from being performed if the condition is false. A concern for this approach is the usability, it might require technical knowledge to be comfortable with such a system. However, when performing user tests with mock-ups of designs, it became apparent that this is not an issue. The users seemed to grasp the idea of trigger-condition-action rather easily and managed to use it effectively throughout the rest of the user tests with the mock-ups provided. Another important factor for the decision to follow this structure is the feasibility of the implementation as pointed out by the client. His concern that a change in the underlying structure will we unfeasible in the 7 weeks allocated for implementation was a valid point that the team took into account when making this decision. Turning the visual configuration in the rulesets used by MORSE will be challenging, but with the same structure it is feasible.

Another aspect of the visual language, that followed from the integration with MORSE is a distinction between team only and general nodes. In MORSE certain condition and actions refer to a certain team. However, not every trigger concerns a certain team. For example a trigger that fires when the timer has paused does not have any team, while a trigger that fires when a puzzle has been answered, does. The team only nodes often offer more configuration and some team only nodes do not have a general version. To keep a valid configuration, one that can be exported correctly, a general trigger cannot be connected to any team only condition or action. The transitions between screens are considered to be team-only, as a certain team would be moved to another screen.

A big strength of this design is the ability to define the general flow of the escape room, without needing to specify the detailed logic or the exact properties of the screen. This is in line with the sketching guideline described by Resnick et al. (2005), which notes that a composition tool, like the visual editor, should allow for quick and rough sketching. The usefulness of being able to make sketches is further emphasised by the workflow of the designers at RSG. They start with a lot of rough sketches when creating an escape room, which are later worked out further.

3.2. Design Changes

Over the course of the project, the design of the visual language has been adapted to solve unforeseen issues or keep the implementation feasible.

First, a notable change is the removal of the configuration of the schedule items themselves. The editor only allows the configuring of the logic of the screens. Instead of defining the screens in the visual editor, they can be imported from the existing schedule items. This decision comes from the prioritisation made in the requirements. When devising the requirement list and general planning of the project, the client noted that the definition of the logic and general flow is more important than the configuration of the screens themselves. At the end of the project no more time was left to implement the configuration of the properties of the screens.

Second, the visuals of the nodes have changes considerably. The main cause of this change is the switch of the framework used for the visualisation. The reason for the switch itself is explained in section 4.2. The design of the nodes was mostly based on what the initial framework offered and the extend to which the nodes could be changed in the editor. Even though the nodes look different, as can be seen in section 3.3, their purpose and general structure remain the same.

Finally the freedom offered in the editor has been restricted throughout the project. As the compiler from the visual configuration to the MORSE rules was coming together, it became clear that the editor had to be restricted in some ways. There are two main ways to restrict the editor. One way is to allow invalid configurations, but add a visual indicator that this configuration is faulty (for example by displaying an error message). The other way is to disallow configuration to become invalid. In the research we found conflicting ideas on what way would be better. Sketching is easier when invalid configurations are allowed, following one of the guidelines by Resnick et al. (2005), but MacLaurin (2011) argues that the tool is more accessible when no syntax errors can be made at all. As stated earlier sketching is very important in the workflow that we create the tool for. However, general sketching is possible as long as the flow of the escape room can be configured without filling in all the details. Moreover, not allowing invalid configurations of the detailed logic will greatly simplify the compilation of the visual configuration to the rulesets. In the end, a balance must be found. For the detailed logic, we decided to disallow any syntax errors. When such an error is made, the user is notified of what went wrong. When configuring the connections between screens, there is not a lot that can go wrong, except for the connections with the start node. The start node should be connected to a single screen. This

cannot be enforced from the start, as there are no screens initially, so an invalid configuration of the start node is allowed. Then when the rules are exported, the user is notified if the start node is misconfigured.

At the end of each sprint the deisgn choices made were discussed with the client, to make sure that the product would still meet the expectations. Usually a demo showing the consequences of the choices was shown, after which the importance of and reasoning behind the choices were discussed. Any remarks or needed adjustments could then be taken into account for the next sprint.

3.3. User Interface Design

When designing the user interface of the new system, we took great inspiration from the design of the brainstorming tool our client uses on a daily basis called Miro. The general UI of Miro is displayed in figure 3.3

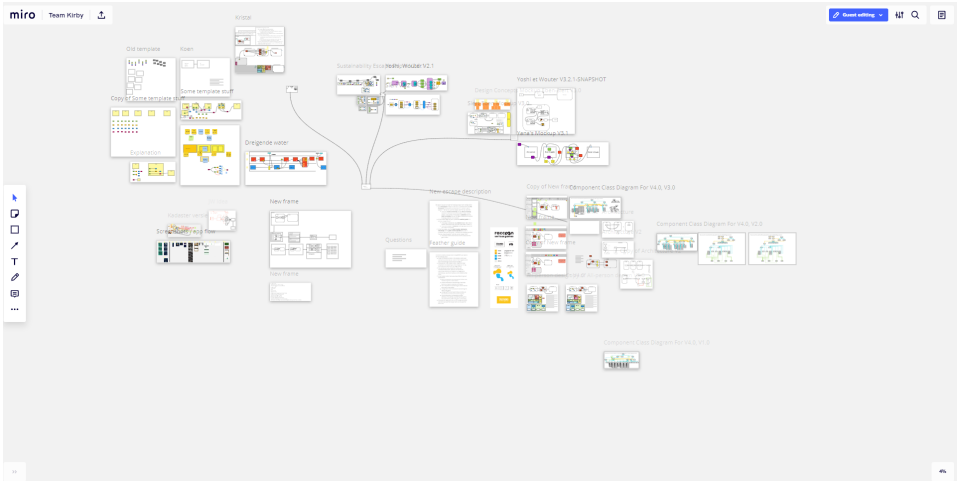


Figure 3.3: The UI of Miro.

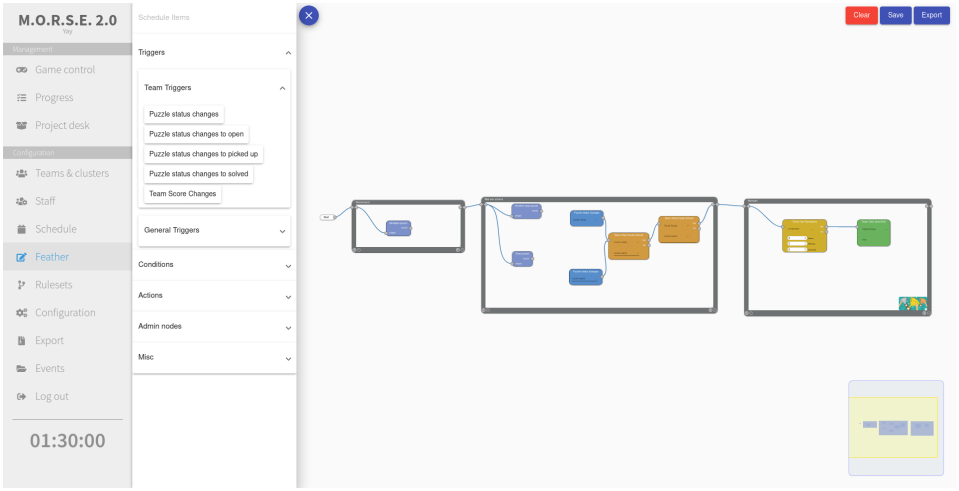


Figure 3.4: The Feather visual editor.

The Feather editor consists of four main interface components: the editing space, a mini-map, a menu, and action buttons. The entire editor is shown in figure 3.4. The editing space is where the user can place the nodes he/she wants and create the game configuration. It allows zooming in and out and has an infinite work space so the user will never run out of space. The mini-map is, as the name suggests, a small map in the bottom right corner that shows the outline of the elements present in the editing space and their relative positioning. It also shows which parts of the space are currently visible with respect to the rest of the editor.

The side menu is the place where one can find all the possible nodes that can be used in the editor. From there the user can drag and drop these nodes to the desired position in the editing space. In the side menu a

special menu, named ‘Schedule Items’, contains the already pre-defined schedule items, like a puzzle screen with puzzles inside, a text screen, or an image screen. From there the user can place them inside the editor and use them in their configuration.

Finally the editor has three buttons in the top right corner that the user can click. The button ’Clear’ as the name suggests clears the editor and removes the current configuration. This is only performed after the user confirms this action in the confirmation dialog by typing out the word ’DELETE’, as shown in figure 3.6. This feature was implemented on request of the client, since unintentionally removing everything should not be easily possible. Requiring someone to type a word makes sure that they notice that they are going to delete the whole configuration. The ’Save’ button saves the current state of the editor in the database. ’Export’ is the button that one should press if they want to convert the visual representation present in the editor into rules compatible with the rest of the MORSE’s functionalities.

Furthermore, to make a distinction between the user created and generated rules in the rulesets tab we have attached a special badge with the word ’Generated’ to the ones that have been created by the Feather editor, as can be seen in figure 3.5. Per request of the client, these generated rulesets are moved to the bottom of the list of rules. This way the manually created rules are easy to access.

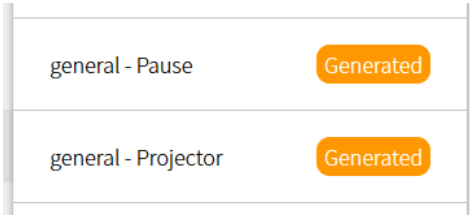


Figure 3.5: The ‘Generated’ label indicating that a rule in the ruleset editor has been automatically generated by Feather.

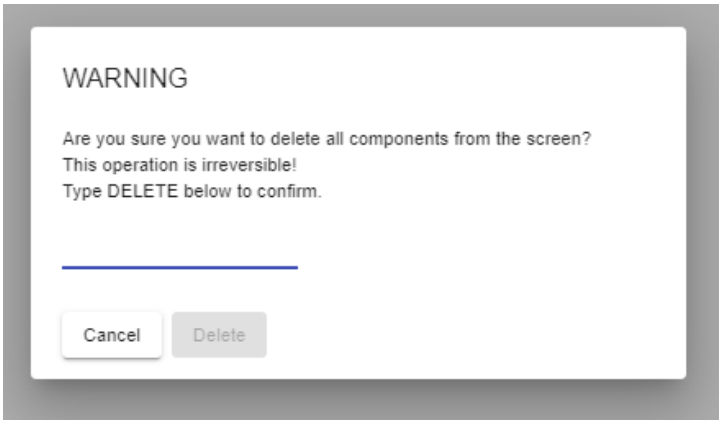


Figure 3.6: The dialog that appears when a user presses the ‘Clear’ button.

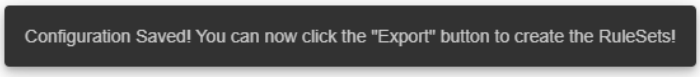


Figure 3.7: Notification design.

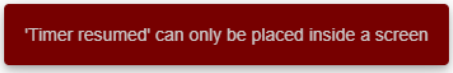


Figure 3.8: Error design.

Another user interface feature of the editor is the notifications that can be shown to the user. There are two types of notification that can be displayed: regular notifying messages and error messages. The notifying

messages are small texts usually consisting of a single sentence, which inform the user that a certain action was successful, e.g. saving/exporting. The error messages are similar to the notifications but have a red background and inform the user that a certain action is not allowed, e.g. when trying to connect incompatible nodes.

Throughout the whole interface the same visual library was used that provides material themed UI components for Angular applications. All input fields, drop-down menus, confirmation dialogues, notifications, error massages, and buttons were implemented using this library for a more coherent feel of the whole system.

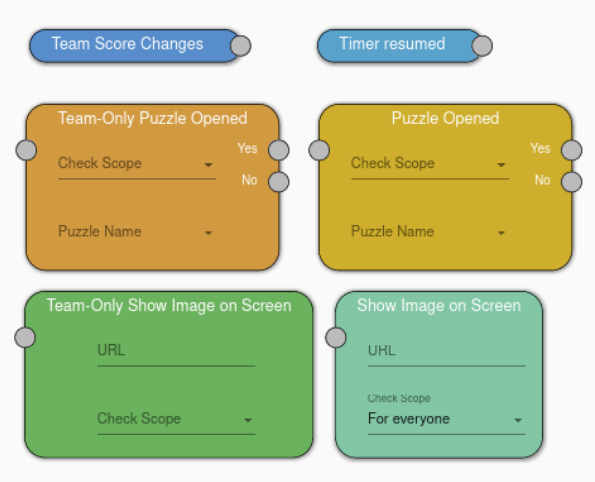


Figure 3.9: The node design.

The final aspect of the user interface is the design of the nodes themselves. The rounded rectangular shape is the standard shape defined in the library we use and we think it suits well with the rest of the application. We decided to correlate the background colour of the nodes with the colours used in the rulset definition page. In there the triggers are blue, the conditions yellow, and the actions green, so in our system the trigger nodes are blue, condition nodes yellow, and action nodes green as well. We made a further distinction between team only nodes (nodes that concern only a specific team) and general nodes. To show the difference, apart from putting the 'Team-only' prefix in the node's name we made these nodes a darker shade of their corresponding colour. Furthermore the editor has admin nodes such as an admin screen for game host related logic, admin buttons for defining admin actions, and a start node for indicating the starting point of the whole event.

Even though the biggest differentiation point between the nodes is their colour, we have made it explicit to what category they belong in the side menu. There the user can decide what node (trigger, condition, action, other) he/she wants to create and what the node's scope should be. Furthermore, the dots that connect the nodes together are different in the different kinds of nodes. For triggers there is only one on the right side, for actions one on the left side, and the conditions have three dots. This separation together with the node's name should be enough to help out visually impaired people in using the system.

4

Implementation

This chapter describes the implementation of the system. In the first section a general overview of the complete system is given. Then the front-end is explained, which includes the frameworks and different components used to build the visual editor. After that the database used to store all information is explained. A brief overview is given on the compiler, the system responsible for deriving MORSE rules from the visual configuration. Chapter 5 gives a more in-depth look into the compiler. Lastly, some explanation is given on how the system can be extended with new nodes.

4.1. Overview

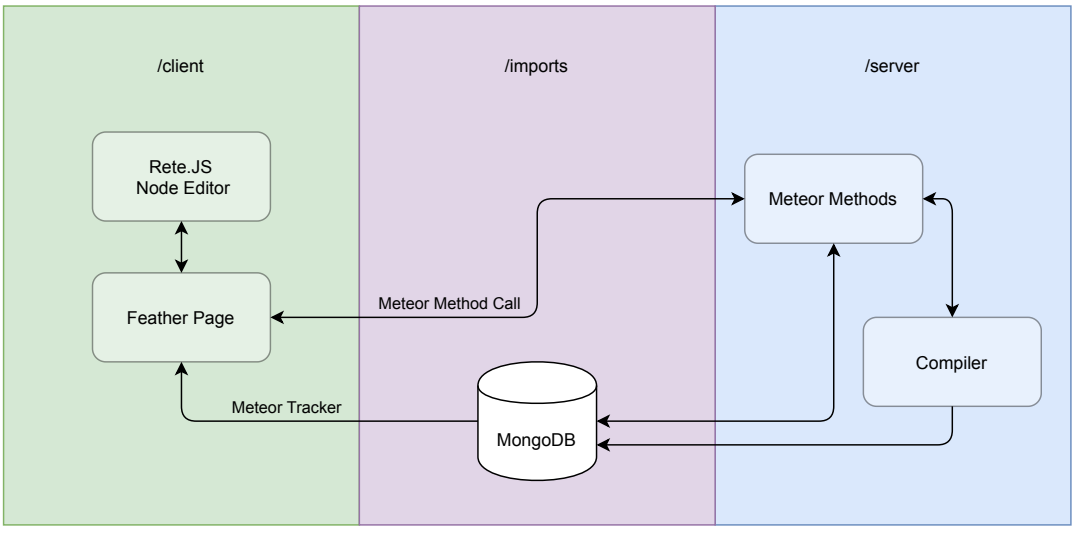


Figure 4.1: General overview of the structure of the entire system.

Given that the Feather editor is implemented on top of an already existing application – MORSE – the architecture of this existing system has played a major role in shaping the high-level implementation of Feather. Feather had to be implemented in the TypeScript[1] programming language using the Meteor framework[2]. These are the programming language and web application development framework used by MORSE. Meteor is a framework based on Node.js that takes care of many of the vital systems in a web application such as the connection between front-end and back-end of the system but also the application's permanent storage. Additionally, MORSE uses Angular[3]for its front-end, a library that streamlines the development process by

[1] https://www.typescriptlang.org/
[2] https://www.meteor.com/

separating the front-end functionality into components that can be combined and reused. Feather's front-end is therefore also implemented using Angular.

As is the case with most web applications, the front-end and back-end code are mostly separated from each other. Following the naming convention in Meteor, all the code needed for the client is in the `/client` folder and the code needed for the server is stored in the `/server` folder. However, Meteor also introduces a folder with code that is accessible by both the client and the server, located in the `/imports` folder. The shared code mainly contains the database model, but also some additional helper methods or objects that can be used by both the server and the client. A graphical representation of this can be seen in figure 4.1.

Besides enabling shared code, Meteor also allows for the communication between server and client. This is done through so-called Meteor Methods. These methods are defined on the server, with a certain name as string, and a function that is executed when the method is called. The client can then use Meteor to call these methods, by giving the name and any required arguments. Meteor automatically gives the result, which can be used in a callback function.

Finally, Meteor synchronises the database from the server to all the clients. The Meteor Tracker is used for this purpose. The Tracker is set up to load the configuration whenever one of the collections used for feather are updated. This can be done without any calls to the server, as each client keeps a cache of the MongoDB database. This cache is what Meteor updates when the server changes data in the actual MongoDB database.

The other parts of the system, such as the node editor and compiler, will be discussed in more detail in the following sections and chapters.

4.2. Front-end

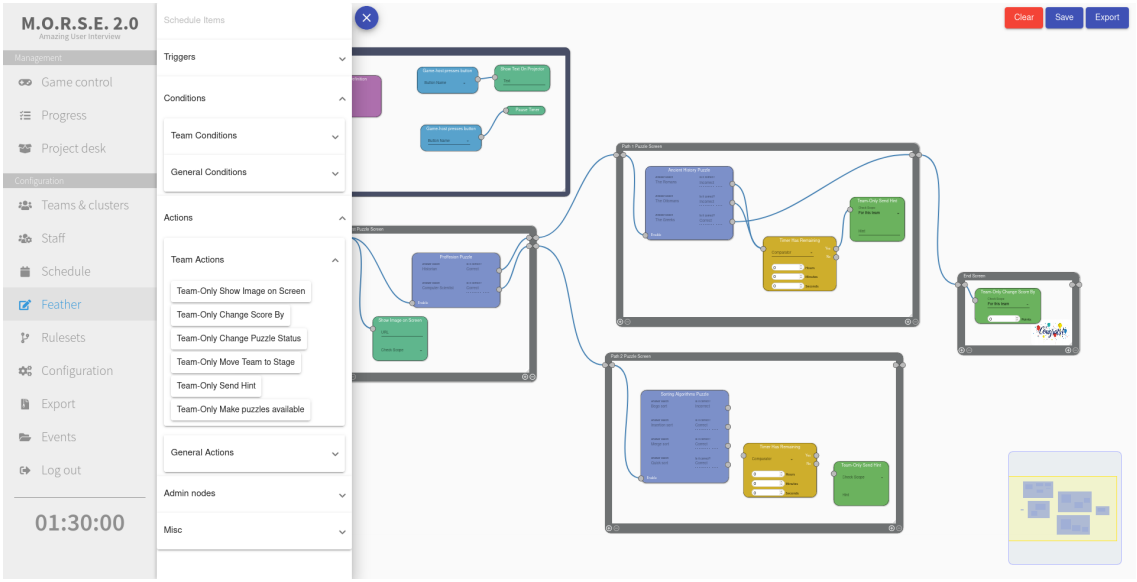


Figure 4.2: A screenshot of the Feather editor web page.

The front-end portion of the application contains the implementation of the UI elements that make up the editor and can be seen in figure 4.2. This of course concerns elements such as buttons and menus, but the majority of the front-end is dedicated to defining the 'nodes'. These components are the basic building blocks of the editor. They are the visual elements that can be connected together to build logical statements inside the editor. Nodes reside within different types of 'screens', each of which represent a stage. Screens can also be connected with each other and with nodes contained within them to conceptually and logically create the connections between different stages.

Nodes are divided into three categories: trigger, condition, and action nodes. Each category has a differing colour scheme to differentiate them from the others. There is also a special type of composite node called the 'puzzle' node which is a trigger, condition, and action node all at once.

[3] https://angular.io/

4.2.1. Frameworks

Given that the project had to be an extension to an existing code base, some design decisions were out of the scope of the project. Most notably the use of Angular and Meteor in the project was non-negotiable given that the entire existing code base was built on it. Furthermore, the fact that the project was in Angular limited our choice of visual library further to just ones compatible with Angular.

Using Angular however has the benefit of having a Material Design[4] theme library built-in native to the project. With a little bit of work all components could be given a modern look.

Another library central to the project is Rete.js[5]. This a library that provides the most fundamental functionalities for the creation of flow-chart editors. Rete was very well-suited for the purposes of what we were trying to build.

4.2.2. Switch from Storm React to Rete.js

During the research phase we looked into various frameworks to use for the visual editor. Among others we found Storm React Diagrams[6], a library specifically built for visualising flow charts. As this software fit all our requirements, we decided to use it. However, one thing we did not realize at the time is that it being built for React – yet another web development framework –, it would not work with Angular, which is used by MORSE. We tried to find a way of making this library work within the MORSE system, but quickly learned this would be impossible and if not very difficult.

In the research phase we carefully documented the pros and cons of different libraries we could use for the system. The Rete.js library was among those and by using reading back our findings, we were able to determine that this library would better fit our use case. Thus, because of our thorough investigation earlier in the project this issue was resolved rather quickly.

4.2.3. Rete.js

Specifically designed for the purposes of creating flow-based visualisations, the intended use of Rete.js matches the design for the visual editor. The library offers various features, some of which did not suit the project. These features were left untouched and can be used if the editor is extended at a later point in time.

The Rete.js library uses two main elements: components and controls.

Component A component describes the features of a specific type of node inside the editor. The component describes the sockets for incoming and outgoing connections of a node as well as its controls, which are explained below. Each component can have its own styling and functionality. The nodes created from these components can be connected to each other to represent different logical 'flows' through the escape room.

Control Controls are used to customise data inside components. Examples include the numeric field to select a specific score and the text field to enter the URL for a video to be displayed. For some controls, like the control to select a specific puzzle, Angular services are implemented to keep track of the relevant data. The component can access the data entered in its controls can then use that data to perform operations on or to update its HTML.

4.3. Database

The database used for storing the escape room configuration made by the user and the rules generated from it, is MongoDB[7]. MongoDB is a database variant that is integrated by default in Meteor and also already used by the MORSE system. It thus made sense to also use it for the Feather editor.

MongoDB stores its data as documents inside collections. The Astronomy for Meteor[8] package is used to allow easier definition and modification of the MongoDB documents. This is also a library that is already being used by MORSE. To store the configuration of the editor two collections and various document schemas have been added, which will now be discussed in more detail. Figure 4.3 illustrates the added schemas and how they relate to each other.

[4] https://material.io/design
[5] https://rete.js.org/
[6] https://github.com/projectstorm/react-diagrams
[7] https://www.mongodb.com/
[8] https://jagi.github.io/meteor-astronomy/

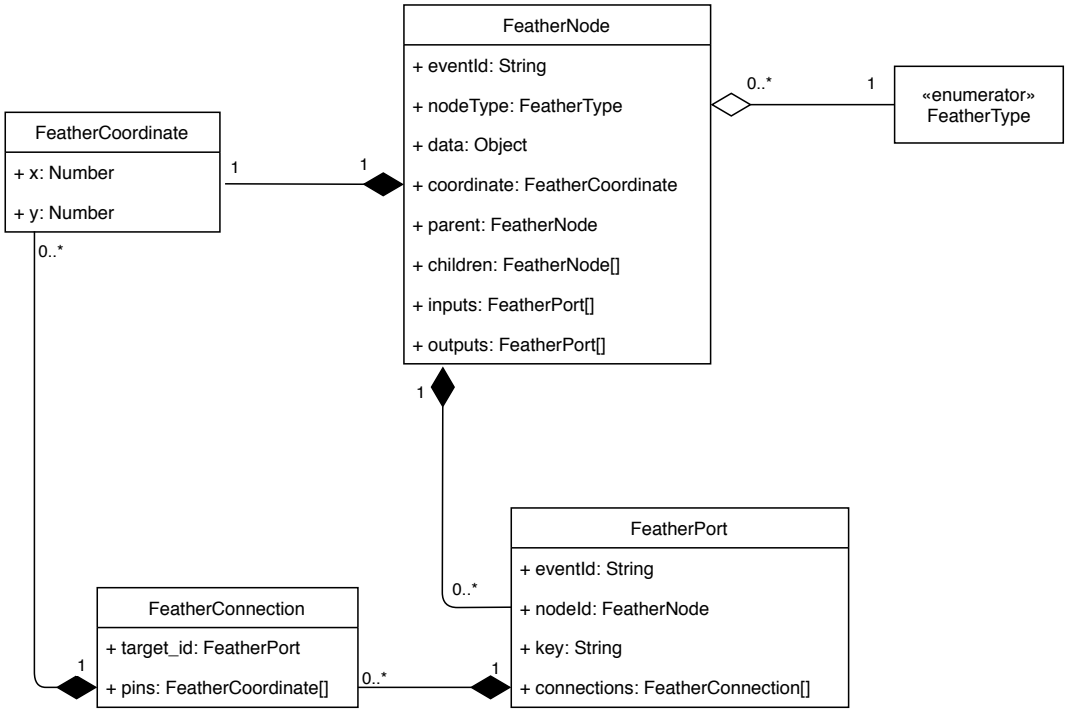


Figure 4.3: UML Diagram of the database model.

The first collection stores documents with the `FeatherNode` schema, which is created to store the data about the nodes that can be inserted and used in the visual editor. A node has the following attributes, each with their own purpose:

- `eventId`: The id of the event the node belongs to.
- `nodeType`: The type of the node. This is a string that is in one of the enumerators, which are combined by the FeatherType.
- `data`: An object containing any data needed for the node. The data that is stored in this object varies between the different types of nodes.
- `coordinate`: An x and y coordinate embedded in the FeatherCoordinate class specifying the position of the node in the editor.
- `parent`: The id of the parent of the node. This parent usually is the screen a node is placed into. The string should always correspond to one of the other FeatherNode's ids.
- `children`: A list of ids giving all the children of the node. For screens this is a list of all the ids of the nodes that are placed inside of it. Each string in this list should point to a FeatherNode's id.
- `inputs`: A list of strings corresponding to ids of FeatherPort documents. The ports represent the input sockets this node has in the editor.
- `outputs`: Similar to the inputs, but the ports represent the output sockets.

The second collection stores documents with the `FeatherPort` schema. Again each of the ports attributes have their own purpose:

- `eventId`: The id of the event the port belongs to.
- `nodeId`: The id of the node that this port belongs to.
- `key`: A string that specifies which socket of the Rete.js node this port corresponds to. This is important for nodes with multiple ports such as condition nodes. Something that is connected to the 'Yes' output, should not suddenly become connected to the 'No' output when reloading from the database. For screens and multiple-choice puzzles, there is a variable amount of ports. In this case an index is embedded in the key to make sure each port of a node has a unique key.
- `connections`: A list of `FeatherConnections` that each represent a connection in the editor. Such a connection has a `target_id`, referencing the other `FeatherPort` of the connections, and a list of coordinates that are the pins placed on the connection.

The connections are stored in both directions: If A has a connection to B, then B also has a connection to A. Adding this connection both ways requires minimal effort, but eases the process of loading the configuration from the database.

The main choice in designing the database model was whether to create a separate schema for each node or to use a general schema like defined above. The advantage of creating a separate schema for each implemented node instead of a general one is that it allows the definition of fields specific to each node which means it is not necessary to have a generic data field. Additionally, helper methods can be created for each of the nodes.

However, defining a separate schema for each type of node comes at the cost of modularity and extensibility. There is a clear difference between the way the nodes are used client-side and the way they are used server-side. Placing the code for these functionalities in one place is not preferred. The general model is also chosen for its extensibility. Storing a node works the same way for all the types of nodes. While special behaviour of a node still needs some extra work, the general procedure of storing and loading can work the same way for every node. Another important factor in this decision is that the Astronomy for Meteor package is written in JavaScript, not in TypeScript. This means that we do not lose any typing information we might have had with separate schemas.

4.3.1. Loading and Saving from Client

In the editor a user can save the visual configuration which is then loaded automatically when the editor is opened. Meteor allows for easy reading from the database without the need to explicitly communicate to the server.

When Angular creates the view, it injects a service that is used to load the configuration from the database. Whenever Meteor notifies the client that something has changed in the database, the service is used to load the new configuration. The loading happens in a couple steps. First the Rete.js nodes are created. To create a node, the service uses a map to select the correct custom Rete.js component based on the type of the node. This component is then used to create the needed node with the correct data. When a node has been created the ports of this node will be loaded. Finally, after all the nodes have been created, the nodes are connected to each other.

Saving the configuration to the database is done on the press of a save button. Inserting or updating documents in the database cannot be done directly by the clients, so a Meteor call to the server is needed. First the current configuration is removed. Then all the nodes are saved, the client sends the following information for each node: type, eventId, data, x-coordinate, y-coordinate. The server returns the id of the newly created node, which can then be used in the saving process. With the IDs the parent and children fields can be updated for each of the nodes. The saving continues with saving the ports, where the client sends a list of input keys and output keys to the server for each node. The server then creates a port for each of these keys and adds it to the node documents created earlier, returning the ids of the newly created ports. Finally, the connections are saved by sending the IDs of the connected ports to the server, which will create the connection in both directions in the database documents.

4.3.2. Access from the Server

Accessing the database server-side works the same as accessing it client-side, with the exception that the server is now also allowed to insert and update documents. The new schemas and collections are mainly used in two places: the Meteor method calls by the client and the first part of the compiling process (5.5). The methods allow the clients to update the configuration and the compiler reads the configuration from the collections to start the compilation.

4.3.3. Concurrent Use of the Editor

Support for multiple users editing the current configuration was decided to be a ‘won’t have’ requirement at the start of the project (section A.7). Even so, we would like to emphasise that this is not supported in the editor.

Multiple users can connect to the server at once and they can have the editor open at the same time. However, changes made to the configurations are only saved when the save button is pressed. These changes are then automatically loaded by all clients, overwriting any local changes they had made. Furthermore, saving two different configurations at the same time, can lead to problems such as the ports of nodes saving incorrectly or nodes being saved twice. This might then also cause errors when the loading from the database again.

A second limitation is the saving and compiling of a configuration at the same time. It is technically possible to save and compile at the same time, by pressing the two buttons (possibly on multiple clients). However, the saving will add new nodes to and remove old ones from the database. While it is changing the configuration, the database could hold an invalid configuration when the compile is started, causing it to fail. In such a case, the export button can simply be pressed again, when the saving is completed.

4.4. Compiler

To actually allow an escape room designed in the Feather editor to be used and played, it needs to be converted to the existing system that MORSE uses in the back-end for defining escape rooms. This system works in the earlier mentioned if-this-then-that rule format. Thus, besides allowing the user to design, save and load escape room configurations, another major component of Feather editor is the compiler: the system that is responsible for compiling the graphical escape room definition on the user's screen into MORSE rules.

When the 'Export' button on the Feather editor is pressed by the user, the compiler is invoked. This will cause the following things to happen in the background:

1. The escape room configuration is fetched from the database. This saved configuration might be different from the one on screen and thus it is important for the user to save their design before pressing 'Export'.

2. The graphical format in which the escape room is defined is converted to MORSE's rule format by the compiler.

3. The generated MORSE rules are added to the database, allowing them to appear in MORSE's ruleset editor. Generated rules are stored in the database with a special flag. This allows the system to differentiate between these two types of rules and to overwrite previously generated rules while leaving manually created rules untouched.

Due to its complexity, the most important feature of the compiler, the system responsible for performing the conversion between the two formats, is discussed in a separate chapter (5).

4.5. Adding New Nodes

An important aspect of Feather is that it is relatively easy to add new nodes. This was useful during the project itself, as a lot of nodes had to be created, but also makes the product extensible. If new elements are added to MORSE, it does not require a lot of extra work to add these elements to the visual editor. It is however important that all the necessary behaviour is specified, so this section comprises a list of functionalities/classes that need to be added to add a new node, with the places they should be added in.

1. Type: First a new entry in one of the type enumerators should be created.

2. Component: The component is a TypeScript class that extends the `FeatherComponent` class.

3. Creator Service: The component has to be added to the component creator service to make sure it can be utilised.

4. Intermediate model: A new class in the intermediate model should be created for the node, implementing all abstract methods. The intermediate model is a data structure used by the compiler and is discussed in more detail in 5.4.

5. Intermediate Node Creator: Similar as the creator on the client side, but now the `fromDatabase` method should be added to the map. This is also something that is necessary for the compiler to properly handle the node (5.5).

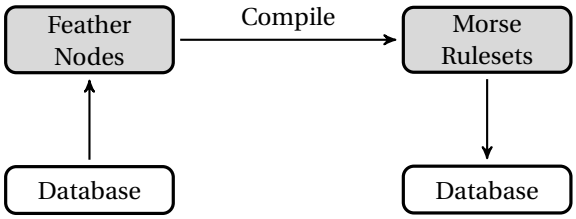
5

Compiler

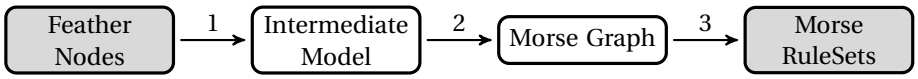
The Feather editor allows the user to design and define an escape room but of course it also needs to allow defined escape rooms to be executed such that they can be played or tested. To allow this, the graphical node structure in the Feather editor needs to be converted to MORSE rulesets, as these rulesets are what is used by MORSE to define and run escape rooms. The nodes that the user sees in the visual editor and the rules in MORSE's ruleset editor are not directly compatible, which is why a separate conversion system is required.

5.1. Overview

The compiler works closely together with the database. It retrieves the Feather escape room definition from the database, converts it to MORSE rules and then stores those rules again in the database to be retrieved when running the escape room.



Implementing a compiler between these two data formats is complex. Not only are there many different nodes in our visual editor and many different rule elements (triggers, conditions and actions) in the MORSE system, but some elements in the visual editor do not have a simple one-to-one mapping to existing rule elements. To make the task of designing and implementing the compiler a easier, we decided to split the compiling operation up in a number of smaller compile steps that are serially chained together.



Splitting it up like this does not only come with the benefit of reduced complexity and therefore reduced effort from our side, but also results in the code being modular and thus more extendable and understandable. Furthermore, it also allowed us to more effectively divide the work of implementing the compiler among team members.

In the first step of this compile chain, the graphical representation stored in the database as directly derived from the nodes on the user's screen is compiled to an intermediate data structure, which we will refer to as 'the intermediate model'. This data structure is different from the raw graphical representation in the sense that it omits all details from the latter representation that are not relevant for the expected behaviour of the editor configuration. An example of data that is omitted in this step is positional data. After all, the exact coordinates of a node should not matter for rules resulting from compiling the node. What matters is the type of nodes that are being used in the editor configuration and how they are combined with each other. This first compile step will be discussed in more detail in 5.5.

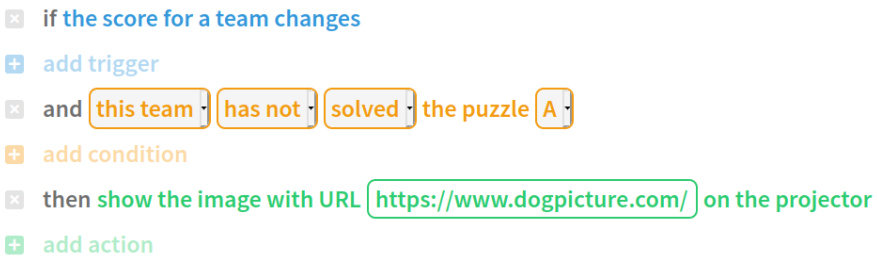


Figure 5.1: Example of a rule in MORSE's ruleset editor with one trigger (in blue), one condition (in yellow) and one action (in green).

The second step of the compile chain transforms the intermediate model into a 'MORSE graph'. This is a directed graph where the nodes are 'rule elements'. These rule elements are the building blocks of MORSE rules and can either be a trigger, a condition or an action. Each rule consists of a set of triggers, a set of conditions and a set of actions. The exact meaning of these rule elements and how they are used to construct rules in the MORSE editor will be discussed in 5.2. The MORSE graph thus more closely resembles the compiler's final output, a set of MORSE rules, but is still represented as a graph that derived its overall shape from the original Feather editor configuration. More details on this compile step will be given in 5.6.

The final compile step in the chain, transforms the MORSE graph into actual MORSE rules. This is done by employing a path finding algorithm on the MORSE graph. This will be discussed in 5.7.

After the Feather nodes are successfully compiled into MORSE rules, the rules are added to the database to be used when running the escape room or to be edited again in the MORSE ruleset editor. The generated rules are given a special flag in the database. This flag allows the system to differentiate between generated and manually created or edited rules. When compiling a Feather editor configuration to MORSE rules, the system can then override the previously generated rules while not destroying the manually created ones. This allows the user to try out multiple versions of their escape room design in Feather without losing their manually added rules on every compile.

5.2. Background on MORSE Rules

MORSE rules follow a simple if-this-then-that structure. Besides a couple of extra options, they consist solely of a non-empty set of triggers, a set of conditions and a non-empty set of actions.

A trigger describes an event that might occur in the escape room. Adding a trigger to a rule means that the rule will execute when its corresponding event occurs. An example of a trigger is: the timer reaching five minutes left.

A condition describes some condition within the escape room context that can evaluate to either true or false. The condition will prevent actions defined in the rule from being executed if it evaluates to false. An example of a condition is: the team has not solved puzzle A.

An action describes some change to the state of the escape room. Examples of actions are: make a number of puzzles available for a team, display an image on the projector screen or stopping the timer.

Each rule allows multiple rule elements of each kind to be added. When adding multiple triggers, they are treated like an OR relation: if one trigger fires, the rule is executed. When adding multiple conditions or actions, they are treated like an AND relation: all conditions need to be satisfied and, if that is the case, all actions will be executed.

This is the way in which rules are presented in the MORSE ruleset editor (figure 5.1) and, internally, MORSE rules are represented in a similar way. Thus, reasoning about compiling Feather nodes to MORSE rules as seen in the ruleset editor is not much different from reasoning about compiling them to MORSE's internal ruleset representation.

Some rule elements allow an input field to be filled in with a "this team" value which will then refer to the team in the trigger. These correspond with the team-only nodes in the Feather editor. Using conditions or actions with the "this team" value selected together with a trigger that does not support the usage of this value is illegal in MORSE. When compiling from Feather to MORSE, such things should be taken into account as to not generate illegal rules. This specific issue is handled by the Feather editor already: team-only nodes can only be placed inside screens and team-only conditions and actions can only be connected to team-only triggers. More details on this will be given in 5.3.

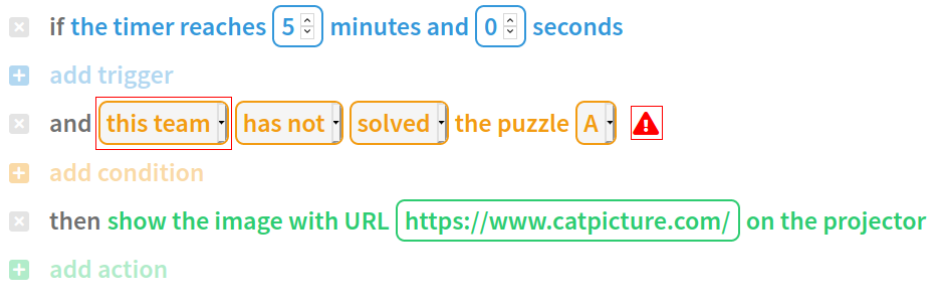


Figure 5.2: Example of an illegal rule in MORSE's ruleset editor. A warning is given because a "this team" value is used in the condition, while the chosen trigger does not support the use of this value.

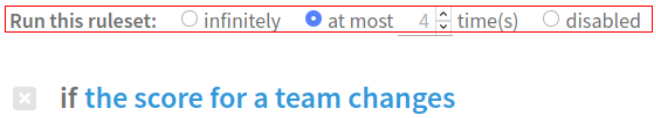


Figure 5.3: The extra options that can be set by the user above every MORSE rule.

Lastly, there are couple more options that the user can set for a rule: putting a limit on how often a rule can be executed or disabling a rule altogether.

These options are, however, not included in the Feather editor. Compiled rules will have the option 'infinitely' selected by default and if a user wanted to use a different option, they have to update the rule manually.

5.3. Constraints for the Compiler in the Front-End

Rete.js – as a general purpose library for making graph editors – does by default not put many restrictions on how nodes are connected and where they are placed. Any in-port can be connected to any out-port. This is a problem because it allows the user to construct many configurations in the Feather editor that will compile to illegal MORSE rules or ones that are semantically undefined.

To deal with this problem without making the the compiler too complex, we put constraints directly into the front-end of the editor to prevent configurations that are problematic to the compiler. Another benefit of this is that it allows the editor to provide more intuitive, direct feedback on errors rather than simply showing error notifications generated by the compiler. To illustrate: a connection between two ports cannot cause an error in the compiler if the editor does not allow the two ports to be connected in the first place.

As said earlier, out-of-the-box Rete does not put many constraints on configurations. Rete does offer cycle detection and can disallow the creation of edges that complete a cycle, which is useful because the graph in the Feather editor should not contain cycles in order to be compiled.

However, even with this restriction, it is still possible to construct many problematic configurations in the Feather editor. To combat this, the following constraints have been added to the front-end of the Feather editor:

- External screen in-ports are only allowed to be connected to external screen out-ports. These are screen edges and they represent transitions of teams throughout the escape room. Similarly, internal screen ports can only connect to nodes inside the screen. These are logic edges. These two types of edges are conceptually different and their ports should not be mixed as this will cause problems during the first compile step.

- In general, any node in-port can be connected to any node out-port, but team-only conditions or actions can never be (in)directly connected to a trigger that is not for teams. This is to prevent the compiler from generating illegal rules as discussed in 5.2. Because a screen transition is also a team-only action in MORSE, the same applies for screen out-ports.

- Team-only nodes should only be placed inside screens. As discussed in 3.1, allowing *all* nodes to only be placed inside screens was already a design consideration, but team-only nodes are more so required

to be constrained this way because they need to be scoped inside a screen. The compiler ensures this scoped behaviour by generating conditions that check if a team is in the correct screen. This means that nodes must be properly nested inside a screen for the compiler to know on which screen to perform this check.

5.4. The Intermediate Model

The intermediate model serves the purpose of being a data structure that exists between the first and the second compile step. Additionally, the intermediate could also be used for doing analyses on an escape room configuration. An example of such an analysis could be a script that detects loops or patterns that will lead to illegal MORSE rules in the escape room definition and warns the user about it. However, we have implemented few of those analyses in the current system and those work directly on the Rete nodes in the Feather editor. Performing this analysis on the intermediate model instead is an option for future extension.

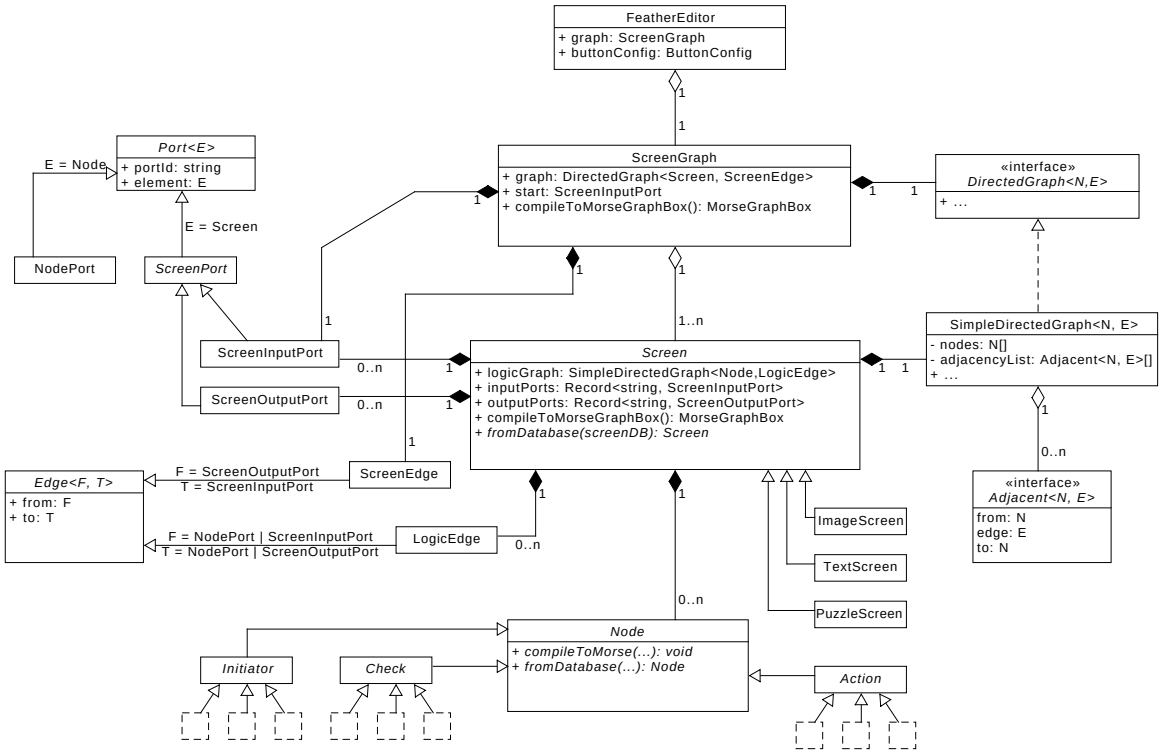


Figure 5.4: A diagram illustrating the structure of the intermediate model. The diagram mostly follows the UML standard, but adds some non-standard notations. A class extending another class and thereby assigning a generic type (e.g. `NodePort extends Port<Node>`) has this assignment annotated on the extension arrow (E = Node). Also, for simplicity's sake, the numerous extensions of the three `Node` subtypes are omitted and represented by dashed boxes.

The intermediate model closely follows the node structure as seen in the Feather editor but omits details that are irrelevant to the workings of the escape room and nests structures more logically. A graphical overview of the class structure of the intermediate model can be seen in figure 5.4.

While in Rete there is a single class for nodes where properties of the type of node are stored in an untyped data field, the intermediate model has a class dedicated to each type of node in the Feather editor, each extending an abstract `Node` class. As can be seen in figure 5.4, `Nodes` are nested in `Screens` and `Screens` are again nested inside a `ScreenGraph`.

For this, a generic `DirectedGraph`[1] class is used, representing a standard graph data structure. The `ScreenGraph` contains a graph with instances of the `Screen` class as nodes and `ScreenEdges` as edges. This is representation follows from the fact that screens are layed out as a graph in the Feather editor. Those `Screens` again contain a graph but this time with `Nodes` or `ScreenPorts` as nodes and `LogicEdges` as edges. These correspond to the logic nodes found inside screens in the Feather editor.

[1] We made a custom implementation of a simple graph ourselves due to the lack of properly typed graph libraries for TypeScript.

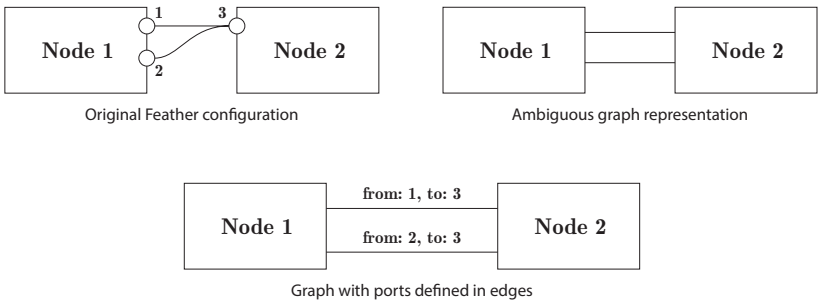


Figure 5.5: Three representations of a node configuration with two nodes with two edges between them, where the first edge connects from the upper port of the first node and the second edge from the lower port. In the ambiguous graph representation, it is defined that there are two edges between the nodes but information about how they are connected to the ports of nodes is lost. By adding the data about ports as fields of the edges, the information is preserved.

Both types of edges used in these graphs, `ScreenEdge` and `LogicEdge`, contain data about a from-port and a to-port as can be seen on the far left in figure 5.4. One might get the impression that storing this data is unnecessary as the graph already defines how nodes are connected to each other. It is however indeed necessary to include these additional "from" and "to" fields in the edges, because without it the graph is ambiguous as to with what ports, screens or nodes are connected to each other. This is illustrated in figure 5.5.

It might have been more elegant to extend the generic `DirectedGraph` class to differentiate between connections to different ports. However, this was discovered in hindsight and changing this was not as much as a priority as other features.

The `Screen` and `Node` classes both contain an abstract `fromDatabase()` method. This method is related to compiling feather nodes and screens from the database to the intermediate model and will be further discussed in section 5.5.

The `compileToMorseGraphBox()` and `compileToMorse()` methods found in `ScreenGraph`, `Screen` and `Node` are part of the second compile step, compiling the intermediate model to a MORSE graph. These will be covered in section 5.6.

5.5. The 1st Compile Step: Database to Intermediate Model

This section will discuss the implementation of the first compile step in the three-step process as explained earlier. The two following steps will be covered in the sections after this one.

5.5.1. A Concrete Example

To illustrate the workings of the compiler as a whole, a single example of an escape room configuration in the Feather editor is given that is then taken through every compile step until it eventually is fully transformed into a list of MORSE rules. This example configuration is shown in 5.6. The list of MORSE rules that the compiler is expected to have as output is shown in 5.7.

The example configuration has two screens of which the first one is connected to the start node. Thus, Screen 1 is the first to be entered, which is reflected in rule 1 in the resulting rules (5.7). Rule 2 incorporates the fact that the enable of Puzzle A is connected to the screen port to which Start is also connected. Given that rule 1 and 2 share the same trigger, they could be merged into one rule that has both actions. In 5.7 will be elaborated why these rules are nonetheless separate in the compiler output.

Screen 1 contains one multiple choice puzzle node with two answers. If a team picks the first answer with the text "First Answer" then a transition happens to Screen 2. This behaviour is defined in rule 3 in the resulting rule. It also includes a condition that checks whether or not a team is in Screen 1. This is to prevent the rule from firing in other screens than Screen 1.

Screen 2 contains one condition node and two action nodes, one for each condition outcome. If a team is fast enough to arrive at Screen 2 before the timer hits 20 minutes, they get more points (100) than if they are too slow (25). This logic is represented by rule 4 and 5 in the resulting ruleset. Two rules are required, one for the yes-case of the condition and one for the no-case.

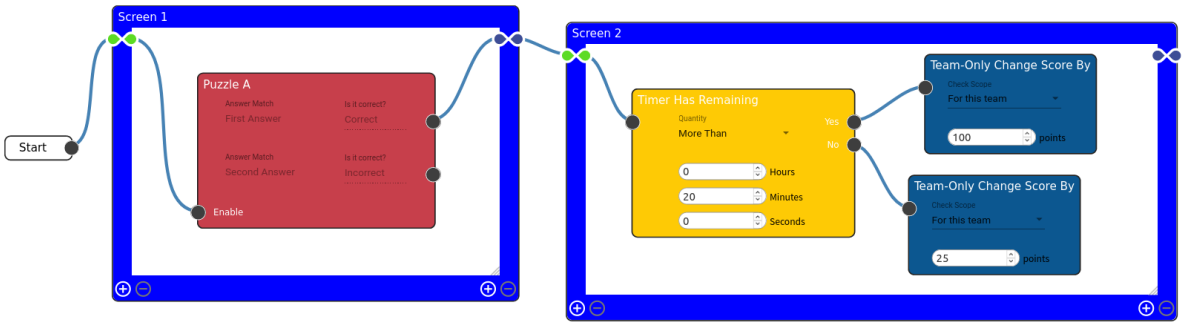


Figure 5.6: An example of an escape room configuration in the Feather editor that has two screens.

(1)
if a team logs in
and change the status of the puzzle Puzzle A to open for this team

---

(2)
if a team logs in
then move this team to the stage Screen 1

---

(3)
if the status of the puzzle Puzzle A changes for a team
and this team is in the stage Screen 1
and this team gave an answer which equals First Answer to the puzzle Puzzle A
then move this team to the stage Screen 2

---

(4)
if the status of the puzzle Puzzle A changes for a team
and this team is in the stage Screen 1
and this team gave an answer which equals First Answer to the puzzle Puzzle A
and the timer has more than 20 minutes and 0 seconds remaining
then update the score of this team with 100 points

---

(5)
if the status of the puzzle Puzzle A changes for a team
and this team is in the stage Screen 1
and this team gave an answer which equals First Answer to the puzzle Puzzle A
and the timer has at most 20 minutes and 0 seconds remaining
then update the score of this team with 25 points

Figure 5.7: The expected output of the compiler when compiling the example configuration in figure 5.6.

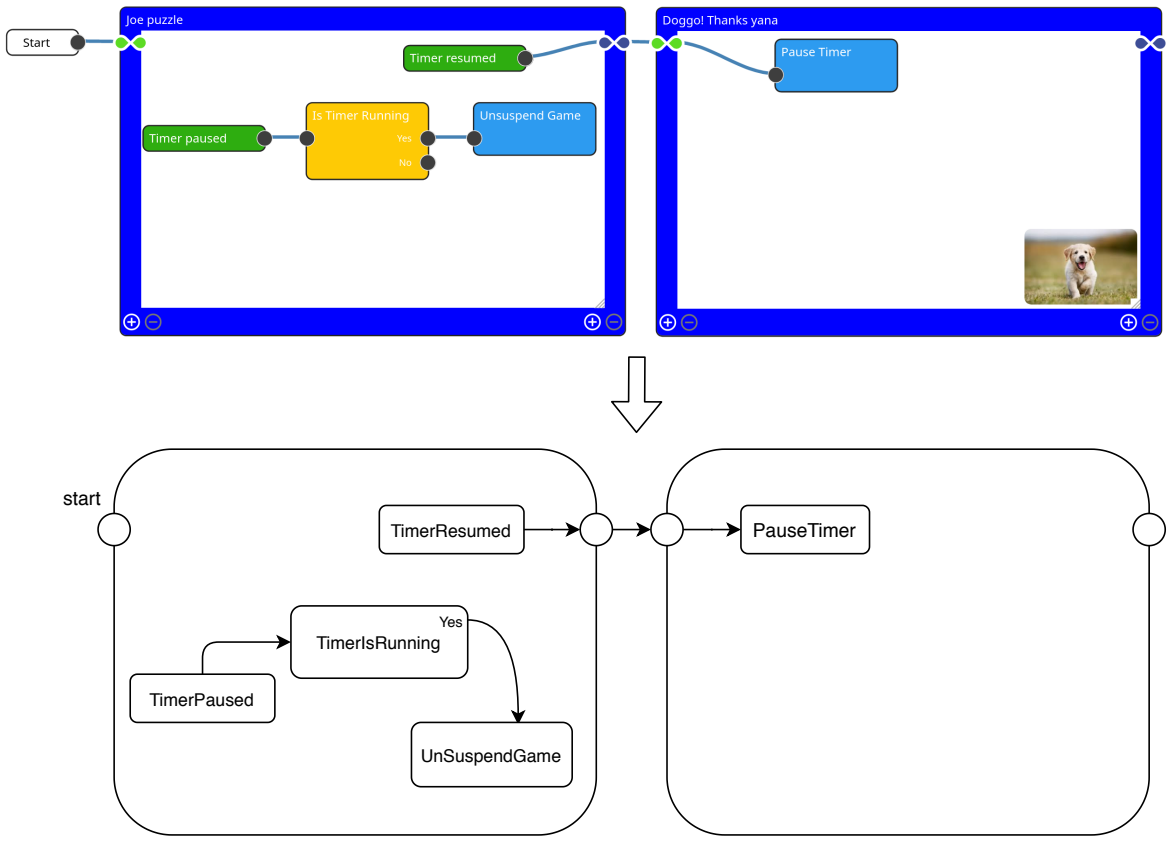


Figure 5.8: Visualisation of the effect the first compile step has on the configuration.

5.5.2. Compiling to the Intermediate Model

In the first compile step, the Feather nodes are fetched from the database and turned into the intermediate model. In this model, all the escape room data is fully contained within one instance of the `FeatherEditor` class. This compile step is responsible constructing a `FeatherEditor` from the Rete nodes found in the database. This process can, again, be divided into three steps.

First, the screens and their ports are created. Where each port of the screen has two separate sockets (one on the outside and one on the inside), each pair of these sockets is turned into one port. The ports on the left of the screens are `ScreenInputPort` and on the right turn into `ScreenOutputPort`. After the screens are created, the connections between screens can be added to the graph in the intermediate model. These connections are directed from a `ScreenOutputPort` to a `ScreenInputPort`, so from the right side of a screen to the left side of another screen. The connections assume that screens are connected with their outer sockets, which is enforced by the visual editor.

The next step is setting the start of the configuration. If the start node is not present, not connected, or has more than one connection, the compilation step throws an error. Like the connections between screens, the start node should be connected to an outer socket of a screen. Additionally, the start node should be connected to the left side of a screen. Again this is already enforced by the editor.

Finally, the nodes are created, added to the screen they are in, and connected to each other and the ports of the screen. Each node, that is not a screen, is assumed to have a screen as parent. In other words, each node should be placed inside a screen. Nodes can be connected with ports of each other and with the inner ports of the screens.

While this step does not change the configuration itself a lot, it changes the way the configuration is stored. The untyped database model is turned into a graph data structure, in which all the vertexes (corresponding to nodes) have their own type. Additionally, the start node is removed by just setting the port it's connected to to a field. This process is illustrated in figure 5.8.

We will not go into detail of the first compile step of the example, as its representation is not changed drastically. The only significant change is the data structure it is stored in.

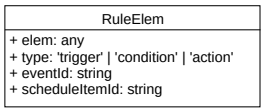


Figure 5.9: The UML diagram for the RuleElem class. RuleElem contains besides the actual rule element also the type of rule element and the IDs of the event and the schedule item that the rule element belongs to.

5.6. The 2nd Compile Step: Intermediate Model to MORSE Graph

The purpose of the second compile step is to transform the intermediate model as discussed in 5.4 to a MORSE graph. A MORSE graph is a directed graph data structure with 'rule elements' for nodes and no data on edges. These rule elements are the elements that MORSE rules consist out of: triggers, conditions and actions. Since MORSE's data structures for these rule elements are not typed – they are all of the `any` type –, we introduce a new `RuleElem` class (figure 5.9) to serve as a wrapper around actual MORSE rule elements.

This compile step makes the following assumptions about the intermediate data structure it takes as input:

- General triggers can never be followed up by a team-only condition or a team-only action or a screen transition. This will lead to an illegal MORSE rule for the reasons discussed in 5.2.
- The amount of incoming or outgoing ports of nodes in the intermediate model does not exceed the number of inputs or outputs that this type of node has in the Feather editor, respectively. E.g. a condition node that only has two outgoing ports ("yes" and "no") should not have three outgoing edges with unique from-ports in a `Screen`'s logic graph.

5.6.1. Challenges in the 2nd Compile Step

The resulting MORSE graph from performing the second compile step should only contain rule elements and edges without any data. This means that in this format, the concept of nodes with ports and the concept of screens is gone. The disappearance of these concepts suggests that there are non-trivial mappings between elements in the intermediate model and the MORSE graph. And indeed, a single node might map to multiple rule elements as can be seen in figure 5.10. In this example, a single condition node will compile to two condition rule elements. This is necessary because a single condition in a MORSE rule cannot account for both the true and the false case of a condition node in Feather. A separate condition element – and thus also a separate rule – needs to be constructed to account for both cases.

Furthermore, screen transitions contain implicit rule elements for changing stages and certain nodes such as the puzzle node don't have a rule element counterpart in MORSE at all. An example of the former phenomenon can be seen in figure 5.11. Here, the stage transition – caused by the screen edge – is added as an extra action.

To go back to the example in figure 5.6, the expected MORSE graph resulting from performing this compile step on this configuration is shown in figure 5.12. Here can also be seen that the condition node in the Feather editor has been split up into two separate rule element nodes in the MORSE graph. The screen edge between Screen 1 and Screen 2 has been converted to a stage transition rule element. The Feather node for Puzzle A has even been split up in two rule elements: a trigger that checks if the status for Puzzle A changes – this trigger fires when a user submits an answer – and a condition that checks if the given answer is indeed "First Answer". Lastly, the bottom two islands in the directed graph also contain a condition for checking if the team is on Screen 1.

5.6.2. GraphBoxes

To actually implement this mapping behaviour, we introduce a "GraphBox" data structure. This is a directed graph with a list of inputs and a list of outputs to which nodes from the graph can connect. It can be seen as a subgraph that defines through its inputs and outputs how it is connected with outer structures. A graphical representation of a GraphBox can be seen in figure 5.13.

For GraphBoxes, a merge function can be implemented that takes two GraphBoxes as an input and returns a single, combined GraphBox, by connecting nodes from the separate graphs that have matching inputs and outputs inside the combined graph. This merge operation is illustrated in figure 5.14.

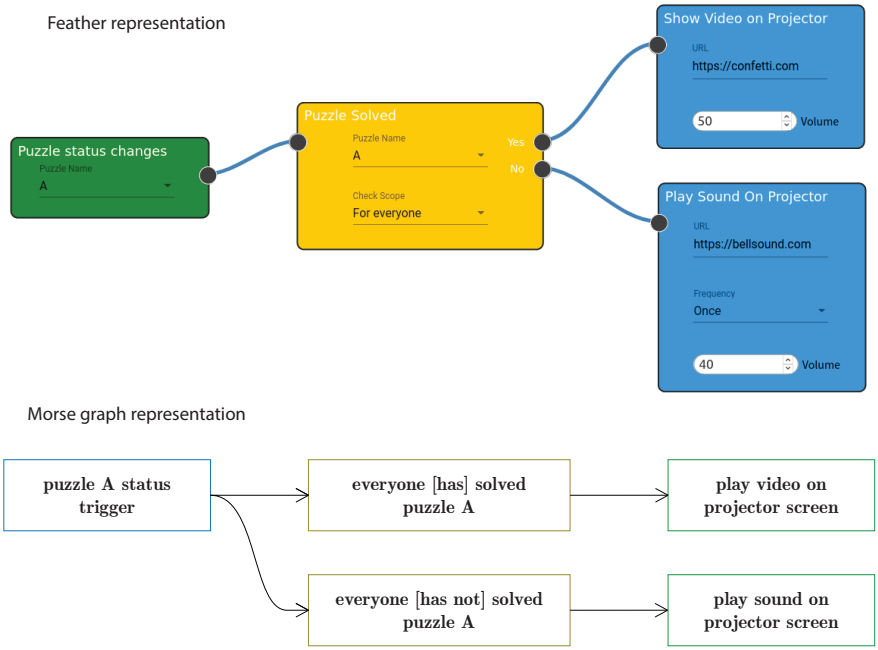


Figure 5.10: A node configuration in the feather editor (top) and the desired output for second compile step when compiling that configuration (bottom).

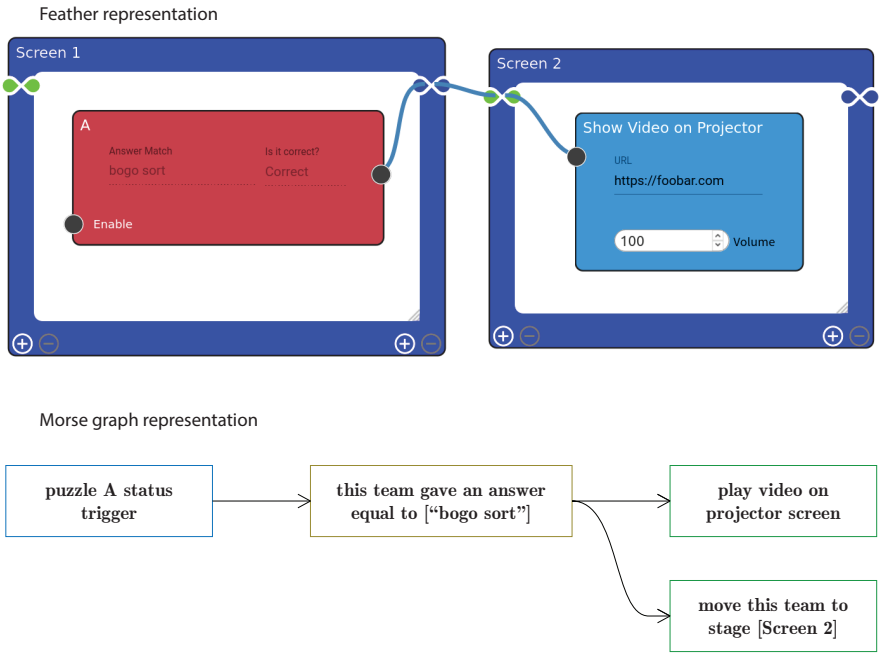


Figure 5.11: Two screens in the feather editor with a transition edge between them that is triggered when the answer on puzzle A is "bogo sort" (top) and the desired output for second compile step when compiling these screens and their contained nodes (bottom). The MORSE Graph should strictly also contain conditions for checking whether or not "Screen 1" is the current stage of this team, but this is omitted in this figure for the sake of simplicity.

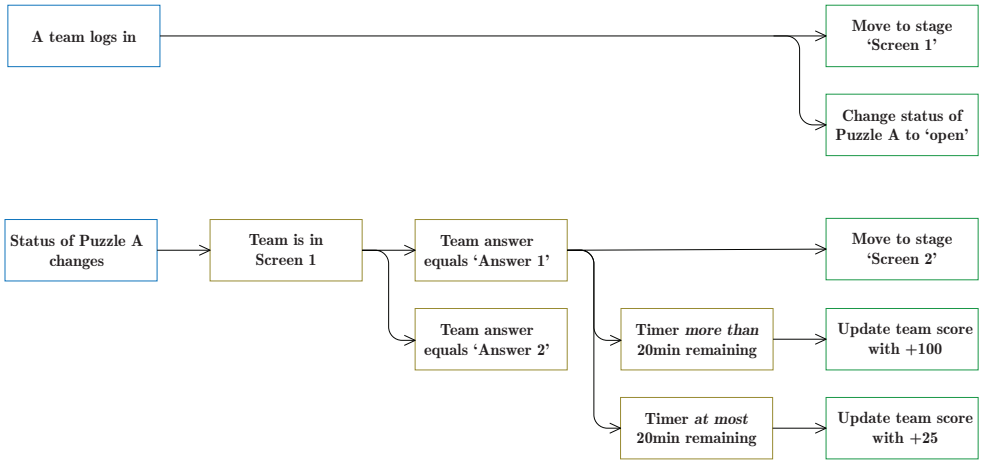


Figure 5.12: The resulting MORSE graph after having the finished the second compile step on the example configuration in figure 5.6.

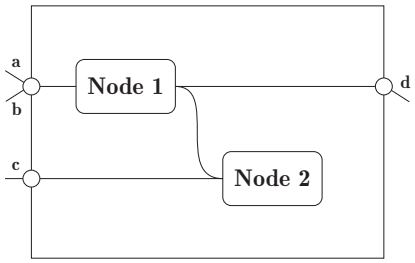


Figure 5.13: A graphical example of a "GraphBox". It has an internal graph containing two nodes. Node 1 is connected to Node 2, to an input port that is linked to two edges, a and b, and to an output port that is linked to one edge, d. Node 2 is connected to Node 1 and to an input that is linked to one edge, c.

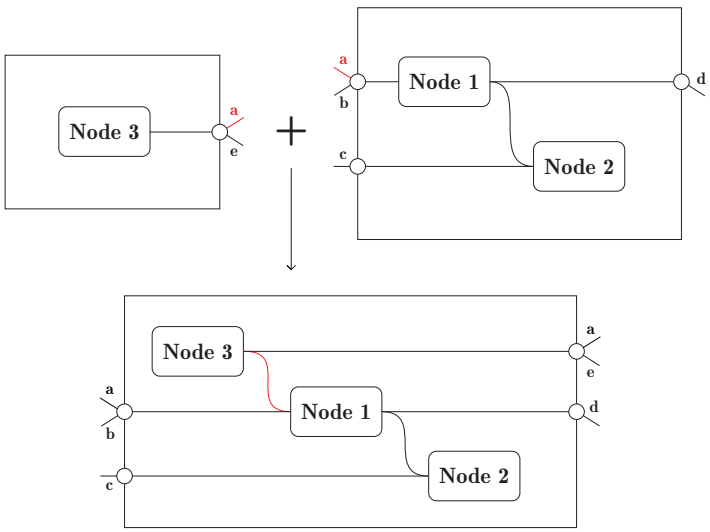


Figure 5.14: A visualisation of merging two GraphBoxes. The GraphBox on the left has an output with edge a and the one on the right has an input with the same edge. This means that, when merging, the nodes linked to the respective input and output will be connected to each other.

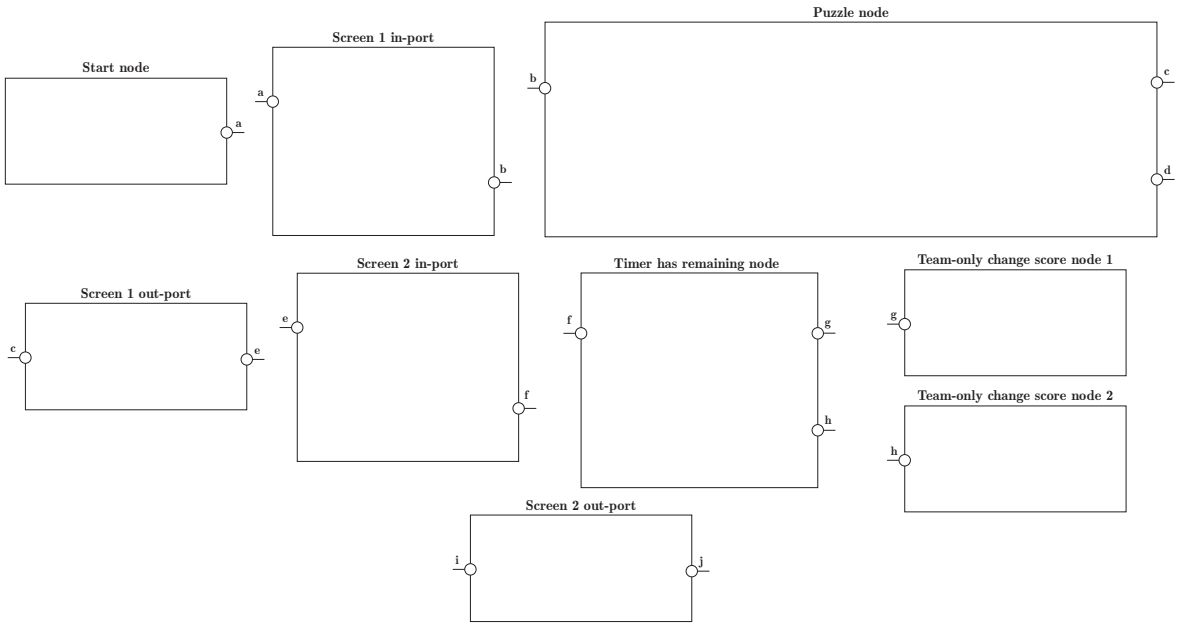


Figure 5.15: The empty GraphBoxes generated from the example configuration in figure 5.6. Every node and screen port has one corresponding GraphBox.

The core idea of the second compile step is to implement a method for every type of Feather node in the intermediate model that allows them to be mapped to GraphBoxes with rule elements as nodes and then merge all these GraphBoxes together to obtain a single, large GraphBox that contains the complete MORSE graph. This allows us to, as is required, map a single Feather node to multiple rule elements.

5.6.3. Performing the Second Compile Step

Then, the actual execution of the second compile step is done as follows:

1. To get the required GraphBoxes, for every node as well as every screen in-port and out-port an empty GraphBox is generated. These empty GraphBoxes have inputs and outputs corresponding to the incoming and outgoing edges of their respective node or port. The effect of this operation on the overarching example is shown in figure 5.15. It is necessary to generate GraphBoxes for screen ports as well because these are also involved in generating or connecting rule elements in the final MORSE graph.

2. Then, each GraphBox is populated with rule elements. The amount and type of rule elements that are generated inside a GraphBox depends on the type of its respective node or screen port. Performing this operation on the example gives us the collection of GraphBoxes seen in figure 5.16. Screen ports will always generate an 'Empty' rule element inside the node. This is not an actual rule element and these will be filtered out in a later step. For now, the empty elements are required because they nonetheless specify how inputs and outputs of a GraphBox are connected.

3. After that, all the GraphBoxes are merged together using the same strategy as seen in figure 5.14 to obtain one single GraphBox. This GraphBox now contains the logic for the whole escape room configuration. The final GraphBox obtained for the example is shown in figure 5.17.

4. Lastly, the inner MORSE graph inside the final GraphBox is extracted. This is done by removing the inputs and the outputs of the GraphBox as well as the edges connecting them to nodes. Simultaneously, the 'Empty' elements introduced in a previous step are filtered out again as these do not belong in the final MORSE graph. The path that they are a part of is not destroyed, however, because all neighbours to the left and the right of an empty element are connected to each other. Regarding the example, this gives us the graph seen in figure 5.18 which is in fact equal to the MORSE graph that we set out to create (figure 5.12). With this, the second compile step is concluded.

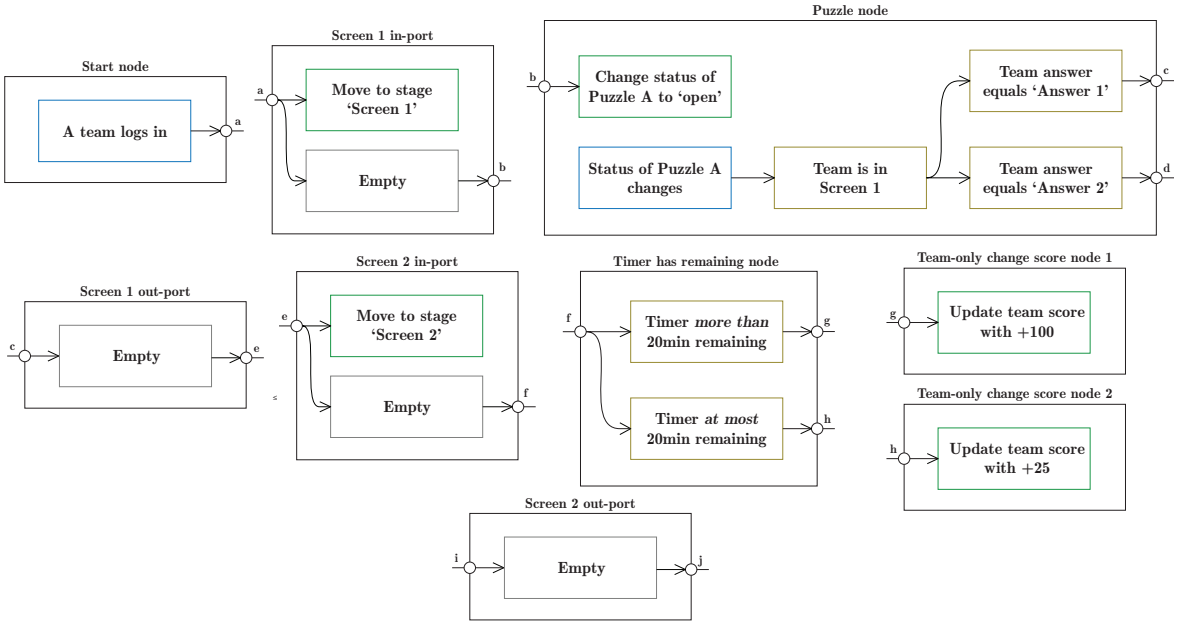


Figure 5.16: The GraphBoxes generated from the example configuration in figure 5.6 after being populated by each node's (or screen port's) specific `compileToMorse()` function.

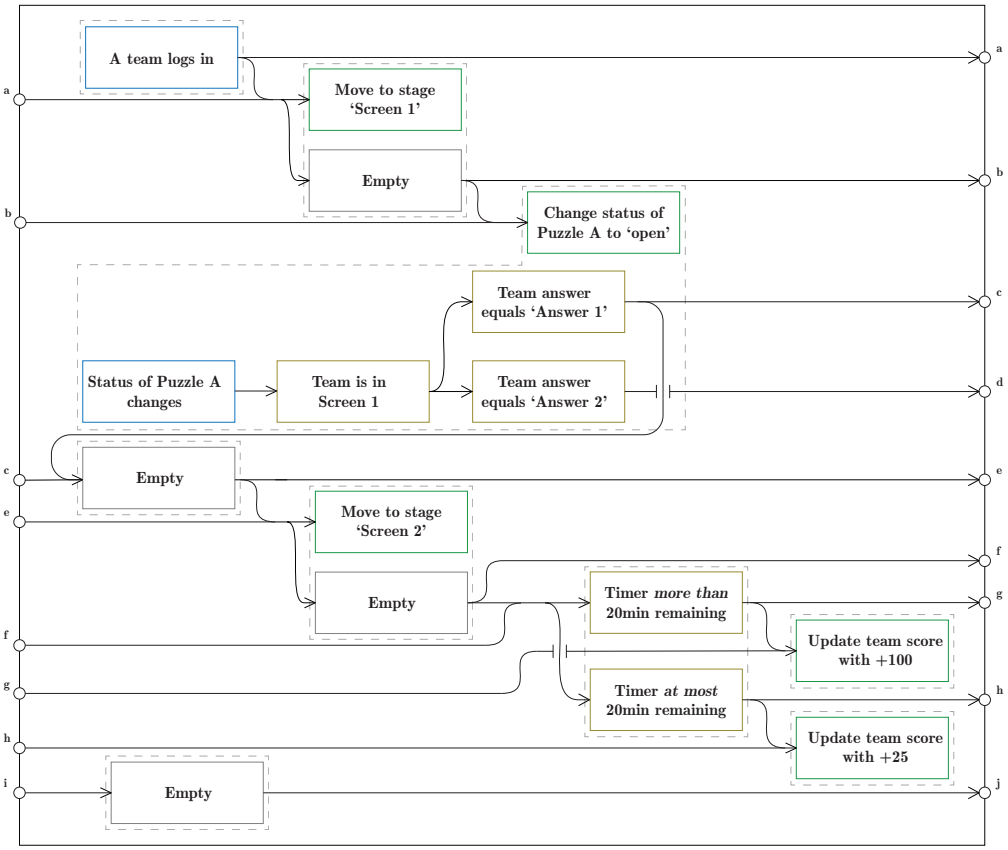


Figure 5.17: The final GraphBox obtained by merging together all smaller GraphBoxes in figure 5.16. This one GraphBox contains all logic defined in the example configuration in figure 5.6. The dashed boxes denote the groups of rule elements that were originally part of the same GraphBox and were thus generated from the same node.

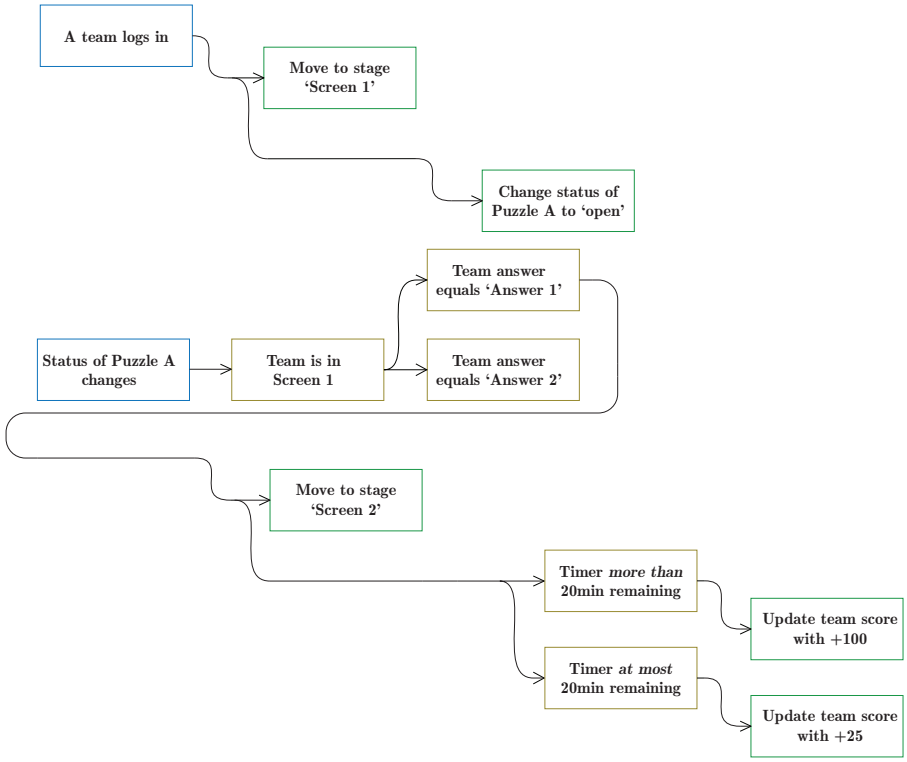


Figure 5.18: The remaining MORSE graph after sanitising the GraphBox from figure 5.17 by removing empty elements and its inputs and outputs. This graph is equivalent to the one seen in figure 5.6.

5.7. The 3rd Compile Step: MORSE Graph to Rules

The previous compile step resulted in a graph of MORSE rule elements. From this graph, a list of rules needs to be generated. In this step the graph is assumed to only have valid rules in valid orders. This means all paths in the graph follow the order trigger - condition - action. Furthermore illegal rules should be filtered out by the previous steps. Partial rules can be left in the graph: paths without a trigger or an action will not be made into a rule.

The initial idea to extract rules from the graph was the simplest implementation possible: create one rule for each path from a trigger to an action in the graph. This implementation is illustrated in Algorithm 1.

This system works, but it potentially creates more rules than necessary. MORSE allows rules to have multiple triggers, but this implementation will always create one rule for each trigger. The same also holds for actions: it is possible for rules to contain multiple actions, but this algorithm will never generate such rules. This is a problem because, while it will not affect the running behaviour of the escape room, a system that generates more rules will result in MORSE's ruleset editor becoming cluttered when exporting the rules. That will in turn make it harder for the user to understand or make manual changes to the generated rules. It is thus desirable to minimise the amount of MORSE rules that is generated from a given MORSE graph.

It is not hard to show that this initial implementation idea performs poorly when it comes to this. To illustrate this problems, the graph of figure 5.19 will be used. For this graph, six rules would be created:

1. T1 → C1 → A1
2. T1 → C1 → A2
3. T1 → C1 & C2 → A3
4. T2 → C1 → A1
5. T2 → C1 → A2
6. T2 → C1 & C2 → A3

To resolve the duplication of rules for multiple actions, the system should merge all actions that follow the same trigger and conditions. Instead of creating a new rule for each following action, the new implementation would combine all directly following actions, see Algorithm 2.

The new result consists of four rules:

Algorithm 1 Initial implementation

```
1: function COMPILE(graph)
2:     for each trigger in graph do
3:         for each condition in outgoing edges of trigger do
4:             COMPILECONDITION(trigger, condition, empty list)
5:         end for
6:     end for
7: end function
8:
9: function COMPILECONDITION(trigger, current, accumulator)
10:    accumulator.ADD(current)
11:    for each condition in outgoing edges of current do
12:        COMPILECONDITION(trigger, current, accumulator)
13:    end for
14:    for each action in outgoing edges of current do
15:        CREATERULE(trigger, accumulator, action)        ▷ This instantiates a resulting MORSE rule
16:    end for
17: end function
```

Algorithm 2 Adding all actions to a rule

```
1: function COMPILE(graph)
2:     for each trigger in graph do
3:         for each condition in outgoing edges of trigger do
4:             COMPILECONDITION(trigger, condition, empty list)
5:         end for
6:     end for
7: end function
8:
9: function COMPILECONDITION(trigger, current, accumulator)
10:    accumulator.ADD(current)
11:    for each condition in outgoing edges of current do
12:        COMPILECONDITION(trigger, current, accumulator)
13:    end for
14:    acts ← actions in outgoing edges of current
15:    if acts.size > 0 then
16:        CREATERULE(trigger, accumulator, acts)          ▷ This instantiates a resulting MORSE rule
17:    end if
18: end function
```

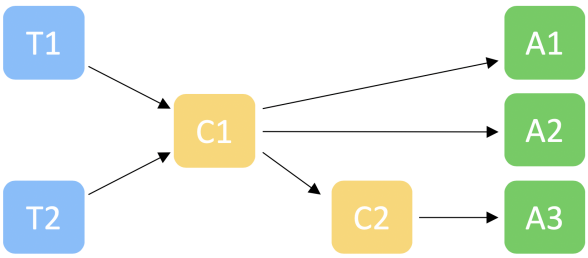


Figure 5.19: Example of a possible graph of rule elements. Names of elements are abbreviations of Trigger (T), Condition (C), and Action (A).

1. T1 → C1 → A1 & A2
2. T1 → C1 & C2 → A3
3. T2 → C1 → A1 & A2
4. T2 → C1 & C2 → A3

The last step is to merge the triggers that lead to the same conditions and actions. For this the implemented solution is the reverse of the solution for the actions: look for all triggers directly before a condition. That would mean that in the process of creating the rule 1 of the list above, at C1 the system would check for other incoming triggers and add T2 to the list. However, when performing the algorithm again for T2, the exact same rule would be created a second time.

In order to prevent the same rule from being created multiple times, the procedure was changed to start the creation of rules with conditions instead of triggers, see Algorithm 3.

Algorithm 3 Adding all triggers to a rule

```
1:  function COMPILE(graph)
2:      for each condition in graph do
3:          trigs ← triggers in incoming edges of condition
4:          if trigs.size > 0 then
5:              COMPILECONDITION(trigs, condition, empty list)
6:          end if
7:      end for
8:  end function
9:
10: function COMPILECONDITION(triggers, current, accumulator)
11:     accumulator.ADD(current)
12:     for each condition in outgoing edges of current do
13:         COMPILECONDITION(triggers, current, accumulator)
14:     end for
15:     acts ← actions in outgoing edges of current
16:     if acts.size > 0 then
17:         CREATERULE(triggers, accumulator, acts)        ▷ This instantiates a resulting MORSE rule
18:     end if
19: end function
```

This results in the minimal amount of rules required to represent the given graph:

1. T1 & T2 → C1 → A1 & A2
2. T1 & T2 → C1 & C2 → A3

However, this system has a limitation: as it starts creating rules at conditions, a rule without conditions cannot be generated. A separate step should be added to create such rules.

When converting rules that include conditions, optimising the rule to obtain the minimal amount of rules results in only one possible configuration. However, this is not necessarily the case for rules without conditions. As an example, for the graph in figure 5.20 optimising for the minimal amount of rules is not enough to obtain a single configuration:

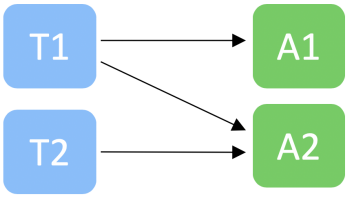


Figure 5.20: Example of a possible graph of rules without conditions. Names of elements are abbreviations of Trigger and Action.

Rule 1:

1. T1 → A1 & A2
2. T2 → A2

Rule 2:

1. T1 → A1
2. T1 & T2 → A2

In the first rule is trigger-based: for each trigger all connected actions are added to one rule. The second rule is action-based: for each action all connected triggers are added to one rule.

Both rules form valid configurations for MORSE, but the second one has a benefit over the first one: with the second implementation it is possible to allow multiple triggers to fire one action, but only allow this action to be executed once, while this is impossible with the second one.

In the ruleset editor a user can configure a maximum amount of times a rule can be executed. For example, a team could get a specific hint only once. With the first rule, say A2 was an action to send a hint, it could be triggered by both T1 and T2. Putting an execution limit of 1 time on these rules, the hint can still be triggered twice: once by T1 and once by T2. With rule 2 this problem is eliminated, as all triggers for an action are in the same rule. For this reason the second option was adopted. The final algorithm is displayed in Algorithm 4.

Algorithm 4 Final implementation

```
1:  function COMPILE(graph)
2:      for each condition in graph do
3:          trigs ← triggers in incoming edges of condition
4:          if trigs.size > 0 then
5:              COMPILECONDITION(trigs, condition, empty list)
6:          end if
7:      end for
8:      for each action in graph do
9:          trigs ← triggers in incoming edges of action
10:         if trigs.size > 0 then
11:             CREATERULE(trigs, empty list, action)        ▷ This instantiates a resulting MORSE rule
12:         end if
13:     end for
14: end function
15:
16: function COMPILECONDITION(triggers, current, accumulator)
17:     accumulator.ADD(current)
18:     for each condition in outgoing edges of current do
19:         COMPILECONDITION(triggers, current, accumulator)
20:     end for
21:     acts ← actions in outgoing edges of current
22:     if acts.size > 0 then
23:         CREATERULE(triggers, accumulator, acts)          ▷ This instantiates a resulting MORSE rule
24:     end if
25: end function
```

The algorithm returns a list of all rules generated from the graph, so these can be stored in the database. Each of these rules is an unsaved document of the RuleSet astronomy schema. All that is left to do is setting

the `generated` field for each rule to true, as these are the generated rules. With this field generated rules can be distinguished from manually created rules. Lastly the `save()` method, provided by Astronomy, is called on each of the rules to store them in the database and the compilation is completed.

From that point on the rules are also editable from the ruleset editor that is present in the MORSE system. However, manual changes to generated rules are overridden when regenerating the rules later. Important to note is that a user can still add rules manually in the ruleset editor. These rules will not be overridden by the generated rules, only generated rules (identifiable by the `generated` field mentioned earlier) are overridden.

To be able to find a certain rule in the ruleset editor, the algorithm to create rules from the graph also adds a name to each rule. We consider the action to be the most important element in the rule, as it describes what is going to take place). For that reason the name consists of the name of the stage the action of this rule is in, followed by the type of the action. If the rule consists of multiple actions, the data from the first one is used.

When applying the final algorithm to the graph in figure 5.12, the following steps are taken:

1. Starting from the condition "This team is in 'Screen 1'", find all directly preceding triggers. This is only the trigger "The status of Puzzle A changes". When traversing the graph to the right, two conditions are found: "Team answer equals 'Answer 1'" and "Team answer equals 'Answer 2'". Inspecting the latter condition tells us that there are no conditions or actions to the right of it. Since it is not useful to create a rule without an action, this path is ignored. The former condition *does* have connections to nodes on the right and when moving in that direction, three different paths can be taken. One separate step is taken for each path:
   (a) The "Move this team to 'Screen 2'" action is found, and a rule is created with the mentioned trigger and conditions and this action. This is rule 3 of figure 5.7.
   (b) For the "timer has *more than* 20min remaining" condition, the action to add 100 points is found. A rule is created with the previously found trigger and conditions, and the timer over 20 minutes condition and the add 100 points action. This is rule 4 of figure 5.7.
   (c) For the "timer has *at most* 20min remaining" condition, the action to add 100 points is found. A rule is created with the previously found trigger and conditions, and the timer has at most 20 minutes condition and the add 25 points action. This is rule 5 of figure 5.7.
2. The other conditions have no triggers directly before it, therefore no rules are created starting from these.
3. Starting from the action "Move this team to 'Screen 1'", find all incoming triggers. This is only the trigger "A team logs in". A rule is created with this trigger and action. This is rule 2 of figure 5.7.
4. Starting from the action "Change the status of Puzzle A to open for this team", find all incoming triggers. This is only the trigger "A team logs in". A rule is created with this trigger and action. This is rule 1 of figure 5.7.
5. No other actions have a directly preceding trigger, therefore no rules are created starting from these.

This results in the desired output, which is displayed in figure 5.7. This is the set of rules that was expected to be the result of compiling the example configuration. With this, the general workings of the Feather compiler have been discussed in full.

5.8. Edge Cases

In this section some specific elements of escape room configurations are discussed that lead to problems with the described compile procedure. Also discussed are the suspected cause of the problem as well as the solution that we implemented or that we suggest as future work.

5.8.1. Triggers in Puzzle Nodes

When designing the puzzle node, we decided that it should combine a number of MORSE functionalities that are related to puzzles together in one node (3.1). Thus, the puzzle node was given trigger outputs for each of its answers. This allows the designer to let different actions to be performed depending on the answer given by a team. The usage of a puzzle node can be seen in figure 5.21.

When compiling the puzzle node, the output ports – if connected to anything – will result in rules that start with the following three elements:

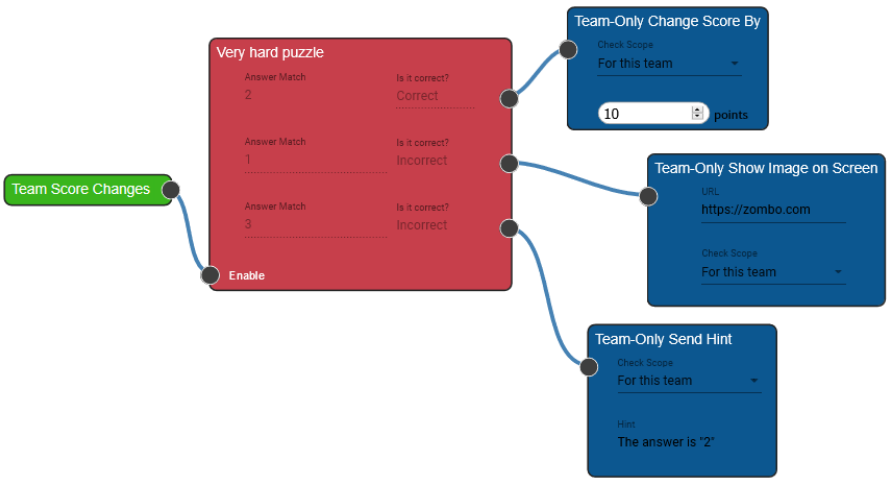


Figure 5.21: A puzzle node (red) connected to a trigger node (green) on the left and a number of action nodes (blue) on the right. The puzzle is enabled when the score of the team changes, and a different action is performed depending on the team's answer when an answer is submitted.

1. A trigger for the status of the puzzle changing. This trigger fires when a team submits an answer for the puzzle.
2. A condition that checks if the user is in the correct screen.
3. A condition that checks if the given answer is equal to the answer that the output is associated with.

This can also be seen in rule 3, 4 and 5 in figure 5.7.

The problem with this way of compiling the puzzle node is that it doesn't work well with MORSE's open questions. MORSE offers the possibility to be more lenient with the answers given to open questions. It allows the answer to slightly differ from the expected answer regarding its capitalisation, punctuation or white space usage (figure 5.22). However, the third rule element that is generated by the puzzle node – the condition that

Figure 5.22: A cropped screenshot of MORSE's schedule editor. Shown are the options for how lenient the system is regarding a team's answer on an open question.

does an equality check on the answer – does not respect these options. It simply does an equality check and that check will fail if the two strings do not exactly match. This leads to confusing situations where a team's answer is considered correct by MORSE system, but where the actions connected to the respective output port of the puzzle node will not be executed.

After the client pointed out this problem, we decided that it would be best to make two types of puzzle nodes: one for multiple-choice questions and one for open questions. The open question version would then not have an output for every answer but would always have just one output. This output is fired when the puzzle has been solved. To correctly compile this behaviour, the rule elements at the start of the generated rule should be a trigger for when the puzzle is solved and then just the check if the team is in the correct screen (figure 5.23). At the time of writing, this has not been implemented yet.

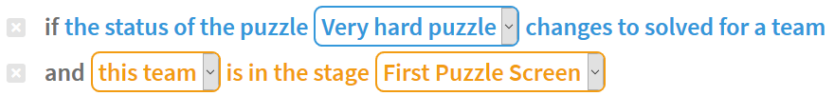


Figure 5.23: The trigger and the condition that should be generated when the output of an open question puzzle is connected.

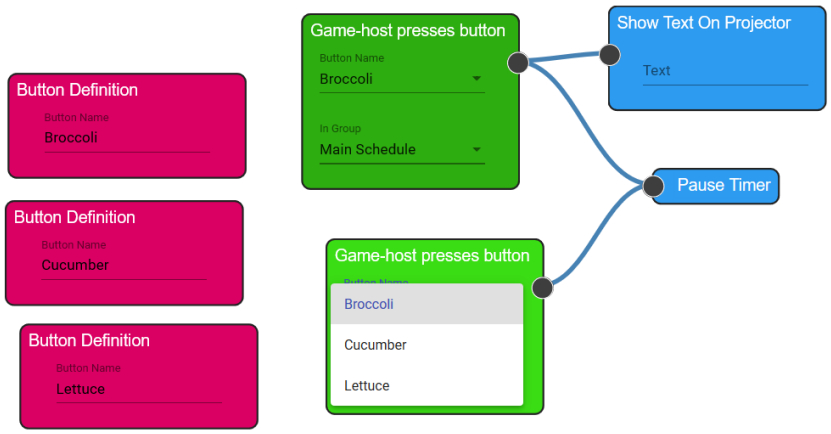


Figure 5.24: An example of how button definitions can be used in Feather. Three buttons are defined which will then show up in the drop-down of a 'host presses button' trigger node. There are two 'host presses button' trigger nodes for the 'Broccoli' button. These nodes have three connections to actions and therefore three buttons named 'Broccoli' are generated in the host's dashboard (figure 5.25).

5.8.2. Screen Transition Conditions

An important feature of the Feather editor is that nodes can only be placed inside screens and that the logic they describe – if it doesn't lead to illegal rules – is scoped inside screens. In other words, rules generated from a team-only trigger node will only be executed if the team is actually in the screen that the node resides in. The compiler enforces this by generating rules that include a condition that checks if a team is in the correct screen. This can be seen in rule 3, 4 and 5 in figure 5.7.

From rule 3 in the same example it however becomes clear that this scoping can lead to problems. This rule namely includes an action for moving the team to a new 'Screen 2' stage. And while rule 4 and 5 have the same trigger as rule 3, we can not be sure if their actions get executed. If rule 3 gets evaluated first by MORSE then the team will move to another screen and then the actions of rule 4 and 5 can no longer be executed due to the scoping condition. The correct behaviour would be that no actions will be blocked by the scoping condition because the 'puzzle status change' trigger fires when the team is in screen 1 but depending on the order in which MORSE evaluates the rules, the system might deviate from this behaviour.

In order to solve this problem without removing the screen scoping feature altogether, some modifications have to be made to the existing MORSE rule evaluation system. It could be altered such that rules with the same trigger are (virtually) performed in parallel. No changes to the game's state – such as the stage that teams are in, the score of teams, the timer value – should be made until all conditions of the respective rules are evaluated. Another simpler but less elegant solution would be to alter the order in which MORSE evaluates rules. It can be made that screen transition actions are always evaluated last in cases where multiple rules are triggered at the same time.

5.8.3. Button Triggers

While in Feather buttons need to be defined explicitly by using a button definition node (figure 5.24), MORSE will simply generate a button in the host's dashboard when there exists a rule containing a 'host presses button' trigger. MORSE will generate a new button every time such a trigger is used. However, it does so even if triggers reference the same button name and the same button group. This will result in multiple buttons with the same name being generated that are each only linked to one rule in the ruleset editor.

This is problematic because one would expect that if there is only one button defined using the button definition node in Feather and that button is referenced in trigger nodes throughout the escape room, that only one button would be generated that controls all these triggers simultaneously. However, due to this limitation of MORSE, multiple buttons will appear in the host's dashboard, each controlling only one of the triggers. And it will not be clear which button is linked to which trigger as all buttons will have the same name (figure 5.25).

This problem is made worse by the fact that the third compile step (5.7) prefers to generate rules with few actions over rules with few triggers. This means that if the a trigger node is (indirectly) connected to multiple action nodes, then multiple rules will be generated with the respective trigger element duplicated across the different rules. If the trigger node is a button trigger node then this will result in multiple buttons with the

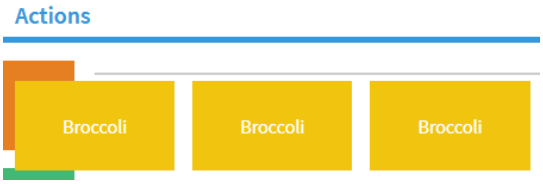


Figure 5.25: Three buttons in the host's dashboard in MORSE with the same name that should have been just one button.

same name, even if this is the only trigger referencing this button.

A proper solution to this problem will involve updating the existing MORSE system such that the 'host presses button' trigger no longer generates multiple buttons if they specify identical names and groups. This solution has not been implemented yet.

5.9. Time Complexity

To conclude this chapter, a brief worst-case time complexity analysis is done on the compile operation to find an asymptotic upper bound to the run-time of the compiler (in big O notation). This is a rough estimation of the complexity and by no means a formal proof. Knowing the exact time complexity is not of great importance to this project as in a typical Feather escape room configuration the number of nodes and edges is unlikely to grow into the hundreds. This analysis' main purpose is to give an intuition for the running behaviour of the compiler. The following values will be used throughout the analysis:

- N : The number of nodes and screens in the escape room configuration.
- P : The number of node ports and screen ports in the escape room configuration.
- E : The number of edges (including screen edges) in the escape room configuration.
- $n = N + P + E$: The total input size of the compile operation.

First, the time complexity of the individual compile steps (as discussed in 5.1) will evaluated. Those are then combined to obtain the complexity of the full compile operation.

1. Both the operation of saving to the database and the operation of performing the first compile step can be done in $O(N + P + E)$ time as they are not much more than a linear mapping of all elements of the configuration.

   Thus, we get $O(n)$ in terms of the input for the first compile step.

2. In the second compile step, the converting of the intermediate model to GraphBoxes is again a linear mapping, so $O(N + P + E)$. The number of generated GraphBoxes is $O(N + P)$. The merging operation is somewhat inefficient, $O((P + E)^2)$, due to the fact that as the GraphBox grows, the number of inputs and outputs also keeps growing (as is visible in figure 5.17). Merging two GraphBoxes together namely requires a pass over every pair of input edges and output edges. The amount of merges done grows linearly with the amount of GraphBoxes, $O(N + P)$. Extracting the MORSE graph and removing the empty elements can again be done in $O(N + E + P)$ time.

   This all combined gives $O((N + P) \times (P + E)^2 + (N + E + P))$ which is $O(n^3)$ in terms of the input for the second compile step.

3. We assume the number of nodes and edges in the MORSE graph to be $O(N + P)$ and $O(N + P + E)$, respectively. The number of rule elements generated in a GraphBox by a node is either constant or grows with the amount of ports of the node. The edges include besides the ones generated inside the initial GraphBoxes also the edges from the original Feather configuration.

   The third compile step iterates over all trigger-to-action paths found in the MORSE graph. These paths possibly overlap. The length of such a path is at most $O(N + P)$ – the length of a path can not surpass the amount of nodes – and we make the assumption that there are at most $O(N + P + E)$ of those paths.

   This gives us a time complexity of $O((N + P) \times (N + P + E))$ which is $O(n^2)$ in terms of the input for the third step.

When combining these three values, it can be concluded that the compiler is expected to run in $O(n^3)$ time. The second compile step is the bottleneck and also seems the best place to look for performance improvements. Especially the merge operation of GraphBoxes can probably be implemented more efficiently.

However, given that the escape room configurations are unlikely to grow very large, having a compiler with a time complexity of $O(n^3)$ is acceptable.

5.10. Summary

In order to be able to use the visual editor for configuring escape rooms, it must be integrated with MORSE. This is achieved by means of the compiler, which creates MORSE rules from the visual representation. The process of compiling these rules has multiple steps. First the configuration is loaded from the database and transformed into the intermediate model. From this model a graph of MORSE rule elements is compiled. Finally, that graph is transformed into a set of rules. After the rules are stored in the database, the escape room is ready to be tested and played.

6

Quality Assurance

In this chapter the main practices for quality assurance will be discussed. More information will be provided on both the code quality of the created product as well as the overall usability and feature characteristics. The chapter will be divided into two important sections concerning quality assurance: verification and validation. In the section on verification the main methods and techniques used will be discussed, explaining how it was ensured the product is up to the best coding standards. The section on validation focuses on the usability and functionality of the system and the process taken to test the suitability of the new feature for the client's needs.

6.1. Verification

Throughout the project different types of code verification techniques have been employed.

Static analysis has been used to ensure the code produced is up to the best coding standards. For this purpose ESLint has been used. This is a static analysis linter tool that can identify bad coding practices in JavaScript and TypeScript code.

For the dynamic testing, unit tests have been developed together with the code. These tests ensure that the system works as expected and no unexpected behaviour is present. The extensiveness of these tests is then taken into account for the total coverage in the project. As of the time of writing of this report the overall coverage on the master branch of the repository is 88%.

The above mentioned technique, has been used in the CI/CD pipeline on the GitLab repository of the project. There two different pipelines have been ran on each push. One running in the context of the merge request independent of the merge result, and one on the merge result of the merge request.

Furthermore, each new feature and thus merge request, has been thoroughly reviewed by the group's members and have been merged only after the approval of at least two group members, who have not been working on that specific feature.

6.1.1. SIG

In week 6 of the project the system's code has been sent to the Software Improvement Group (SIG) for evaluation. This group performs advanced static analysis on the code and reports any possible problems with the quality or maintainability of the codebase. The provided feedback is divided into 8 main categories: Volume, Duplication, Unit size, Unit complexity, Unit interfacing, Module coupling, Component balance, and Component independence.

- Volume concerns the size of the codebase, as stated by SIG larger systems can introduce maintainability risks.
- With duplication the code duplication of the project is measured, meaning that the bigger the amount of duplicated code is, the lower the score.
- Unit size as the name suggests tells more about the size of the units (functions, methods, classes). Short units with single responsibility are preferred and thus score higher in the analysis results.

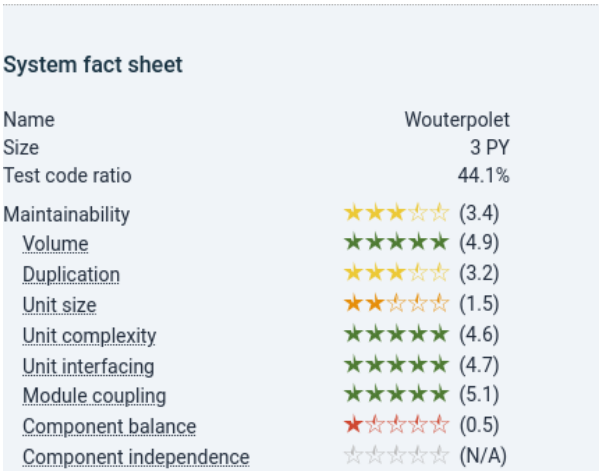


Figure 6.1: SIG results of the first submission.

- Unit complexity looks at the amount of paths possible in a single unit of code. The more complex the code is, the harder it is to be understand and tested.
- Unit interfacing concerns the number of parameters a unit has. Larger interfaces could result in error prone and hard to modify code.
- Module coupling evaluates the separation of concerns in the project. Modules with single responsibility are better in terms of project maintainability.
- Component balance and component independence are metrics concerning the project architecture and thus together with volume will not be taken into account for the purpose of this project and report.

The results of this initial analysis have shown some concerns with the unit size as well as some small code duplication.

Our Response To account for the concerns mentioned above we have extracted the relevant for our product candidate classes for refactoring. This showed a couple of problems that were present in our codebase.

First was code duplication, which was present in the code for the client side. Each node that was used in the editor had its own angular component, even though they were very similar. To minimise the code duplication, we have abstracted the main functionalities of the Node and Control classes.

Then there is the unit size, which concerns among others the size of the methods. While the ESLint settings already enforce a maximum amount of lines of the methods, these rules were suppressed in some places. All these methods that were actually too long have been refactored, mainly by extracting some functionality to a separate method. After this refactor, the code review has been made stricter to not allow any unnecessary suppression of ESLint rules.

Finally, while the rating of the unit complexity was not bad, we noticed some room for improvements. Some methods had a complexity that was too high. To allow us to easily find and prevent new method with high complexities, we added a new ESLint rule limiting the complexity of a method to five. This means that a method is allowed to have up to five possible paths. This rule was mostly useful, though in some cases it proved to be a bit too strict. For example for methods with a very readable and understandable switch case statement. In these instances it was chosen to suppress the rules.

With the report received from SIG there were some concerns present from the work of the previous BEP group, the one that developed MORSE. Those issues were outside of the scope of our refactoring as the code base was being used by the other group performing their BEP at RSG. However, we do believe that those problems should be addressed in the future as they can hinder the maintainability of the system.

At the time of writing this report, the feedback of the second hand-in to SIG has not been received yet.

6.2. Validation

Throughout the project a series of user tests have also been conducted to ensure that the product is best fit for the client's needs.

In the beginning of the development a user test using development mock ups has been made with the designers at RSG. Using a white board, brainstorming system called Miro, we have asked them to move around and configure an escape event using the initially designed building blocks in the form of bubbles representing different functionality. The feedback received from this was incorporated in the actual design considerations of the system.

Around the middle of the project another user test was conducted, this time using the actual, working system. During this experiment the designers were asked to configure the following escape room :

> We would like you to create the following escape room configuration:
>
> - In the first phase ("Phase 1") the user is presented with new the first puzzle. The question of the puzzle is "Is winter a warm season?". "No" will be a correct answer and "Yes" an incorrect.
>   - If the user answers correctly and the timer is running he/she will transition to the next phase ("Phase 2") and unlock 2 new puzzles, if the answer is incorrect and the timer is running, the user will again transition but only the following puzzle will be unlocked.
> - In the second stage the user is presented another puzzle: "Is summer always in July?".
>   - If the player chooses the only correct answer "No", then it he/she transitions to the third and final phase ("Phase 3").
>   - If the user's answer is incorrect, nothing happens and the user stays in this phase.
> - In the final phase, once the timer reaches 30 minutes remaining and the game is not suspended, the user is presented with the final puzzle. It contains the following question: "Did you enjoy the game?" with the correct answer "Yes".
>   - If the player selects that he/she exits the phase and the game is over.

A couple of issues came forward with this user test. First was that the performance of the editor was not good enough; it would slow down to the point that is was unusable. Aside from the performance itself, there were some critical limitations: Screens could not be resized or connected to other nodes for instance. Second, the assignment was confusing. The configuration that was to be made was not entirely clear. The schedule items that were created had some dummy values in them that were very confusing. Finally, the explanation of the system (both of the editor and MORSE itself) given at the start was not complete enough.

Still, from the results and feedback received during the tests we have evaluated the main issues present in our system as well as some possible design decisions that we have made but were not well perceived by the user. In the final weeks of the project these issues have been worked upon and design principles have been reevaluated. On top of that the main issues with the user test itself were taken into account for the new user test. This includes explaining the system more and providing a smaller, more logical configuration to be made. Additionally, the most critical problems with the program had been fixed.

The final user test was conducted with one of the designers of RSG at the end of the project. They were asked to configure a small escape room in both the ruleset editor as well as the visual editor. This allowed for a better comparison between the old and new way of configuring the logic and seeing if the new editor actually improved the usability. The escape room rules were as follows:

- The event should start with the puzzle screen.
- When it starts, the puzzle should be enabled.
- If a team answers incorrectly, a hint should be displayed to them with the text: 'Helaas'.
- If the team answers correctly, they move to the 'End Screen'

During this user test the designer reported they were quite happy with the way the visual editor worked. Compared to the ruleset editor it gives a better overview of the event. The performance and intuitiveness were deemed good, respectively graded with a 7 and an 8 out of 10.

The most important improvement points that came forward from this meeting were about usability and design. Regarding the design we tried to get as much feedback as possible, as this falls within their area of expertise. Most of this feedback was implemented the same day, drastically improving the looks of the editor.

The most valuable information we got from this user test is that the designer is no longer afraid to use the configuration system now that there is a visual editor.

Taking into consideration the above mentioned techniques we have ensured the quality of the product throughout the entirety of the project. The quality of the result has been confirmed by both the reports of SIG and the satisfaction of the client with the usability of the product. Future extensions of the product should account for the methods of validation and verification used and try to incorporate them into their process as well.

7

Process

This chapter describes the way in which the project was performed. First the general development method that was used is explained. Then the way the group communicated, both within the group and with the involved parties outside of the group, is described. Finally, some concrete problems that were encountered during the project and a general reflection on the performed development process are presented.

7.1. Development Methods

During the research phase, the group decided to use the agile development methodology SCRUM. In SCRUM, the development is divided in sprints, each lasting a certain period of time. In this project the duration of the sprints was set to 1 week. At the start of each sprint the group would create the backlog together.

At the end of the sprint, which is just before the start of a new one, the progress of the tasks is evaluated. In terms of SCRUM, this is called the Sprint Review. Any tasks that are still open can be moved to the next sprint or, if needed, modified or delayed. Problems that were encountered and their solutions during this sprint are documented. This can include specific problems for certain tasks, but also general problems encountered with the process. Especially the latter kind of problems invite for discussion during the sprint review, so that a way to prevent it in the next sprint can be found. Finally, any major design changes are documented for later reference.

SCRUM also offers guidelines for the meetings that take place. As explained in section 7.2, the group implemented the daily stand-ups. Furthermore at the same day of the sprint review, a demo is given to the client to show the achieved progress.

7.2. Communication

With the project taking place during the outbreak of COVID-19, everyone had to work from home. This obviously brings challenges to the communication. To ensure we all work together as a team, we settled on fixed working hours each day. During these hours we would also be online on Discord. Discord offers many features but most notably multiple voice and text channels, allowing us to use separate voice chat rooms. More towards the end of the project we started to use separate voice channels more often, so we could have fewer distractions while keeping the option for quick communication. To allow for flexibility in the planning, it was decided that missing hours are acceptable so long as they are made up another time.

At the beginning of each day we would have a stand-up meeting, during which everyone tells what they worked on and achieved the previous day and what their plans are for the current day. Such a stand-up meeting was also held with the company: Raccoon invited us to their daily stand-up meetings. In addition to that, the group held weekly meetings with Jan-Willem, our contact person from Raccoon. Another weekly meeting was held with the other BEP group working on the MORSE system, to keep each other up to date on the progress and make sure there would not be any issues when merging the code later. Additionally every week we tried to merge the progress both groups have made together in a BEP branch of the original MORSE repository. This would be helpful for our client in the end of the project when both products need to be incorporated together in the current system. Lastly once every two weeks we had a meeting with our TU coach.

This communication style has proven efficient and effective for the team's dynamics. In the future, in case of remote work restrictions, a similar idea can be employed. The benefits of the whole team being approachable during work hours result in better communication among the team and allows the client to reach out the group easily. The separate voice chat rooms provide the team members with enough privacy to concentrate on their work, while still being approachable. Finally the morning stand-ups both between the team and with the client provide all involved parties with a clear overview of the progress and the tasks being developed.

7.3. Encountered Problems

A few issues arose during the project that caused us to not be able to follow the initial planning. In this section a few concrete encountered problems that caused the delay in the schedule and the way the group dealt with them are described. Later in section 7.4 some general reflection on the process as a whole is performed. With this reflection the group mainly looks back on the development process and how this changed over time.

First, the impact of these problems should be made clear. The initial timeline was as follows:

- **Sprint 1** Groundwork has been laid, e.g. pipeline and needed frameworks are setup.
- **Sprint 2** Must-have feature set is partially implemented.
- **Sprint 3** Must-have feature set is (almost) completely implemented.
- **Sprint 4** Must-have feature set complete, should-have features' implementation started
- **Sprint 5** Should-have feature set nearing completion, could-have consideration starts.
- **Sprint 6** Rounding off of the codebase and feature set freeze at start of week.

While some issues arose during the start of the project, the work done in the first two sprints roughly followed the timeline. After that, the timeline became too ambitious, because of the concrete problems explained in the next sections and the more general issues explained in section 7.4.

7.3.1. Getting Familiar with the Existing System

As the MORSE system is proprietary to RSG, the team had never seen it before the project started. During the research phase we tried to get to know the system as much as possible. We documented all functionality of MORSE relevant to our project. Furthermore we received an example configuration for an escape room from RSG, so we could see the system in use.

In the beginning of the implementation phase we ran into issues with the original codebase several times. Sometimes we were unable to locate or comprehend certain parts of the code. Fortunately one of the developers from last year's BEP group offered his help and he answered most of our questions. From that point on we continued on our own and learned about the system as we got along.

7.3.2. Remote Working

As mentioned earlier, the project was largely performed remotely. Event though this change was anticipated from the start, some unforeseen challenges were caused by this different way of working. This sections describes the two main issues that arose from the remote work environment during the project.

Review Code reviewing is an important activity to maintain the quality of the code base. However, from time to time some frustrations arose from the code review. These frustrations had two main causes:

1. High response time for the changes of the review. (especially towards the start of the project)
2. Impatience when one's merge request had to be reviewed.

Due to the distance involved and lack of physical presence during working hours we had to deal with the challenge of communicating code review and editing. Sometimes this would cause feedback to be received slowly, issues to be resolved slower, and for people having to wait on each other before they could get a merge request merged and start building on that work. This was mostly solved in the later weeks by being more attentive to changes in merge requests.

Communication Even with the measures taken to ease the communication, some problems were observed. First is the focus that someone can have while being in the voice call. Often it is easier to focus when you don't hear people talking in the background. In person someone would usually use headphones, or sit somewhere else for a moment. The solution that was found within the communication platform Discord, was to make more use of separate voice channels. Every team member could be in their own voice channel and whenever someone wanted to discuss something they could join others in a channel. This worked very well, allowing for more focus without cutting down on any important communication.

The second issue we didn't expect, however, is the level of verbosity in issues and merge request descriptions and discussions. While descriptions and comments on issues and merge requests should always be as verbose as possible, it was even more important with the communication over discord. In some cases the descriptions and/or comments lacked this verbosity, occasionally leading to confusion and misunderstandings. These were mostly resolved verbally, after which the comments or descriptions would be updated to take away the confusion.

7.3.3. Reevaluation of requirements

Due to the above mentioned issues in week 6 the team had to reevaluate the initially set requirements and together with the client make a plan for the rest of the project. At this point the schedule had already had about a week of delay. Most of the initial Must Haves as specified in section A.7 have already been implemented, but a new prioritisation of the rest of the features was needed for the successful completion of the project.

We came up with a draft proposal for the new priorities of requirements and after a discussion with the client the final version looked as shown in Appendix E. One of the main points to notice is that a lot of the general features that were present in the should have requirements were discarded from the planning. This came from the fact that there were some critical issues with the editor. We discussed with the client and agreed that it would be better to have a well functioning editor with a minimal amount of features, rather than a bad editor with a lot of features. Furthermore some suggestions from the second round of user tests were incorporated, such as including a warning when the user leaves the page with unsaved changes.

In retrospective this was a feasible plan and the team managed to stick to it and implement almost all of the features specified.

7.4. Reflection

Besides the concrete problems that were encountered during the project, the progress of the project was also influenced by the general development process. In this section we reflect on the way we organised and planned the project. We describe what could have gone better and what has been improved during the course of the project.

7.4.1. Sprint Planning

An important part of SCRUM is the planning of the sprints. Planning a sprint effectively is a challenge and the sprint backlogs that were made to plan each sprint have improved during the project.

Initially, the sprints that were planned were rather big. First, the estimated time that was taken for certain issues was too low. Being inexperienced with web development and the frameworks that were to be used increased the amount of time needed to complete the tasks. Second, the total amount of planned time was too high. At first the planning aimed at filling most of the hours in the week with tasks. After the first two sprints this proved to be infeasible, as a lot of time is needed for other tasks of the project, like meetings with various parties, or code review. The main actions taken to improve the planning of the sprints was adjusting the estimated effort needed for a task to have room for solving unknown or complex problems and by planning less features in one sprint.

Having sprints with a feasible amount of tasks is also very important for the prioritization of tasks. Too many tasks in one sprint introduce the risk of getting caught up in tasks that can wait, as opposed to tasks that should be completed as soon as possible.

Besides the size of the backlogs, we also found that the creation of backlogs and retrospectives was not as effective as possible with the strategy described in the research. The initial strategy was to have both backlogs and retrospectives in the GitLab issues and milestones. This caused the process of creating the backlog to be very long, as essentially only one person could create issues at a time. Of course the remote working environment partly complicated this as well. Furthermore it was hard to find a good place to indicate the time spent on issues and to create an overview of unfinished tasks. To remedy these issues it was decided to

use Google Spreadsheets to create the sprint backlogs and retrospectives. This increased the speed at which the backlogs were made, as every team member could type at the same time, and also gave a good place to create the retrospectives. The issues and milestones on GitLab were still created to easily be able to see what should be done in certain merge requests.

With the improved backlogs we have seen that the focus on the important tasks improved and that the sprints we more successful in general, meaning that the progress made in a sprint became larger.

Some parts of the development process with respect to the agile methodology, were performed properly from the start. The planned meetings, with the client, coach, other group, and between the member of this group itself were beneficial to the project from the start. In general the proposed frequency of these meetings have largely been followed and served their purpose well.

7.4.2. Anticipation of Feature Size

A second area of improvement is the initial analysis of the size of certain features or parts of the system. As (part of) a web application, the product is divided into two main parts: the client-side and the server-side. On the server-side we also had to implement the exporting of the visual configuration to the rules. This is one of the most challenging parts of the system. Even though we were aware of the complexity of the exporting system, this should have been acted on earlier.

The complexity and size of the export system should have been considered more, starting from the research phase. While the capabilities of the MORSE system and the design of the visual language were described and thought about in detail, the bridge between them was missing. Furthermore, after the research phase more priority should have been given to such a big and important feature. With an earlier high level overview of the export system, work on it could have started more and could have been divided over multiple tasks earlier. This would have also allowed for more gradual progress towards this part of the product.

8

Discussion

This chapter will discuss some main points concerning the project and product developed in the past 10 weeks. It will touch upon the the overall feeling about the created system and how satisfied the involved parties are. Future extensions and improvements will be proposed as well as general tips about maintainability of the product will be given. The chapter will end with an important discussion about the ethical implications of our system and the measures taken in ensuring its responsibility.

8.1. Evaluation of Requirements

At the end of the project we have evaluated the initial requirement list. The list with implemented features can be found in section F.1. A list with the requirements that have not been implemented in the project is present in section F.2. These lists also include the features that were added in week 6, when a new plan for the final weeks of the project was made.

8.2. Team Satisfaction

At the start of the project the team was hesitant as to what to expect from the final product. With little experience in web development and a number of unfamiliar frameworks, we were uncertain how the project would go. Although nothing is ever completely finished and you can always make more improvements, the final result exceeds our expectations. From the final interviews with both on of the designers at RSG and our client it was apparent that the new system is a great improvement upon the previous one. This really reassured us as a team and made us proud of what we have accomplished. Despite the minor tweaks that can still be done, the team is satisfied with the results.

Furthermore, we enjoyed the process of developing this project to the fullest. The team at RSG welcomed us warmly into their day-to-day work routine and made us feel like part of the company. We have enjoyed our time both working with them as well as getting to know each other outside of the work environment, during after work activities such as online yoga and virtual Friday drinks.

8.3. Suggested improvements

In week 6 the team created a new requirements list as described in subsection 7.3.3. With this re-prioritisation we deemed it unfeasible to work on many of the improvements for the editor that were set in the should-haves. These features are the main suggested improvements upon the editor and can be found in subsection F.2.1. All of these requirements are non-vital for the smooth usage of the system, however they could provide some quality of life improvements to ease the design process. One especially notable one is the undoing and redoing of actions, as this was suggested in one of the user tests.

As discovered in the end of the project, the MORSE system has some limitations on the ruleset creation. Currently rules having team related conditions or actions, such as stage transitions or checks, can only be connected to team triggers. We find this limitation quite restricting for the type of rules that can be created and believe this can be improved upon and eliminated in the future. For example, any rule should be able to be configured in such a way that it only has an effect in a specific screen, but currently such rules require a team trigger.

Furthermore the incorporation of the schedule item creation into the Feather editor could be developed. This will allow the user to fully configure escape events using only the visual editor and simplify the whole process. This feature was not developed due to time constraints, but the team believes it will be a nice improvement of the Feather visual editor.

8.4. Future extensions

In the future the MORSE system can be extended to accommodate more types of triggers, conditions, and actions. A concrete example would be a 'Team enters a stage' trigger, which would greatly simplify the generated rules by providing a starting trigger for each state.

Another feature that the designers at RSG suggested during the user interviews, but was outside of the scope of this project, was the possibility to have a preview of the player's screen while configuring the escape event. This will nicely fit with the visual representation present in the Feather editor and give an idea of the feel the game will give to the teams.

Finally, a system similar to Feather could be developed for the other configuration software our client has Sensory Communication Inside Live Escape Rooms (SCILER). This will allow the user to visualise the game flow and ease the design process. We believe great inspiration can be taken from Feather in order to integrate a visual editor into SCILER.

8.5. Maintainability

With the addition of the Feather editor the maintainability of the whole MORSE does not increase much, but is also did not decrease. We have made sure that the libraries used are compatible with the existing system and thus only the the regular version updates would be required. Furthermore, the system is implemented in such a way that it allows easy extensibility of the features present. If new features are added to the editor the development of those should follow the principles described in chapter 6 of this thesis such as unit tests, static analysis, and usability tests. In addition to the present maintainability techniques, the group has updated the project's README file so it corresponds to the projects functionalities. This file contains examples of command-line commands that one can execute to quickly and easy run the project, its tests, and static analysis.

8.6. Ethical Implications

As the product of this project builds upon the already existing MORSE system, the ethical implications of the new system are closely related to those of the old one. Bakker et al. (2019) have discussed the overall system's implication from ethical point of view and those have not been changed by the new additions to the system made by our team. Furthermore, the new feature interacts only with the game host and does not require any further data collection, minimising any possible morally related issues.

The discussions mentioned in this chapter are good reference points for future maintainability and extensibility of the MORSE system as well as the Feather visual editor. Further developments and improvements upon the system should take into account the recommendations proposed in the sections above. Future teams should make sure that the maintainability, extensibility, and ethical implications are not compromised, but rather extended and improved upon.

9 Conclusion

After 10 weeks of intense work we believe our product is an adequate solution to the client's needs. The problem of not having a good visualisation tool, that is easy to use even by people without technical background has been an issue for Raccoon Serious Games for a while. The current system was not intuitive to the designers at the company and thus was not used in the day-to-day creation of escape events. The only real user of the MORSE system was one of the co-founders of RSG, who has a background in computer science. This fact led to the dependency on said person for the configuration of all digitally based events.

With the development of the Feather editor, we ensure that the designers can easily configure such events themselves. The visual editor presents a more intuitive way of creating rulesets by dragging, dropping, and connecting nodes. This design eliminates the algorithmic feel of the old system, while still supporting its full functionality.

To ensure the developed solution is of the proper quality the team has used various validation and verification techniques as mentioned in chapter 6. Testing and static analysis prove the code quality and functionality of the system, and user interviews assessed the usability and fitness of the editor to the clients needs. The real validation point was during the final interview with one of the designers at RSG, when they said they are no longer scared to use the system by themselves. On top of that the client is satisfied with and impressed by the end product.

This thesis provides insight not only on the product developed but also on the process taken in creating it. Doing so it answers the questions posed in chapter 2.

The first question posed is "How do we integrate the new UI in MORSE, such that it can be used as another page in the admin view of MORSE?" The only way too allow for full integration in the system, is changing and extending the existing code base. This determined the use of certain programming languages and frameworks and limited the choice of frameworks and libraries that could be added. More details on the added page and editor are explained in chapter 4.

The second question is "How do we make sure that the visual representation can be used to configure the escape events of MORSE?" and is answered by creating the compiler described in chapter 5. This compiler takes the graphical representation that is stored in the database and turns it into the MORSE rulesets, that can be used by the existing MORSE system. Besides the compiler, this also imposed restrictions to the design of the visual editor itself. These restrictions mainly concern the types of nodes that can be connected to each other and where they can be placed.

The third question devised in the problem analysis is "How do we preserve all the functionality that the current editor of MORSE offers?" First, the visual editor can be used alongside the existing ruleset editor. This means that any functionality lacking in the visual editor can still be configured. In the MORSE ruleset editor, it is clearly indicated which rules are generated by the editor and which are manually created. Even though the old system can still be used, the most used functionality can all be configured in the visual editor.

The fourth and final subquestions is "How do we create a UI that fits the workflow of RSG employees and that is usable by users without a technical background?" The answer to this question leads back to the research phase at the start of the project. At this stage, the initial design of the visual language was determined, with the help of literature and user tests with the designers at RSG. The research was used as reference throughout the project when design choices had to be made. In the user test at the end of the project, this proved to be successful, as it indeed simplifies the configuration for an employee at RSG.

By answering these four subquestions, this thesis has answered the main question it posed: “How can we design and implement an integrated User Interface (UI) that facilitates the configuration of events in MORSE, which is usable by users without a computer science background or similar, while maintaining all of the functionality that the current editor has?” This thesis has described the product and process of the 10-week Bachelor’s End Project.

A

Research Document

A.1. Introduction

For this Bachelor End Project (BEP) we have been tasked to design and implement a visual editor for Raccoon Serious Games (RSG). Raccoon Serious Games is a game company offering, among others, various large-scale escape rooms in both physical and digital format. To facilitate the creation and monitoring of these escape rooms, multiple software systems are used. Unfortunately, most of these systems, including M.O.R.S.E. 2.0 (MORSE), require the user to be familiar with technical concepts from the field of software development. Since the designers of escape rooms with RSG usually do not have this background knowledge, these otherwise powerful toolsare not easily configurable by all the employees of RSG.

To solve this issue, RSG assigned us the task of creating a new visual editor for designing escape rooms that is more intuitive to use and better fits within the rest of their development cycle. The editor will primarily be built for the MORSE system. MORSE is a web-based tool used by RSG for running, monitoring, and building large-scale digitally-based escape rooms. MORSE already has an editor for implementing escape rooms, but the current implementation uses a text-based rule system that is cumbersome to use and expects too much technical knowledge from the user. The new editor will be integrated into MORSE and will coexist with the current editor or (preferably) even replace it entirely if the new editor is feature-complete compared to the current one. The other functionalities of MORSE should remain mostly unchanged.

Additionally, the new visual editor might be used with S.C.I.L.E.R (SCILER), another system used by RSG for building escape rooms. SCILER is currently used for physical escape room configuration (in contrast to MORSE which is used for online escape rooms). Adding support for SCILER to the editor is, however, not the main focus of the project. Therefore, SCILER will be not be further elaborated on here, but will be discussed in subsection A.3.2.

This document contains the research that has been done prior to the implementation of the product, the implementation of the visual editor, and the writing of the formulated list of requirements for the product.

Section A.2 discusses the general structure of escape rooms and the findings from studying the scientific research that has already been done on escape rooms. Section A.3 provides an overview of the systems currently used by RSG, such as MORSE and SCILER. For the editor we will create a visual domain-specific language for the user to implement escape room games in. The properties that such a language should have in order to be useful, accessible and powerful are discussed in section A.4. It would be labor intensive and therefore unwise to build a visual modeling system from scratch so in section A.5 a number of existing tools and frameworks for making web-based interfaces for visual modeling are covered. Section A.6 concerns the subject of user interface testing and provides guidelines for both user testing and automated testing. Then, based on the findings from all previous sections a requirements list for the product is presented: Section A.7 and section A.8 contain the functional and non-functional requirements, respectively. Lastly, section A.9 describes the software engineering methods that will be used during the project and the motivation for choosing these methods.

A.2. Escape Rooms

As the name suggests, an escape room is an experience where a team of players tries to 'escape' from a locked room by solving different puzzles or doing various tasks. Usually the team has to complete this task within a

given time span or by collecting points. Depending on the design and purpose of the game, the activities can be in a single room or in multiple ones.

A.2.1. What things tend to be included in escape rooms?

Escape rooms can be very different, but most still share some common aspects. Usually escape rooms feature a storyline. This can be a general theme, or a more detailed story about some event that happened.

Most importantly, escape rooms feature many different types of puzzles. We found the following ones to be the most common:

- Hidden objects
- Required communication (give one person the information, while another person has to solve the puzzle)
- Hiding in plain sight (players often look for very hidden stuff, they might miss the obvious)
- Hiding clues in images
- Math
- Pattern recognition
- Riddles
- Sound (music, vocal clues)
- Mirrors (e.g. mirror text)
- Require research (and provide sources)
- Perspectual illusions (looking at something in a specific perspective reveals a clue)
- Finding the correct order
- Matching items (e.g. place an object over an image to find a clue)
- Physical puzzles (e.g. untying a knot)

(Pedersen, 2019; Pedigo, 2019; *Puzzle Types Commonly Found in Escape Rooms*, 2019)

A.2.2. What is the general structure of an escape room?

When it comes to escape room organisation there are different aspects that have to be taken into account.

Depending on the design such a room can fall under one of the following categories: rooms without a leading theme, themed but not narrative rooms, narrative rooms, and fully narrative rooms (Stasiak, 2016), where narrative means rooms with a prominent story-line, themed rooms have an evident theme for the whole experience but not necessarily a story-line, and fully narrative rooms have both a story-line and a theme connected to it. However this is only one of the aspects concerning the structure of most escape rooms.

The general structure of the puzzles in most escape rooms follows the challenge, solution, reward composition (Wiemker et al., 2016). The players are given a challenge and asked to find a solution for it. Upon achieving this they will be given a reward for the accomplishment. According to the type of tasks performed, the puzzles are usually split between cognitive and physical, though some challenges can be a combination of both. Cognitive puzzles require the player's thinking and logic skills for their completion, whereas physical ones, as the name suggests, ask the team to perform physical actions for solving them. The final generic structural type of puzzle present in most escape rooms is the so called 'meta puzzle'. The solution for those types of challenges is a combination of answers from previously solved tasks.

Another important factor is the way the puzzles are arranged in the game itself. As researched by Nicholson (2016), the main structural organizations of tasks in escape rooms are open, sequential, path-based, or a combination of them.

In openly organized rooms the contestants have to solve tasks that are not related to each other, but together contribute to the final result. They can do this in whichever order best fits the contestants so long as, in the end, they solve all of the tasks.

Sequentially structured rooms follow, as the name suggests, a concrete line of events. In these the team starts with a predefined initial task that, once solved, will lead them to the next task in line. In this structure the team has little to no freedom when it comes to the order in which the tasks have to be solved. Usually this is due to current tasks relying on previously solved ones.

With path-based puzzles the participants can choose which path they want to solve first or, if they want, they can solve them all in parallel. The only constraint here is that the paths themselves are sequentially organized meaning that the tasks of a concrete path have to be solved in a given order.

In practice a lot of the escape rooms contain a mix of these three commonly used organization techniques, making the experience of the players more pleasurable.

A.2.3. Escape Rooms at Raccoon Serious Games

Raccoon Serious Games (RSG) offers educational games, among others in an escape room format.

One type of escape rooms RSG offers are digitally based escape rooms. These escape rooms can still consist of physical puzzles and can be tailored to fit the customer's needs. However, all teams have to enter their answers on a tablet, which runs software that controls the game flow. These tablets can for example show hints, determine what tasks the team can complete, and show the team's score. The screens shown on the tablet, as well as the rules for the flow of the game, can be configured in MORSE. Besides the teams, a game host is present. The game host will mostly use the game control screen (figure A.1), where they can press buttons, change the timer, or reset the escape room. Finally, a projector screen is shown to all teams during the escape room. This projector screen can show the time left, a scoreboard, or a configured image.

This type of escape room is a hybrid of traditional physical and digital escape rooms: the puzzles can be physical, so long as they result in an answer that is then entered digitally. This mix of technology and physical puzzles allows the escape rooms to be hosted for a large group of people, with multiple teams playing at once.

Besides these digitally based escape rooms, RSG also offers the more traditional physical escape rooms. Here the answers are given through the physical elements present in the escape rooms itself. These escape rooms are designed for smaller groups of people, forming one team at a time.

The software that is used to create and run these different types of escape rooms is further discussed in section A.3.

A.3. Current Software Systems

Due to the unique nature of the two types of escape rooms on offer at RSG, two separate software systems are required: The system used for the digitally based escape rooms is called M.O.R.S.E. 2.0 (also referred to as MORSE), which will be analysed further in subsection A.3.1. The physical escape rooms are configured using S.C.I.L.E.R. (also referred to as SCILER), which is analysed further in subsection A.3.2.

As escape rooms in general often follow the same structure, the new editor could be used for both types of escape rooms. To meet the client's needs, the new editor should primarily support the escape rooms configured in MORSE and support for SCILER should only be considered a plus.

In this section the systems' functionalities and configurations are analyzed separately, and then the two systems are compared to find their similarities and differences. Furthermore, the frameworks and programming language used for MORSE are analysed, as this determines what will be used for the editor.

A.3.1. M.O.R.S.E. 2.0

M.O.R.S.E. 2.0 was developed by Bakker et al. (2019) and is the system currently used by Raccoon Serious Games to run their large scale, digital-based escape rooms. In MORSE the configuration is split into two parts: the schedule and the rulesets. This distinction was made to enable the high flexibility needed to be able to use the system for various escape rooms.

Schedule The schedule contains various schedule items. These items are the puzzle, the schedule group, and various kinds of screens. The screens are directly tied to what the players of the escape room will see. As the front-end is not desired to be changed in this project, the product should support all these items in their current format. Each of the items are listed below:

- Group - Contains other schedule items
- Puzzle Screen - Shows a puzzle to the player. There can be multiple puzzles in one puzzle screen, from which the player can then use one through a menu
- Text Screen - Shows a predefined piece of text
- Image Screen - Shows a predefined image
- Register Screen - Shows fields that can be used to register a person
- Puzzle - Can only be added to the puzzle screen.

All these items have some properties that are configured in an editor screen (figure A.2). The properties and the functionalities of each of these screen are listed in more detail in section A.11. The schedule does not define anything about the flow of the escape room. The path that a team takes through the schedule is defined with rulesets instead.

Rulesets These rulesets contain the logic of the escape room and follow an if-this-then-that structure. This structure also inspired the current UI for editing the rulesets (figure A.3). The ruleset is activated by a trigger. A lot of different kinds of triggers are available, which for example include a team logging in, the status of a puzzle changing, or a webhook coming in.

When the ruleset is triggered, the conditions are checked. Conditions include, among others, whether a certain answer has been given, whether the team is in a certain stage, or how many points a team has. If these conditions are all true, the actions will be executed. These actions denote what happens in the escape room, like playing a sound, moving the team to another part of the escape room, or calling a webhook with a certain value.

While the rulesets are a fundamental part of the system and offer great functionality, it is not very intuitive to set them up correctly. Therefore we try to focus on the functionalities they offer, instead of their exact implementations. A risk introduced by this approach is that the editor could be limited in functionality compared to the original system. Therefore, the editor should coexist with the existing configuration to prevent any loss of functionality.

The list of functionalities offered by the MORSE system can be found in section A.11. This does not follow the exact elements available in MORSE, but still gives a complete view of the configurations that the current configuration process offers.

Used Language/Frameworks In this part the currently used language and frameworks are discussed. As these make up the current system, we will also use them in the development process.

TypeScript TypeScript (TS) is a language built on top of Javascript (JS). Its syntax is a superset of Javascript's, allowing JS developers to easily move to TS. TS is an object oriented language, making it more suitable for building large scale applications. The code is transpiled to JS, thereby giving it the same browser support.

Angular Angular is a framework for developing web applications. Built on TypeScript, it provides templates to support different concepts within web applications much easier than otherwise possible with HTML and JavaScript. Its main feature is the separation of all different components: it has templates to define the HTML for a specific part, and a component to retrieve the data.

MongoDB MongoDB is a schemaless database system. This fits the project very well, as MongoDB stores object in JSON and is queried using JavaScript. Using Mongo instead of SQL eliminates the painstaking process of changing the DB scheme whenever the data classes are changed.

Meteor Meteor is a platform that aims to speed up the web development process drastically. It integrates well with the aforementioned tools and provides a low latency. The biggest advantage is that Meteor takes away the syncing process between the different tools: It takes care of database integrations, client/server messaging and data synchronization to the web pages. One of the great features is a very low latency. Traditionally users have to wait for their actions to reach the server, be processed in the database and then for the server to reply. Meteor speeds up this process drastically by providing a client-side mock database (MiniMongo) that the data is processed in, and later these changes will be relayed to the server in the background.

A.3.2. S.C.I.L.E.R.

SCILER is a software system created by Hanou et al. (2020) and is used to monitor a escape room where a small group of people are locked in a room and need to solve puzzles to escape. Like MORSE, the escape rooms have to be configured before they can be used with SCILER. Where MORSE provides a UI to enable this configuration, SCILER requires a JSON config file to be written. With complex escape rooms, this JSON file becomes very long.

As the desired editor only applies to the configuration of the escape room, the research will not analyze the entire SCILER system. Instead, it highlights the structure of the configuration files, and tries to find similarities and highlight the differences between the configuration of SCILER and the configuration of MORSE. This is the only part that is important for the integration with the editor.

In SCILER each device has a configuration file and the server has the main configuration file. This main configuration file contains information about the following components:

- General information about the escape room itself
- The cameras in the escape room
- Devices used in the escape room
- Timers, which can be used to set time constraints on solving parts of the escape room
- Puzzles themselves, containing their rules and possible hints
- General events, consists of a list of rules, that define the behavior of the event.
- Button events, consists of conditions and rules. These are executed when a button is clicked in the monitor.
- Rules, that define the logic of the escape room.

A.3.3. Similarities and Differences

MORSE and SCILER show some similarities. In both systems the escape rooms are divided in two parts: The components and the logic.

In MORSE the components are the represented as the schedule items and the logic is configured using the rulesets. SCILER, on the other hand, stores the components as puzzles, devices, and cameras. Here the components are more related to physical objects, rather than the screens that are shown on for example an iPad. Like MORSE, SCILER uses rules to define the logic of the escape room. These rules are triggered by interaction with the devices in the escape room, and can check conditions before they execute the actions. This has a very similar structure as the ruleset of MORSE, with one of the few differences being the explicit notation of triggers in MORSE's rulesets.

The similarities in general structure and more importantly in the representation of logic of the two systems encourages the use of the same editor for both systems. The differences in technical representation between the two systems can be accounted for by designing the software properly. When the new software is designed to be extensible, support for the different technical representation will not require a large rewrite of the system. The difference in components poses a larger challenge. This would require changes in the editor itself to allow an accurate visual representation for configurations of both MORSE and SCILER.

Taking these similarities and differences into account, we are convinced that extending the editor to support SCILER is possible, but it is not straightforward. Moreover, as the client also suggested, it should not be a big priority for this project. Finally, the potential support for SCILER should be taken into account while implementing the editor.

A.4. Visual Modeling Languages

As part of the project, it will be necessary to design and implement an interface in which employees of RSG can design and define logic for escape rooms and games. Consequently, we must design a visual language in which designers without an in-depth technical background can intuitively produce such escape rooms and games. We looked for existing work about the design of such a language to arrive at design principles to use during development.

This section is divided into five parts: First, some guidelines and best practices are discussed when considering the visual editor as a Domain Specific Language (DSL). Then, in a very similar subsection, we go over some guidelines and best practices when considering the visual editor as a tool for creativity. Next, some existing tools are highlighted that can serve as examples to the DSL that we will be designing. After that, we present our findings from our conversations with employees from RSG and letting them work with mock-ups. Lastly, based on these findings, guidelines, and examples we come to some conclusions about the design of the visual DSL.

A.4.1. Domain Specific Language (DSL) Design Principles

In their paper Karsai et al. (2014) lay out several design principles when designing a domain specific language. These principles were split into 5 categories with differing amounts of "recommendations" that they then made:

1. **Language purpose** "Identify language uses early", "Ask questions", and "Make your language consistent".

2. **Language realization** "Decide carefully whether to use graphical or textual realization", "Compose existing languages where possible", "Reuse existing language definitions", and "Reuse existing type systems".

3. **Language content** "Reflect only necessary domain concepts", "Keep it simple", "Avoid unnecessary generality", "Limit the number of language elements", "Avoid conceptual redundancy", and "Avoid inefficient language elements".

4. **Concrete syntax** "Adopt existing notations domain experts use", "Use descriptive notations", "Make elements distinguishable", "Use syntactic sugar appropriately", "Permit comments", "Provide organizational structure for models", "Balance compactness and comprehensibility", "Use the same style everyewhere", and "Identify usage conventions".

5. **Abstract syntax** "Align abstract and concrete syntax", "Prefer layout which does not affect translation from concrete to abstract syntax", "Enable modularity", and "Introduce interfaces".

These categories are organized in the order that one would follow when developing a DSL. As mentioned in the paper, these guidelines (especially when read in detail and not summarized) can be in conflict with each other, needing to be balanced with each other (Karsai et al., 2014).

Questions such as whether to use textual or graphical realization are already decided for us by the product constraints. However, there are several guidelines which are interesting to discuss here.

"Permit comments" is an interesting one for our project due to the nature of what we are trying to create. We will be creating a 2D interface with drag-and-drop features that a user can use to create executable programs. The support for comments will be added in the form of a kind of sticky notes with text in the editor screen as well as allowing components to be given names.

"Use descriptive notations" is another interesting design principle. How do we effectively show features of the 'language' visually? Using a lot of text will not help to make the editor any clearer, but there are other visual elements to consider. The rules are made up of trigger, conditions, and actions. Difference in these components can be expressed by using different colors. After talking to a designer at RSG, we realised that using icons for components in the editor would be even more effective. After all, unlike colors, icons have a meaning themselves already. Using icons can then link this meaning to the components of the escape room.

"Identify language uses early" and "ask questions" is what we are doing in the first two weeks of the project. Talking to members of RSG, analysing existing escape rooms configured in MORSE, drawing mock-ups, and interviewing designers of RSG helps us to identify the main usages and needed features for the editor. This is all done before the actual development phase of the product, so that changes can be made without any rewriting of the codebase.

"Adopt existing notations domain experts use". While we do analyse currently existing programs that serve a somewhat similar purpose in subsection A.4.3, the knowledge and notation that domain experts might have cannot be used in the editor. The simple reason is that the editor will be used by people that are no experts in the domain that the editor works in.

"Reflect only necessary domain concepts" in other words, only implement in the system that one is going to use. The prioritization of certain features come down to this rule. Many features of the old system were considered and core features were identified to prioritize the implementation of. The rest of the features from the old system, which we would still like to implement were then deemed to be 'should haves' (see section A.7).

A.4.2. Creativity Tool Considerations

In this section the findings of Resnick et al. (2005) and how these relate to the project are discussed. In their paper (Resnick et al., 2005) list and discuss the qualities of "composition tools". These are applications that allow users to interact with a search space and build their own creations within that space. The MORSE

tool and especially the component that we will be modifying in this project, the escape room editor, nicely fits within this definition and, thus, the qualities and guidelines provided in the paper are applicable to our product.

Exploration and Experimentation Because exploration and experimentation are key elements of creativity, it is important for composition tools to allow users to easily explore and experiment. Design principles that could be followed to achieve this include:

- *The tool should allow immediate modifications of all aspects of the design* (Resnick et al., 2005). It will thus be best to display the whole escape room on one screen as a composition of components and allow the user to add and freely interact with components.

- *All actions done by a user should be reversible, preferably automatically* (Resnick et al., 2005). This will manifest itself as an undo feature in our application. We see this as a high-priority feature as it greatly improves the user's ability to experiment and is unlikely to be challenging to implement for this system.

- *The user should be able to automate repetitive tasks* (Resnick et al., 2005). The simplest feature that we want to implement in the editor to improve it in this aspect is allowing the duplication of components and their configurations. What would also help in this regard is allowing users to save presets for components, but this will already require more work because it will require a storage system that works across projects. We might also look into allowing two components to be partially linked such that changes in the one node will result in the same changes in the other. This is however probably quite tricky to implement and thus hasn't the highest priority.

- *The tool should give an immediate, intuitive impression of the search space that can be explored* (Resnick et al., 2005). The work space should occupy the majority of the screen with the most fundamental tools (e.g. adding a component) also being displayed most prominently.

- *The tool should allow the user to sketch: make quick, rough designs without having to fill in all the details* (Resnick et al., 2005). Users should be able to experiment with the high level features and general shape of an escape room (e.g. the screens that a player visits and in what order) without having to define low level features and without the escape room having to be operational/syntactically correct. We find this an important feature and will try to incorporate it from the start. It will however require us to also implement error messages when a game-breaking syntax error occurs.

- *The tool should clearly reflect changes made by the user* (Resnick et al., 2005). It would thus be useful that, even if components are configured in a separate menu, changes to the configuration are still visible through the use of color or icons in the overall view. This is a feature we would like to implement but other features have more priority.

- *The app should avoid prompting the user to go down particular paths in the creative process.* E.g. by providing and prominently displaying three optional fields, users might feel compelled to always fill in all three of them (MacLaurin, 2011). We could improve our editor in this regard by greying out optional inputs and connection points or even make them invisible. This is however not the biggest accessibility issue and therefore not a priority, because our target user is an employee of RSG and not - as is the case in the study of MacLaurin (2011) - a child.

Accessibility versus Power Another quality of a creative application is that it is accessible: it should be easy to pick up for new users. This accessibility should not compromise the feature set however, as experienced users shouldn't feel that their ability to express themselves is limited because of the tool or application.

- *The tool should have "wide walls" which means that the tool offers and encourages a lot of different ways of being creative.* A sign of a tool not doing this properly is that all creations made by users appear very similar (Resnick et al., 2005, p.3). This thus means that our app should be able to be used to create interesting and distinct escape room designs. If that freedom is not given by the feature set currently offered by MORSE it would be good to extend on that. This is likely outside the scope of this project; we are tasked with reimplementing the current system visually, not extending its functionality.

- *The tool becomes more accessible for new users if it is impossible to make a syntax error.* If the combining of high-level concepts automatically forms coherent behavioral statements, then that allows the user to forget about the abstract grammatical structure and focus on the design itself (MacLaurin, 2011). Following this guideline, our new escape room editor should minimize the possibility of users creating these non-operational designs. Doing so, however, clashes with the guideline about sketching as is discussed in subsection A.4.2. Connecting high level components in the editor freely should be possible to allow the sketching of overall progress through the escape room. However, when configuring the actual logic of the escape room, it would be preferential to prevent all errors. Implementation-wise, this can be challenging, which is why at first just showing errors based on the changes could be acceptable.

- *The tool should prevent users from having to deal with abstract, technical concepts such as "compile time" and "run time".* This increases the accessibility for users without a technical background (MacLaurin, 2011); In our app, there should be no difference between the programming environment and the finished program. When designing an escape room in the editor, it would be already running and operational. This means that ideally, the visual representation will be transformed in the existing back-end configuration, whenever a change is made in the editor. It is quite likely that this is feasible to implement due to MORSE being implemented in Meteor which has strong support real-time changes.

- *Abstract concepts should be illustrated with real-world metaphors.* This is again to increase accessibility and will also help users to communicate about these concepts (MacLaurin, 2011). It is not a good idea, for instance, to include plain variables in our tool like in a regular programming language as this is quite a technical concept already. Furthermore, choosing terminology for elements that the user is already more familiar with can help speed up the learning process.

- *The tool should be as simple as possible.* One should be aware of "feature creep". The addition of new features might allow for more creative possibilities but can also obfuscate the core goal and intention of the tool, harming accessibility and user curiosity (Resnick et al., 2005). This guideline poses a challenge with regards to the requirements of the product. It should support all existing functionality of the MORSE system. This functionality would result in a lot of different components, allowing for a feature creep. To solve this, the editor should still support all the existing features, but will discourage or hide some of the current elements. The main focus can then be placed on the elements that are most frequently used when configuring the escape rooms. Additionally, some components that are now mainly used together can be combined into one. This way, the use of these components separately should be discouraged, but still possible to again avoid limiting the functionality.

Personal Styles of Users When designing a tool for creativity, one should be careful to not just cater to one small subgroup of users.

Resnick et al. (2005) discuss how MIT researchers tested the programmable LEGO pieces that they were developing by letting fourth-grade students build a LEGO amusement park using these pieces. What these researchers found is that one group of kids started off in a much more structured manner, not stopping to work on their design until it was operational, whereas another group let themselves be guided more strongly by their passion and impulsive creativity and therefore took a break from working with the programmable blocks to build an environment around their Ferris wheel.

The main takeaway from this is that users have different ways of being creative. Therefore it is important to not tailor our escape room tool around just one type of user or one style of working but to support many different styles.

In order to account for this, user tests should be performed on as wide a range of users as possible. Preferably, feedback will be gathered from all employees at RSG that are likely to use the editor in the future. More on user testing will be discussed in subsection A.6.2.

Collaboration and Compatibility The quality of an application for creative work also depends on the extend to which it facilitates collaboration, since in most real-world work environments people work in teams. When people with different talents and ideas collaborate, their efforts generally add up to a better product (Resnick et al., 2005).

Resnick et al. (2005) mention a number of ways to improve an app in this aspect, but not all of them are relevant to our use case. The fact that we build an escape room editor for one company that will not be openly

shared with the world, means it is not useful to add functionality that allows users to share their creations and problems with an online community, even though, in general, this is a good feature for creativity tools.

Collaboration related guidelines that *are* relevant, however, are:

- *The tool should allow users to work together on the same project* (Resnick et al., 2005). Ideally, users should be able to edit a project at the same time much like *Miro*, a tool already used at RSG for brainstorming. Making this experience work well when multiple people are working on the same escape room simultaneously could require a lot more effort to develop and is not one of the main goals of the project. However, due to the nature of the Meteor framework it will be easy to switch from one machine to another when configuring the escape room.

- *The tool should be compatible with other applications and file formats that are commonly used during the design process* (Resnick et al., 2005). One of the reasons that the editor is built, is to make the visual design directly applicable to the escape room configuration in MORSE. One could argue that it therefore is compatible with at least the MORSE application. Other design tools commonly used at RSG include Adobe Illustrator and Miro. Nevertheless, adding support for the file formats or exports offered by these programs is outside the scope of this project.

- *Allowing users to make and use custom plugins for the tool, will increase its usefulness.* Resnick et al. (2005) mentions this point mostly within the frame of online community-driven plugin development. However, because RSG also has employees with a computer science background and because it could very well be that MORSE will be further iterated on by BEP groups in later years, it is good to leave the system open for extensions. This will not result in direct support for plugins, but the system itself will be designed with extensibility in mind.

A.4.3. Examples

With the above mentioned in mind, a few examples of existing systems will be explored, examining their strengths and weaknesses.

Pipes Pipes, the spiritual successor to Yahoo! Pipes is a program that allows one to connect together different Web 2.0 services (*Pipes*, 2020).

Strengths Pipes includes a principle that could be useful in our design: The system uses four different types of blocks: Inputs, Manipulation, Control, and Output. The Inputs are from any feed on the internet which can then be manipulated, controlled, or output. The Manipulation blocks are about changing or filtering an input stream (e.g. replacing words, filtering out only pieces with relevant keywords). The Control blocks are about merging or duplicating input streams such that one can apply multiple filters on the same stream, or just apply one filter to multiple streams, etc. The Output blocks are about creating a new output feed from all the input feeds and streams that one has then manipulated using pipes.

Our DSL will be very different from the ones used in Pipes but some takeaway principles are useful: You have inputs, which can be manipulated, combined, or duplicated and are then put into an output (which of course can become an input again). This input, change, output system is one similar to one we are trying to implement.

Weaknesses In its current form Pipes can only take as input data streams from websites, manipulate, split, and combine these streams, and output that as another data stream. This is a quite expansive capability but it is still quite a ways off from being useful when setting up instructions for computers. There are no if-then statements or for-loops of any sorts and the design is likely only intuitive when one has experience with programming.

Kodu Kodu is a visual programming language that was developed for children by Microsoft (MacLaurin, 2011).

Strengths An interesting language aspect that is mentioned is the impossibility of a syntax error. Combined high level concepts automatically form coherent behavioral statements, allowing the user to forget about "abstract grammatical structure or sequencing rules" (MacLaurin, 2011). Another interesting language

feature to draw inspiration from is that there is no difference between the programming environment and the finished program. When working on the program, it is already running and operational.

Weaknesses One of the weaknesses that Kodu has is inherent in its design goals: Its simplicity. The same system that made Kodu so intuitive was the same thing that stopped it from becoming a truly flexible VPL. MacLaurin (2011) mentions that the designers of Kodu for a short time considered implementing subroutines and function calls but ended up discounting it due to the complexity that that would add. Instead they went with 'pages'. When a robot is initially created in Kodu it just presents the user with one programmable 'page' in which the user can start creating robot behaviour. "Most new Kodu programmers will spend several hours without realizing that a robot can have more than one page" (MacLaurin, 2011). Apart from the obvious positive of simplicity that this adds it does still very much limit the user as no parameters can be passed during a 'page switch' making traditional-style functions calls unwieldy.

Scratch Scratch is a visual programming language developed by MIT for people without programming experience and children: "Scratch helps young people learn to think creatively, reason systematically, and work collaboratively" (*Scratch*, 2020).

Strengths Scratch is a great implementation of a system that allows people to learn about the basic concepts of programming without being put off by the visual difficulties that it often presents. Users don't have to memorize programming constructs (or, worse, look through documentation) to be able to start writing executable code. The system employs a drag-and-drop, puzzle-like system to guide the user in the creation of what he or she wants to make. Furthermore, scratch has a system that allows users to share and build upon their and other's creations all within the same website.

Weaknesses The language that Scratch uses, however, is very general-purpose. This is not something that we should be replicating given what we are trying to build. Our language will be more domain-specific and should thus be able to work with higher level concepts without getting too many statements/blocks/instructions.

Flowgorithm A visual programming language that can compile to several other languages (*Flowgorithm*, 2020). It includes constructs such as If, For, and While and allows the user to arrange blocks with directed edges between them to set up a program.

Strengths In a way this is very similar to Scratch in the sense that one can use well-made overview-displaying visuals to familiarize themselves with basic programming constructs. This makes it, like Scratch, more accessible to the average non-technically minded individual who would like to make a start on programming. Furthermore, the fact that it can compile to several different languages makes it useful in some other narrow applications, such as teaching people the connection between the visuals they may be getting used to and how one would actually program something.

Weaknesses Like Scratch and Kodu, Flowgorithm is not particularly expressive. It cannot create classes nor objects. While it can create functions and has appealing visuals (both of which can be a useful takeaway) it is too low-level for the DSL that we are trying to make (in the same way that Scratch is).

A.4.4. Interviews with RSG employees

We had a number of interviews with employees from RSG, in which they explained their design process, we let them create their own visual representation of an escape room design, and we asked them to build something with one of our own mock-up languages. These sessions provided crucial insights that were more specific to the problem at hand than the insights from the earlier discussed research.

Player-oriented design The first thing to take away is that it is intuitive for designers to reason and design from the perspective of the player. That makes a sense, they build an experience for the player, after all. This player-oriented design should be a leading factor when designing the DSL. If we, for instance, would decide to call a particular syntactic concept an "action" then the user will assume it is a player action and not an e.g. system action. The meaning of concepts should align with initial intuition that designers have when reasoning from the player perspective.

While the naming of concepts should definitely not go against their defined language semantics, we should make sure that we do not make them too abstract and technical. To make terms less technical it is helpful to use analogies and metaphors. This follows from a guideline seen in subsection A.4.2. But again these names should make sense within the world of escape rooms from a player perspective or should at least be established metaphors in the world of user interfaces (such as gear standing for configuration or a pencil standing for editing).

The design process In their design process, the designers at RSG have a heavy emphasis on prototyping. They make quick mock-ups of escape rooms and then let colleagues comment on how it looks, feels, and plays. Based on that feedback they can then iterate on the design. It would thus be ideal if our DSL allows the designer to quickly come up with semi-functional prototypes without having to worry about details such as exact logical implementations. This is in line with the earlier discussed "sketching" guideline (subsection A.4.2).

Additionally, it became clear that the "look and feel" of an escape room is an important part of the design process that is also continuously iterated upon. This means that for designers it would be very helpful if not only the escape room's functionality but also the visual aesthetics of an escape room are presented in the editor. The emphasis on both prototyping and on designing the visuals means that a feature that allows designers to quickly and easily hop in to the game and play the escape room will also be of great help.

Rulesets One of the main considerations that arose while starting with the design of the visual modeling language was whether to visualize the current ruleset system or to replace this system in the editor entirely. This led to very different modelling languages, as can be seen in the mock-ups initially created in figure A.4 and figure A.5.

Some members of the team were concerned that the approach with the current rulesets would still be too technical and too complicated, even when visualized. In the interview the designers were asked to use the system with the rulesets to configure a small escape room, presented by some textual requirements. The visual rulesets turned out to be very usable and the principles of the system were picked up swiftly, even though the visual modelling language was not worked out at all. This let us to believe that using the rulesets visually in the editor is feasible. Moreover, this allows most, if not all, of the functionality of the current configuration UI to be transferred to the visual editor.

Visualizing the if-this-then-that structure of the rulesets is also in line with the client's preference.

Ambiguities In the experiment, we noticed that when a visual representation of an escape room is made without constraints, ambiguities are easily introduced. In a normal design this is not a real problem, but it poses great problems for the logic of an escape room. Trigger arrows would for instance all be connected to the same input node, with the designer intending them to be combined with a logical AND. To the system, however, it could be both AND or OR as the designer's definition is ambiguous. If the DSL were to default to OR, then that might lead to unexpected behavior and leave the designer wondering why it doesn't work the way he/she intended.

A way of solving this problem is to disallow ambiguities by treating them as syntactic errors. The user can then be informed about the error with a warning in the editor. A drawback of this approach is that it harms the creative work flow. Having to solve errors before being able to see the result interrupts the user's experimentation.

Other possibilities are to design the DSL in such a way that the defaults chosen for ambiguities are always in line with intuition or to define the syntax in such a way that the number of ambiguities is brought to a minimum.

A.5. Visualisation Frameworks

To display the flow of the escape room intuitively, we want to work with a flowchart-like visualization. Furthermore this chart should be editable, preferably using drag and drop tools. With these requirements we looked for and found a couple of potentially useful tools. For each of these tools we will examine sample projects to see to what extent they suit our requirements.

A.5.1. D3

Data Driven Documents (D3) allows "efficient manipulation of documents based on data" The examples listed on their website drew our attention. The tool provides an easy way of interacting with the DOM. However, D3 does not provide a graphical library: it is a tool to aid with modifying existing shapes with a focus on using external data for this. As we do require proper visuals as well, it is probably better to go with another library.

A.5.2. Go.js

Go is a library specifically made for creating interactive diagrams. It provides the main classes for objects required to create graphs of the type that we need and easy ways to modify the data. Additionally it integrates well with Angular, which is used in the existing MORSE system.

When building a small trial application, we found out that syncing data from changed nodes (by the user) back to our system would be relatively simple. There is a built-in tool to extract node and link data easily. Furthermore, there are event listeners for modifying links and nodes, clicking an object or changing selections which will also be useful for our purposes.

A.5.3. Storm React Diagrams

Based on Typescript and React, React Diagrams provides all components for designing diagrams. We were very impressed by their demos, as they show great flexibility of the different nodes. One of the demo applications shows the use of various ports for each node and clear flow paths from these.

The main thing that stands out from this library is that all nodes feature specific in and out links. Go.js seems to be made more for generic graphs, while React Diagrams is seemingly geared towards flowcharts.

A.5.4. Rete

Rete simplifies visual programming by providing a node-based structure. This would fit well for our flowchart-like visualisation. The package also comes with other useful additions, such as a dock menu and selection of modules.

The demo applications on the website show a number of nice features. However, when trying out some simple diagrams we encountered some issues. These included links that would dangle across the screen rather than follow the cursor and accidental zooming when trying to edit nodes. As our application should be as intuitive as possible, we will not use Rete.

A.5.5. Final verdict

Of the four investigated libraries, Storm React Diagrams seems to be the winner for our use case. This is due to its great support for flowchart-like diagrams and its extensibility.

A.6. Testing

When it comes to testing for our project the relevant parts concern user interface (UI) testing as this is the main focus of our product. There are two main topics we will be looking at belonging to UI testing. First, different user interface testing techniques will be discussed and how they are applied in practice. Then, a deeper look will be taken into what user testing is and how it is conducted. This section will be concluded by the main takeaway and design choices made from the findings.

A.6.1. How does one test UI

In order to verify the correct behaviour of a system, the developer usually writes different types of tests. The process of testing the general functionality and logic is usually simple and straightforward. Testing user interfaces, however, is often cumbersome and unintuitive. There are two main techniques that are used in practice when it comes to UI testing.

The first one involves manual usability tests, conducted by the development team and executed usually by the user of the system. Even though this method is easy to implement for the developers, it could be time and resource expensive and it does not guarantee that most of the existing issues and bugs will be caught. A further constraint is the fact that the system needs to be fully functional in order to perform user testing, leaving this only as a post development option. This approach will be discussed in more detail in subsection A.6.2.

The second option is usually employed during the development process together with the regular testing practices for the back-end logic of the system. It involves writing automated test suites that execute parts of the UI and try to simulate user behaviour. When it come to this approach the main way of constructing these tests for user interfaces is using the model-based technique (Bowen & Reeves, 2013). The workflow of it starts with constructing models such as flow charts, transition figures, and attribute schemes, which can later be used for the test creation. These models should be an exhaustive overview of the system's functionality so they can allow for full system coverage by the test suite. This test suite will reflect these models and test for the necessary attributes and transitions of the UI in an automated way. Within this second approach there has been extensive research on how this process can be simplified. The most popular technique involves configurable test strategies similar to the ones used for back-end testing.

Just like regular software architecture design principles, user interfaces have developed design patterns to help developers in implementing regularly reoccurring requirements. What researchers have shown is how one can use these design patterns to create generalized test strategies, which can be further configured to the needs of the system (Moreira et al., 2013). In this way the test creation process is simplified to the extend possible for the commonly used UI elements and designers can spend more time and effort on testing the other functionalities of the system. Even though this technique has for the most part been studied exclusively in academic research, there exist tools created by the scientific community such as the PETTool developed by Cunha et al. (2010).

A.6.2. User Testing

When software is created for a particular target group the developers have to make sure that the group's needs are met. In doing so one of the most often used techniques is user testing. This involves evaluating the product through user experience in various forms.

How is user testing usually performed? When it comes to user testing there are several different approaches that can be considered. The main techniques include questionnaires, interviews, and experiments and constitute for a part of the empirical method of usability testing (Brinkman, 2019).

Questionnaire When the development team wants to conduct a test that has a broad target test group and has a relatively shallow scope of the questions asked, the best approach will be to create a questionnaire. For this unmediated remote evaluation to be efficiently done a good set of questions has to be developed, which in most cases can be difficult. The developers have to take in mind that the questions have to be clear, neutral, and appropriate, as well as that all relevant terms have to be explained. In the end the results have to be analyzed such that appropriate conclusions can be drawn. This process is useful for big scale feedback on the system, but lacks on the depth of the evaluation.

Interview In contrast to questionnaires, interviews are generally executed on a smaller scale of users and consist of more in-depth questions about the product. With them the design team is present during the answering part and can follow up with more questions if needed, thus this is a moderated testing technique that can be done both remotely and in-person. Because of this structure the feedback from the user can be more concrete and the interviewer can filter out only useful information, making the analysis part less time consuming. The main drawback is the fact that conducting these interviews could be time consuming if the user set size is relatively big.

Experiment When using this testing technique the designers ask the user to use the system as he/she would do in everyday life. This could include asking questions in the process or making the user 'think aloud' while executing the tasks. For this purpose a working version of the software is required, as well as the presence of both the user and developer. During the process usually a log is kept of the user's actions that can later be analyzed. In the end a small interview can take place for the user to evaluate the experience.

Question's structure This section answers the question: "How do you structure your questions such that the responses are as informative as possible?" Depending on the purpose of your user test and the data that is desired to be collected there are two main types of questions one can ask. When the main goal is for quantitative data to be collected the main types of questions should be scale and multiple choice questions. On the other side, if one wants to receive more qualitative information, the questions should be structured

in a more open ended way, with 'yes/no' questions always being followed by explanation inquiry (*Usability Testing Questions: Asking the Right Questions*, n.d.).

Furthermore there are three main times when one should ask questions: pre-test, post-task, and post-test (Barnum, 2011). The developers should use an appropriate combination of these question times to gather the relevant information they need.

The use of pre-test questions is beneficial for a better understanding of the background of the user, such as technical expertise, habits, preferences, etc. This type of questions can give valuable information as to why the user responds to the system in a particular way.

When talking about post-task questions, we mean questions that are asked immediately after the user has executed a certain task. This type of questions aim at gathering the immediate feedback and reaction of the participant on a certain aspect of the system. They target specific functionality separately from the general workflow of the system usually constructed as a scenario by the designers.

Finally there are post-test questions, which contain inquiries about the whole system and all the scenarios presented to the user. Usually these questions take up majority of the interview time and thus are usually open-ended questions. For this purpose they can be structured as open-ended statements rather than questions to introduce variety in the interviews/questionnaires.

A.6.3. Design choices

As mentioned above, extensive functional user interface testing is usually complicated and rather time consuming. For that reason, apart from the regular unit testing techniques, the project will employ extensive usability testing together with the employees at Raccoon Serious Games. These tests will mainly consist of interview style questions with possible designs of the system presented to the participants. Furthermore, once functioning interfaces are developed small experiments will be conducted for design evaluation.

A.7. Functional Requirements

In this section we will go into the functional requirements of the escape room editor. We will be following the MoSCoW method.

A.7.1. Must-haves

General editor features. The editor must:

- Be able to represent rules such as triggers, conditions, and actions as components/nodes in the visual editor.
- Be able to display a 2D space in which components/nodes can be:
  - dragged
  - dropped
  - added
  - removed
  - connected
- Be able to display an overview of the detailed logic and the system flow corresponding to it.
- Be able to extract the rules from the visual representation and store them in the existing database model.
- Allow the user to continue with the same visual representation as they stopped with earlier.
- Be able to display error messages.

Expressiveness features. The editor must:

- Have a visual representation of the puzzle screen and its corresponding questions/puzzles inside.
- Allow the user to link screens (just puzzle screens for now) of the escape room together, determining the flow of the escape room.

- Allow the user to configure a starting point.
- Allow the user to configure triggers/conditions based on the timer (both value and whether it is running).
- Allow the user to add triggers based on the status changing of a puzzle.
- Allow the user to add and configure actions that change the status of a puzzle.
- Allow the user to add conditions based on the answers of a team to a puzzle.
- Allow the user to add actions regarding the state of a puzzle, which are executed given that a certain trigger is present and the underlying condition(s) are met.
- Allow the user to configure a button system for admins.
- Allow the user to link buttons to actions.
- Allow the user to configure the skippable field of screens and use conditions based on that property.

A.7.2. Should-haves

General editor features. The editor should:

- Allow undo and redo actions.
- Allow the user to duplicate components and their configurations.
- Allow the selection of multiple components at once and performing an action (such as delete/duplicate) on all of them.
- Allow using “abstract” components: high-level components that do not have their details defined yet but can still be used in the project as if defined.
- Have pretty visuals (e.g. appealing color palette and fonts) that are coherent with the existing system.
- Update the underlying rule sets in real-time.
- Have the ability to read existing/newly added rule sets and drop them in the visual at a random spot.
- Have a dynamic transition between detailed and general overview (e.g. with zooming by scrolling the mouse wheel).
- Have a system that checks for syntax errors and displays them to the user.
- Be able to include a commenting system (e.g. sticky notes or block highlighting)
- Be able to display an overview of the whole game, where detailed logic (e.g. number of questions and question flow) is abstracted away.
- Be able to give components names that are used in the visualization.

Expressiveness features. The editor should:

- Allow the user to configure projector actions (image, timer, audio, video).
- Allow the user to configure image screens.
- Allow the user to configure text screens.
- Allow the user to configure hints.
- Allow the user to configure actions that set the value of the timer, or pause/resume it.
- Allow the user to configure register screens.
- Allow the user to link an action to the event of media such as a video finishing.

- Allow the user to configure triggers/conditions/actions based on team scores.
- Allow an action to suspend the game (which can be bound to a button) and an action lifting this suspension.
- Allow the user to configure conditions that check whether the game is suspended
- Allow the user to configure actions that show an image to a team
- Allow the user to use an action to set the start time of the event.

A.7.3. Could-haves

General editor features. The editor could:

- Support keyboard shortcuts for important actions.
- Support auto formatting of the visualization.
- Allow users to save component presets.
- Allow components to be "linked" to each other. Similar to duplication but linked components keep their properties synced when they are modified.
- Allow the user to hide/show certain elements of the rule system.
- Make components still visually reflect their configuration when zoomed out or minimized through the use of colors and icons.
- Load JSON configurations of SCILER.
- Export JSON configurations that SCILER can use.
- Show errors that arise in the SCILER configurations.

Expressiveness features. The editor could:

- Allow a game admin to revert the ending of the game.
- Allow the user to add a web hook.
- Allow the user to configure the properties special to SCILER that are not in MORSE.
- Allow the user to use a MORSE configuration action.

A.7.4. Won't-haves

General editor features. The editor won't:

- Have dark theme.
- Have mobile device support.
- Have Multiple language support
- Allow multiple people to edit the same project simultaneously like Miro.
- Have a speech-to-layout feature
- Support import of file formats from other design applications, like Abode Illustrator

A.8. Non-functional requirements

A number of non-functional requirements have also been formulated. These are the following:

- The editor should be built on top of the MORSE system and extend its functionality without compromising code qualities of the original system.
- The editor's implementation should adhere to software engineering principles in order to ensure its maintainability and extensibility.
- Every employee at RSG should be able to learn to work with the editor by providing them with a user manual.
- The usability of the system will be assessed by the execution of user testing.
- The editor should run sufficiently fast to provide the user a pleasant editing experience.
- The editor should be modular such that it will be compatible with the modifications to MORSE made by the other group.
- The new editor and the old rule-based editor should be able to coexist within the renewed MORSE system.

A.9. Development Methodology

To ensure a successful completion of the project, a certain development methodology should be agreed upon with the members of the team.

A.9.1. Agile

In software development it is crucial to ensure that the right system is built for the client. Often this means that changes need to be made when the client uses the software. The further the team is along with development, the harder it gets to apply these changes.

Using an agile methodology allows the client to see a working version of the product early in the development process, making it easier to apply needed changes. One of the best known agile methodology is SCRUM. As the members of this group are all already familiar with SCRUM, this is the agile methodology that will be followed in the project.

The sprints will each last one week, from Tuesday to Tuesday. In principle, this allows for a total of 6 sprints in the time span of the project. At the end of each sprint, a meeting with the client will take place, where the product so far will be shown. Any feedback from this meeting can be incorporated in the planning of the next sprint.

Besides the meeting with the client, there are weekly meetings with the other group to coordinate the changes made to the codebase. Furthermore, every two weeks the progress so far will be discussed with the TU Delft coach.

The planning of each sprint will be made using the issue board on GitLab. In this issue board the status of an issue can be updated when they are being worked on.

Every morning the group will join the stand-up meetings at RSG. In these stand-up meetings, a highlight of the previous day and the main goal of that current day will be discussed. These stand-ups also enable the group to easily ask for help of one of RSG's members in case this is needed. Furthermore, to keep each other up-to-date on the progress a short meeting will take place at the start of each morning and afternoon. In this meeting the progress, plans, and obstacles can be discussed.

A.9.2. Code Quality and Git Usage

Besides the validation, the product should also be programmed well. To ensure a high quality of the delivered code, a couple of agreements have been made.

Continuous integration will be used to automatically run tests, check the codestyle, and determine the coverage the tests have. The group decided to use GitLab instead of GitHub, on which the current MORSE repository is hosted, to be able to use extra features like a built-in CI, and issue boards. Since there already is a Travis configuration in place, this will have to be converted to a GitLab configuration file at the start of the development phase. The test coverage will be determined using codecov, as this is already in place for

the current MORSE code base. For static analysis ESLint is the standard for both JavaScript and TypeScript. This has also been used in the existing system and a configuration for it is already in place. The current configuration offers a good ruleset, so this will be used in its current state. Naturally, if any issues come forward with the current configuration, changes can be made during the project.

Furthermore, some agreements on the git usage during this project have been made to allow easy analysis of the versioning of the software. Git commit messages follow a certain format, by finishing the sentence: If this commit succeeds it will <Commit message>. This implies the use of the imperative mood. Commits should be made frequently, and should be pushed frequently to get fast feedback from the CI. If the CI finds any problems, these should be fixed as soon as possible.

Finally, to maintain code quality throughout the project, all the code written will be reviewed by other members of the team. This will also ensure knowledge about the entire code base among all member of the group. For the code review the merge requests in GitLab will be used. A merge request needs at least two approvals of team members that did not work on the branch itself before it can be merged.

A.9.3. Timeline

The development process consists of six sprints, each lasting one week. Based on the requirements, below is a rough timeline of the development. This timeline is subject to change, as unforeseen difficulties or successes could be encountered during the project.

- **Sprint 1** Groundwork has been laid, e.g. pipeline and needed frameworks are setup.
- **Sprint 2** Must-have feature set is partially implemented.
- **Sprint 3** Must-have feature set is (almost) completely implemented.
- **Sprint 4** Must-have feature set complete, should-have features' implementation started
- **Sprint 5** Should-have feature set nearing completion, could-have consideration starts.
- **Sprint 6** Rounding off of the codebase and feature set freeze at start of week.

A.10. Conclusion

This research report highlights the main topics associated with our bachelor end project at Raccoon Serious Games. It started with an introduction to the problem the client currently has and the idea of how it can be solved by our project in section A.1. After that in section A.2 the general structure behind most escape rooms was discussed as well as the type of events RSG creates.

Section A.3 explains the currently available system employed by the developers at Raccoon Serious Games, their functionalities, what makes them different, and what underlying frameworks were used for their development. In section A.4 we summarize the main guidelines for creating a domain specific language and provide some visual modeling languages as examples. The following section, section A.5, compares four different visualization frameworks suitable for this project.

It states their strong and weak points and determines the best one that will be used throughout the project. The next section section A.6 provides more information on testing techniques used during software development. It explains how user interfaces can be tested during development and what and how usability tests are conducted. Those findings will be put to use during the process for the creation and execution of user tests.

Ultimately, the research led to the requirements in section A.7 and section A.8 along with their respective prioritization, allowing for a more successful completion of the project. Finally in section A.9 we discuss the software development techniques and methodologies we will be using to ensure the quality of the final product.

These research discoveries will be used as reference during the project.

A.11. Functionalities of MORSE

To determine the functionalities the editor should contain, we have analysed the current Schedule items, Triggers, Conditions, and Actions of MORSE. The list contains most of these, while grouping some of them together to focus on the actual functionality, rather than the exact implementation.

Global/Monitor screen

- Game host is able to press buttons in the monitor screen to perform actions in the escape room.
- Based on the timer, some actions should be performed automatically

Screens:

- Puzzle Screen:
  - Function: Displaying one or more puzzles
  - If there is more than one puzzle in this screen, it automatically creates a menu containing the puzzles. The players can select one of them, which opens that puzzle. Then they can go back to the general overview.
  - If there is more than one puzzle in this screen, on the left it shows a list of all the puzzles through which the players can switch puzzle. This can be hidden by enabling auto-advance, which moves to the another puzzle when the current one is solved automatically.
  - Shows a scoreboard on the right, that can be hidden
  - Shows the timer on the right, can also be hidden, if so the scoreboard is also hidden
  - (Title is displayed)
  - (Skippable)
  - Milestones
- Text Screen:
  - Function: Display text in the middle of screen
  - Text should be configured
  - (Title is not displayed)
  - (Skippable)
- Image Screen:
  - Function: Shows image on the screen
  - URL: to point to the image
  - Styling: background color, fit image the screen: enlarge the image maintaining the aspect ratio
  - (Title is not displayed)
  - (Skippable)
- Register Screen:
  - Function: Updates the information in the profile of this team
  - Team name is a special field
  - Email field is a special field that checks the formatting of the input
  - Text of the button can be changed
  - General text is displayed between the title and the fields
  - Additional fields can be added, with
    - ◇ Label: defines the name of text that is input so it can be used within the escape room (e.g. use the team name)
    - ◇ Question: The piece of text left of the input
    - ◇ Type:
      - · String
      - · Number
      - · Boolean
      - · Phone number
      - · Dropdown
  - Shows a scoreboard on the right, that can be hidden
  - Shows the timer on the right, can also be hidden, if so the scoreboard is also hidden
  - (Title is displayed)
  - (Skippable)
- Puzzle
  - Function: displays a puzzle
  - Is embedded within a Puzzle Screen
  - Can customize puzzle title, subtitle and description
  - Can customize points for (in)correct answer
  - Can have a description
  - Can customize appearance of the question
  - Can specify answers and their grading
  - Can add pages before the actual question is shown

Rulesets:

Triggers:

- A team logs in
- Host presses a button
- Media [x] has finished
- Score for a team changes
- Timer reaches [x]
- Timer is paused/resumed
- Status of a puzzle changes
- Webhooks coming in

Conditions:

- Team answers/solves puzzle
- Team is in stage
- Team score
- Timer (running/time remaining)
- The game is suspended
- Stage is skippable

Actions:

- Play sound on projector
- Show image to a team
- Change config option
- Send hint
- Pause / run / set / change the timer
- Show text / image / video (uninterrupted or not) / timer on projector
- Update score of team
- Move team to different stage
- Make a stage skippable
- Make puzzle available
- Set start time of the event
- (Un)suspend the game
- Call webhook

A.12. Figures

A.12.1. MORSE

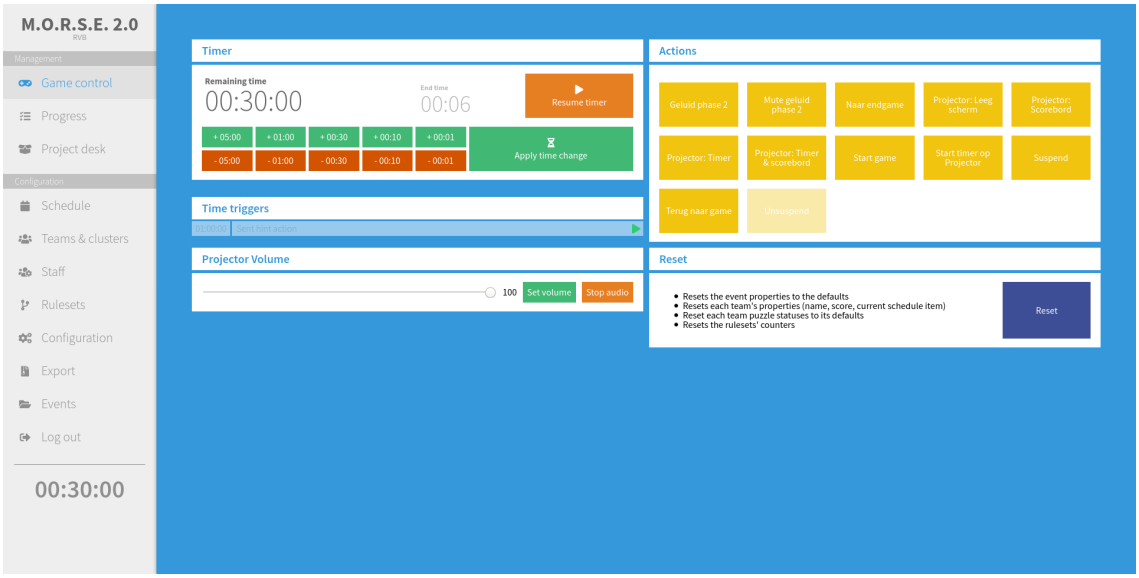


Figure A.1: Game Control Screen

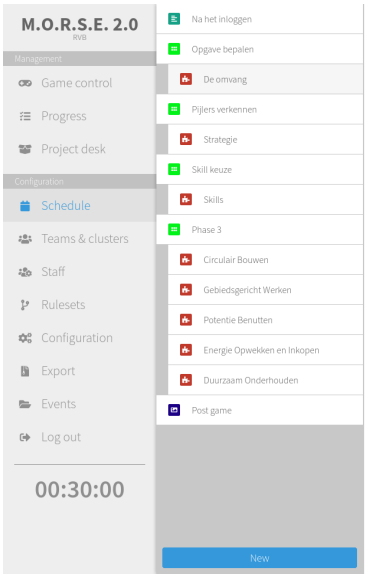


Figure A.2: Configure Puzzle UI

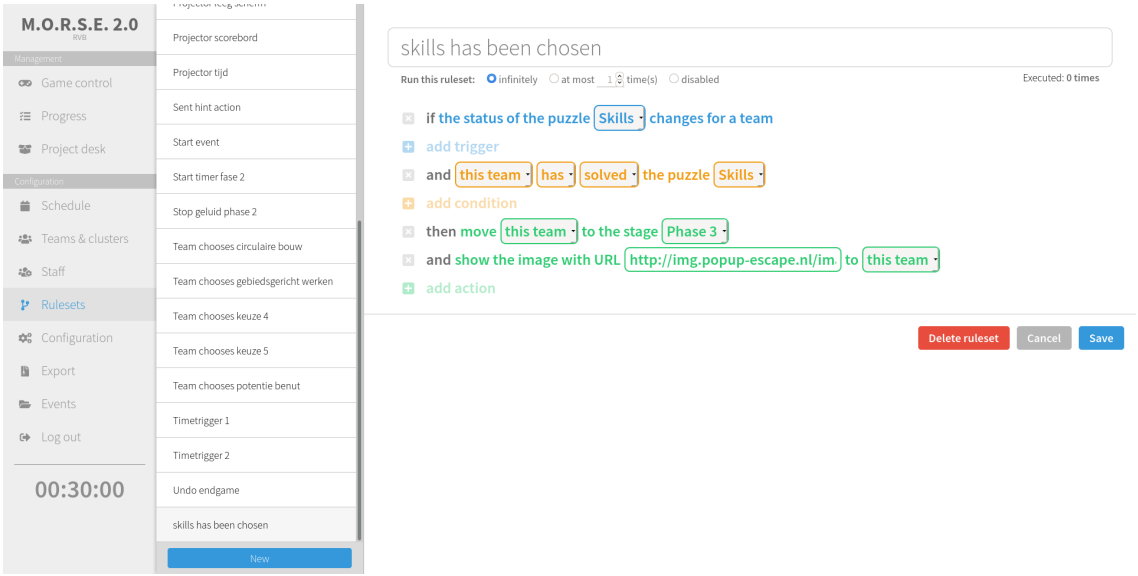


Figure A.3: Configure Ruleset UI

A.12.2. Visual Modelling Language

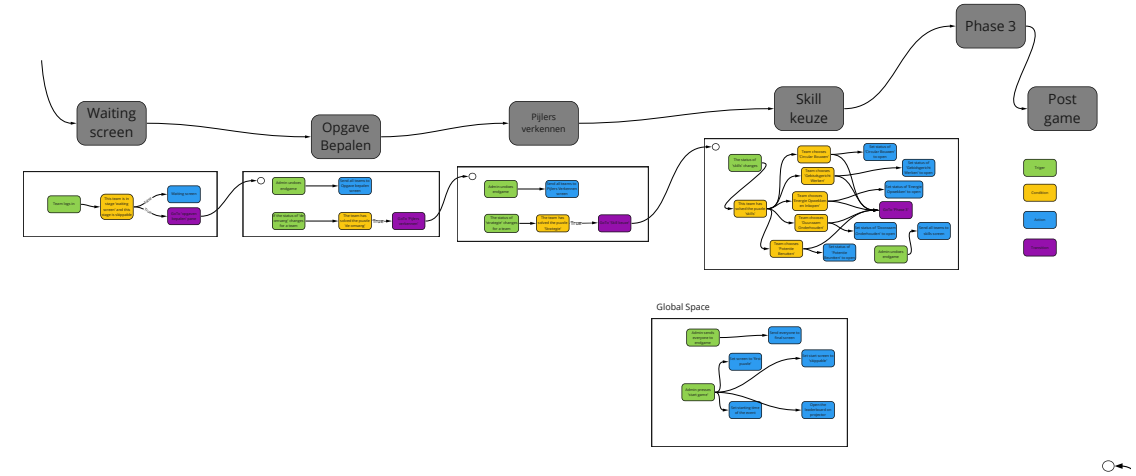


Figure A.4: Mock-up with Rulesets

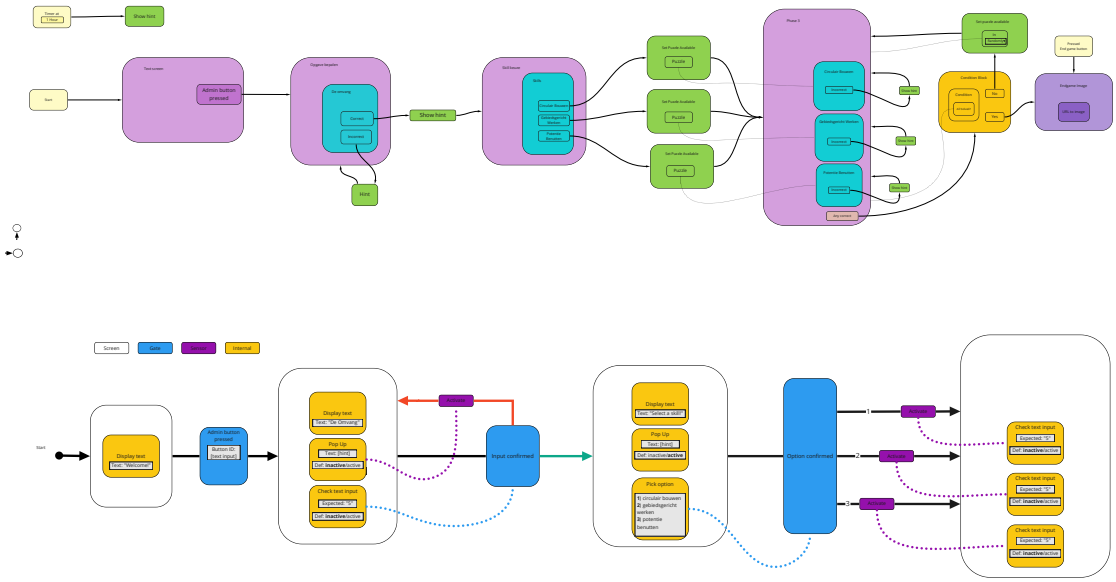


Figure A.5: Mock-ups without Rulesets

B

Project Description

B.1. Het project

Raccoon Serious Games heeft verschillende software systemen wat ze gebruiken voor het hosten van grootschalige en fysieke escape rooms.

Op dit moment gaat het configureren van deze systemen nog moeizaam: Vaak is er IT Kennis, of IT gedachtengoed nodig om een escape room in elkaar te draaien.

De uitdaging van dit project is het maken van een visuele escape-room editor voor M.O.R.S.E 2.0 .Het doel is dat alle medewerkers van raccoon een escape room kunnen maken in het systeem.

Daarnaast kan het zijn dat er extra uitdagingen te voorschijn komen onderweg waardoor het nodig is om aanpassingen te maken in het MORSE systeem. Hiervoor wordt een kritische en adviserende blik verwacht.

B.1.1. Extra uitdaging

Daarnaast hebben we S.C.I.L.E.R, een software systeem wat fysieke escape rooms kan verzorgen. De software wat tijdens deze BEP gemaakt wordt, kan ook ingezet worden om het maken van een escape room te vereenvoudigen in SCILER.

B.2. About Popup-escape

Popup-escape is a young company/startup started by a Delft Computer Science student two years ago. It started out as a hobby and it is now grown to a full-time job with multiple part-time employees. (Also some Computer Science students). We have designed and made about 30+ escape rooms over the past two years and at least 3000 people have played the different escape rooms.

C

Info Sheet

This page intentionally left blank.

FEATHER: Visual Editor for Escape Rooms

Eben Rogers Siert Sebus Wouter Polet Yana Angelova Yoshi van den Akker

Name of the client organization: Raccoon Serious Games
Date of the final presentation: 1 July 2020

Description:

Challenge: Raccoon Serious Games develops and hosts educational activities such as escape room events and serious games. For their digitally hosted events the Massive Online Reactive Serious Escape 2.0 (MORSE) system is used for the creation and configuration of the underlying rules of the event. This system, although a great improvement upon the previous implementation, has proven unintuitive to most of the employees at Raccoon Serious Games.

Research: The research for the project focused on visual programming language design and real-life examples of these editors; to learn what constituted as 'intuitive' and the way in which we could find it through our own user testing.

Process: The team used the agile methodology for development, doing weekly sprints and check-ins with the client. The first week or two focused on getting familiar with the pre-exising frameworks and languages used within MORSE. After that development started in earnest. The entire project was coordinated remotely using online communication platforms, such as Discord, due to the restrictions imposed by COVID-19.

Product: To solve the unintuitive nature of MORSE, our team designed and developed Feather: A graph-based visual editing tool that is integrated into MORSE. It can generate rule and ruleset logic needed for the client's escape events. It uses visual components and presents the user with a graph of the whole game during the design process. The editor can be used together with all other, earlier existing, features for creating rulesets of the MORSE system. This tool has most of the functionality the current system has, with the possibility of easily extending it with new components.

Outlook: The Feather editor can be used by Raccoon Serious Games moving forward. However, it will likely need some more improvements. The product can be used in combination with the product created by a BEP group, that worked on the system at the same time, as throughout the project both products have been merged together on a regular basis.

Members of the project team:

Yoshi van den Akker
Interests: Photo-/videography, education, lifeguarding on the beach, solving puzzles (possibly with scripts).
Contributions: Front-end development, ruleset compiler

Yana Angelova
Interests: Reading, golf, breaking software, cyber security, collecting knives.
Contributions: Front-end development, user-tests preparation and execution

Wouter Polet
Interests: Fantasy books, (board) games, solving random problems with programming.
Contributions: Database integration, ruleset compiler, client-server interaction.

Siert Sebus

Interests: Purely functional programming, programming language design, game design, creative writing, drawing, cycling.
Contributions: Front-end development, ruleset compiler, visual language design.

Eben Rogers

Interests: System/server maintenance, reading sci-fi and man/wiki pages, Project Euler, cycling/running/rowing, travelling.
Contributions: Front-end design and integrity checking system, abstract class design and implementation.

All team members worked on the research report and the final report.

Contact:

Name	Role	Email
Jan-Willem Manenschijn	Client at Raccoon Serious Games	jan-willem@raccoon.games
Thomas Overklift	TU Coach, Teacher @ CSE TU Delft	t.a.r.overkliftvaupelklein@tudelft.nl
Eben Rogers	Team member	eben.rogers1@gmail.com
Siert Sebus	Team member	siertsebus@gmail.com
Wouter Polet	Team member	wouterpolet@gmail.com
Yana Angelova	Team member	yy.angelova@gmail.com
Yoshi van den Akker	Team member	yoshivda@gmail.com

The final report for this project can be found at: http://repository.tudelft.nl

D

Project Plan

D.1. Project Problem Description

D.1.1. Background

Raccoon Serious Games is a game company offering, among others, various large-scale escape rooms, in both physical and digital format. To facilitate the creation and monitoring of these escape rooms, multiple software systems are used. Unfortunately most of these systems, including M.O.R.S.E. 2.0 (MORSE), require background knowledge in software development to create the escape rooms. This restricts the designers greatly and puts extra responsibilities on the technical team members at the company.

D.1.2. Goal

In this project the goal is to improve the process of creating the escape rooms for the existing systems. This will be done by creating a new visual editor integrated in the existing MORSE system. To create that we will look into developing a visual domain-specific modeling language suitable for the system's usage.

This editor should be intuitive in design and easy to use for all of the client's employees. The designers should be able to drag and drop elements for the escape room in the editor. It should be possible to connect these elements (like a puzzle or a certain screen) to each other in the editor itself and incorporate the rules in these diagrams. These rules can become complex and the escape room can have different paths that players can follow. It is important that the configuration of the escape rooms stays clear in these cases.

Furthermore the editor will be made primarily for MORSE. The editor will be fully integrated in this system, preferably as another entry in the menu bar of the MORSE monitor panel.

Ideally, the new system will replace the current method of creating escape rooms in MORSE, i.e the new editor will not lack any functionality of the current, less intuitive, editor.

Extra Goal(s)

Additionally, the visual editor might be used with one of the other systems used by Raccoon Serious Games named S.C.I.L.E.R (SCILER). SCILER is currently used for physical escape room configuration. The configuration of the escape rooms in this system currently happens using a JSON file, which is unwieldy to read or correctly edit. If time allows, the editor should support the generation and importing of the JSON format that SCILER uses. Priority will be given to integrating the visual editor into MORSE such that configuration can be simplified. However, if there is enough time, the additional features such as the proper JSON generation system for SCILER will also be implemented.

D.1.3. Initial Project Forum Description

The initial project description on Project Forum by Raccoon Serious Games (as translated into English) is as follows:

> Raccoon Serious Games has a collection of software systems that are used to host large-scale physical escape rooms. Currently the configuration of these systems can be unwieldy: Often IT knowledge or IT-like thinking is needed to be able to create an escape room in this system. We

would like to challenge the client to make a visual escape-room editor for MORSE. The goal being that all employees of Raccoon Serious Games can make an escape room in this new system. Furthermore, it is possible that extra challenges are encountered while making this new system within MORSE, requiring changes in the current system. For this we expect a critical yet constructive view of our codebase.

D.2. Collaboration with the other group

We are not the only Bachelor End Project group doing a project for our client, Raccoon Serious Games. The other group will, like us, also work on modifying and improving the MORSE system. However, they will be solving a different problem that the client has with the system.

D.2.1. The project description of the other group

An issue that the client has with MORSE is that the system does not offer enough flexibility when it comes to working across different domains and changing the front-end design for particular puzzles or games. If this limitation wasn't there and the designers at Raccoon Serious Games were able to tweak these aspects, they would be better able to make the games more immersive. The other group is, thus, tasked to restructure the internals of MORSE to make it more modular and flexible such that working cross-domain and with different front-ends is possible and does not take much effort.

D.2.2. Where our projects overlap and collaboration will be required

Both groups will be extending the already existing MORSE system. In doing so, both groups have to take into account each others' changes when developing their own products. To facilitate this, both groups will be having weekly meetings to keep each other up-to-date.

D.3. Planning

The timespan of the project is defined to be one quarter or ten weeks. During this time the team will produce three deliverables: a research report, the software product, and a final report. In order to create each of these parts, we propose the following schedule:

- In weeks 1 and 2 the main focus will be on the research of the problem and the possible solutions. This includes writing the research report based on these findings.
  - **Week 1** Research visual design languages, current systems (similarities and differences), escape rooms, testing, and frameworks.
  - **Week 2** Interview employees of Raccoon Games to analyze the current workflow for designing the escape rooms.
- From week 3 to 8 the main focus will be on software development of the new editor of the system. This will include a sprint length of one week from Tuesday to Tuesday for a total of 6 sprints. We will start by defining the visual modelling language and designing the application, followed by the implementation of a mock-up prototype. User testing shall also be completed on the employees of Raccoon Serious Games to receive feedback on our design.
  - **Week 3** Define a visual modelling language and design a mock-up.
  - **Week 4** Implement part of the editor: Puzzle screen and puzzles.
  - **Week 5** Add image and text screens and centralize the styling of elements.
  - **Week 6** If needed, add a way to create groups in the editor. Conduct user testing.
  - **Week 7** Consider implementing extra features if time allows (i.e. SCILER integration).
  - **Week 8** Buffer week for unforeseen difficulties or extra features.

  Detailed planning will be created on per-week basis during the sprints.
- In weeks 9 and 10 the focus will shift to the final stage, writing the report and working on the presentation.

In week 5 the team will have a midterm meeting with both the client and the coach to discuss the progress and evaluate the work done so far. Around this time the team will send its software for the first of two checks by the Software Improvement Group (SIG). The exact moment of this meeting will be determined later on in the project. The second check will be done in week 8, after the development phase is over and the results will be reflected upon in the final report.

D.4. Contract

- Scrum:
  - Sprint planning: in GitLab using the issue boards
  - Daily meetings: at start of each block (i.e. morning and afternoon), what have you done? Any problems? What are you going to do next?
  - We start with the scrum sprints when the research phase is done, i.e. week 3.
  - Task rotation: When planning we will make sure that each sprint everyone works on a different part of the system than previous sprints. This will give everyone a better overview of the project and thus makes it easier to review each other's code.
- Git Repo:
  - Use template for issues
  - Make relevant labels for issues/merge request
  - Discussion in the merge request are resolved by the person that opens them, except for the ones where suggestions are applied
  - Continuous Integration: static analysis, tests, test cov
    - Deploy pipelines on MR and master
    - ESLint
    - Codecov
  - Use milestone for each sprint with deadline for this sprint
  - Make small commits, make sure to push often and check the CI request.
  - Make one MR and branch per issue
  - Commit messages: If this commit succeeds it will , in general follow this
  - Write tests for your own code, in the same MR
  - Code formatting: follow ESLint
  - Every merge requests needs at least two approvals, and should be merged by the second approver
- While working: stay on discord (muted)
- Contact with client/supervisor:
  - Weekly meetings with client on Tuesday (which is the end of the sprint as well)
  - Bi-weekly meetings with supervisor, preferably at the end of the week
- Schedule:
  - Sprint: 1-week, from Tuesday to Tuesday
  - 42 hours a week, 8.5 hours a day: 8:30-13:00 and 13:30-17:30
  - Yoshi, Yana, Wouter are TA'ing Monday and Wednesday afternoon.
  - Missed hours should be made up within the same week, or two weeks when there is not a lot of extra work. This can be taken into account when dividing the tasks.
  - Extenuating circumstances should be discussed with the group.
  - National holidays: we take them off

E

Reevaluated Requirements List

E.1. Current Progress

- Done
- Almost done/Partially present
- Not done/started
- ~~No longer in the planning~~

Must Haves:

- Be able to represent rules such as triggers, conditions, and actions as components/nodes in the visual editor.
- Be able to display a 2D space in which components/nodes can be:
  - dragged
  - dropped
  - added
  - removed
  - connected
- Be able to display an overview of the detailed logic and the system flow corresponding to it.
- Be able to extract the rules from the visual representation and store them in the existing database model.
- Allow the user to continue with the same visual representation as they stopped with earlier.
- Be able to display error messages.
- Have a visual representation of the puzzle screen and its corresponding questions/puzzles inside.
- Allow the user to link screens (just puzzle screens for now) of the escape room together, determining the flow of the escape room.
- Allow the user to configure a starting point.
- Allow the user to configure triggers/conditions based on the timer (both value and whether it is running).
- Allow the user to add triggers based on the status changing of a puzzle.
- Allow the user to add and configure actions that change the status of a puzzle.

- Allow the user to add conditions based on the answers of a team to a puzzle.
- Allow the user to add actions regarding the state of a puzzle, which are executed given that a certain trigger is present and the underlying condition(s) are met.
- Allow the user to configure a button system for admins.
- Allow the user to link buttons to actions.
- Allow the user to configure the skippable field of screens and use conditions based on that property.
- Allow the user to configure projector actions (image, timer, audio, video).
- Allow the user to configure image screens.
- Allow the user to configure text screens.
- Allow the user to configure hints.

Should haves:

- ~~Allow undo and redo actions.~~
- ~~Allow the user to duplicate components and their configurations.~~
- ~~Allow the selection of multiple components at once and performing an action (such as delete/duplicate) on all of them.~~
- ~~Allow using "abstract" components: high-level components that do not have their details defined yet but can still be used in the project as if defined.~~
- Have pretty visuals (e.g. appealing colour palette and fonts) that are coherent with the existing system.
- ~~Update the underlying rule sets in real-time.~~
- ~~Have the ability to read existing/newly added rule sets and drop them in the visual at a random spot.~~
- ~~Have a dynamic transition between detailed and general overview (e.g. with zooming by scrolling the mouse wheel).~~
- ~~Have a system that checks for syntax errors and displays them to the user.~~
- ~~Be able to include a commenting system (e.g. sticky notes or block highlighting).~~
- Be able to display an overview of the whole game, where detailed logic (e.g. number of questions and question flow) is abstracted away.
- ~~Be able to give components names that are used in the visualisation.~~
- Allow the user to configure actions that set the value of the timer, or pause/resume it.
- Allow the user to configure register screens.
- Allow the user to link an action to the event of media such as a video finishing.
- Allow the user to configure triggers/conditions/actions based on team scores.
- Allow an action to suspend the game (which can be bound to a button) and an action lifting this suspension.
- Allow the user to configure conditions that check whether the game is suspended.
- Allow the user to configure actions that show an image to a team.
- Allow the user to use an action to set the start time of the event.

Could haves:
We think there is no time left for this

E.2. Towards the end

- High priority
- Medium priority
- Low priority

Bugs:

- Connecting two screens together is almost impossible, as there is an issue with detecting whether the mouse hovers the connection point or not.
- Resizing the screens does not work on chrome and is not saved in/loaded from the database.
- Removing connections between components work weirdly and is very unintuitive. There should be an easier way to select a connection and remove it. With fixed connections this become at least doable.

Functionality:

- Exporting the visual configuration to the rulesets that can be used by the MORSE system.
- Allow rulesets to only be executed a certain amount of times, in the current system a number can be set for this per ruleset.
- When an event is duplicated the feather configuration should be copied as well.
- Auto-saving the feather configuration while someone is using it. Alternatively at least a confirmation dialog should pop up when the user tries to leave the page with unsaved changes.

Components:

- Existing in MORSE:
  - Allow the user to configure projector actions (image, timer, audio, video).
  - Allow the user to configure an action that shows text on the projector.
  - Allow the user to configure image screens.
  - Allow the user to configure text screens.
  - Allow the user to add and configure actions that change the status of a puzzle.
  - Allow the user to link an action to the event of media such as a video finishing.
  - Allow the user to configure actions that show an image to a team.
  - Allow the user to configure triggers/conditions/actions based on team scores.
  - Allow an action to suspend the game (which can be bound to a button) and an action lifting this suspension.
  - Allow the user to configure register screens.
  - Allow the user to use an action to set the start time of the event.
  - Allow the user to configure actions that set the value of the timer, or pause/resume it.
  - Allow the user to configure hints.
- New ones by other group:
  - Page Visited Trigger.
  - Button Pressed Trigger.
  - Site Redirect Action.
  - Page Hint Action.
  - Domain Change Action.
  - New Domain schedule item, which has pages in it and in there puzzles.

Usability:

- Remove the current dock and place it as a menu at the top of the page.
- Give a warning when the user leaves the page with unsaved changes.
- Complete manual for the system.
- Cheat-sheet with the essential information nicely summarised.
- Coherent, nice looking styling for the components that can be placed in the editor.
- A clearer distinction, placement wise, for the nodes that are general and nodes that need a certain stage/team.
- Tutorial for the system (via a video or a interactive tour in the system).

F

Final Evaluation Requirement List

F.1. Implemented

F.1.1. Must-Have

- Be able to represent rules such as triggers, conditions, and actions as components/nodes in the visual editor.
- Be able to display a 2D space in which components/nodes can be:
  - dragged
  - dropped
  - added
  - removed
  - connected
- Be able to display an overview of the detailed logic and the system flow corresponding to it.
- Be able to extract the rules from the visual representation and store them in the existing database model.
- Allow the user to continue with the same visual representation as they stopped with earlier.
- Be able to display error messages.
- Have a visual representation of the puzzle screen and its corresponding questions/puzzles inside.
- Allow the user to link screens (just puzzle screens for now) of the escape room together, determining the flow of the escape room.
- Allow the user to configure a starting point.
- Allow the user to configure triggers/conditions based on the timer (both value and whether it is running).
- Allow the user to add triggers based on the status changing of a puzzle.
- Allow the user to add and configure actions that change the status of a puzzle.
- Allow the user to add conditions based on the answers of a team to a puzzle.
- Allow the user to add actions regarding the state of a puzzle, which are executed given that a certain trigger is present and the underlying condition(s) are met.
- Allow the user to configure a button system for admins.
- Allow the user to link buttons to actions.
- Allow the user to configure the skippable field of screens and use conditions based on that property.

F.1.2. Should-Have

- Have pretty visuals (e.g. appealing color palette and fonts) that are coherent with the existing system.
- Have a system that checks for syntax errors and displays them to the user.
- Be able to display an overview of the whole game, where detailed logic (e.g. number of questions and question flow) is abstracted away.
- Allow the user to configure projector actions (image, timer, audio, video).
- Allow the user to configure image screens.
- Allow the user to configure text screens.
- Allow the user to configure hints.
- Allow the user to configure actions that set the value of the timer, or pause/resume it.
- Allow the user to configure register screens.
- Allow the user to link an action to the event of media such as a video finishing.
- Allow the user to configure triggers/conditions/actions based on team scores.
- Allow an action to suspend the game (which can be bound to a button) and an action lifting this suspension.
- Allow the user to configure conditions that check whether the game is suspended
- Allow the user to configure actions that show an image to a team
- Allow the user to use an action to set the start time of the event.

F.1.3. Reevaluation Requirements

- Give a warning when the user leaves the page with unsaved changes.
- When an event is duplicated the feather configuration should be copied as well.

F.2. Not Implemented

F.2.1. Should-Have

- Allow undo and redo actions.
- Allow the user to duplicate components and their configurations.
- Allow the selection of multiple components at once and performing an action (such as delete/duplicate) on all of them.
- Allow using “abstract” components: high-level components that do not have their details defined yet but can still be used in the project as if defined.
- Update the underlying rule sets in real-time.
- Have the ability to read existing/newly added rule sets and drop them in the visual at a random spot.
- Have a dynamic transition between detailed and general overview (e.g. with zooming by scrolling the mouse wheel).
- Be able to give components names that are used in the visualization.

F.2.2. Could-Have

- Support keyboard shortcuts for important actions.
- Support auto formatting of the visualization.
- Allow users to save component presets.
- Allow components to be “linked” to each other. Similar to duplication but linked components keep their properties synced when they are modified.
- Allow the user to hide/show certain elements of the rule system.
- Make components still visually reflect their configuration when zoomed out or minimized through the use of colors and icons.
- Load JSON configurations of SCILER.
- Export JSON configurations that SCILER can use.
- Show errors that arise in the SCILER configurations.
- Allow a game admin to revert the ending of the game.
- Allow the user to add a web hook.
- Allow the user to configure the properties special to SCILER that are not in MORSE.
- Allow the user to use a MORSE configuration action.

F.2.3. Reevaluation Requirements

- Nodes for new items added by the other BEP group:
  - Page Visited Trigger
  - Button Pressed Trigger
  - Site Redirect Action
  - Page Hint Action
  - Domain Change Action
  - New Domain schedule item, which has pages in it and in there puzzles
- Complete manual for the system.
- Cheatsheet with the essential information nicely summarized.
- Tutorial for the system (via a video or a interactive tour in the system)

References

Bakker, M., Braam, A., Morssink, W., Nederveen, T., & Sterk, A. (2019, Jul). Supporting large-scale escape rooms with a modular system. *Supporting large-scale escape rooms with a modular system*. Retrieved from http://resolver.tudelft.nl/uuid:e64f75d5-9bad-4b4a-9ecf-34426853bcf3

Barnum, C. (2011). *Usability testing essentials : ready, set– test*. Amsterdam Boston: Morgan Kaufmann Publishers.

Bowen, J., & Reeves, S. (2013, May). UI-design driven model-based testing. *Innovations in Systems and Software Engineering*, *9*(3), 201–215. Retrieved from https://doi.org/10.1007/s11334-013-0199-6 doi: 10.1007/s11334-013-0199-6

Brinkman, W.-P. (2019, May). *Delft university of technology*.

Cunha, M., Paiva, A. C. R., Ferreira, H. S., & Abreu, R. (2010, October). PETTool: A pattern-based GUI testing tool. In *2010 2nd international conference on software technology and engineering*. IEEE. Retrieved from https://doi.org/10.1109/icste.2010.5608882 doi: 10.1109/icste.2010.5608882

Flowgorithm. (2020). Retrieved from http://flowgorithm.org/index.htm

Hanou, I., Smitskamp, G., & Schipper, M. (2020, Jan). Designing an escape room sensory system. *Designing an escape room sensory system*. Retrieved from http://resolver.tudelft.nl/uuid:31969411-1053-4bc1-830d-33354febd95a

Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., & Völkel, S. (2014). *Design guidelines for domain specific languages*. Retrieved from https://arxiv.org/abs/1409.2378

MacLaurin, M. B. (2011). The design of kodu. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages - POPL 11*. ACM Press. Retrieved from https://doi.org/10.1145/1926385.1926413 doi: 10.1145/1926385.1926413

Moreira, R. M. L. M., Paiva, A. C. R., & Memon, A. (2013, November). A pattern-based approach for GUI modeling and testing. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*. IEEE. Retrieved from https://doi.org/10.1109/issre.2013.6698881 doi: 10.1109/issre.2013.6698881

Nicholson, S. (2016). *The state of escape: Escape room design and facilities*. Retrieved from http://scottnicholson.com/pubs/stateofescape.pdf

Pedersen, F. (2019, Mar). *101 best escape room puzzle ideas*. Retrieved from https://nowescape.com/blog/101-best-puzzle-ideas-for-escape-rooms/

Pedigo, V. (2019, Dec). *18 mind-bending escape room puzzles and clues to look out for*. Big Escape Rooms. Retrieved from https://www.bigescaperooms.com/escape-room-puzzles/

Pipes. (2020). Retrieved from https://www.pipes.digital

Puzzle types commonly found in escape rooms. (2019, Jan). Quest Reality Games. Retrieved from https://questrealitygames.com/puzzle-types-commonly-found-in-escape-rooms/

Resnick, M., Myers, B. A., Nakakoji, K., Shneiderman, B., Pausch, R. F., Selker, T., & Eisenberg, M. (2005). Design principles for tools to support creative thinking.. Retrieved from https://www.cs.umd.edu/hcil/CST/Papers/designprinciples.pdf

Scratch. (2020). Retrieved from https://scratch.mit.edu/about

Stasiak, A. (2016, June). Escape rooms: A new offer in the recreation sector in poland. *Turyzm/Tourism*, *26*(1), 31–47. Retrieved from https://doi.org/10.1515/tour-2016-0003 doi: 10.1515/tour-2016-0003

Usability testing questions: Asking the right questions. (n.d.). Retrieved from https://www.hotjar.com/usability-testing/questions/

Wiemker, M., Elumir, E., & Clare, A. (2016, Nov). *Escape room games: "can you transform an unpleasant situation into a pleasant one?"*. Retrieved from https://thecodex.ca/wp-content/uploads/2016/08/00511Wiemker-et-al-Paper-Escape-Room-Games.pdf