The Design of a Modular Game Platform for the QuantumRules! Lab

M. Kolenbrander

Supervisors:
Dr. A.W. Laarman
Dr. H.P. Buisman

BACHELOR THESIS **Informatica**

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl 02/02/2021

Abstract

This thesis presents the new software design for the QuantumRules! project. This new design aims to accomplish two goals: The creation of a stable game platform in the form of a webapplication and the implementation of an extension system to help this aforementioned game platform to support the implementation of new ideas and developments through future development cycles. The development was based on the waterfall model. Several elicitation techniques were applied to quantify the two main goals into smaller requirements through the use of user stories. The requirements were applied through the development of a game structure and modular system implemented in the Django Framework. The end result of this software development is that set goals and elicited requirements were all met and satisfied according to performed evaluations and quality assurance tests.

Acknowledgements

I first and foremost want to greatly thank my two supervisors, Alfons Laarman and Henk Buisman, for giving me this special opportunity to work on this very interesting project.

Henk Buisman has not only been a great supervisor and product owner to work with, but he also acted as a true mentor. Programming was always something that was up to me, but planning and concept wise, and figuring out the heading of this project was something Henk has been a tremendous help in. He always liked to take me in on the newest developments in his field and we always had very pleasant discussions about the very potent future of the QuantumRules! program. The atmosphere was always very pleasant whilst working with Henk, and to that I am incredibly grateful. He made this project very special to work on.

Alfons Laarman significantly helped improve me professionally. His sharp and honest critique not only made me a better academic writer and developer, but they also helped to shape me in general during the final stages of my bachelor degree. His help and feedback made, not only this thesis, but this entire project, a project to be incredibly proud of. For that I am especially grateful.

This project had the majority of its run taking place during one of the strangest times of the twenty-first century so far: Year of the Covid-19 pandemic. This fact alone should merit a great degree of gratitude to everyone who has so patiently worked with me during these strange times. The constant working from home, mailing and video calling have not been easy, but in the end, with a lot of effort, a very satisfying result has been reached.

Lastly, but most certainly not least, I would like to thank my friends, family and especially my ever loving and supporting girlfriend. They have all supported me extensively, especially my girlfriend who was always by my side to help me throughout this project.

To everyone involved, thank you very much for this great experience!

Contents

1 Introduction **1**
1.1 Context: QuantumRules! . . . 1
1.2 The problem statement: Software limitations . . . 2
1.3 Goals of the new software . . . 2
1.4 Approach . . . 2
1.5 Overview . . . 3

2 Requirements **4**
2.1 Requirement elicitation . . . 4
2.2 Defintions . . . 4
2.3 Functional requirements . . . 5
2.4 Non-Functional Requirements . . . 8

3 Design **9**
3.1 Game structure . . . 9
3.2 Module system . . . 11
3.2.1 Modules . . . 11
3.2.2 Module selector . . . 12
3.2.3 Module interface . . . 14
3.3 Game rule components . . . 15

4 Non-Functional design **16**
4.1 Django . . . 16
4.2 Websockets . . . 16
4.3 Front-end . . . 17

5 Proof of Concept **18**
5.1 Extendibility through the module system . . . 18
5.2 Multiple Choice, Multiple Answer: Proof of concept . . . 18
5.2.1 Data-structure design . . . 19
5.2.2 Logic design . . . 20
5.2.3 Front-end design . . . 21
5.2.4 Registering a module . . . 22
5.2.5 Wrapping up . . . 22

6 Evaluation **24**
6.1 Software testing . . . 24
6.1.1 Communication through the use of websockets test . . . 25
6.1.2 Websocket failsafe test . . . 25
6.1.3 Group use testing over multiple networks and multiple devices . . . 26
6.1.4 Module development testing . . . 26
6.1.5 Interface testing . . . 26

6.1.6 Wrapping up 26
6.2 Documentation 27

7 Conclusion **28**
7.1 Current status 28
7.2 Future work 28
Bibliography 30

1 Introduction

This thesis describes the design of a new software environment for the Quantum Rules! educational program. With this introduction we will first lay out foundations such as the *context of the project*, *the problem statement*, *the goals of the new software*, *the development approach* and lastly, we will discuss the *overview* of this thesis.

1.1 Context: QuantumRules!

Before we can discuss the goals and approach of this project, we need to understand the context for which this project was developed.

This project was developed for the QuantumRules! educational program for highschool pupils. The QuantumRules! program was founded by Dr. H.P. Buisman, a Dutch high school physics teacher and an associate for the Leiden Institute of Physics (LION) for high school-university programs.

QuantumRules! was developed to help high school pupils, from preparatory scientific education, with creating a more in depth understanding of the world of quantum physics through a fun experience. QuantumRules! seeks to teach quantum physics through the gamification of quantum experiments.

This gamification of quantum experiments is done in an experimental physics laboratory in the University of Leiden, run by Dr. H.P. Buisman. In this laboratory, Dutch high school pupils are spilt in small teams of 4 to 6 pupils and tasked with conducting numerous experiments, in order to complete certain aspects of a greater physics game they are playing. In total, about 30 pupils, divided into the smaller teams, participate simultaneously in a game.

These physics games take shape as an escape room, where the pupils advance further through the game by (successfully) completing experiments. Before the pupils can advance to the next set of experiments, they need to complete a few preliminary experiments. This concept of having to complete multiple smaller tasks, experiments, in order to advance, is what makes the game concept of QuantumRules! an escape room like environment.

In QuantumRules!, custom software is used to help guide the pupils through the experiments. The software serves as a basis for the escape room *"layout"* by serving a structured way in which the quantum physics experiments follow. The pupils interact with this software by receiving questions from it, in combination with pointers to the next experiment they have to conduct in the real world lab in order to answer these questions, and the pupils interact by answering the questions.

This software is entirely web-based and designed in such a way that every student with a phone, which is connected to the internet, and a correct room code (provided by the QuantumRules!

organizers), can follow along.

1.2 The problem statement: Software limitations

The software for QuantumRules! was initially developed and released in the spring of 2018 by C.J.H. Meijerink, an economic and computer science bachelor student at the time, who took on this project as his bachelor thesis [Mei18]. The software was developed in the very early stages of the QuantumRules! project. Because the software was developed alongside the initial development of the QuantumRules! project, it was difficult to accurately capture all the requirements early on. As the software was released and being used by the QuantumRules! staff and visiting pupils, several unforeseen limitations and new requirements started to surface.

In order solve the limitations and implement these new requirements, teams of, mostly student developers, were employed to develop these new components. However, because the original software was not designed with extendibility in mind, many of these requirements were hard to implement. When these requirements were finally implemented, it was often done in an unstable manner with inadequate documentation on the process. Because of these multiple development cycles, the software became unstable in use, so much so that the end-user experience started to suffer.

Another major shortcoming in the software is the way its experiment types were implemented. The original software only supports two experiment or question types: *multiple choice* and *teaching assistant questions*. The former is a simple multiple choice structure where there are multiple answers to a question of which only one is correct. The latter is a question type where the pupils provide their answer to a teaching assistant in the Quantum Rules laboratory, whom then progresses the question by entering a hardcoded password.

Because of the core logic and database structure of the original software, combined with the effects of the poor development cycles later on, it is not feasible to expand this structure with more experiment or question types. Therefore, a completely new software project will be started.

1.3 Goals of the new software

The main goals, on which the requirements for this project are based, of the software can be distilled down into two parts. The first goal is the **Creation of a game environment** that should have at least the same functionalities as the previous software as it stands, but with a higher degree of stability and reliability. The second goal will be **added ability for modular extendibility** to allow for new question types and game components. In Chapter 2 the full extend of the requirements are laid out and discussed. These requirements will serve to achieve the two main goals of the new software.

1.4 Approach

For each software project, it is generally speaking a good idea to decide on a development method. Because the requirements were set at the beginning of the project, the waterfall development method [Vli08] seemed most suited for this project. The main focus throughout this

project remained creating an online game environment and to construct an extendible framework.

Based on the goals and the requirements, a design will be constructed for the new software. The full detail of the project design and implementation is given in the design sections of this thesis (Chapter 3 and Chapter 4). As an introduction we will briefly discuss the main concepts from the design sections.

To achieve extendibility within the software, a modular system will be designed and developed to allow future developers to implement more and different types of questions and game components. Effort will also be put into separating the components that make up the entire game engine into smaller components, one such example is the separation of the game rules component.

This new software environment will be implemented primarily in Python through the Django framework. For websocket communications, the Django package Django-Channels is going to be employed to enable async communications between the front-end and the back-end. In order to counteract the instability issues of the websocket communication, which was experienced in the previous software, a fall-back communication options will be designed and implemented as a failsafe as well.

1.5 Overview

In this thesis the following sections are discussed: In Chapter 2, the ***requirements*** are discussed in more detail. There, the two main goals stated in this introduction will be broken up into more quantifiable requirements. After discussing the requirements, the ***design*** of the new software is discussed in Chapter 3. There the design of the entire game-structure and the concept of the module system, the structure designed for extendibility, will be introduced and explained in more detail. In Chapter 4, the ***non-functional design*** and the implementation will be discussed. There, the reasons for certain software and package choices, as well as API and framework choices, are further elaborated on. After discussing the implementations, the *maintainability* and *extendibility* are discussed in Chapter 5 ***proof of concept***. Besides the evaluation on the second major goal of the project, the overall *quality* of the software and the development of the software is discussed in Chapter 6 ***evaluation***. Finally in Chapter 7 the project is looked back upon in the ***conclusion***, there the current status and future works are discussed.

2 Requirements

Now that the two main goals have been set in Chapter 1.3, they need to be quantified into smaller requirements. In this chapter we will first focus on how these requirements are elicitated and then we will list the functional and non-functional requirements resulting from the elicitations techniques.

2.1 Requirement elicitation

In order to elicitate the requirements from the project goals, several elicitation techniques were applied. These techniques will be briefly discussed here:

- **The Interview Technique**: In order to figure out what the product owner had in mind when they initially envisioned the software for QuantumRules!, several interviews were held with the product owner. Not only was the desired state of the original software discussed, but also the desired direction for the software in the future.
- **The Brainstorm Technique**: Brainstorm sessions were held with the product owner in order to come up with possible applications for the software. Knowing the future envisioned applications for the application helps with finding out what the software is supposed to be able to do.
- **Documentation Analyses & Review**: To further understand how the previous software was supposed to function, and therefore what the baseline for the new software would be, development documentation was analysed. One major source was the documentation on the original development by C.J.H. Meijerink [Mei18]. Other sources were the development documentations of the student development groups.

The elicitation techniques described above resulted in a set of requirements for the new software, which will be listed under Chapter 2.3 and Chapter 2.4.

2.2 Defintions

In this thesis a few terms are used to express certain constructs. In order to help understand these terms, they will be (pre-)defined here.

- **Game-object** A game-object, within the context of this thesis, is an object of a module of a certain type such as a **question**, a multiple choice questions for example, or a **content object**, such as a complete mini-game or simple informational content such as a text or video object. These game-objects can be part of the digital-games as described in Chapter 1.1. Game-objects are added to a game by means of an ItemLink, which is explained later on in this thesis. Chapter 3.1 and Chapter 3.2 further explain how these objects fit within a game.

- **Module-object** A module-object is the **instance** of a **module**, which can used within a game. Note that this is not an **ItemLink**! An **ItemLink** points to these module-objects to link them to a game, as is described in Chapter 3.2.1 and Chapter 3.1.

2.3 Functional requirements

The requirements for this software are summed up below in the form of *user stories*. In order to indicate which requirement belongs to which goal, several user stories and requirements are postfixed with **GE** and **ME** for the first and second goal respectively. Because these requirements are directly linked with to two goals, these will be seen as "must have" requirements. Requirements not categorised by either of the two goals will be seen as optional "nice to have" requirements.

These user stories are categorized by four user roles:

- **Game-Creators** The Game-Creators are the hosts of the QuantumRules! program. They come up with and set up the games and experiments for the pupils.
- **Game-Masters** The Game-Masters are the teaching assistants who help out the hosts of QuantumRules! during a game. As is described in Chapter 1.1 they are for example responsible for accepting answers during TA questions.
- **Players** The players are the high school pupils who play the game in the QuantumRules! lab.
- **Future-Developers** The Future-Developers are the developers that are going develop new *module types* and *game rule components*. Because this project is largely about making the software more extendible, the developers will be seen as users in order more easily quantify their needs. Future-Developers will however not be seen as *end-users* who end up using the software during release.

In addition to these four user roles, there are also **general requirements** which apply either to all users or to the entire application.

Game-Creators

F-GC1 Has an account to enter the game-creator panel. ***GE***

F-GC2 Can publish a game. Publishing a game resets certain metrics, such as obtained scores, and it calculates total score per category used for score-bar displaying as described by **F-GC11** and **F-P7**. ***GE***

F-GC3 Can change settings to open and close a game for **players**. Opening a game makes it accessible for players by means of the methods in **F-P2** and **F-P3**. Closing a game makes the game inaccessible. When a game is closed with **players** inside, the **players** receive a notice that the game is closed and that their input will be ignored until it is opened again. ***GE***

F-GC4 Can change settings to shorten or prolong the total game time, during which **players** are able to play the game. ***GE***

F-GC5 Can apply (predefined) game rules and change the parameters of said rules to change game types. ***GE*** & ***ME***

F-GC6 Should be able to add new *module-objects*, such as new *game-objects*, of pre-existing module types. Examples of game-objects such as *questions* of a certain type are *multiple choice* or *open* questions. Examples of game-objects such as *content-objects* of a certain type are mini-games such as *a Quantum Tic-Tac-Toe minigame* or informational blocks such as *text*, *video*, *music* blocks. ***GE*** & ***ME***

F-GC7 Should be able to create new games. ***GE***

F-GC8 Should be able to create new categories and add those to games. ***GE***

F-GC9 Should be able to add *module-objects*, such as a **game-objects**, to categories by means of so-called **ItemLinks**. ***GE***

F-GC10 Should be able to dictate the progression rules for categories. This progression rule dictates how **players** are progressed when finishing a game-object as is described by **F-P9**. ***GE***

F-GC11 Should not have to worry about scores having to add up to x points in order to have the score-bar, as described in **F-P7**, working with a 0 to 100% scale. The sum of the total score per category can be arbitrary. ***GE***

F-GC12 Should be able to see how many points have been gathered per category in *open* and *published* games.

F-GC13 Should be able to use the software on a wide range of screen sizes, such as phones, tablets, laptops and desktop PC's. ***GE***

Game-Masters

F-GM1 Has an account to enter the game-master panel. ***GE***

F-GM2 Can change settings to open and close a game for **players**. See **F-GC3** for the full explanation on opening and closing a game. ***GE***

F-GM3 Can change settings to shorten or prolong the total game time, during which **players** are able to play the game. ***GE***

F-GM4 Should be able to accept answers from **players** during a **teaching-assistant question** on their **own** account through their **own** device. ***GE***

F-GM5 Should be able to assign a score between *zero* and *max-score* when accepting an answer to a **teaching-assistant question**. ***GE***

F-GM6 Can use a ranged score selector, also known as a slider control, when assigning a score.

F-GM7 Should be able to broadcast messages to all **players** in game. This broadcast message should show up on all the **players'** screens.

F-GM8 Should be able to see the status of each ItemLink in an open game. This status entails: Has this item been visited by **players**, has this item been completed by **players**, how many points have been gathered by the **players**, how many answers or solutions have been provided by the **players** and how many of these answers and solutions are wrong.

F-GM9 Should be able to see how many points have been gathered per category in in *open* and *published* games. ***GE***

F-GM10 Should be able to use the software on a wide range of screen sizes, such as phones, tablets, laptops and desktop PC's. ***GE***

Players

F-P1 Does not need an account to join a game. ***GE***

F-P2 Should be able to join a game using a game-id or *TagName* given by either the **game-masters** or **game-creators** in the QuantumRules! lab. ***GE***

F-P3 Should be able to join a game by selecting a game from the game list on the home page of the web-application. ***GE***

F-P5 Should be able to see which parts of the game, or more specifically which game-objects, have already been visited. This information must update in real-time. ***GE***

F-P6 Should be able to see which parts of the game, or more specifically which game-objects, have already been completed. This information must update in real-time. ***GE***

F-P7 Can see how many points, percentage wise, have been obtained so far on a score-bar. This information must update in real-time. ***GE***

F-P8 Receives a hint, only when the game is set-up this way by the **game-creator**, when they have provided an incorrect answer or solution. ***GE***

F-P9 When finishing a game-object, the player should either be redirected back to the overview page of the game they were in, or be redirected to the next game-object. This redirection behaviour is set by the **Game-Creator**. ***GE***

F-P10 Should be able to use the software on a wide range of screen sizes, such as phones, tablets, laptops and desktop PC's. ***GE***

Future-Developers

F-FD1 Should be able to define and implement new module types such as question types and game-object types. (Please see **F-GC6** for a further definition and example statements on module types). ***ME***

F-FD2 Should not have to work outside of the sandboxed module environment, so an environment separate from the other code inside the software in which a module can be freely developed, in order to implement a module. This implies that they should have enough tools, such as an interface, available from the framework in order to integrate a module into a game. ***ME***

F-FD3 Should be able to define and implement new game rule components which can later be used by the **game-creators** to create different game types. ***ME***

F-FD4 Should get comprehensive debug-feedback whilst testing their modules in the software.

General

F-G1 Throughout the application Markdown formatting must be applicable on all *end-user editable textfields*. ***GE***

F-G2 LaTeXmath formatting should be applicable in addition to Markdown formatting as described by **F-G1**. ***GE***

F-G3 The game webpages should be interactive and dynamic. ***GE***

2.4 Non-Functional Requirements

NF-1 In the case that the interactive and dynamic components, as described by **F-G3**, should fail, a fallback mode, should activate in order to keep the software operational and games playable.

NF-2 The software should be documented for the *game-creators* and *game-masters*, with explanations on how to set up and run a game.

NF-3 The code of the software should be documented, explaining which sections do what.

NF-4 A *how to get started* documentation for development should be provided. The steps to implement new modules and game-rule components, how their interfaces work, should also be included into this development documentation.

3 Design

In the introduction the two main goals for this project were set: **Creation of a game environment** and the **added ability for modular extendibility**. These two goals have been quantified into smaller requirements in the requirements section. The following section proposes a design to meet the requirements described in Chapter 2.

First, the **structure** and **design** of a **game** within the concept of the game in QuantumRules! will be given in Chapter 3.1. This structure and design definition mainly aims to achieve the first goal of *creating a game environment*. This game environment is the environment in which the high school pupils, the **Players**, play the games as introduced in Chapter 1.1. The **Game-Creators**, the hosts of the QuantumRules! lab, create these games in order to support the experiments the **players** perform in the QuantumRules! lab, as is described by requirements **F-GC7**, **F-GC8** and **F-GC9** in Chapter 2.

Once the game structure has been defined, a design aiming to achieve the second goal of this project *adding modular extendibility*, will be given in Chapter 3.2. This design is for a modular system, the **module system**, will be introduced in order to expand the question types and game components. By implementing new question types and game components in this module system, more types of experiments and games can be hosted in the QuantumRules! lab, which should help expand the capabilities of this software in the QuantumRules! lab. In order to incorporate these modules into the game, a design will be proposed which will relate these **modules** to the **game structure**.

The last aspect of the design which we will discuss is the concept of **game rule components**. These will add an extra degree of configurability to the games, in order to support different game types which would widen the range of gamification application of this software. In turn, this helps to further broaden its possibilities and applicability in the QuantumRules! lab.

All the proposed structures and components within this design section are going to be implemented in the **Django** and **Django-Channels** framework. The Django framework works on the basis of a MVC design pattern, but due to the nature of the *module system*, the MVC design pattern will be altered slightly, by adding the module system into the MVC design. The reasons behind choosing Django will be discussed in Chapter 4.

3.1 Game structure

In this software, a game, as described in Chapter 1.1, is going to consist of multiple smaller child objects. In total, three larger database structures will be implemented. The schematic in Figure 3.1 described the abstract design of this proposed game structure.

Game The Game-Class is the main starting data-structure of a game. This main structure contains a few fields to set up game-rules, which helps to accomplish requirements **F-GC5** and

F-FD3 as described in Chapter 2, and it contains zero or more child structures called *categories*. The aforementioned game-rules can be applied by the **Game-Creator** but are predefined by the **Developers**, as described in Chapter 3.3.

Category The Category-Class represents a structure which will act as a collection of game-objects of a certain category. For example: five questions about the sun. Note that game-objects, or rather their pointers through *ItemLinks*, can only be connected to a game through a Category.

ItemLink The ItemLink-Class will act as a pointer to a module and a module-object, an instance of that module, within that module. Modules will be explained later on in Chapter 3.2, but for now it can be assumed that modules are sub-programs hosting game-objects of different types, as described by **F-GC6** and **F-FD1** in Chapter 2. The ItemLinks are partially aimed to support the implementation of the two aforementioned requirements but is moreso implemented to support requirement **F-GC9**. An ItemLink is *not* module-object such as a game-object itself, it only acts as a pointer to one. A *category* holds zero or more ItemLinks. Note that modules can only be linked to a game through itemlinks.

For sequential progression in a game, as described by **F-P9** in Chapter 2, the **FirstItem** and **NextItem** structures are introduced. These act as the head and the links within a linked list respectively. This linked list structure is *only* relevant when the *chained* option in categories is enabled. If the chained option is disabled, the player gets redirected back to the overview with all the categories, questions and game-components. A linked list structure, instead of for example an order field, will be used, because Django is able to take advantage of linked list structures in data-structures.

The ItemLink has one final purpose besides being a pointer and progression list organiser. The ItemLink has a few settings which can be changed by the **Game-Creator** in order to alter the properties of a game-object. The following fields are used in the ItemLink to apply settings to game-objects.

- **MaxNumAttempts** MaxNumAttempts indicates how often **players** may attempt to give a correct answer or solution. If a wrong answer is given, an attempt has been used. If **players** should have infinite amount of attempts, this setting should be 0.
- **NumAttempts** NumAttempts is a counter to keep track of how many answers or solutions, which were wrong, were given by the **players**. This is a field to support the setting in *MaxNumAttempts*. Note that it is up to the specific module, as defined by the **developers**, to decide which player input counts as an attempt. For example, in the case of *multiple choice*, providing one of the choices in your answer is an attempt. When this value exceeds the value set in *MaxNumAttempts*, the game-object gets flagged as finished in *Completed* and 0 points are assigned in *ObtainedScore*.
- **ContinueByGM** ContinueByGM is a setting to turn any game-object, most often informational game-objects such as objects from a text-block module, into a Game-Master, or Teaching Assistant, question. These questions can only be progressed by the approval of **Game-Masters**, as described in **F-GM4** in Chapter 2. This setting can be turned on by the **Game-Creator**

- **MaxScore** MaxScore is a parameter, set by the **Game-Creator**, to assign a certain amount of point to a game-object, which **players** can obtain by submitting a correct answer or solution to a game-object.

- **ObtainedScore** ObtainedScore is a field which, during a game, keeps track of how many points **players** have obtained. The number of obtained points depend on the *MaxScore*, *FaultPenalty* and *NumAttempts*.

- **FaultPenalty** FaultPenalty is a parameter the **Game-Creator** can set to dictate how many points of the *MaxScore* **players** with each wrong answer, as is tracked by *NumAttempts*. The final score when successfully completing a game-object, which is stored in *ObtainedScore*, is ObtainedScore = MaxScore − (FaultPenalty · NumAttempts).

- **Visited** & **Completed** Visited & completed are two supporting flags to indicated whether **players** have previously visited this game-object or whether they have completed this game-object. **Players** and **Game-Masters** can see the status of a game-object through these flags, as described in **F-FP5**, **F-FP6** and **F-GM8**.

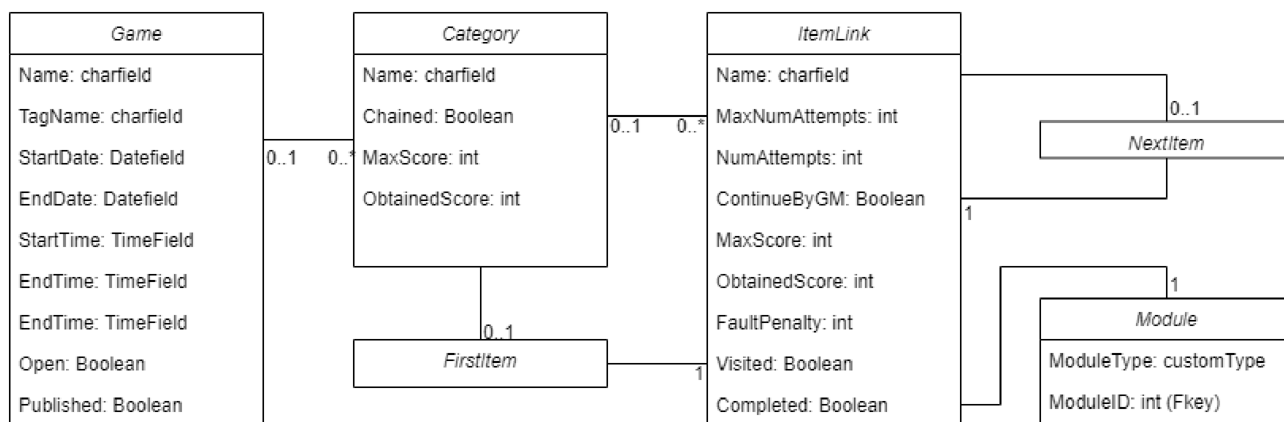

Figure 3.1: *This diagram shows the relations of the different database structures which construct the game structure.*

3.2 Module system

Now that the main game structure of the software has been proposed, we need to discuss how these structures are going to be made extendible. To accomplish this extendibility of game-object types, we introduce the the concept of *modules* and the *module system*.

3.2.1 Modules

A module is a sub-web-application, which runs in a sandboxed environment within this framework. These modules will have their own unique database structure, logic designs and front-end designs/rendering. The concept of a module is introduced to allow **developers** to design new game components and question types, without having to dive into or alter the core database and logic structure of a game, which would limit the freedom of a **developer**, as is required by

F-FD1 and **F-FD2** from Chapter 2. Because the modules are going to be designed with as few restrictions as possible in terms of front-end design, logic and data structure design, **developers** should have a high degree of development freedom. As a result, a module could for example be as simple as a multiple choice question or as complex as a mini-game where quantum computers are simulated. After a new module type has been defined, the **Game-Creators** should be able to create module-objects, such as game-objects, from them, as described by **F-GC6**. These objects, so in the case of for example a multiple choice module: questions, are the objects the ItemLink described in Chapter 3.1, link to.

3.2.2 Module selector

Now that the concept of modules has been introduced, we need to discuss how they are going to be incorporated into games. In Chapter 3.1 we already saw that *ItemLinks* are going to be utilized as pointers to a module. An ItemLink knows the type of a module and the primary key id of an object within a module.

The type of a module will be used to select a module through the **module selector**. When a module is implemented, for example to add a new question type or create a new mini game, which can be added to games, **developers** need to register this module in the module selector. When registering the components a module, a new **type** for the new module should be defined, so that the selector knows which module and module structures belong to which type.

Once a module is selected by the module selector, the primary key, provided by an ItemLink object, of an object within the selected module is passed on to the *selected module*. With the primary key, it can render and process the correct game-component. In the simple example of a *multiple choice* question module, the objects within this module are the different *multiple choice questions*.

This process of selecting modules based on the **type** and **ID** has been visualised in Figure 3.2.

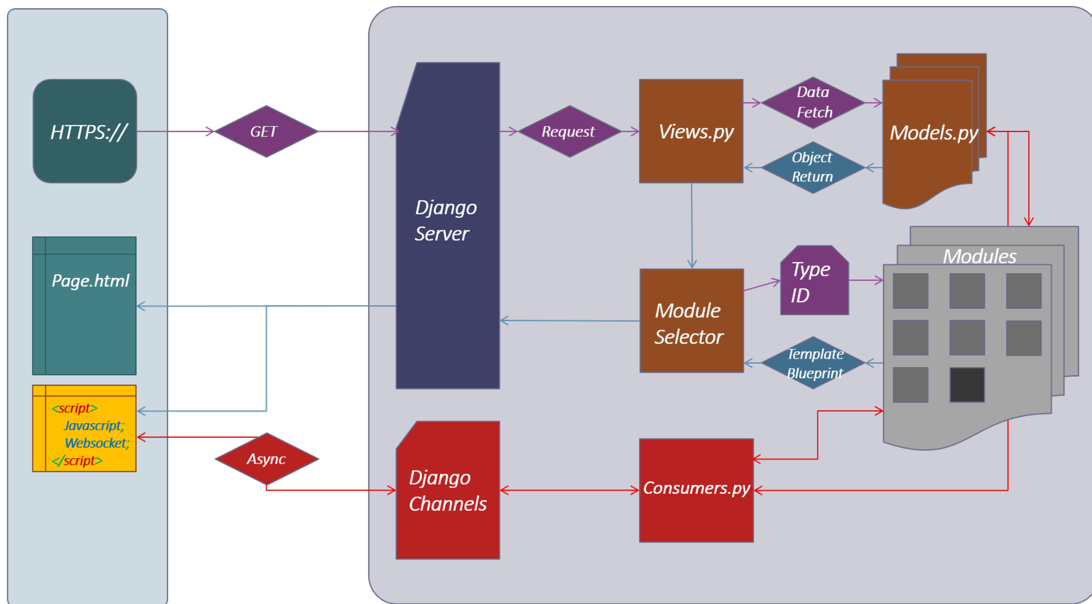

Figure 3.2: *This gives an abstract representation of the process flow inside the server whenever a module is requested, loaded or interacted with. The blue super-block on the left represents the user side, or front-end, and the purple super-block on the right represents the server side, or back-end. The other elements will be described in Figure 3.3 and Figure 3.4*

In Figure 3.2 the selection scheme can be broken down into two parts. First, when **players** want to see or interact with a game-object, the **ItemLink** is fetched from the core database of the program, represented by ***Models.py***, by the view component of Django, represented by ***Views.py***, as can be seen in Figure 3.3.

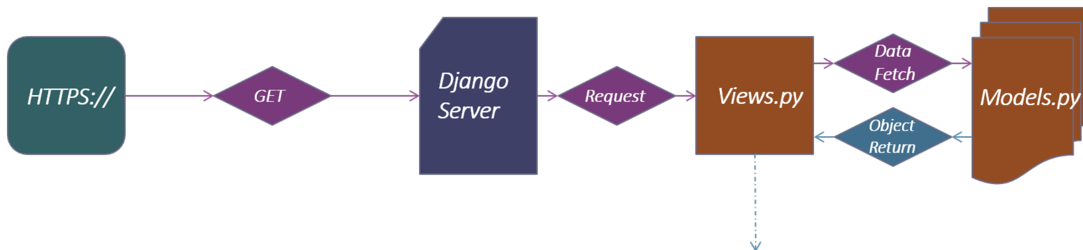

Figure 3.3: *The user-page, represented by the green **HTTPS** block, requests a module from the server through the means of a HTTP **GET** method. This **request** will be passed on to the **views.py** section where the type of the correct module, together with the primary key of the object within a module, housed inside an **ItemLink**, will be fetched from the database, indicated by **Data Fetch**, **Models.py** and **Object Return**. The View will then pass the module-type and object ID to the Module Selector in Figure 3.4.*

Once an **ItemLink** has been fetched from the game-structure, we know the *type* of a module and the *primary key*, or the *ID*, of the correct object within a module. These will be passed on to the module selector which will then fetch the correct module and object. The module will be invoked to either construct its web-application, as a HTML/JS page, or it will be invoked to process user input. The output of a module, a constructed page or response data, will then be

passed on back to the user through the Django server interface, as can be seen in Figure 3.4.

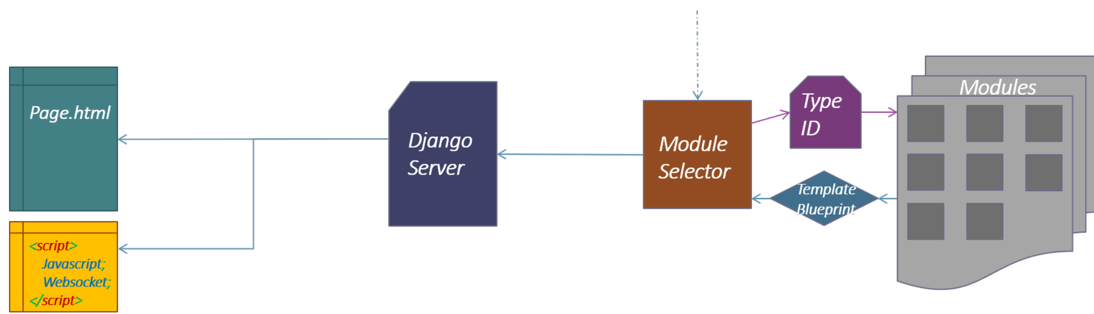

Figure 3.4: *The module will be selected with the* ***module selector*** *using its* ***type*** *and* ***ID****. These modules are housed in a sandboxed environment represented by* ***Modules****. Once a module is selected, it is going to construct an* ***HTML/JS*** *page, which is based on the Django* ***HTML/JS****-Template page for that particular module. This rendered page will be passed on back to the user through the* ***Django server****.*

Once a module is fully served to the user, the module will attempt to set up a direct communication between the user and itself, through the use of **websockets** and the **Djang-Channels** component of the back-end.

3.2.3 Module interface

In order to support the future **developers** in implementing new modules for the software, this software will come with an interface, which will act as the communication layer between modules and the rest of the software, and as a **developers** toolset. This interface contains a set of functions and event generators, which have two main purposes.

The first task of the components inside this interface will be to provide the **developers** with a toolset which should help them to implement new modules with less effort. Example of components inside this toolset would be:

- Base HTML/JS-Template pages, containing all the base components for a web-page, including the required javascript for Websockets, on which the **developers** can build their custom web-pages for the module.
- Base, so called, “renderers”, which populate the aforementioned HTML/JS-Templates with a set of standard data, such as data from the game structure as described in Chapter 3.1, and custom **developer** provided datasets.
- Base database class structures, containing a set of standard fields, required by the software to correcly interact with modules, from which **developers** can inherit when implementing their own data structures for modules.
- Database fetchers which return the correct object from the database of a module, based on the information it gets from an **ItemLink**.
- Communication methods which abstracts the communication of data to the front-end, through for example the means of websockets as described in Chapter 4.2.

The second task of this interface will be to provide a set of event generators to the **developers**, which act on the core code, the game structure as described in Chapter 3.1, and of the rest of the software in a safe and abstracted manner. These event generators could for example be used to *flag* a game-object, or rather an **ItemLink**, as complete when the provided user input is satisfactory.

3.3 Game rule components

Besides the modules, a second concept will be implemented to make the software more extendible. This is the componentization of the **game rules**. Game rules dictate how a game is played and progressed. It dictate how **players** should be be treated and it also dictates when they can progress to the next parts of a game, as indicated by the *chained* setting in categories and the [optional] chaining of ItemLinks.

In order to componentize the rules, a game rule engine will be constructed. Whenever a game, a category or a game-object (through an ItemLink) is requested, the game rule engine is called to test a few conditions, in order to decide whether a game or an object within a game is still valid. One example of such tests could be: *"Is game accessed within its valid time and date frame? (In other words, have the **players** run out of time or not?)"*. These predefine rules can be applied by the **Game-Creator**, who can, depending on the rule, also change settings on that rule, as is described by **F-GC5** in Chapter 2. The **Developers** implement (new) game rules so that the **Game-Creator** can use them as described above, as per requirement **F-FD3**.

The conditions a game or object is tested against are methods which can contain its own logic. That way new rules can be added by the **developers**, is by adding and registering a method to the game rule engine.

4 Non-Functional design

In the previous section we discussed how the game-structure and the modularity components are going to be implemented. In the discussion of the design, it was decided that Django and Django-Channels are going to be used as base frameworks to develop this software in. In this section, we will briefly discuss the reasons behind choosing Django and Django-Channels as our base frameworks. Besides Django, there were a few other frameworks and software packages used in order to satisfy some requirements such as **F-GC13**, **F-GM7**, **F-GM10**, **F-P10**, **F-G1**, **F-G2**, **F-G3** and **NF-1** from Chapter 2. Here we will discuss how these packages are used to satisfy these aforementioned requirements.

4.1 Django

As stated in Chapter 3, this software for QuantumRules! will be developed in the Django framework. There are a few reasons for choosing Django. First and foremost, Django is seen as a *batteries included* framework [Why][Kal18]. This means that Django contains everything you need to create a fully function web-application. Another reason for choosing Django is that it has a strong focus on security and data stability, and by using its provided tools a safe and secure web-application can be made [Dja][Dja21][Kal18]. The last, and perhaps the biggest reason for choosing Django, is that the Django framework is based on Python. The reliance on Python allows **developers** to use the broad (scientific) package base of Python, such as for example quantum computing packages [quT][Qis]. When looking at the environment this software is going to be used in, a scientific one as was indicated by the context setting in Chapter 1.1, using a framework with support for scientific packages makes a strong case in favour of Django.

4.2 Websockets

One of the requirements, **F-G3** from Chapter 2, is the implementation of dynamic web-pages. In this project, websockets will be chosen as the asynchronous, full-duplex communication standard in order achieve these dynamic web-pages. In order to implement websockets in the Django back-end, the Django-Channels package, together with the REDIS-API, is going to be used to add ASGI (Asynchronous Server Gateway Interface) support to Django [Cha]. The use of websockets will allow for continues player to server and player to player connections and communication. This also helps with the implementation of requirements **F-GM7** as this feature relies on direct group websocket communication.

Websocket failsafe option As stated by requirement **NF-1** in Chapter 2, there should be a failsafe option to allow the game to still be playable, even when the websocket connection fails. In order to achieve this, an abstracted input interface layer will be added to the framework, which modules can use to communicate with the user on the front-end. This abstract input interface layer is able to convert both websocket inputs and HTTP post inputs and the output responses to a standardised data dictionary and standardised event-generators respectively. Using this

input interface layer allows for the implementation of a fallback mode on HTTP in the case that the websocket connection should fail, which helps with satisfying the failsafe requirement. The implementation of this abstract interface layer also takes away the more complicated converting of datastreams to and from websocket and HTTP formats. This abstract input interface layer will be part of the module interface described in Chapter 3.2.3.

4.3 Front-end

For the front-end, one of the requirements is that it is responsive and usable on a wide range of screen sizes as described by **F-GC13**, **F-GM10** and **F-P10** in Chapter 2. In order to implement this requirement, the Bootstrap-4 Css/Javascript front-end framework will be employed [Boo]. Bootstrap has a *responsive first* philosophy, which aims to abstract the multi screen size support. Using Bootstrap will allow for rapid development of a responsive web-application user interface.

To implement the requirement for Markdown formatting and LaTeXsupport, as required by **F-G1** and **F-G2**, three javascript packages will be used. MarkDown-It will add Markdown formatting to simple textfields. This will also add the support for in text image and figure rendering. KaTeX will be used to parse LaTeXmath expression within text. Lastly MarkDown-It-Texmath will be used to combine the MarkDown-It and KaTeX parsing engines into one parsing engine.

5 Proof of Concept

For this project, the ease of extendibility in terms of module types is one of the main focusses of this software. To support this goal, the module system was introduced with the intend to allow for the expansion of the software's capabilities without altering the core components of the code. In this Proof of Concept evaluation, we will look at whether the module system makes this software extendible, and whether implementing a new module for the **developers** could be considered easy.

5.1 Extendibility through the module system

To support the evaluation, the following section will be dedicated to designing a new question type, or in more formal terms "a module", and implementing it into the framework. During the implementation of this new question type, an in depth analysis will be provided on how the entire implementation is conducted. In this analysis, a clear description will be provided on what needs to be implemented, and where in terms of the front-end, back-end, data-structure design and logic design.

5.2 Multiple Choice, Multiple Answer: Proof of concept

The *multiple choice multiple answer* question type is a question type where there are multiple correct answers and in order to pass you need to provide all the correct answers. The question in Figure 5.1 will provide an example of how a question in this new type should function.

Cars
There are a lot of different car brands throughout the world originating from many different countries.
Which of the following brands *originally originate* from *Germany*:

A Bently

B BMW

C Trabant

D Citroën

E Opel

F Spijker

Figure 5.1: *This is an example of a Multiple Choice Multiple Answer question. Questions such as these are made by the* ***Game-Creators****. The title of the question is given, followed by a description and then the possible answers. There are multiple brands given in this question which originate from different countries. In order to pass this question, the* ***player*** *has to provide all the car brands which originally originate from Germany. The only correct answer in this case is providing the following three and only these three brands in the answer:* ***BMW****,* ***Trabant****,* ***Opel****.*

This is where *multiple choice multiple answer* comes from: you have multiple choices *and* you **need** to provide multiple answers.

5.2.1 Data-structure design

In this design only two database data-classes will be required: the **question** and the possible **answers**. It should be noted that the **question** class inherits from a base class provided by the module interface as is described in Chapter 3.2.3. The diagram in Figure 5.2 represents the database design of this module.

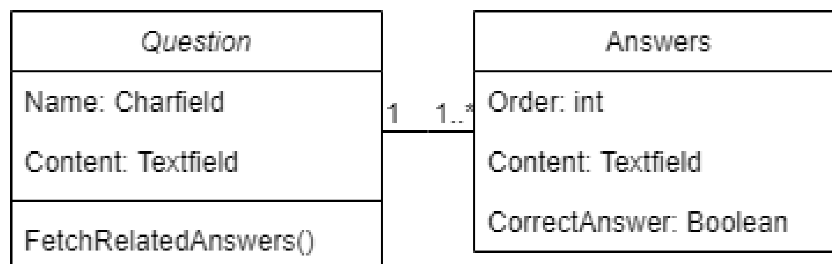

Figure 5.2: Database design for Multiple Choice Multiple Answer

The method `FetchRelatedAnswers()` in the questions class is a method which fetches all the related answers. The order field in the answers table can be used to order the answers. Finally,

CorrectAnswer in answers indicates that this answer must be included in the users answer.

This database design is completely separate from the core database design as seen in Chapter 3.1. When implementing a new module, the **developer** should not have to alter existing tables, which means that this new structure can be imported into an existing database without table, field or relation conflicts. In order to add a question of this type to a game, the **ItemLink**, described in Chapter 3.1, is used to link objects in this database example, and also objects in other modules, to the game structure. In this case, the ID to which the **ItemLink** refers would be the ID of the **Question** class in the database example from Figure 5.2, and the type would be *Multiple Choice Multiple Answer.*

5.2.2 Logic design

In the multiple choice multiple answer module, the logic component will check whether the player's input is correct. This input, consists of a set of id's of the selected answers. These id's are checked against a set of id's of all the correct answers. If the two sets match, the answer is satisfactory. This logic design is described in the code example inside Listing 5.1 below. This code is written in Python inside the Django framework. **Note** that the keyword `Framework` relates to the interface of this project, not to the Django framework.

Listing 5.1: *This is an example of the handle_input function for the Multiple Choice Multiple Answer module.*

```
def handle_input (input, itemID):
  if "mcAnswer" in input:
    # Fetch module object through module/framework interface by feeding
    # it the ID of the ItemLink (itemID). This returns a tuple of three
    # elements: [0] success boolean, [1] module object (the Question in
    # this case) and [2] the ItemLink object related to this particular
    # question.
    if not (fetch_res := Framework.fetch_module_object(itemID))[0]:
      return False

    # Fetch all required answers and convert user input from
    # input["mcAnswer"] into answers objects.
    all_required_answers = fetch_res[1].FetchRelatedAnswers()
                                       .filter(CorrectAnswer=True)
    all_given_answers = Question.objects.filter(ID__in=input['mcAnswer'])

    # All required answers must be given, therefor both querysets need
    # to be equal. Note that the order of elements does not matter in
    # Python sets. (so [3,1,2] == [1,2,3] -> (true)).
    if set(all_given_answers) == set(all_required_answers):
      # If the sets match (answer is correct), tell the framework that the item is
      # satisfied. This will mark it in the game as completed and it will fire
      # an event to all connected clients.
      Framework.handle_item_state.item_satisfied(itemID)
      return true

    # If the lists do not match (answer is incorrect),
    # tell the framework that the item is not satisfied yet.
    Framework.handle_item_state.item_not_satisfied(itemID);
  return false;
```

Just like with the data-structure, this code is sandboxed within the framework into its own compartment. The module manager knows, by using the type of a question/module, which input handler to select. The code of a module has full access to its own database structures, but to interact with the core components of the software, components from the module interface should be used, as described in Chapter 3.2.3. Sending data back to the user (not used in this example) should also be done by using provided event generators from the module interface.

5.2.3 Front-end design

The front-end section of a module is going to consists of two parts within this framework: a renderer and a template.

The rendering of a module will be relatively straightforward in this framework, thanks to the module interface. An example of a renderer is given by the code example in figure Listing 5.2 below. As with the previous listing, this code is written in Python inside the Django framework and the keyword `Framework` relates to the interface of this project.

Listing 5.2: *pseudocode for render_module*

```
def render_module (request, itemID):
  # The template name is the relative path to and name of the
  # template file.
  template_name = "modules/MCMA/index.html";

  # The basic module renderer is a renderer provided by the framework
  # which will fetch all the required objects, such as the module
  # item object, together with some mandatory data. It also contains
  # error handling and (error) feedback to the user.
  return Framework.basic_module_render(request, itemID, template_name,
                                      "Multiple Choice Multiple Answer",
                                      [optional_extra_dictionary]);
```

The use of a provided standard renderer allows for quick deployment, whilst it takes care of abstracted mandatory data inclusion and error handling. The rendering of extra data can be achieved by including an optional, custom data dictionary containing objects and data.

Rendering a module will also require a template in which the data can be populated. **Developers** will be provided with a base templates from the module interface, as is described in Chapter 3.2.3, which they can build upon.

An example of the template for the multiple choice multiple answer module can be seen in the code example in figure Listing 5.3 below. This is an example of an HTML/JS page combined with the Django templating language (which uses the `{{ }}` and `{% %}` operators).

Listing 5.3: *pseudocode for the module template page*

```
<!-- This inherits the template design from the parent template.
The contents below are only a small portion of the total webpage.
By using the blocks (for example block form_content) the contents
of these blocks can be altered at locations preset by the
```

```
parent templates. -->
{% extends "game/base/base_running_item.html" %}

{% block form_content %}
  <!-- Loop through all possible items and display them as
  input options -->
  {% for choice in item.module_item_content.related_choices_content %}
    <div class="form-check">
      <input class="form-check-input" type="checkbox" name="mcAnswer"
              id="Radios" value="{{ choice.ID }}"/>
      <label class="form-check-label" for="Radios"
              id="choice_option{{ choice.ID }}">
              {{ choice.content }}
      </label>
    </div>
  {% endfor %}
{% endblock %}

{% block baseRunningItem_contentRender %}
  <!-- Inherit and extend using block.super from parent block contained
  in the upwards chain in the base templates. -->
  {{ block.super }}
  <script>
    <!-- Loop through all the options to apply Mardown and LaTeX
    formatting to them. -->
    {% for choice in item.module_item_content.related_choices_content %}
      runMarkdownLatexAtDest($("#choice_option{{ choice.ID }}"),
        $("#choice_option{{ choice.ID }}").html());
    {% endfor %}
  </script>
{% endblock baseRunningItem_contentRender %}
```

For simple modules, only a small amount of blocks, such as the ones above, should have to be altered. Thanks to the way how Django handles template rendering, on which this is based, any block up in the parent template chain can be altered, which will give the future **developers** more freedom in their front-end design. At the same time, the use of the standard blocks should allow for a consistent look and feel to the design, and it should allow for a rapid and secure development process.

5.2.4 Registering a module

As stated in Chapter 3.2.2 new modules are going to have to be registered when they are created. Registering a module is necessary to tell the framework where it is able to find the methods, renderers, templates and data structures of a module.

Using this registration scheme in a single registration file is intended to make integrating new modules easy and safe. Core code and structures do not have to be changed, which should help in keeping code safer and more consistent.

5.2.5 Wrapping up

As we have seen above, the implementation of the *multiple choice multiple answer* question type, or module, requires no existing structures to be modified, other than the registration

file. What is notable is that with only a limited amount of code, a completely new question type can be added to the software. This question can, after registering it, immediately be used by the **Game-Creator** in order to create more interesting games. Through the use of the module interface, as is described in Chapter 3.2.3, **developers** did not need to dive into, or even understand the underlying structure of the software, such as the game structure, described in Chapter 3.1. This "layer of abstraction" should prevent **developers** from using "unsafe code" which could alter the core database incorrectly. This should keep the code safer and more maintainable.

Wrapping up, the proof of concept should bear a question: should the implementation of a new module be considered easy? This question is difficult to answer objectively, because the answer relies on personal factors of the **Developers**, such as for example how familiar they are with python, django, the module interface and the software as a whole. However, there are plenty of tools available in this software which take a significant portion of the work in creating a web-application, which is essentially what a module is, away from the **developers**. And, with the right knowledge from the **developers** on how these tools from the module system and the module interface work, implementing a module does neither take a lot of code nor complicated code to complete. Three examples in this proof of concept, which aim to show the simplicity of implementing a module, that support this are: the simplicity of the datastructure design seen in Chapter 5.2.1, the ease of rendering the web-page of a module seen in Chapter 5.2.3, and, as seen in the handling of player input in Chapter 5.2.2, the ease of fetching data and communicating data with the rest of the software through the interface.

6 Evaluation

Software which is intended to be deployed in a production environment should meet a certain amount of quality standards. Quality assurance within software comes in different forms. The following questions are quality assurance questions applied to this project, in order to evaluate the quality of this project.

- How is the ***software tested*** and what is the coverage of the tests? So: which testing paradigms were used; black box or white box, what sections of the software/code were tested, and which and how many boundary conditions were tested?
- What is the level of ***documentation*** provided with the software? Is there documentation on the user side, is there documentation on the development side and lastly is there documentation within the code?

The questions above are discussed in full detail below. When discussing these questions, we will look at why these are important to discuss in context of this project, and how we attempt to answer these questions.

6.1 Software testing

Software testing is an important part of developing software. It can be used as proof that software is secure or reliable. Software testing is also a difficult and time consuming aspect of software development. In software testing, there are different testing types to determine the quality of software. These types of testing can be broken up into two major ideas: blackbox testing and whitebox testing [Vli08].

Due to the time constraints within this project, whitebox testing and automated testing have not yet been applied to this software. Instead the software was tested using the blackbox method. Within the blackbox method, the system testing methodology was applied to see whether the software behaved according the specification set in Chapter 2. These tests were done manually by hand. There were five main tests, each designed to test a different aspect of the software. The following concepts were tested:

- Does asynchronous and continues communication through the use of websockets work as expected?
- Does the websocket failsafe options through the use of an HTTP backup routine work as intended?
- How does the software handle group use over multiple networks and multiple devices?
- Is the module development environment robust?
- Does the interface meet expectations?

For each of the tests above we will briefly discuss what was tested, how these tests were conducted and what the outcome of these tests were.

6.1.1 Communication through the use of websockets test

Setup For this test, four devices were used, each running an instance of the web application. Two of these devices were inside the same question, the a third device was in another question and a fourth device was on the main page of a game.

Tests

1. The first test was whether the main page updates when **players** enter a question. As three of the four instances moved from the main page to the questions, the fourth instance saw the main page automatically update, stating there are **players** inside two different questions.

2. The second test was to see what happens when the question with the two **players** is answered wrong. When a wrong answer was given, all the **players** in that question got a pop up stating that the provided answer was wrong. The other two instances, one in another question and one in the main page of the game, saw no changes.

3. The third test was to see what happens when the question with the two **players** is answered correctly. When this was the case, a popup was displayed for both instances stating that the question has been answered correctly. The score bar updated automatically for all four instances and the instance on the main page saw the question update to complete.

Result The application handled interactions and state updates as intended.

6.1.2 Websocket failsafe test

Setup In order to simulate a websocket failure, the websocket interface on the back-end was turned off. When this was turned off, the instance used for testing inside the game, got a notification stating that there is a connection issue. It also stated that most functionalities still work but that some features might not work as well as intended. When entering a question, a message is displayed, stating the question will still function, but it will not auto-update.

Tests

1. The following test was conducted with two instances. The two instances joined the same question. First, the question was answered incorrectly by one of the instances. This instance got an automatic page reload where the page was updated with a message stating that the provided answer was wrong. The other instance will not notice a change until they reload.

2. The second test was that the first instance would answer the question correctly. This automatically reloaded the page and notified the user that the answer was correct. They were then presented with the next section of the game and they saw an updated score. The second user did not notice a difference. The second user would then provide a correct answer, and in a second run of this test a wrong answer. In both cases, the page reloads and notifies the second user that the question they were in has already been finished. They were then also presented with the question the first instance is currently in. The score did not update as the question was already finished.

Result The failsafe worked as intended. Extra functionalities were disabled, but the game was still playable. This behaviour explicitly satisfies requirement NF-1 as described in Chapter 2.

6.1.3 Group use testing over multiple networks and multiple devices

Setup To see whether a larger amount of different devices, both in numbers and in variance (laptops, phones), were handled smoothly, a group of nine people tested a mockup game with two puzzles/categories each with multiple questions.

Result In this test, we saw that the web-app handled the test with multiple devices without any issues. People were able to answer different questions at the same time, and people on the same question were taken along to the next question when the question was finished.

One note from the users is that the answer format in one of the question types was difficult to use on a smaller mobile phone. This was noted and promptly adjusted in the interface design.

6.1.4 Module development testing

Setup To test whether there were any problems in the module development, a small development team of 7 people was instructed to construct new modules in this framework. They were all given the same starting point with three pre-existing modules.

Test Because they were tasked with using all the mandatory structures provided by the framework, such as the module interface, to construct new modules, they were able to test each method for its intended functionality.

Result During this testing a bug was found where the input interface (see Chapter 6.1.2), did not properly deal with listed inputs *(multiple inputs provided under the same input name)*. This was promptly fixed and updated which solved the issue.

6.1.5 Interface testing

Setup Finally, the administration interface, the interface where games and module-objects are created, was tested by the product owner. The product owner did not have any knowledge on how the software worked, they only knew how it should work based on experiences with the previous software.

Test They were tasked with creating a new game inside the administration interface. This test is aimed to see whether the interface is intuitive and easy to use by being self-explanatory.

Result During this test, it because clear that the interface needed a fair few improvements to work better for the product owner. The comments from the product owner will be used to improve the interface and also construct a well detailed user manual.

6.1.6 Wrapping up

When conducting tests in order to analyse the robustness of the software, it is important to be critical about the process and the results. As was said in the introduction to the software testing, only system testing from the blackbox methodology was applied on this software, without the use of automated tools. These tests were mainly focussed on verifying whether the software adheres to the requirements set in Chapter 2. This however, does not proof that the software is

stable, safe and/or secure. In order to proof this, more advanced tests need to be conducted, with greater code coverage.

The used technologies for this project, such as Django, come with testing tools and should make quantifying the quality of this software's code easier. But, during the development the decision was made, with the product owner, to **not** devote a lot of time in the formal testing of the software and code. Testing code takes time, and that would have been time that can be spend on implementing more requirements and making the software more feature rich. There was a greater desire for feature rich and complete software, than for the formal proof of the stability and security of the software. This however, does not have to imply that the software is inherently not robust, there is just no formal proof that it is robust. Effort has still been put into building stable and secure software. But, corner cases, on which the software might fail, have not been explored through the use of (automated) testing.

6.2 Documentation

Another part of quality assurance is the use of documentation. In this project documentation is a part of the requirements, as described by **NF-2**, **NF-3** and **NF-4** in Chapter 2. This thesis will serve as one source of documentation. All the documentation will be written at the end of this project, in accordance with the waterfall method. The types of documentations are as follows:

- The first form of documentation comes in the form of code documentation. The goal is to have commentary in most sections of code within the software. Especially the pre-existing modules and the interface between the modules and the framework will receive a high amount of attention as these can serve as the basis for future development cycles.

- A second form of documentation comes in the form of development documentation. The development documentation will cover the following:
  - A description of which parts of the code do what.
  - How a module should be implemented.
  - A how to get started guide.
- The final piece of documentation is user documentation, which will describe how to use the software. It will explain how to create games, add module items/objects and how the different functionalities can be used.

The documentations listed above have either been tested on and adjusted to according to feedback of the target users, as is the case with the development documentation and code documentation for the **Developers**, or it has been made in collaboration with the end-users. In the case of the user documentation this was made with input from and in collaboration with the product-owner, who will be the primary end-user of this software as a **Game-Creator** and **Game-Master**. Feedback from all involved end-users have been applied in order to improve the documentation for that target audience.

7 Conclusion

In this thesis, the development of the software the QuantumRules! program has been described. For this project, two goals were set, as is described in Chapter 1.3. These goals have been quantified into smaller requirements through the use of user stories in Chapter 2. In Chapter 3 and Chapter 4 a design and implementation is proposed to satisfy the requirements and goals. We also looked at aspects for upholding reliability and maintainability, and we evaluated the quality of the software in Chapter 5 and Chapter 6. In this conclusion we will discuss whether the goals and requirements for this project were met. We will also look at possible future improvements.

7.1 Current status

Through the implementation of the designs proposed in Chapter 3 and Chapter 4 all the set requirements have been met and the end result accomplishes the two main goals of the project. During a demo session the satisfaction of the first goal, the creation of a game environment, was presented. During this session, the game worked as intended and the optional extra requirements added an extra polish to the finished product. As discussed in Chapter 5, the extendibility of the software through the use of modules also works as intended. Despite these results, it should be noted that this software was never able to be fully "field tested" in game sessions such as the ones described in Chapter 1.1. The reason this was not possible is due in large part to the **Covid-19** pandemic that took place during the development of this project. Several restrictions related to the pandemic made it impossible to host the games which this software is intended for. Only once the restrictions imposed by the Covid-lockdown are lifted, and the QuantumRules! lab is able to invite pupils again, can this software have its first "field test".

7.2 Future work

Even though all the requirements have been met, there are a few points of improvement which can be looked into during later development. During the final stages of the development of this project, a few possible features came up as new opportunities. However, because these features were proposed at the end of the development and due to time constraints, or because these features would deviate too much from the set requirements/goals these features were not yet implemented. In order to preserve these future opportunities, the newly proposed features are mentioned here.

The front-end is currently built using the Django templating engine, raw javascript and jQuery. Even though this is producing satisfactory results, the use of raw javascript and jQuery, as compared to using a front-end framework such as Vue or React does hamper the ease of reactive front-end development. Implementing a front-end framework like Vue or React could improve the flexibility of this software.

A second point of attention is the user sessioning. Currently, the framework does count nor remember who is connected to a game. For future modules, a desired feature could be to keep track of user sessions so that features such as user grouping in modules or specific user feedback can be implemented. Django and Django-channels offer solutions for this, so a user session system could very well be implemented for more complex modules and game modes.

Lastly, the game, as of right now, does not keep track of individual scores and answers of individuals **players**. Creating leader boards and between-user competitions is not possible right now. When implementing the point above, the session tracking of users, a score and answer tracking system for users can also be implemented.

The proposed future work features coincide with the wishes of the product owner for the product's direction. One of the wishes that came up during the later stages of the development is that **players** would be able to compete against one and another in smaller groups. The designed framework is flexible enough to be expanded with the features mentioned above. It should however be mentioned that, unlike the extendibility discussed in this thesis through the module system, the extensions mentioned above do involve the altering of the software's core structure. These proposed *future work* features should be implemented with more care and attention and should still adhere to the generalist approach of this framework: *All components and concepts within this framework should be easy to use and develop for, and completely optional to use. Any features of the framework should not make the development of new modules and game modes any more difficult.*

Bibliography

[Boo] Boostrap-4, build fast, responsive sites with bootstrap. Available at https://getbootstrap.com/docs/4.6/getting-started/introduction/.

[Cha] Django channels: Asgi communication for django. Available at https://channels.readthedocs.io/en/stable/.

[Dja] Security in django. Available at https://docs.djangoproject.com/en/3.1/topics/security/.

[Dja21] Django Software Foundation. *Django Documentation Release 3.1.6.dev*, 2021.

[Kal18] Kalpit. When to use django (and when not to), aug 2018. Available at https://medium.com/crowdbotics/when-to-use-django-and-when-not-to-9f62f55f693b.

[Mei18] C.J.H. Meijerink. Quantum rules! Master's thesis, Leiden Institute of Advanced Computer Science, jun 2018.

[Qis] Qiskit. Available at https://qiskit.org/overview.

[quT] qutip. Available at http://qutip.org/.

[Vli08] H. C. Vliet. *Software Engineering: Principles and Practice*. Artech House, 3 edition, 2008.

[Why] Full stack python: Django. Available at https://www.fullstackpython.com/django.html.