

Università degli Studi di Genova

**Dipartimento di Informatica, Bioingegneria, Robotica
e Ingegneria dei Sistemi (DIBRIS)**

MSc Computer Science
Data Science and Engineering Curriculum

(Draft Copy)

**DubitaC: a serious game to support
undergraduate programming courses**

by

Lorenzo Tibaldi

February, 2022

Table of Contents

Chapter 1	Introduction	7
1.1	Structure of the thesis	7
1.2	Glossary	8
1.2.1	CSV	8
1.2.2	Embarassingly parallelizable	9
1.2.3	Flow state	9
1.2.4	Game loop	9
1.2.5	Positive feedback loop	11
1.2.6	RPC	11
Chapter 2	State of the art	12
2.1	Gamification	12
2.2	Games about programming	13
2.3	Products taking advantage of <i>Gamification</i>	14
Chapter 3	Design and Development	15
3.1	Comparisons with the State of the Art	15
3.2	High level prototype	16
3.3	Initial requirements and Development choices	17
3.4	The prototype	17
3.5	The online multiplayer development	18

3.6	The release	19
3.7	Gamification elements	19
Chapter 4	Description of DubitaC	22
4.1	The Windows Installer	22
4.2	The main menu	23
4.2.1	Server interface	23
4.2.2	Client interface	24
4.2.3	The Cosmetic panel	26
4.3	The first round	26
4.3.1	Server interface	26
4.3.2	Client interface	27
4.4	The hint panel	28
4.4.1	Communication behind the scenes	29
4.5	The second round	29
4.5.1	Server interface	29
4.5.2	Client interface	30
4.5.3	The doubt panel	30
4.5.4	Communication behind the scene	32
4.6	The slideshow	32
4.6.1	Server interface	32
4.6.2	Client interface	32
4.6.3	Communication behind the scenes	33
4.7	The final leaderboard	33
4.7.1	Server interface	33
4.7.2	Client interface	34
Chapter 5	Implementation	35

5.1	The main gameplay	35
5.1.1	Creating a valid cpp	35
5.1.2	Creating a valid Catch2 program	36
5.2	The account management	43
5.2.1	Obscuring the data	44
5.2.2	Security considerations	44
5.2.3	Using a text file as a database	46
5.2.4	Updating the user points	48
5.3	The localization	50
5.3.1	The TextManager and LocalizableText classes	50
5.3.2	Avoiding a common pitfall	50
5.3.3	Fighting Unity's main script execution	51
5.3.4	Final result	52
5.4	The codeQuestions	52
5.4.1	The mandatory labels	53
5.4.2	The recommended labels	55
5.4.3	The limits label	56
5.5	The cosmetics	57
Chapter 6 Unique challenges		59
6.1	Bad Serialization	59
6.1.1	Netcode Serialization	59
6.1.2	Serializing collections	61
6.1.3	One step further	62
6.2	Shrinking the data	64
6.2.1	The biggest data packet	64
6.2.2	Minimizing the strings' impact	65
6.2.3	Avoiding strings where possible	66

6.2.4	Fitting the data inside a packet	66
6.3	Speeding up Catch2	67
6.3.1	Profiling the cycle	67
6.3.2	Compiling the test environment only once	67
6.3.3	Changing the linker	68
6.4	Speeding up DubitaC	68
6.4.1	Size considerations	69
6.4.2	Time considerations	69
6.4.3	Reducing the number of doubts	70
6.4.4	Reducing the waiting time	70
6.4.5	Performance gain	71
6.5	Speeding up Unity	72
6.5.1	Evaluating the extremes	73
6.5.2	Dependency graph	73
6.5.3	Dividing roughly across scenes	73
6.5.4	Final subdivision	74
6.6	Increasing maintainability	75
Chapter 7	Limitations and future work	82
7.1	Limitations	82
7.2	Possible extensions	83
Chapter 8	Conclusions	85
	Bibliography	86

Chapter 1

Introduction

When discussing how to best teach new students about programming, it is common knowledge that an hands-on approach should be preferred.

While online platforms that teach programming with an hands-on approach already exists, what we were interested in developing was a product that would both be educational and entertaining.

To achieve this we had to research, select and implement various *Gamification* elements.

In this thesis we will describe the design choices taken and challenges that we had to face during the development of DubitaC, an educational game with the goal of aiding professors in teaching the basics of the C++ programming language to students following their beginner programming course.

DubitaC was developed as an Open Source project that could be used in the future as an additional teaching tool for the "Introduction to computer programming" course at the University of Genoa. [Uni21]

1.1 Structure of the thesis

The thesis is divided in 8 chapters, after the Chapter 1 introduction, in Chapter 2 we will cite the state of the art research on gamification, followed by examples of games that can be considered "programming" games, alongside different products that successfully employed *Gamification* elements to create a better, or more successful, final product. In Chapter 3 we will give a general overview of our design and development choices. In Chapter 4 we will give a complete overview of everything an end user will see and will be able to interact with during a game session and during the very first installation. In Chapter

5 we will explain some of the more technical details of the main systems that we have implemented. In Chapter 6 we will share some of the unexpected challenges that we had to face and overcome during the development of DubitaC. In Chapter 7 we will point out the shortcomings of the final product and what can be done to extend the game further. Lastly, Chapter 8 will contain a closing statement about this thesis' journey.

1.2 Glossary

TODO: [find a place for the glossary, in a chapter of its own?, maybe right before the bibliography?]

In this thesis we will sometimes use terms or acronyms which the reader should be familiar with but are not assumed to be known, thus we will refer to this section every time it is appropriate, instead of interrupting the current paragraph with the explanation of a term or concept.

All glossary entries are in alphabetical order, acronyms included.

1.2.1 CSV

Acronym for: Comma Separated Values, referring to a very popular text file format containing lists of entries separated by a comma with extension `.csv`.

Although the specifics are not standardized, a `csv` file mainly refers to human readable tabular data.

In this paper we will refer to `csv` file when it has the following properties:

- The first entry is considered the header and it contains a list of column names, implicitly setting the number of values that a valid entry must have.
- Each entry appears on a new line and contains the same amount of values.
- Each value is separated from the following one by a comma and each value should not contain the comma or newline characters.

An example `csv` file could be:

Name	Age	Height	Married	Number of children
Alice	22	1.75m	false	0
Bob	34	1.53m	true	1
Eve	29	1.62m	true	0

1.2.2 Embarassingly parallelizable

Embarassingly parallelizable refers to any task that is composed of many parts that do not depend on each others' completion.

Because of this property, by utilizing the available resources to executes as many tasks as possible concurrently, it is possible to reduce the execution time by a factor equal to the number of resources, provided that the tasks have minimal serial code that cannot be parallelized.

Embarassingly parellizable tasks often depend on the initial input and must recombine a final output, meaning that the rest of the execution might not be parallelizable at all, while the tasks themselves can take full advantage of the available resources.

A, very simple, example of an **embarassingly parallelizable** task might be calculating the number of sunny days over the last year, since the main operations are just sums, 12 tasks calculating the number of sunny days over each months could be executed concurrently and then a final sum would yield the expected result.

1.2.3 Flow state

Referring to the gaming context, the **flow state** is used to describe a moment in time during a player's interaction with the game where they are completely comfortable with the feedback the game is giving to them.

A player that is not sufficiently stimulated, be it because the game is repetitive or too easy compared to the player's current skill level, will phase out of the **flow state** into a state of boredom. Similarly, a player that is overstimulated, be it because the game is introducing many new concepts in a short span of time or because the difficulty of the game became much higher than the player's current skill level, will phase out of the **flow state** into a state of frustration.

Maintaining the player in the flow state is one of the biggest challengest of game development and, because of the different backgrounds that players can have, it often requires expert fine tuning of the game systems so that as many players as possible can stay in **flow state** for as much time as possible.

1.2.4 Game loop

A **game loop** or gameplay loop, refers to the sequence of actions that a user needs to perform to continue playing the game, for example in a game of chess, the **game loop**

would be composed of:

Analyze the board state → Evaluate the best move → Execute the move

It should be clear why it is called a loop, since the execution of a chess move is repeated between 20 and 40 times on average during a chess match, the **game loop** is repeated as much as necessary.

The only requirement for a **game loop** to be considered as such is a distinct repetition of user interaction. If we consider a game where the player wins simply by pressing a button, for example, there is no *sequence* of action, just the singular one of pressing said button, similarly one cannot consider a **game loop** present in a game that, for example, awards the user some points every 5 minutes and the only objective is reaching an high score, the "sequence" of "actions" here consists of:

Wait 5 minutes → Enjoy the increased personal score

Obviously lacking the minimal interaction needed to be considered a **game loop**.

Notice how the 2 previous examples, when combined, create a valid **game loop**, if the player receives points every 5 minutes but only after pressing a button each time, then the sequence of actions becomes:

Press the button → Wait 5 minutes → Enjoy the increased personal score

The user is now "engaging" with the game, even if in a minimal way, this example serves to illustrate that a **game loop** does not need to be exciting and can be arbitrarily simple.

Lastly, games will often contain multiple **game loops** of different sizes, returning to the previous chess example without analyzing the aspects of the game in depth, it is easy to identify at least 3 different **game loops**:

- Deciding and making a move, already mentioned before.
- A smaller **game loop** contained inside the first one that manifests itself during the second action, Evaluating the best move requires analysis of all the possible moves that are at one's disposal in the moment, but the player will continuously simulate each move and opponent's response, create tactics, identify checkmate patterns, recall previous games that reached this position...
- Lastly, the overarching **game loop** of being a chess player, that contains the previous game loops and its sequence of actions can be roughly identified as:

Study chess principles → Play chess matches → Learn from past mistakes

An overarching **game loop** is used in many games to ensure that the player obtains a mastery of the game that is derived by the continuous engagement with game's systems.

1.2.5 Positive feedback loop

A **Positive feedback loop** refers to the phenomenon where the effect of an action increases the frequency, or magnitude, of the action itself, creating a loop.

In games, **positive feedback loops** are very frequent and closely related to the player motivation to continue playing, a typical example is rewarding the player, not only with the usual rewards but additionally with noticeable increases in strength after a certain amount of enemies have been defeated.

The action of defeating enemies has the effect of making the player stronger, making it capable of defeating stronger enemies which in turn will make the player even stronger...

This system, that has been embedded in videogames for decades in many different ways, is usually composed of experience and levels, where some thresholds on an experience counter will trigger a level up, increasing the player's strength or capabilities.

1.2.6 RPC

Acronym for: Remote Procedure Call, refers to a distributed computing technique where a procedure called on a machine is executed on a different machine.

Rpcs require a client-server architecture that can encapsulate the functions call requests and executions.

Once the architecture is set up, developers can write functions that can be executed both locally and remotely with minimal changes, even though they still have to be aware of the consequences of calling a **rpc**, since communicating over the network drastically reduces the speed of the execution.

Chapter 2

State of the art

In the first section of this chapter we will describe *Gamification* and some of the interesting research surrounding it, then in the following sections we will explore some, non exhaustive, examples of products that can be considered:

- **Games about programming**, although we will see that this does not necessarily mean that they teach about a programming language that could be used in the real world.
- **Products taking advantage of *Gamification***, be it in the education domain or otherwise.

2.1 Gamification

Gamification is the strategy of inserting game-like elements in a product and, by doing so, creating a better or more engaging version of said product.

Even though there is no scientific consensus in regards to the possibility of employing gamification in any context with great success [DD17], there have been many examples specifically where *Gamification* increased user performance, user retention and user enjoyment.[Boh15][HYHS13]

In the domain of education, many studies have been conducted in the last decade leading to a consensus regarding the classification of 2 different types of *Gamification* elements:

- Self elements.
- Social elements.

Self elements consist of anything that will motivate the user with a sense of progression, like points and badges. Social elements consist of anything that will motivate the user with a sense of competition, like leaderboards, or ego boost, like allowing the users to show their earned badges to other players.

All these elements are aimed to increase the psychological motivation of the users, not their skills specifically, however the two are closely connected by a Positive feedback loop, as *Gamification* elements push the user to interact with the systems more, their skills increase and as their skills increase, their rewards increase making the user more motivated to continue.[HYHS13]

Additionally, in [SRM⁺20], researchers were able to correlate, on their study group, an overall improvement in all categories for subjects that would be defined as introverts, shy or non confrontational, personality traits that are often contribute to anxiety in standard learning environments and would benefit from *Gamification* as a way to ease them into continuing their regular studies, while the performance of the subjects that would be defined as extroverts did not deteriorate significantly.

During this thesis, we will refer to DubitaC as a "game" instead of the more correct denomination of "serious game", as mentioned in [KAY14], serious games possess all the qualities of regular games but their goal is to achieve a final training objective.

We personally felt that such definition puts the developers at the center of a game's discussion, since the "serious" part refers to the developers' intent, while we think that the players should be at the center of the discussion, meaning that if the players would describe a game as "fun" instead of "serious" then it would not qualify for the original definition.

2.2 Games about programming

The game studio Zachtronics is considered by the gaming community as able, time and time again, to create interesting and engaging programming games:

- Starting from TIS-100, this game released in 2015 asks the user to learn to creatively solve programming problems in assembly. [Zac15]
- One year later, in 2016, they moved away from a real world programming language, their game SHENZHEN I/O asks the user to learn both some circuitry basics and an ad hoc programming language that closely resembles assembly. [Zac16]
- Lastly, in 2017, they released Opus Magnum, where the user has to program machinery using a visual programming language that represents the movement of factory components, like extending and contracting pistons for example. [Zac17]

Another example that came out in 2018 is 7 Billion Humans, in this game the user has to learn a new programming language that governs the movement of characters on the screen. [Cor18]

TODO: [images of the games needed]

2.3 Products taking advantage of *Gamification*

The first example we would like to highlight is the online learning platform Code wars.[DH12]■

Code wars offers user created programming challenges that can be solved using many different languages. Each solved challenge gives points to increase your rank alongside another set of points that measure how much the user has engaged with the community. After each challenge a list of solutions is shown where users can vote on "Clever" or "Best practice" solutions of other users, the first satisfying the curiosity of users that want to learn tricks and quirks of the selected language and the second is aimed at users that are more concerned in learning patterns that they can translate into real world scenarios.

The next examples is the very famous online learning platform Duolingo.[LH11]

Duolingo is, at the time of writing, the biggest language learning platform on the internet, it employs *Gamification* to give users a sense of progress and to improve user retention.

Lastly, taken from [Boh15], specifically chapters 2 and 4, a wide variety of products successfully employed *Gamification* in different domains like: encouraging recycling, doing chores, exercising, reducing energy consumption, keeping track of emotions, exploring public places, army recruiting, learning about binary numbers, answering public questions, learning how to use a tool, keeping track of student's progress and encouraging reading at the library.

Chapter 3

Design and Development

Even before starting the development of DubitaC, we had to ask ourselves which goals we wanted to reach.

We identified 2 primary goals, DubitaC **must**:

- help players in learning C++
- implement *Gamification* elements

Alongside that, DubitaC **should**:

- be fun to play
- be well documented
- be easy to extend and maintain

Although these last goals would only be "measurable" through the feedback of end users and future maintainers.

3.1 Comparisons with the State of the Art

Out of all games mentioned in Section 2.2, only one of them teaches a real world programming language (assembly in TIS-100), this is because, broadly speaking, it is easier to create a restricted language to be used in the controlled environment of a game, instead of wrapping an existing language in a game system.

While it is possible to argue that programming is a problem solving skill that is language agnostic and that those games are honing this skill even in their controlled environment, we felt that giving students a concrete starting point using C++ would make it easier to integrate DubitaC in an educational setting.

On the topic of *Gamification*, we noticed that many of the examples in Section 2.3 were targeted towards user retention, however, we felt they were doing so by taking advantage of **negative emotions**, like the fear of losing a streak or offering the user paid options to "repair" a failed task. [Bil21]

Conversely, Code wars only implements "**positive emotions**" *Gamification* elements:

- A ranked progression system that only improves by completing challenges, giving the user a sense of progression.
- A numerical score that represents community interaction, this score is used to unlock privileges that will encourage the user to interact with other parts of the community
- The only interactions between users happen voluntarily with votes and comments but there is no obvious competitiveness in the system.

It should be pointed out that Code wars does not have any *Gamification* elements aimed at increasing user enjoyment directly, this is because Code wars does not aim to be an entertaining videogame.[dt22]

3.2 High level prototype

We decided to implement the challenge solving part of DubitaC in the same way that Code wars does:

Write code → Compile solution on the fly → Test solution against a test battery

Differently from Code wars, however, we decided to make the game directly competitive for user enjoyment, DubitaC will be a time based competition, where users try to find the solution faster than their opponents.

However, faster is not always best, in DubitaC users will be able to point out mistakes in the other users' solutions, by playing a doubting round [Unk], to gain points that will help them come out on top of the competition.

TODO: [images of the game loop needed]

3.3 Initial requirements and Development choices

DubitaC was developed using a prototype driven approach, the initial prototype was created to outline the basic systems of the game and then the development of the first release started from scratch.

The game we were instructed to develop had to follow the general requirements of:

- Containing a game loop that involved writing and understanding code.
- Avoiding a "trivia" focus for the game, for example: "What variable should go in this spot?"
- Offering the possibility for struggling students to "buy" hints to help them create a solution.
- Contain *Gamification* elements.
- Having a graphical interface.

Later, when the decision was taken to make the game multiplayer, a new requirement was added:

6. The interaction between players should be, at least, in the form of analyzing each other's code.

Because of the need of creating a graphical interface, we decided to use the Unity Game engine [HFA04], motivated by our familiarity with the tool.

Choosing a game engine let us focus on the development of the main moving parts without having to worry about rendering, initialization and garbage collection of objects and the general updating cycle.

3.4 The prototype

The prototype focused on creating the interface that the user would interact with, alongside the logistics of compiling and executing solutions on the fly.

Initially, a pseudo-IDE would try to signal to the user syntactical errors and possible bad behaviours, the idea was scrapped to make the execution more lightweight and we felt that the actual compilation would suffice, as long as the players are adequately interfaced with compiler errors.

The prototype was also completely singleplayer and offered a tutorial that would introduce the basic game mechanics as well as some basic C++ concepts.

As is usual in prototype driven development, the prototype was scrapped and the real development started with the main gameplay systems already outlined.

At the end, we felt that the prototype did not offer clear possibility to integrate a multiplayer competitive aspect, the next iteration of DubitaC was more focused on providing a direct competitive feel.

3.5 The online multiplayer development

The multiplayer part of the game was developed using the Unity package: Netcode for GameObjects [Tec20], taking care of the transport layer and unity integration. While the package took care of the low level parts for us, it was not without its difficulties since the package is, currently, still in a pre-release state.

The package offers the basics of a Server-Client architecture with only 2 ways to interact with eachother:

- RPC calls
- Synchronized Variables

While RPC can be useful to send a message to a different machine, Synchronized Variables can maintain a state that is automatically synchornized over the network.

The package offers the option to ensure that the communications are all reliable, meaning that packet loss is handled by the package automatically by resending packets as needed.

Additionally, thanks to an ownership system, every client can and should instantiate the objects that it manages itself, while every object that has to be synchronized between all machines must be spawned directly by the server.

This approach is called "Server authoritative", meaning that any local action is allowed but any networkwide action must be queried to the server that will execute it with its own logic.

The package also implements an event system that notifies the server of a new connection or disconnection, while also offering the possibility to manage the approval or refusal of any connection using custom logic, for example by requiring the correct user credentials, see Section 5.2 for our implementation of the login system.

3.6 The release

During the main development of DubitaC, the only change to the gameplay came in the form of a Test framework integration, we chose to use Catch2 [Hoř15] to test the C++ code produced by the users.

Additionally, various additional systems came up organically and were implemented to create a more complete final product.

Some of these system, that we will explain more in detail later, are:

- Separation and rebalancing of players in different lobbies.
- A separate server interface.
- A secure system to store user credentials.
- An extendable localization system.
- An extendable system to create new codeQuestions.
- An extendable system for user cosmetics.
- A Windows installer.

3.7 Gamification elements

DubitaC implements the following *Gamification* elements:

- A responsive graphical interface.
- A point system.
- An hint system.
- A series of Avatars to choose from.
- A multiplayer competitive environment.

To remark the importance of each of them, it is necessary to compare the gamified product with the original non-gamified learning task.

We identify the original learning task as a laboratory session for our target beginner programming course:

- The students receive a "codeQuestion" from their instructor.
- The students try to solve the challenge on a notepad either alone or collaboratively.
- At the end of the session, the solution is not revealed.

The differences in personalities and work ethics of the students are a driving factor on the choice to work on the challenge alone or as a group and, when the choice is incompatible with the number of students in the laboratory, might negatively impact the learning experience.

Additionally, the instructor gives the students the possibility of continuing their work at home if needed, however, students that were not able to make minimal progress will often struggle to progress at home as well.

Lastly, students are often asked to hand out their work on a biweekly basis, meaning that the instructor's feedback is delayed and some students might feel disappointed by missing their chance for a good grade.

Compared to the laboratory session, a DubitaC game session would:

- Force players to interact with other players' solution.
- Provide an hint system with minimal drawbacks for any player that might need it.
- Provide immediate feedback on their solution performance relative to their peers.

The possibility of working alone or in groups and the possibility of creating a .cpp file to continue the work from home do not change with the introduction of DubitaC to the laboratory session.

Additionally, the final solution is not shown, so that the student solutions might still be graded if the instructors so chooses.

From a student's point of view, DubitaC provides the possibility of learning from other students' mistakes, something that is not possible in a laboratory setting without some strong cooperation between a network of students, alongside offering the possibility of obtaining hints without having to speak with the instructor directly and receiving at least a minimum amount of points for every game session.

This analysis is not trying to argue that instructors might be substituted by DubitaC, as their expertise on **why** something is not working is much more valuable than the rapid feedback provided at the end of a game session. Additionally, the codeQuestion selection requires that the instructor knows both which knowledge the codeQuestion requires to be

correctly solved and which knowledge are the students expected to have at this point of the course.

We compiled Table 3.1 with the expected results derived by the inclusion of the *Gamification* elements in DubitaC.

	User		
	Enjoyment	Retention	Performance
Graphical Interface	✓	×	×
Point system	×	✓	×
Private hints	×	×	✓
Public Avatars	×	✓	×
Multiplayer competition	✓	✓	✓

Table 3.1: Expected improvements caused by *Gamification* elements. User enjoyment can be considered as "more fun" or "less boring". User retention can be considered as an increase in motivation to play more. User performance can be considered as "easier learning experience" or "additional learning experience". Our considerations depend heavily on each single student, meaning that this table cannot be broadly applied to every user.

Chapter 4

Description of DubitaC

In this chapter we will describe every component of DubitaC following an example game session. At the same time, we will mention the systems listed in Section 3.6 when relevant.

4.1 The Windows Installer

A Windows installer has been created to aid in the installation of DubitaC.

The installer will ask if the installation should be user only or for all users, requiring admin privileges for the latter option. Then the user can choose the language displayed during installation, followed by the choice of the installation folder, lastly it is possible to choose between:

- Server installation (For admins)
- Client installation (For players)

In case the user chooses Server installation, the Server DubitaC build will be installed instead of the Client build and the database file will be setup for use.

The installation includes:

- A DubitaC build, with all the necessary resources to start the game and an easy to use .exe file to play.
- A folder containing all the necessary components that are **required** to play:
 - A complete, usable out-of-the-box, g++ compiler for Windows.

- The Catch2 required header, already in the correct library folder.
- The "lld" linker, already in the correct linkers folder.
- An automatically run .bat file that inserts the previous folder in the environment variables of the machine.

In case of a manual installation from the compressed folder, the only additional action required after the extraction is the insertion, in the machine environment variables under "Path", of the path:

`$InstallationFolder$\UpToDateMinGw\bin`

Where \$InstallationFolder\$ is the folder containing the "UpToDateMinGw" directory.

4.2 The main menu

TODO: [image of the main menu needed]

The main menu consists of a title screen with the name of the game, and 2 buttons:

- A button that will open the correct interface, called either "Continue as server" or "Continue as player".
- A button with a country flag.

In case, during installation of Section 4.1, the "Server build" installation was chosen, the database file would be created and the "Continue" button would open the server interface.

In case the "Player build" was chosen, the "Continue button would open the client interface.

The button with a country flag indicates which localization is currently loaded, if the button is pressed, the next localization in the list will be loaded and all text on screen will relocalize in realtime to the new language.

In the top left corner an exit button represented by a question mark lets server and clients alike to disconnect gracefully and close the application.

4.2.1 Server interface

TODO: [image of the server interface needed]

The server interface presents the user with a "Start" button on the bottom right of the screen, an help button represented by a question mark on the top right of the screen and 4 "blocks" of information filling the remaining space.

- At the top, the local Ipv4 address of the server.
- On the left, the available lobbies and which players are currently connected.
- In the center, an input field where the user can insert the number of seconds to give to each player to complete the codeQuestion.
- On the right, an interactive list of all the available codeQuestion, said list can additionally be filtered by "tags", associated with each codeQuestion, by typing the desired tags in the related input field.

It is very important that the user in control of the server communicates the Ipv4 address that is shown at the top of the screen to all players, so that they may connect to it correctly.

If no number of seconds is inserted, the game will utilize a default value of 600 seconds.

If the user inserts more than one tag in the filtering input field, the subset of codeQuestion shown will only contain codeQuestions associated with all the tags.

The "Start" button will be available only when at least 2 players have connected to the server and the codeQuestion has been selected.

If the "Start" button is pressed, all clients will be instructed on which lobby they belong, who their lobbymates are, the amount of time at their disposal and which codeQuestion they have to solve, before synchronously loading the first round.

The lobbies on the left are not interactable, the server cannot move the players into an arbitrary lobby, instead after pressing the "Start" button, a rebalancing algorithm reassigns the players in as many lobbies as needed to create balanced groups of players.

The help button in the top right activates a color coded overlay explaining where the user has to interact to ensure a correct initialization of the game session.

4.2.2 Client interface

TODO: [image of the player interface needed] The player interface consists of:

- 2 input fields to enter username and password.
- A checkbox to swap between known credentials and new credentials.

- A button that will attempt a connection with the server called "Play".
- Another input field to insert the Ipv4 address of the server.
- A button that will open the tutorial section called "How to play".
- A button that will open the settings section called "Settings".

The "Play" button will be disabled until the user inserts a valid Ipv4 address in the proper input field.

If the user presses the "Play" button, a connection is attempted at the inserted Ipv4 address using the user credentials inserted in the first 2 input fields, on a successful connection, a Cosmetics interface is opened.

If the user enters a username and password that matches an entry in the server's database and there are not 24 clients connected already, the user starts it's own client and connects to the server.

If the user does not insert known credentials or all the lobbies were full, the server will refuse the connection.

After a refused connection, a message will notify the user about the unsuccessful connection and they will be prompted to insert new credentials.

If the user does not insert known credentials but checks the "Create new account" box the connection will be accepted, if not all lobbies are full and the username is not already present in the database, and the username is inserted in the server's database, making the inserted credentials "known" for a future login.

If the user inserts credentials that do not satisfy the requirements, a message will notify them of the requirements and they will be prompted to insert new credentials.

If the user inserts an invalid Ipv4 address, a message will notify them about the unsuccessful configuration and the "Play" button will be disabled.

If the user inserts a valid Ipv4 address, the next attempts at connecting to the server will only happen through that address, the user is notified that they inserted a valid Ipv4 address, however, there is no guarantee that an available server will be listening on the other side.

If the user presses the "How to play" button, the tutorial panel will open, here a series of images, that can be navigated by two arrow keys at the sides of the screen, will present the different important screens of the game, while an helpful message will describe each image's color coded boxes.

If the user presses the "Settings" button, the settings panel will open, here the user can choose the values of 3 options: the volume, between 0 (muted) and 4, at 25% increments each, the timeout parameter between a choice of different seconds from 1 to 9 and the path where to save session cpp files.

4.2.3 The Cosmetic panel

TODO: [image of the cosmetic panel needed] The cosmetics panel is shown twice during the game: after a successful connection with the server and at the very end of the game. In both cases it consists of a message informing how many points they have accrued by playing the game on this account (this information is saved in the database) and a list of all existing Avatars ordered from least to most "expensive", Avatars that are more "expensive" than the user's available points display an obscured sprite instead of the standard one. Differently from the end of the game, after a successful connection the user is waiting for the game session to start, during this time if an available Avatar is clicked, a message will be sent to the server informing it of the user choice, the selected Avatar when the session starts will be shown to all other players in the same lobby.

If an Avatar that is currently unavailable is clicked, the selection will not be sent to the server.

If a player has not chosen an Avatar in time, a random "free" one will be assigned.

4.3 The first round

Once the user managing the server presses the "Start" button, all the clients will load the client interface for the first round, similarly the server will load the server interface, even if the interaction between the server and the users are, at this point, minimal.

The goal of the first round is to be the fastest client to create a solution that solves perfectly the codeQuestion given.

4.3.1 Server interface

TODO: [image of the server interface needed] The server interface divided into 3 parts, all dynamic and responsive depending on the messages that the server receives from the users:

- On the left, the list of lobbies that are currently in use.

- On the right, the time remaining until the end of the round.
- On the bottom, a progress bar tracking the progress until the next round.

The list of lobbies only displays the lobbies that have at least 2 players inside, alongside how many player are "ready" out of the total number of players in that lobby. Every time a client sends a message to set "ready" or "not ready", this display is updated accordingly.

The remaining time displays a simple countdown from the time given to the players to solve the codeQuestion to zero, at which point all clients have to submit their solution, weather they were "ready" or not beforehand.

The progress bar is not tied with the remaining time and instead tracks how many clients were able to communicate their solutions with the server.

The progress bar should fill quickly since all clients should send their solutions more or less at the same time.

4.3.2 Client interface

TODO: [image of the client interface needed] The clients are greeted with a panel in the center of the screen informing them what codeQuestion they have to solve, once dismissed the main interface is revealed:

- A button to create cpp files, called "Export".
- A button to compile and test their solution, called "Test".
- A button to open the hint panel, called "Hints".
- A button to open the log panel, called "Log".
- A timer tracking how much time has passed and how close it is to expire.
- A button to notify the server that the user is satisfied with their solution, called "Ready".
- A big input field where the player will write their solution.
- 2 small buttons that zoom in and out of the solution.

If the user presses the "Export" button, a cpp file will be created, containing only the user solution, at the path that they specified in the setting (see Section 4.2.2).

If the user presses the "Test" button, the user solution is wrapped in a Catch2 wrapper file that was created for this codeQuestion, it is then compiled and run against a base test battery, all the outputs that would show up in the terminal are caught, parsed and shown to the user in the log panel.

If there have been no errors in the compilation and the test battery passed completely, the user is granted the possibility of pressing the "Ready" button.

If the user presses the "Log" button, the log panel will open, here all the communication between the terminal and the user is shown, a button called "Clear" in the top right will empty the log, however this is not necessary, as the log can be infinitely filled and the user can scroll through its content as they would on the terminal itself.

Unfortunately, because of the limitations of the localization system, see Section 7.1, the log could not be translated and the default english language is used.

If the user presses the "Ready" button, the server is notified that the user has finished, the input field of the solution stops being interactable as the user will wait the end of the timer.

If the user wants to modify its solution after having pressed "Ready", it is possible to do so by clicking on "Ready" again, notifying the server that the user should be removed from the list of "ready" clients.

The input field for the solution behaves mostly like a standard notepad, it supports scrolling, copying, pasting, cutting, undo and redo with the most common key combinations, zooming is instead delegated to the 2 small buttons at the side.

4.4 The hint panel

TODO: [image of the hint panel needed] The hint panel has a dual purpose: First it contains the description of the codeQuestion, so that players can always refer to it in case they forgot some specification and second it given the opportunity for users that are not sure how to proceed to "buy" some hints. At the cost of reducing their final payout, players will be able to read hints that have been predetermined during the codeQuestion creation.

This system can be seen as a *Gamification* element, since the equivalent non-gamified action would involve asking the professor for hints directly, a portion of students would "avoid the embarrassment" of asking publicly or might think that the professor would not be willing to help privately. This system is completely automated instead, and should have a positive impact on struggling students since, with a little hint in the right direction, they might be able to start competing and maybe even create a better solution than other more confident students. At the same time, since the cost is not too steep and there are

no gameplay related disadvantages in getting an hint, hints should never be purposefully avoided, making it even more likely that students would use the hints to create a better solution, no matter the motivation for asking one.

4.4.1 Communication behind the scenes

The only communication that clients and server have during this round are:

- The clients can send "ready" and "not ready" tokens to the server.
- The clients must send their solutions at the end of the round.
- The server shares all the solutions with the clients that are in the same lobby before loading the next round.

4.5 The second round

The second round will start after the available time has run out and the clients have all shared their solution.

The goal of the second round is to create correct "doubts", so that clients that are behind may hope to get ahead by finding flaws in their competitors' solutions.

4.5.1 Server interface

TODO: [image of the server interface needed] The server interface is exactly the same as Section 4.3.1, except for 2 differences in what type of data is tracked:

- The ready player counters have been substituted with received doubts counters.
- The progress bar now tracks the retrieval of doubts, creation of solutions, compiling solutions, execution of solutions, sharing of execution results and lastly sharing the updated leaderboard.

The received doubts counter contains, for each lobby, the amount of doubts the clients of that lobby have sent, differently from last round, this counter should fill quickly since the users' doubts will all be sent more or less at the same time when the time at their disposal is finished.

The progress bar is instead filled more gradually, as the steps necessary to finish this round are many and might take some time, we felt it was important to keep the bar responsive so that the user managing the server knows that the game is not hanging.

4.5.2 Client interface

TODO: [image of the client interface needed] The clients are greeted with a panel in the center of the screen informing them how to continue with the doubting "round", once dismissed the main interface is revealed:

- A list of Avatars on the left side of the screen.
- A button to open the doubt panel, called "Doubt".
- The same timer as Section 4.3.2.
- The same zoom buttons and input field as Section 4.3.2.

The Avatar list shows the current leaderboard, the first place player at the top.

If the user selects one of the Avatars, the corresponding player's solution will be shown in the input field.

If the user selects their own Avatar, the "Doubt" button is disabled, since creating a doubt for your own solution is not allowed.

Once the user has read a solution, it can choose to doubt it by pressing the "Doubt" button at the top of the screen.

When the timer runs out, a loading screen will appear, signaling to the players that there might be some wait time.

4.5.3 The doubt panel

TODO: [image of the doubt panel needed] The doubt panel contains all the necessary interactive components to create a doubt:

- A button to return to the previous screen, called "Back".
- A button to remove a previously created doubt from the currently selected target, called "Remove doubt against current player".

- An input field reserved for the inputs.
- An input field reserved for the expected output.
- A button to signal the prediction of a failed compilation, called "Does not compile".
- A button to signal the prediction of an non termination, called "Timeout".
- A button to signla the prediction of an unexpected termination, called "Crashes".
- An input field reserved for the expected correct output.
- A button to confirm the creation of the doubt, called "Doubt".

The player will be able to create a doubt only if all the following requirements are met:

- The input field reserved for the inputs is filled with a string.
- One, and only one, of the following:
 - The input field reserved for the expected output is filled with a string.
 - The "Does not compile" button is pressed.
 - The "Timeout" button is pressed.
 - The "Crashes" button is pressed.
- The input field reserved for the expected correct output is filled with a string.

Additionally, every input field must be filled by respecting the corresponding part of the function's signature: The inputs must resemble the function's arguments in number and types , while the outputs must only match the function's return type.

If the user wishes to create a doubt by supplying a string to the expected output, said string must be strictly different from the one supplied to the expected correct output, since, if they were equal, the player would be "doubting" a solution by expecting it to return the correct result.

To aid the user in creating a valid doubt, the UI elements in this panel will activate and deactivate to avoid the creation of a badly formatted doubt: The "Doubt" button will be deactivated until all requirements are met, while the four possible ways to doubt will deactivate if they weren't the one chosen to enforce the "One, and only one" restriction.

If the user creates a new doubt against the same player, the old doubt is lost and the new one will take its place, without needing to remove the original with the "Remove doubt against current player" button first.

Lastly, the "Remove doubt against current player" button will be deactivated until a doubt against the target has been created and deactivated again after it has been removed.

4.5.4 Communication behind the scene

The communication in this round is more frequent, and the data shared bigger, than last round:

- The server sends the ranking of all clients in a lobby to the clients in said lobby.
- The clients must send their doubts at the end of the round.
- The server sends a prepared string that contains the entirety of their solution, and the doubts made against it, to each client.
- The clients send to the server the result of the compilation and execution of their solution.
- The server shares the new leaderboard of the lobby to all clients in said lobby.

4.6 The slideshow

TODO: [image of the slideshow needed] The slideshow will start after the available time has run out and clients and server executed all solutions.

From here onward, there is no more interaction between the players and the game.

4.6.1 Server interface

TODO: [image of the server interface needed] The server interface contains only the usual progress bar, this time tracking the execution of the final test batteries before loading the final part.

4.6.2 Client interface

TODO: [image of the client interface needed] The client interface is composed of 3 main parts:

- On the left, the leaderboard of the clients in the lobby representing the solutions that have been doubted.
- On the top, the leaderboard of the clients in the lobby representing the doubts against the solutions.

- In the center, all the text required to describe the doubts, the results of the execution and a final verdict.

The slideshow will start by highlighting 2 clients, one on the left leaderboard and one on the top leaderboard.

The central text will be populated by the information of the doubt that was created by the client highlighted at the top towards the solution of the client highlighted on the left.

Additionally, the information about what the server returned when executing the perfect solution and what the user solution returned when given the doubt's inputs is also shown.

The final verdict on the doubt is then used during the final leaderboard update.

4.6.3 Communication behind the scenes

Every client has enough local information available to run the slideshow by itself, however:

- The clients send requests to synchronize the highlighting of the players in the leaderboards.
- The clients signal to the server that they are ready for the final part at the end of the slideshow.

4.7 The final leaderboard

TODO: [image of the final leaderboard needed] The final leaderboard is the complete end of the session, all the points coming from doubts and executions have been calculated and a final leaderboard is shown.

4.7.1 Server interface

TODO: [image of the server interface needed] The final server interface contains:

- A button to turn off the server, called "Disconnect".
- Some information about the session.
- A button to save all the clients' solutions on the server's local machine, called "Export all solutions".

The sessions' information contains the number of clients that participated in this session and which codeQuestion they had just solved.

4.7.2 Client interface

TODO: [image of the client interface needed] The client interface contains:

- A button to disconnect from the server, called "Disconnect".
- A button to save on the local machine the player's own solution, called "Export my solution".
- A button to save on the local machine the solution of the player that got first place, called "Export best solution".
- A display in 3 tiers containing the clients ordered by their final rank.

If the user presses on the "Disconnect" button, the client is disconnected from the server and a very similar panel to Section 4.2.3 will appear, with the only difference that the user can only press a button called "Back to menu", in the top left corner, to be sent back into the first interface described in Section 4.2.2.

If the user presses either "Export" button, a cpp file will be created at the path that they specified in the setting (see Section 4.2.2).

Lastly, every Avatar contains 2 different sprites that are only shown in this interface, the first one is a "positive" sprite that is shown for users that finished in the top 3 and the other is a "negative" sprite that is shown for users that finished in the bottom 3.

This small detail can be considered a *Gamification* element, since it should encourage users to try different Avatars during future sessions to see the corresponding "positive" and "negative sprites".

Chapter 5

Implementation

In this chapter we will explain some more technical details about the different systems that are working together before, and during, a game session.

5.1 The main gameplay

The main game loop consists in creating a valid C++ program that will be compiled and then run against different test batters.

5.1.1 Creating a valid cpp

Creating a compilable cpp file would not require much work, as a matter of fact, provided the users are able to write a compilable C++ solution, it would be possible to save to a file directly the user solution and compile it.

Every time a solution does not compile, it is caused by mistakes on the user's part and the game would punish them accordingly.

However, because we want to take advantage of the Catch2 test framework [Hoř15], we need to make sure that the user solution is both a valid C++ program and a valid Catch2 program.

Furthermore, we wanted to be able to recognize when an execution would not terminate, since infinite loops are one of the many pitfalls of programming for beginners and the terminal would not offer any indication that such an undesirable behaviour is occurring.

5.1.2 Creating a valid Catch2 program

Catch2 prides itself of being lightweight, easy to use and easy to integrate, even if our use case is a bit different from the most common ways to use a test framework, we found that those claims are not, for the most part, unfounded.

The 3 requirements for integration are:

- Add the Catch2 header to the possible libraries that the g++ compiler can link.
- Include the Catch2 header file in the source code of all the programs that will need to be tested.
- Either create a new file or extend the existing target file with a Catch2 directive that would signal to the compiler where the test are located.

Regarding the first point, during installation, we provide a folder with all the necessary to correctly compile the user solutions, refer to Section 4.1 for the contents of said directory.

Regarding the second point, it is trivial to create a new file that will contain in the first line:

```
#include "catch.hpp"
```

and then append to it the user's solution to satisfy the requirements.

Lastly regarding the third point, a series of compiler directives can be added to enable or disable Catch2 features, but most importantly the directive:

```
#define CATCH_CONFIG_MAIN
```

Is needed so that the compiler knows that this file will both contain the source code to test and the tests themselves.

Regarding the creation of the tests, Catch2 offers a variety of different tools to test code executions, from the most basic, testing the returned value of a function, to more sophisticated ones like catching exceptions that match a given name.

Because of the options given to the player, See Section 4.5.3, we needed to make sure that we were able create a test for all the possible doubt types. The doubt types that we considered were:

- The target function will return a wrong value.

- The target function will not compile.
- The target function will timeout.
- The target function will crash.

Unfortunately, except for the first type, there is no Catch2 function that could be used to easily test these doubts and some out-of-the-box thinking was required to tackle correctly each one of them.

5.1.2.1 Detecting non compilation

Firstly, since Catch2 tests are run during execution, it would be impossible to detect non compilation using the framework only. Luckily, we have to try to compile all solutions to being able to test them in the first place, as such it is possible to intercept the result that would be displayed to the standard output after the execution of the compilation command and parse it for our needs.

In particular, a successful compilation does not return anything on standard output, while an unsuccessful one will display the errors encountered during compilation. With this information, it is now trivial to check for solution that do not terminate because the returned string to the standard output would have a length bigger than 0.

However, it is not trivial to detect doubts that expected a solution to not compile, but said solution compiles correctly instead.


For this we create a dummy test for which the success or failure is not relevant, what is relevant is only the presence of said test in the solutions, so that the corresponding doubt is not lost.

5.1.2.2 Detecting non termination

Regarding the timeout, we had to find a way to tackle the Halting problem [T⁺36], which is a classic problem in programming stating that it is impossible to create a machine that can classify if a program will terminate or not given no additional restrictions.

The obvious solution was to introduce a restriction in the form of a time limit after which any program that is still running will be flagged as non terminating and aborted. However, such a task is not trivial and required the creation of a general wrapper for any user solution and then an additional transformation of said wrapper so that it would work specifically for the given user solution.

Figure 5.1 shows the whole wrapper that we wrote for the task, additionally, we will explain briefly all of its parts:



OfflineImages/Wrapper.png

Figure 5.1: The whole wrapper used to make sure that any compilable C++ code can be run as a Catch2 executable with a maximum timeout before being aborted of TIMEOUT seconds. Every piece of "comment" between double forward slashes will be substituted with the relevant characters to convert the wrapper from its general form to a specific one.

```
//_type_// //_name_//(int __timeout__, //_full_arguments_//){
```

The first line is the signature of a new function, notice how we indicate that a part of this code needs to be replaced by using the pattern:

```
//_label_//
```

All labels refer to different parts of the function that users were asked to complete to solve the codeQuestion. In this line in particular, type represents the returning type of the function, name represents the name of the function and full arguments represent the

arguments of the function in the form: type name and separated by commas. Notice the presence of an argument called `__timeout__`, this represent a user definable amount of seconds that the function will be allowed to run before being aborted.

```
std::mutex __mutex__;
std::condition_variable __conVal__;
std::exception_ptr __exPtr__ = nullptr;
bool __wakeUp__ = false;
//_type_// __retVal__;
```

Inside the function we declare the variables that we need: a mutex, a condition variable, an exception pointer, a boolean and a variable used to store the return value of the function.

The exception pointer is necessary to catch and rethrow an exception called by an internal function, while the other variables are used in the management of the time limit.

```
std::thread __thread__([&__conVal__, &__exPtr__, &__wakeUp__,
                        &__retVal__, //_&_arguments_//]() {
    try{
        __retVal__ = //_name_//(_//_name_arguments_//);
    }catch(...){
        __exPtr__ = std::current_exception();
    }
    __wakeUp__ = true;
    __conVal__.notify_one();
});

__thread__.detach();
```

The final part of the setup requires the creation of a sepatate thread, said thread is supplied with the condition variable, the exception pointer, the boolean, the variable that will store the result of the function and all the arguments of the target function in the for `&` name and separated by commas.

The thread will execute the target function inside a try-catch block, if an exception is thrown, it is saved in the exception pointer, otherwise the boolean is set to true to represent a correct termination and the condition variable is notified.

After the setup the thread is immediately started.

```
{
    std::unique_lock<std::mutex> __lock__(__mutex__);
```

```

        if(__conVal__.wait_for(__lock__, std::chrono::seconds(__timeout__), [&__exPtr__,
                                                                           &__wakeUp__]() {
                return (__wakeUp__ || (__exPtr__ != nullptr));
            })){
            if(__exPtr__ != nullptr){
                std::rethrow_exception(__exPtr__);
            }
        }else{
            throw std::runtime_error("Timeout");
        }
    }

    return __retVal__;

```

Lastly, the time managing part is started on the main thread, the mutex is locked until the condition variable is notified or until a `__timeout__` amount of seconds has passed.

Whichever condition is satisfied first will trigger the return line, that in return will evaluate the condition to true if the the function terminated or crashed and false if the time expired.

The true branch of the code checks if the exception pointer has been filled, if yes it rethrows the exception. The false branch of the code will throw a runtime error called "Timeout".

If neither happens, the execution was successful and the function will return the value that the function in the other thread had returned.

This wrapper will now let us execute the target function and be able to return a value if the execution terminates successfully, be able to catch the exceptions if the execution did not terminate successfully and be able to catch the custom "Timeout" exception if the execution took too long.

This method has 2 limitations, but we minimized their impact as best as we could:

- The function that needs to be tested is now the new wrapper function and not the original one, as long as the doubts are created programmatically, this is not a problem, but when creating new codeQuestions the tests will need to be written by an admin that must keep in mind such limitation.
- Since we need to insert in the wrapper the name of the function arguments to transform it from general to specific, a user that renames any of the function's arguments to the same name as any variable's name reserved to the wrapper (`__mutex__`, `__wakeUp__`, ...), will cause the solution to not compile correctly. To reduce the chance that a non malicious user would stumble into this limitation, all the variables are preceded and followed by a double underscore, reducing severely the chance of a collision.

5.1.2.3 Detecting bad termination

With "crash" we refer to both an unexpected exception, that is not the custom "Timeout" exception, and the program terminating fatally, for example with a SEGFAULT error.

Thanks to the efforts taken to make sure that the thrown exceptions were not lost in Section 5.1.2.2, we can test for unexpected exceptions by simply creating a test that expects an exception different from the custom "Timeout" exception.

Regarding fatal crashes, nothing can be added to the source code to be able to detect them. However, Catch2 already contains a feature specifically to intercept system failure signals during execution and count the test as a failure.

The feature is disabled by default for windows machines but can be enabled by adding the following compiler directive:

```
#define CATCH_CONFIG_WINDOWS_SEH
```

5.1.2.4 Creating the tests

Having overcome all limitations, we can now create the template for all 4 types of doubt.

The variable names that we will use in this section are:

- `clientId`, the id of the client that created the doubt.
- `targetId`, the id of the client that was the target of the doubt.
- `functionName`, the name of the function.
- `TIMEOUT`, the amount of seconds before a running execution is aborted.
- `givenInput`, the inputs given to the function for the doubt.
- `expectedOutput`, the output that the client that created the doubt expects from an execution of the function with `givenInput` as arguments.
- `correctOutput`, the output that the client that created the doubt expects from an execution of a function that solves the `codeQuestion` with `givenInput` as arguments.

A Catch2 test case is composed of 4 parts:

- `TEST_CASE`, indicating that a new separate test is about to be declared.

- The test's name, as the first argument of the test case.
- The test's tags, as the second argument of the test case.
- The content, where we can use all the functions that Catch2 gives at our disposal.

In our use case we never used more than a single function inside the tests and kept all test cases completely separated, but the possibility to create more complex test structures exists.

The doubt expecting a wrong value to be returned can be tested with:

```
TEST_CASE("clientId", "[user]"){
    CHECK(functionName(TIMEOUT, givenInput) == expectedOutput);
};
```

The doubt expecting the solution to not compile is tracked with:

```
TEST_CASE("clientId", "[user]"){
    CHECK(functionName(TIMEOUT, givenInput) != correctOutput);
};
```

The doubt expecting the solution to timeout can be tested with:

```
TEST_CASE("clientId", "[user]"){
    CHECK_THROWS_WITH(functionName(TIMEOUT, givenInput), "Timeout");
};
```

The doubt expecting the solution to crash can be tested with:

```
TEST_CASE("clientId", "[user]"){
    CHECK_THROWS_WITH(functionName(TIMEOUT, givenInput), !Contains("Timeout"));
};
```

And for fatal crashes, Catch2 will handle them automatically.

Additionally, because we consider a doubt to be completely correct only when it also predicts correctly the output of a perfect solution, it is necessary to run the execution of said perfect solution (by the server) with all the user doubts appended to it.

Because all type of doubts contain a correctOutput and the server solution will never fail, it is enough to add on the server solution this test for each doubt:

```
TEST_CASE("clientId->targetId", "[user]") {
    CHECK(functionName(TIMEOUT, givenInput) == correctOutput);
};
```

5.1.2.5 Executions and Results

Catch2 offers the possibility of executing only a part of the tests in a file by specifying which tags to execute, we take advantage of this to reduce the waiting time that might be generated by a longer execution.

After the relevant tests have been added, the executions proceed as follows:

- If a user is testing its own solution, the base tests are appended and then the solution is compiled and executed.
- After all doubts have been appended, the server sends back to each client their own solution, then each machine (server and clients) compiles their own solution and executes only the tests with tag "[user]".
- For the very last executions, the clients execute from their own executables, that were created previously, only the tests with tag "[final]".

After an execution, Catch2 outputs one line for each test, containing if the test passed or failed and a brief recap of the test contents if it did not fatally crash, or the system raised error, otherwise.

Lastly, all the results that would be displayed on the standard output are intercepted, like in Section 5.1.2.1, and parsed to calculate the points that needs to be assigned to the clients based on the performance of their solution and their doubts.

5.2 The account management

Persistency is vital to any game to keep track of user progress, while the game sessions are self contained enough to not need a persistency system, the progress is mainly used as a *Gamification* element, where users will want to use the points that they have earned in previous sessions to use a different Avatar in the next ones.

For developers using Unity, the main methods suggested to save player data are the following:

- Use Unity's PlayerPrefs.

- Serialize data into a file.
- Create a JSON file.

All of these are local methods creating a file on the user's machine, while we wanted to create a system tied to user credentials, meaning that storing and validation would be done on a different server machine.

5.2.1 Obscuring the data

PlayerPrefs and JSON are not obscured at all, while using a serializer would mean that the final result is completely obscured, being made up of 0s and 1s.

The level of obfuscation that we wanted to achieve was "inbetween", where the names of the players and their progress could easily be modified by an admin, but the password would be undecipherable for anyone accessing the database or intercepting the communication.

We decided to take a simple approach where the "database" consists of a single CSV file stored on the server, then, when the user credentials are exchanged, the communication consists of a byte array that is formatted to contain 3 pieces of information:

- One leading bit used as a flag to check if the credentials belong to a known or a new account.
- A series containing between 1 and 20 bytes, representing the username, in plain text.
- A series of 32 bytes, containing the user password after salting and hashing.

The server then saves the credentials as a prepared statement in a new row of the database.

An example of a serverside validation of the user credentials is shown in Figure ??.

TODO: [image of the

5.2.2 Security considerations

Although it is possible to devote much more resources to database security than what we have for this account management system, we still wanted to follow solid security principles.

We wanted to defend against the following malicious actors:

- An "eavesdropper", someone able to intercept the communication between a client and the server.

- A "peeker", someone able to read the database, either fully or in parts.

Since DubitaC does not deal with sensitive data, such attacks have little to no impact, however, it is important to remember that a high amount of users reuses usernames and passwords regularly, meaning that any password that a malicious actor can retrieve is a possible breach in one of the other user's accounts.[DBC⁺14]

This is why we decided to follow a 3 steps approach:

- Force the users to choose their password to be at least 8 characters long.
- **Before** sending the password to the server, it is salted and hashed clientside.
- Before being saved in the database, the password is salted and hashed once more.

If the password were to be sent as plain text, an eavesdropper or peeker would trivially be able to retrieve it alongside the client's username.

If the password were to be sent as hashed text and the server saved it as is, an eavesdropper could try to employ cracking techniques like lookup tables or using rainbow tables. Additionally, a peeker would be able to immediately identify which users use the same exact password, since they would have the same exact hash, and employ the same tactics as the eavesdropper a bit more efficiently.

Some cracking attempts will be more efficient the shorter the password is before being hashed, although our 8 characters limit is still not sufficient to completely discourage a malicious actor, it will slow down their attempts.

If the password were to be sent as hashed text and the server saved it after hashing once more, a peeker would not be able to take advantage of the techniques working on short passwords anymore, since the database would contain the hash of a 32 character long password, that is the result of the hashing of the original password.

Unfortunately, this has no effect on an eavesdropper and does not prevent the peeker to learn of which passwords are the same in the database, since identical string, once hashed, are still identical.

Lastly, if the password were to be salted and hashed on both sides before being stored it in the database, both eavesdroppers and peekers would not be able to use neither lookup tables nor rainbow tables, as the original password got extended by a completely random string before being hashed, the result cannot possibly have been precomputed.

The salt used by the user is a combination of its username and a predetermined string, while on the server a new completely random salt is calculated during the account creation for each new account.

The salt is stored in plaintext since there is almost no advantage in knowing which salt has been used.

Additionally, a peeker can no longer understand if users are using the same password, since the salt is random and unique for all clients, the hashes stored in the database are going to be different no matter what the original password was.

We are aware that no system is truly secure but we believe that taking the precautions that we did was the least we could do for a system handling user credentials.

Table 5.1 summarizes the considerations of this section.

	Eavesdropper	Peeker
Plaintext communication and storage	No action needed	No action needed
Serverside hashing only		Short password attacks, Lookup tables, Rainbow tables + deducing identical passwords
Clientside hashing only	Short password attacks, Lookup tables and Rainbow tables	
Hashing on both sides		Lookup tables, Rainbow tables + deducing identical passwords
Salting and hashing on both sides	Dictionary attacks or brute force	

Table 5.1: Table detailing the possible techniques that the 2 types of malicious actor, that we considered in Section 5.2.2, could perform depending on the different decisions taken when handling the users credentials, specifically their passwords.

5.2.3 Using a text file as a database

While storing data, the most obvious candidate is the use of characters, since the messages sent over the network by the user are a simple sequence of bytes, it is possible to cast the byte values between 0 and 255 into a 256 dimensional array of characters, this is, in essence, what Encodings do, even if using more complex systems, like being able to use more than one byte for a character.

This incurs in an incompatibility with the CSV format, csv stands for Comma Separated Values, meaning that 2 characters have a special meaning:

- The comma, used to separate values on each row, represented on the ASCII table as

character number 45.

- The newline character, used to separate each row, represented on the ASCII table as character number 10, preceded by a Carriage return control sequence character on Windows machines, represented on the ASCII table as character number 13.

Because of this, any CSV standard parsing would not work, as a byte containing values 10, 13 or 45 would be perfectly valid but would violate the standard structure of the file once stored as a character.

Our initial approach was exploring some of the encodings offered by .NET, in particular, we wanted to satisfy the following properties:

- The encoding should not contain the characters that have special meaning in a csv, the comma, the newline and carriage return.
- The encoding should not contain control sequence characters or whitespace, to avoid mishaps during read operations.
- The encoding should only contain characters that can be visualized in the default text editors of our choice, in our case, all characters should be contained in the standard font assigned to an english speaking european installation of Visual Studio (Cascadia Mono) and Notepad++ (Courier new).

Unfortunately, any 256 encoding that we could find did not have the first 2 properties, indeed, because of backwards compatibility with ASCII, using UTF-8, UTF-16 and ASCII itself would incur in the leading 31 control characters and will contain a comma at number 45.

We then searched for a character block in UTF-8 and UTF-16 encodings that could be completely visualized, even if the task did not look promising from the start, since many of these blocks are trying to represent non european characters.

We were not able to find any block that would contain at least 256 characters that could be visualized, and in the few were it might have been possible, they would not be contiguous or following an obvious pattern, making the use of such encodings more akin to a lookup table than an easy and ready solution.

We then conceded that a 1 to 1 correspondence between byte values and characters could not be easily achieved.

It was possible, since we do not mind if the database entries are longer than expected, to convert the bytes into base64 which, instead of the usual 8 bits, only requires 6 bits to

output a character, meaning that, for example, our 32 bytes hash will be converted in 43 characters where last one will need 2 bits of padding.

This meant that the Encoding of choice, in this case UTF-8, will convert all the characters from plaintext to bytes, but the storing and loading of data will only deal with base64 strings that, conveniently, do not have whitespace nor commas in them.

5.2.4 Updating the user points

One last challenge that we had to face regarding persistency was the necessity to update the user's points in the database.

In the context of file writing, "updating" is better known as "overwriting" or simply "writing" on top of data that was already there.

The 2 standard methods are:

- The "append" method:
 - Read the file until the desired spot.
 - Write what has been read into a new file.
 - Write, into the new file, the new data.
 - Read the rest of the original file.
 - Append what has been read into the new file.
 - Delete the original file.
 - Rename the new file to the same name that the original file used to have.
- The "Seek" method:
 - Read the file until the desired spot.
 - If the buffer read past the spot, use the Seek function to move back to the desired spot.
 - Write the new data.

We opted for the second option, as we thought it would be cleaner and the only additional code that we needed to add to a simple file reader was the calculation of how many places we needed to move the cursor to reach the correct spot.

However, we wanted to write the new data over the old data, while at the same time not losing any information around it.

Unfortunately, there is no standard function to ensure that the writing process is bounded, for example a database like this:

```
Alice,42,Hash1,Salt1
Bob,5,Hash2,Salt2
Eve,100,Hash3,Salt3
```

Would incur into overwriting problems when all users are awarded 100 points, since some new lengths of the "points" field are longer than the old ones, characters around them are lost:

```
Alice,142Hash1,Salt1
Bob,105ash2,Salt2
Eve,200,Hash3,Salt3
```

This would violate the structure of the CSV (each row should have 4 comma separated values, but 3 were found) and should be avoided at all costs.

While at first we thought it would be possible to know when an overwriting was about to happen, we quickly realized 2 major setbacks:

- Since the username and the length of the points are variables, it is impossible to execute any calculation of cursor position before reading the row.
- Even if that was possible, the moment we detect that an unwanted overwrite was about to happen, we would either fallback to the append method or would need to rewrite the rest of the database from this point onwards.

We avoided these problems by fixing both the maximum and minimum length of the "points" field to the same value.

By setting the maximum length to 4, we are allowing a maximum of 9999 user points and, to maintain the minimum length of 4, numbers under 1000 are padded with leading zeros, that do not interfere with the parsing to integer during gameplay.

This fixed length can be modified by a constant in the appropriate class, although doing so would require that the administrator of the database changes all the "points" fields accordingly, so that the next overwriting attempts are not problematic.

5.3 The localization

In game development, and software development in general, the easiest approach to localization is to create a 1 to 1 table containing a "label" on one side and the equivalent localized string on the other.

Specifically, the localization of a game requires that localization files are loaded at runtime so that every "label" in the game is swapped for a region appropriate version of itself by a localization manager.

To be easier to maintain, it should be immediately obvious which strings in the project actually represent "labels" and which do not.

In DubitaC we decided to adhere to the following pattern for all localization "labels":

`_descriptive_name_with_no_spaces`

5.3.1 The TextManager and LocalizableText classes

Unfortunately, Unity does not support a Localization system out of the box, so we had to create 2 classes specifically for the task.

TextManager is tasked with loading the localization on a static dictionary and acts as a general manager class for all the LocalizableText components in the scene.

LocalizableText acts as an extension of a standard component that displays text, with the added possibility of having a label assigned to it, this label can be assigned externally and, everytime it changes, the text will localize itself by quering the currently loaded dictionary.

We have also given to the users the possibility of changing the current localization by pressing a button on the main menu, at each keypress the next localization in the list will be loaded and the user will immediately notice the change since all the text on the main menu will be localized in real time.

5.3.2 Avoiding a common pitfall

Strings compose the vast majority of text that would need a localization, however, there are some situations where some images, audio or video might need to be used to convey information, if that is the case, the development time to localize these parts will be considerably higher compared to the string localization.

We had to consider this when we decided to create a small tutorial for DubitaC, initially

we wanted to utilize some small videoclips with audio explaining the main parts of the game, but quickly scrapped it for a more static approach.

We decided to use images of the main game scenes but, even if the images are in english, there should be no problems in following along, since the important parts of the images are boxed in a color coded rectangle and some text at the bottom of the screen will explain what the tutorial is trying to show the player.

5.3.3 Fighting Unity's main script execution

Because of Unity's lack of an in-house localization system, any attempt to create one will need to manouver around Unity's main script execution, see Figure ??.

TODO: [image of unity's script execution loop needed]

The main difficulties that we encountered are:

- Loading a localization at runtime can happen at the earliest in an Awake() or Start() function call of the localization manager.
- All text that needs to be localized can fetch the correct localization at the earliest in their Awake() or Start() method.
- All text that is not currently enabled when the localization dictionary changes, cannot localize itself.
- The localization dictionary should persist between scenes.

We were able to overcome all of them in the following ways:

To avoid possible out of order initializations, the loading of the localization must happen before the LocalizableText have a chance to query for it, this meant that the TextManager's localization loading was inserted in its Awake() call, while the LocalizableText will try to localize themselves in their Start() call.

To avoid the non localization on disabled text, the TextManager keeps a reference to of all text in the current scene that is disabled, so that when the user presses the button to change the localization, all LocalizableTexts can be updated even if disabled.

Because of the previous approach, we reduced the amount of needed references by making it possible to change the localization only in the first scene.

To remove the need for the localization to be reloaded in every scene, we moved the dictionary containing the current localization into a static class that will persist through all scene changes.

5.3.4 Final result

At release, DubitaC contained 2 fully localized languages, English and Italian, and an easily extendable system where, in the future, any new text that will be added can simply be inserted in the 2 already existing localization files, while any new language will just require a new txt containing the localization of all the already existing 128 labels.

5.4 The codeQuestions

A codeQuestion represents a challenge that the players will need to solve, an admin that wants to include a new codeQuestion needs to make sure that it has created a "perfect" solution for the problem and then create a txt file following the series of rules detailed in this section.

When an admin creates a codeQuestion, it is mandatory that it contains:

- The label corresponding to the description of the question.
- The full solution of the question.

It is heavily recommended that it contains:

- At least one tag, to help the search of the codeQuestion.
- At least one hint, to help the players that are stuck.
- A test case with tag "[base]" containing at least one check for the function.
- A test case with tag "[final]" containing at least one check for the function.

It is optional that it contains, if it makes sense:

- The limits for the function's inputs.

Each component, with one exception, must be prefaced by the same pattern used in the wrapper in Section 5.1.2.2.

The list of possible labels are:

- `//_main_//`

- `//_question_//`
- `//_tags_//`
- `//_hints_//`
- `//_base_//`
- `//_final_//`
- `//_limits_//`

5.4.1 The mandatory labels

The localization label describing the codeQuestion must be in the first line of the file and must be **enclosed** in the previously mentioned pattern, not prefaced by it, for example:

A file beginning with:

```
//_helloWorld_description//           is a valid codeQuestion
```

A file beginning with:

```
//*anything*//_helloWorld_description   is NOT a valid codeQuestion
```

The `//_main_//` label must precede the function that will be tested. Note that the actual function that will be tested will be the wrapper corresponding to the function that was prefaced by this label.

The `//_question_//` label must precede the function that the players will need to write and doubt. Note that this label is only necessary when the tested function and the target function differ.

Additionally, when the `//_question_//` label is used, the users will be tasked to complete and doubt the function that takes the same arguments as the one after `//_main_//`.

For example:

```
//_main_//
int sum(int a, int b){
    return a + b;
};
```

Is a valid use of the `//_main_//` label only if this function is the target function for the users and the tests have been written for it accordingly.

In a more complex case:

```
int sum(bool returnEarly, int a, int b);

//_main_//
int sum(int a, int b){
    bool returnEarly = false;
    if(b > a){ returnEarly = true;}
    return sum(returnEarly, a, b);
};

//_question_//
int sum(bool returnEarly, int a, int b){
    if(returnEarly){ return -1;}
    return a + b;
};
```

The players will be asked to complete a function called `sum` taking in input 2 ints (the signature of the `//_main_//` function), however, the user solution will fill the secondary function under the `//_question_//` label.

The tests will be written referring to the `//_main_//` function and the players will be instructed to use assume the existence of a boolean called "returnEarly" that has been declared in the global scope.

Using this trick, it is possible to write arbitrary code and then task the players to solve only a small subset of it.

Note how, the first line in the complex example above, containing the secondary function's signature, sits above the `//_main_//` label, that is the only unlabeled space in a codeQuestion, whatever is written after the localization label is added to the cpp files of the users, but is part of the codeQuestion setup.

For example, a setup might include the declaration of a struct, that the users can assume exists, or the importing of a library necessary in the `//_main_//` function:

```
struct Home {
    int residents;
    int age;
};

//_main_//
bool fun(int capacity){
```

```

    Home one = new Home();
    one->residents = 12;
    one->age = 2;

    Home two = new Home();
    one->residents = 8;
    one->age = 8;

    Home street[2] = {one, two};

    return fun(street, 2, capacity);
};

//_question_//
bool fun(Home* h, int len, int capacity){
    while(len > 0){
        len--;
        if(h[len]->residents / h[len]->age > 2){ return true;}
    }
    return false;
};

```

Here the players must be told that they can use a struct called Home containing one field called residents and one called age, additionally an already existing array of Homes called street needs to be accessed, and the length of said array is stored in the global variable called len.

The obvious drawback is that more complex code would give the users a big load of information to keep track of and the admin creating the codeQuestion would need to remember to add all the necessary information to the localization strings.

5.4.2 The recommended labels

A codeQuestion missing any of these labels will lose some of its features outside of the user solution and doubt creation, but otherwise would not disrupt a normal session execution.

The `//_tags_//` label must precede a list of comma separated tags, it is suggested to write a small amount of short tags, to aid visualization during the codeQuestion selection, and to avoid using whitespace.

The `//_hints_//` label must precede a list of comma separated localization labels, it is

suggested to write between 1 and 3 hints to help struggling players.

The `//_base_//` label must precede a test case containing the base tests for this codeQuestion, it is suggested to include tests of some obvious example executions without including any of the edge cases. Note that the base tests are run only at each user's discretion.

The `//_final_//` label must precede a test case containing the final tests for this codeQuestion, it is suggested to include all possible edge cases and without repeating the tests in the "[base]" test case. Note that if the final test is missing, the leaderboard after the doubting "round" will be exactly the same as the final leaderboard.

All tests in the codeQuestion should test a function that takes as a first argument `TIMEOUT`, so that they would be executed on the overload that can be terminated after `TIMEOUT` amount of seconds.

5.4.3 The limits label

To make sure that players would not be able to crash other user's solutions by giving inputs that are unexpected or even told to ignore by the admin, we decided to implement a limits system.

An admin can limit the inputs that can be given to a function using the following syntax:

```
variableName0, variableName1, ...:: leftDelimiter value0, value1, ... rightDelimiter
```

Where, the list of variableNames contains at least one variable and all of them must use the same identifier of one of the arguments in the function prefaced by the `//_main_//` label.

The left and right delimiters can be respectively:

- Open and closed parentheses `'(' '')`, representing a non inclusive interval limit.
- Open and closed square brackets `'[' '']`, representing an inclusive interval limit.
- Open and closed braces `'{' '}'`, representing a set of allowed values.

Following math notation, interval limits represent a minimum and maximum value that the variables can have, either including or excluding the limits, while the set notation only allows the variables to assume the values explicitly mentioned.

The values must be exactly 2 if the delimiters represent an interval, with the left one being strictly smaller than the right one, and at least one, with no upper limit, if the delimiters represent a set.

The values can only be integers in an interval but can be any value of the correct type when inside a set.

In case the variables are strings, an interval limit will be applied to its length.

In case it is necessary to have an interval be unbounded on one of the sides, a colon ':' can be inserted and the interval will be considered as one sided.

For example, all the following limits on the left are valid, considering variables starting with 's' as string, 'f' as floats, 'c' as char and 'i' as int, and on the right is displayed their meathematical representation:

In the case of booleans and char intervals, the limits are ignored.

5.5 The cosmetics

Cosmetics, and specifically Avatars, can be considered one of the major *Gamification* elements in DubitaC.

Each player is nudged towards playing a new session by the possibility of using a new Avatar that they had not used before, or have just obtained by gaining points in a previous session.

The system handling the loading of the cosmetics takes advantage of a special Unity asset called a SpriteAtlas, which is able to load, compress, store and reference all the sprites that are contained in a selected folder.

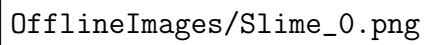
By using a SpriteAtlas, we reduce the amount of space taken up by each sprite, additionally it is possible to query it at runtime to obtain the sprites directly by name and the whole system is easily extendable by just adding the new sprites in the folder that the SpriteAtlas is monitoring.

Unfortunately the SpriteAtlas cannot return the names of all the sprites at its disposal, for this reason we had to create a text file containing the names of all the Avatar sprite that we have used, so that it would be possible to show the users all the possible Avatars that are in the game, while the ones that are unavailable are obscured making the reward of understanding what the Avatar represents more interesting.

Each Avatar sprite is a 64x16 png image that can be split into 4 different 16x16 "states":

- A standard Avatar that is shown throughout the game.
- An obscured version of the Avatar that is only shown if the player has not enough points to select it.

- An "angry" state that is only shown when the player does not reach the top 3 at the end of a game session.
- An "happy" state that is only shown when the player reaches the top 3 at the end of a game session.

A rectangular box containing the text "OfflineImages/Slime_0.png", which serves as a placeholder for the Slime Avatar's sprite.

OfflineImages/Slime_0.png

Figure 5.2: The "Slime" Avatar's sprite that can be selected during the game. The Avatar is repeated 4 times, once for each of the possible states: normal, locked, angry and happy.

Chapter 6

Unique challenges

In this chapter we are going to describe the biggest challenges that we had to face that are not directly tied to the implementation described in Chapter 5.

Some of the following sections will detail problems that we did not expect to have to solve during the development of DubitaC, while others are necessary steps that we had to take so that, in the end, we would have developed a smoother final product.

6.1 Bad Serialization

Netcode for GameObjects [Tec20], is the only option that game developers have if they do not want to rely on third party packages during their development of an online game using Unity.

However, the package is still in development and it is not unreasonable for the end user to be able to find bugs and unexpected behaviours.

Specifically, during the development of DubitaC we had to implement some low level code, that the package could call, to fix some shortcomings of their native Serialization system.

6.1.1 Netcode Serialization

Serialization is the process of transforming any data in a sequence of bytes, this is mainly useful during storage and transport of data.

During RPC calls, the parameters given to the function need to be serialized so that they can be properly sent over the network.

However to be able to read serialized data, it is needed to know in which order the elements of the original data have being written, so that a corresponding reader might retrieve them and recombine them correctly.

In our case, the 2 main functions that Netcode for GameObjects uses are:

```
public static void ReadValueSafe(this FastBufferReader reader, out *TYPE* value);  
public static void WriteValueSafe(this FastBufferWriter writer, in *TYPE* value);
```

We do not have to concern ourselves with the FastBufferReader or FastBufferWriter classes, what we want to highlight is how it is possible to extend said classes by overloading these 2 methods and changing the *TYPE* so that the serialization of said type would be possible.

For unknown reasons, but one can speculate that this is a temporary solution to reduce development time in this area, the package only offers ReadValueSafe and WriteValueSafe functions for:

- All single item value types, like ints but also FixedString32Bytes.
- Arrays containing only value types.
- Structs containing only value types.
- Strings, even if they are a reference type.

Most notably, some very common datatypes in Unity development are missing:

- Any collection type excluding arrays, from List to Custom collections.
- Structs containing a mix of value and reference types.
- Generic GameObject references.

While all of these limitations can be circumvented in some way:

- Send each element in the collection by itself.
- Coerce, when possible, the reference types into their corresponding value types, like string into FixedString32Bytes.
- Manage GameObject instantiation so that client and server can obtain the reference locally without having to send anything over the network.

It would still be a quality of life improvement, for developers, if the package could handle all datatypes natively.

In particular, collection types are ubiquitous in game development and sending each element by itself could prove taxing on the bandwidth of the clients' connections.

6.1.2 Serializing collections

Even by carefully selecting the data that we wanted to send over the network, we could not avoid the use of string arrays, and such has to write an extension method for it.

```
public static void WriteValueSafe(this FastBufferWriter writer, in string[] value) {  
    writer.WriteValueSafe(value.Length);  
  
    foreach (var item in value) {  
        writer.WriteValueSafe(item);  
    }  
}
```

The write function follows a simple approach, every collection needs to declare its length, usually as the first element, so that a reader can prepare a correctly sized collection as soon as possible.

After having written the length of the array, we loop through its elements and, since strings are part of the types that are serializable natively, call the overload of the WriteValueSafe method, taking a string, for each element.

```
public static void ReadValueSafe(this FastBufferReader reader, out string[] value) {  
    reader.ReadValueSafe(out int length);  
    value = new string[length];  
  
    for (var i = 0; i < length; ++i) {  
        reader.ReadValueSafe(out value[i]);  
    }  
}
```

The read function follows the same logic as the WriteSafeValue method we just wrote, the first thing we are going to read is the length of the array, then we create the array that we will output using the length we just obtained, then for as many elements as the length value, we read an element, that is a string so it is possible to use the ReadValueSafe overload, and then save it in the correct cell inside the array.

6.1.3 One step further

Before settling on only sending string and int arrays, we used to send string and int Lists.

For that purpose we had created one overload each for `WriteValueSafe` and `ReadValueSafe` with code that is almost identical to the previous one:

```
public static void WriteValueSafe(this FastBufferWriter writer, in List<int> value){
    writer.WriteValueSafe(value.Count);

    foreach (var item in value){
        writer.WriteValueSafe(item);
    }
}

public static void ReadValueSafe(this FastBufferReader reader, out List<int> value){
    reader.ReadValueSafe(out int length);
    value = new List<int>();

    for(var i = 0; i < length; ++i){
        reader.ReadValueSafe(out int val);
        value.Add(val);
    }
}

public static void WriteValueSafe(this FastBufferWriter writer, in List<string> value){
    writer.WriteValueSafe(value.Count);

    foreach (var item in value){
        writer.WriteValueSafe(item);
    }
}

public static void ReadValueSafe(this FastBufferReader reader, out List<string> value){
    reader.ReadValueSafe(out int length);
    value = new List<string>();

    for(var i = 0; i < length; ++i){
        reader.ReadValueSafe(out string val);
        value.Add(val);
    }
}
```

Similarly to before, we take advantage of the single element's overload methods and create

the Lists by adding said elements to it.

However, at the moment of testing, said implementation would crash with a "Bad Serialization" error. After a long debugging session, we were able to narrow down the problem to a bug in the package.

C#, and Unity using it, make use of a powerful tool to reduce the amount of development time and lines of code needed to support different datatypes, making it possible to call a function over some data of which the type is not known until checked.

This "check" is called *Reflection* and can be usually relied on in regards of type deduction, however for custom datatypes, classes and methods, it is not always possible to use native *Reflection*, the developer needs to nudge the machine in the right direction instead.

And the problem lies here:

Because the implementation to nudge *Reflection* developed by the Netcode for GameObjects developers has a mistake in it, when serialization is invoked on a generic List, all the Lists are treated as containing the same datatype, creating the error.

To be sure that such claim is not a developing blunder on our end, we tested our theory with different experiments:

- Trying to serialize an int List.
- Trying to serialize a string List.
- Trying to serialize either List when their corresponding overload is not declared.
- Trying to serialize either List when both overloads are declared.

The results that we obtained matched exactly what we expected:

- The int List is serialized perfectly, meaning that our overloads work correctly.
- The string List is serialized perfectly, meaning that our overloads work correctly.
- The serialization is not attempted, as the package notices that it does not know how to serialize a datatype with that signature.
- The serialization fails with a "Bad Serialization" error because Unity attempted to serialize the string List as an int List.

Instead of delving deep into the source code, we simply switched the Lists to arrays and that was enough for us.

However, we opened a Github Issue for it on the Netcode for GameObjects Repository here: <https://github.com/Unity-Technologies/com.unity.netcode.gameobjects/issues/1582>

At the time of writing, the Issue has neither been tackled nor resolved yet.

6.2 Shrinking the data

While Netcode for GameObjects is very efficient in sending the data required for synchronization between all connected machines, game developers can use RPC to send data of any arbitrary size over the network, this means that a developer that makes many implementation choices prioritizing ease of development instead of data reduction, might incur in problems both in sending the data and transporting the data over the network.

Netcode for GameObjects limits the maximum packet size at 5120 bytes, corresponding to just 5 kiloBytes of data, after having included the necessary header.

6.2.1 The biggest data packet

To start an analysis on the size of the game's sent packets, it is necessary to take a look at the largest single piece of data that we used.

The biggest struct that DubitaC uses is the struct saving a user doubt:

```
public struct doubt {
    public STATUS currentStatus;           byte sized enum = 1 byte +
    public ulong clientId;                 8 bytes +
    public ulong targetId;                 8 bytes +
    public FixedString32Bytes input;       32 bytes +
    public FixedString32Bytes output;      32 bytes +
    public FixedString32Bytes expected;    32 bytes +
    public DOUBTTYPE doubtType;           byte sized enum = 1 byte +
    public FixedString128Bytes clientDoubt; 128 bytes +
    public FixedString128Bytes serverDoubt; 128 bytes =
}                                           370 bytes
```

While 370 bytes are almost negligible, when the server shares all the doubts of all clients in a lobby, an array of doubt structs with a length between 2 and 30 is sent.

This means that the data that should fit in the RPC packet will potentially have a size of:

$$(370 * 30) + 4 = 11104 \text{ bytes} \sim 11 \text{ kiloBytes}$$

The 4 additional bytes represent the integer that would store the length of the array.

This is already **more than double** the size of the allowed packet.

Additionally, the same RPC sends 2 arrays of strings of the same length, since strings are reference types and can be unbounded in size, it is not possible to make any precise statement on the final size of the RPC packet.

6.2.2 Minimizing the strings' impact

While it is true that string are theoretically unbounded, in practice, during development, the possible content of a string is known, and this information can be used as bounding estimate.

As an example, in DubitaC we send over the network the following strings:

- The name of the sprite that the user has chosen.
- The name, description label, tags and content of the selected codeQuestion.
- The user solution.
- The users inputs, expected wrong output and expected correct output of a doubt.
- The result of the execution of a user solution.
- The results of the tested functions against each doubt for server and client.

Of these, some are easily bounded by a mindful admin:

- The sprite name is chosen by an admin, we recommend under 20 characters to aid visualization.
- The codeQuestion components are written by an admin, we recommend under 20 kiloBytes for the content and under 20 characters for the other parts.
- The user inputs can be bounded by an admin using the `//_limits_//` label in the codeQuestion.

The others are in the hands of the players, and, if they intend to be malicious actors, there is nothing stopping them to act in such a way that creates huge strings and, unfortunately, not much can be implemented short of truncating them, since such a string could represent a good faith attempt to solve the codeQuestion, said approach might prove disadvantageous.

6.2.3 Avoiding strings where possible

Because structs cannot contain reference types, See Section 6.1.1, any string that can be converted in a value type should be.

The family of `FixedStringBytes` is able to exactly represent any string, provided that they fit in a predefined length.

In the `doubt` struct, see Section 6.2.1, we are using `FixedString32Bytes`, to store the user inserted strings during the `doubt` creation, and `FixedString128Bytes` to store the text representing the test cases, one for the server and one for the client.

These conversions are possible as long as the admin has been mindful in naming the function inside the `codeQuestion` and limited the length of the inputs, since we calculated that the creation of any test case will fit in 128 bytes as long as the function names and maximum user inputs are shorter than 25 bytes.

6.2.4 Fitting the data inside a packet

We have shown that the data that might be sent by `DubitaC` can be as big as 11 kiloBytes and that `Netcode` for `GameObjects` would limit the data sent on a single packet to 5 kiloBytes.

We have thought about splitting the data over more packets, but `Netcode` for `GameObject` was already taking care of the transport layer for us, making the prospect of splitting manually the data, adding a UDP header and an ordering index, sending the packets, recombining the original data and implementing redundancy measures, rather unappealing.

We have also thought of removing the transport of any collection, sending each element by itself and then recreating the collection on the receiving end but, like we have already mentioned in Section 6.1.1, this would be taxing on the bandwidth and waiting times would increase.

Instead, the possibility of extending the packet limit was added to the package while we were developing `DubitaC`, although not recommended, the corresponding UDP packet sent by a RPC could now hold more than 44 kiloBytes, if enabled to do so.

Thanks to that change, we would have no problem in sending our data anymore, provided that user created strings do not exceed 30 kiloBytes.

6.3 Speeding up Catch2

At the beginning of DubitaC development, the full cycle of creating a solution, compiling it and executing it would take up to 3 minutes. Since we wanted to create a videogame around such cycle, this amount of waiting time for the players would be obviously unacceptable, as such, we had to find a solution to make sure that a complete cycle would not take more than 30 seconds.

6.3.1 Profiling the cycle

The first step of optimization is always to look where a process takes the longest and check if it is possible to improve performance in these sensitive areas first.

Unsurprisingly, compilation takes around 99.99% of the time of the cycle, what is a bit more surprising is that the long compilation is caused by a slow "linking" step.

With the help of the Catch2 documentation, we were able to pinpoint 2 causes:

- We were reinitializing the Catch2 environment at every compilation.
- We were using the standard g++ linker.

6.3.2 Compiling the test environment only once

One major drawback of using Catch2 is the need to compile a "main" Catch2 file that would initialize the test framework.

In Section 5.1.2, we mentioned adding the main directive at the top of the user solution, while this is not incorrect, it would mean that the environment is reinitialized at every compilation, increasing significantly the compilation time.

What we do instead, is creating a simple auxiliary cpp, that will contain all the necessary directives that we mentioned before:

```
#define CATCH_CONFIG_MAIN
#define CATCH_CONFIG_FAST_COMPILE
#define CATCH_CONFIG_WINDOWS_SEH
#include "catch.hpp"
```

The "fast compile" directive, that was not mentioned previously, also contribute to a, admittedly small, speedup during compilation.

We then compile this cpp, only once, using the following terminal command:

```
g++ catch_main.cpp -c
```

Where the `-c` flag is used to create a linkable file with the `".o"` extension instead of an executable file with an `".exe"` extension (on Windows).

We can then compile our solutions, as many times as we want, alongside this linkable file to obtain a considerable speedup during compilation, from around 3 minutes to a bit more than 2 minutes.

6.3.3 Changing the linker

Because Catch2 creates a great number of symbols, any compiler that is slow in the symbol linking step will not be able to compile a Catch2 file in a reasonable amount of time.

The standard g++ compiler comes with the possibility of using 3 different linkers:

- The standard linker.
- The `"bfd"` linker.
- The `"gold"` linker.

Unfortunately, all of these linkers take, more or less, the same amount of time to compile a Catch2 file.

The Catch2 community often suggests to use the LLVM project [LA04] linker called `"lld"` since, differently from the others, it is very efficient in the symbol linking step.

After having downloaded the `"lld"` linker and compiled our `solution.cpp` with:

```
g++ -O3 -fuse-ld=lld catch_main.o solution.cpp
```

The total compilation time decreased from around 2 minutes to around 20 seconds.

6.4 Speeding up DubitaC

When we decided to limit the maximum amount of players that can be connected at once, we wanted to find a compromise between the number of students that could be in an "Introduction to programming" class and the amount of resources that the game would require to run smoothly.

6.4.1 Size considerations

From our own experience, we considered the average first year bachelor in computer science class to be at around 100 students, often split into 2 or 3 rooms during laboratory practice.

Although 100 connections are unattainable without a dedicated server or little to no data being shares over the network, we reasoned that the real bottleneck is the number of machines at the laboratories' disposal, which, in the University of Genoa, would amount to around 25 machines each.

Furthermore, it is not unusual to promote the creation of groups between 2 or more students, as the situation demands.

Since DubitaC is more akin to a puzzle game than a more fast paced game, the cooperation of more than 1 student to represent a single client would have no disadvantage for the integrity of the game.

One could argue that the collaboration of more students to produce a single solution and analyze and doubt other solutions would be an advantage, as the need to agree between groupmates should lead to a more careful explanation and better understanding of the problem, and solutions, at hand.

While in the early stages of development we thought it could be possible to have 24 concurrent players, we later realized that our system for doubt creation would create too much data for the task.

Equation 6.1 shows the formula to calculate the number and size of the doubts created with N number of players:

$$\begin{aligned} |D| &= N * (N - 1) \\ \text{size}(D) * |D| &= 370 \text{ bytes} * (N^2 - N) \end{aligned} \tag{6.1}$$

Since the number of doubt follows a quadratic function, the bytes needed to share them over the network would grow unmanageably.

6.4.2 Time considerations

We also wanted to consider the length of each game "round" of a game session for 2 reasons regarding the Flow state of the players:

- Long waiting times are detrimental to the user engagement.
- Repetitive game loops, that are not short and rewarding, need to be sufficiently different from one another.

While the first one is evident, waiting times are devoid of interaction, the second point, referring to the doubt creation part, is more nuanced.

Since each user can create a doubt for every other player, this smaller game loop consists of the sequence:

Select user → Understand user solution → Identify mistakes → Create doubt

With the whole cycle taking anywhere between 30 seconds to some minutes and the players receives no positive feedback at any part of the sequence, it is necessary to make sure that performing the loop more than once is different enough to maintain the user engagement.

Although we have no control over the similarity between the player solutions, what we can control is the maximum amount of times this loop needs to be repeated so that, in case the solutions are similar or easy to understand, the whole process would not take too long.

6.4.3 Reducing the number of doubts

Taking the Equation 6.1 as an example with $N = 24$, the number of doubts would be 552 with a total size of 204240 bytes \sim 200 kiloBytes.

This was obviously unacceptable but we also did not want to reduce the number of possible players, we decided to take a simple but effective approach. While the server would still need to manage 24 players, they would all be split into 4 groups, or lobbies, making the number of doubts more manageable and reducing the repetition of the doubting game loop, achieving both of our goals.

Equation 6.2 shows the improved formula for calculating the amount of doubts:

$$|D| = \sum_{i=0}^3 n_i * (n_i - 1) \quad \text{for } 2 \leq n_i \leq 6 \quad (6.2)$$

Where n_i represents the number of players that have been assigned to the i th lobby.

Using this technique of lobby subdivision, we could reduce the maximum amount of doubts to 30 for each lobby, for a total of 120 but, more importantly, we reduced the amount of solutions that a user might have to analyze and doubt from 23 to just 5.

6.4.4 Reducing the waiting time

Originally, the server would compile the user solutions on its own machine sequentially so that we could minimize the amount of strings that were passed over the network.

This approach would have the side effect of increasing the length of a game session significantly, even with the improvements of Section 6.3, the time required to compile 24 user solution + the server solution would be between 300 and 600 seconds.

It was obvious that having a waiting time between 5 and 10 minutes was unacceptable, however, by how we created the game infrastructure, the task of compiling and executing a solution is Embarassingly parallelizable, meaning that, if the inputs and outputs are shared correctly between clients and server, the actual execution can be offloaded to each client, reducing the total waiting time to between 12 and 24 seconds.

The total length of a game session t would now follow Equation 6.3:

$$\begin{aligned} T_2 &= T_1 + \max(n) * 0.15 \\ T_4 &= (\max(n)^2 - \max(n)) * 4 \\ t &= T_1 + T_2 + T_3 + T_4 \end{aligned} \tag{6.3}$$

Where T_1 is the admin selected time to create a solution, T_2 is the increased time given to the doubting "round", T_3 is the compilation and execution time of a single solution (~ 20 seconds) and T_4 is the time it takes to show all the relevant doubts to the users.

We used the $\max(n)$ instead of n_i because we wanted to synchronize the rounds between each lobby, so the smaller lobbies will experience a small additional waiting time, to note how, because of the lobby rebalancing of Figure ??, the maximum difference, in the number of clients, that 2 lobbies can have is always 1, adding at most the time that it takes to compile one solution more and the time it takes to display $n_i - 1$ doubts more.

TODO: [image of lobby rebalancing needed]

6.4.5 Performance gain

It is important to remember that the calculation of the total time also depends on 4 parameters that could be changed to fit one's needs:

- L , the maximum amount of lobbies, currently set at 4.
- P , the maximum amount of user per lobby, currently set at 6.
- α , the percentage increase of the time available depending on the number of clients in the lobby, currently set at 15%.
- W , the number of seconds given so that the players can read each doubt in the final slideshow, currently set at 4 seconds for each doubt.

With these considerations, Equations 6.2 and 6.3 can be parametrized, as shown in Equation 6.4.

$$\begin{aligned}
|D| &= \sum_{i=0}^L n_i * (n_i - 1) \quad \text{for } 2 \leq n_i \leq P \\
T_2 &= T_1 + \max(n) * \alpha \\
T_4 &= (\max(n)^2 - \max(n)) * W \\
t &= T_1 + T_2 + T_3 + T_4
\end{aligned} \tag{6.4}$$

In conclusion, thanks to the efforts detailed in Sections 6.3 and 6.4, we were able to reduce waiting times, total execution time and the size of the data to be created and sent over the network to a more appropriate level.

6.5 Speeding up Unity

This section details the efforts of keeping the Unity compilation times manageable.

Unity compilation system works as follow:

- If a change has been detected in a script, search for user defined assemblies.
- If no user defined assemblies are set, all scripts will be recompiled.
- If the relative user defined assemblies are found, recompile the folders that are associated with the modified scripts.
- If the recompiled folders are referenced by other assemblies, their corresponding folders are recompiled aswell.
- Repeat the last step until every script with a dependency on the modified scripts has been recompiled.

For small projects, there is usually no need for user assemblies, as the total compilation would not usually exceed 10 seconds.

In our case, we noticed a substantial increase in compilation times (up to 2 minutes and a half) as the development of DubitaC proceeded and especially as more systems, that would interact with eachother, were getting implemented.

In the interest of reducing the compilation times both for the development of a first release and for future changes related to maintainance or extensions of DubitaC, we decided to create some user defined assemblies.[dt19]

6.5.1 Evaluating the extremes

The final release of DubitaC contains a total of 28 scripts, this means that with no user defined assemblies, 28 scripts (all of them) will be recompiled on average for every single script that gets modified.

On the other extreme, if every single script was in a folder of its own, with a user defined assembly associated with it, the average number of compiled script would not be 1, since all dependencies would need to be compiled aswell.

This means that creating a dependency graph will help us in determining a subdivision in folders that would reduce the average amount of recompilation needed, while at the same time respecting the following 2 rules:

- Scripts that depend on eachother cyclically, cannot be split into different folders.
- Folders should have a minimal logic that is shared with all the contained scripts, so that the subdivision could be considered "logical".

6.5.2 Dependency graph

Figure 6.1 shows the dependency graph of our project, it is easy to see that the structure is very complex and that, as it stands, there are cyclic dependencies, meaning that each script cannot belong to its own folder.

Notice how the graph, with 28 vertices and 68 edges, contains a small number of nodes that have a higher degree than average (~ 5), especially the triple "Cosmetics", "DataManager" and "Required Structs" are the core static scripts from which every other script can access data that should persist between scenes.

Another outlier is the "SlideshowManager" script that, since it needs a lot of data to manage the final slideshow but does not create any concrete information, has an outdegree of 10 but an indegree of 0, it is the script that depends on the most scripts and at the same time is also one of the scripts with no dependencies.

6.5.3 Dividing roughly across scenes

Some articles or guides online suggest to organize your Unity project scripts based on which scenes they are used on, in our case we have 5 scenes, with one of them (the last one) being a subset of another (the first one) in regards of scripts used.

Additionally, many scripts are shared across all scenes and 2 scenes are particularly close in regards to script used.

This meant that, our first attempt at subdividing the scripts, created 4 user defined assemblies alongside that many folders.

Figure 6.2 show the subdivision and condensation of this first subdivision.

Assuming that any script has the same probability of being modified, although this might be an erroneous assumption, with this new subdivision the average number of scripts that get recompiled is 17.25, only 61.6% of the original.

The compilation times that we were experiencing were still bothersome at around 1 minute and a half.

6.5.4 Final subdivision

Not satisfied with the result, we decided to divide the scripts with a more "purpose" oriented approach, this time we identified 16 groups depending of their scripts' function.

Figure 6.3 show the subdivision and condensation of the final subdivision.

This final subdivision that we deemed acceptable recompiles on average only 9 scripts, 32.1% of the original and 52.2% of the previous subdivision.

In conclusion, we were able to reduce the compile times that we were experiencing from around 150 seconds to at most 20 seconds, a final summary of the results can be found in Table 6.1.

	#User defined assemblies	Average compilation time	Comparison with first row	Average #scripts compiled	Comparison with first row
Original	0	~150s	100%	28	100%
First subdivision	4	~90s	60%	17.25	61.6%
Final subdivision	16	~20s	13.3%	9	32.1%

Table 6.1: Comparison between the compilation time, average scripts recompiled assuming a uniform distribution of modifications along the project's scripts and improvements obtained with the 3 different script divisions. Notice how the improvements on the average scripts recompiled is smaller than the improvement on the average execution, this is probably because the compilation times that we observed do not follow the assumption that all scripts are edited equally often.

6.6 Increasing maintainability

Since there is no single way to calculate the real world maintainability of a project, we had to decide which metric to use if we wanted create code that will one day be maintained, easily, by someone else.

During our development we used Visual Studio as our editor and one of the features that it offers is a calculator of code metrics, in particular:

- Cyclomatic Complexity, representing the number of possible branches of execution that can taken when running the script.
- Depth of Inheritance, representing the deepest chain of inheritances used in the script.
- Class Coupling, representing the number of classes that are referenced in the script.
- Lines of Source Code, the number of lines of the script.
- Lines of Executable Code, an approximation of the number of executable lines of code in the script.
- Maintainability Index, a final general score, considering all the above, between 0 and 100.

From the documentation [con21], The maintainability index is grouped in 3 categories:

- Above 20, the code is considered with a good maintainability.
- Between 10 and 20, the code is considered with a bad maintainability.
- Below 10, the code is considered problematic and in urgent need of refactoring.

As development progressed, we tried to keep the maintainability index high thanks to constant refactoring efforts.

TODO: [image of the code metrics needed] The code metrics calculated before the final release can be seen in Figure 6.4.

For each folder containing scripts, the ones detailed in SectionFinal subdivision, and each script itself, the results were promising, with the lowest maintainability index being 51 in the DoubtManager class.

It is also possible to go 1 step more in depth and analyze the code metrics of each function inside the classes, this time, however, the DoubtManager class contained a troubling method, highlighted in Figure 6.5.

We decided to start from there to refactor one last time the more problematic classes before release.

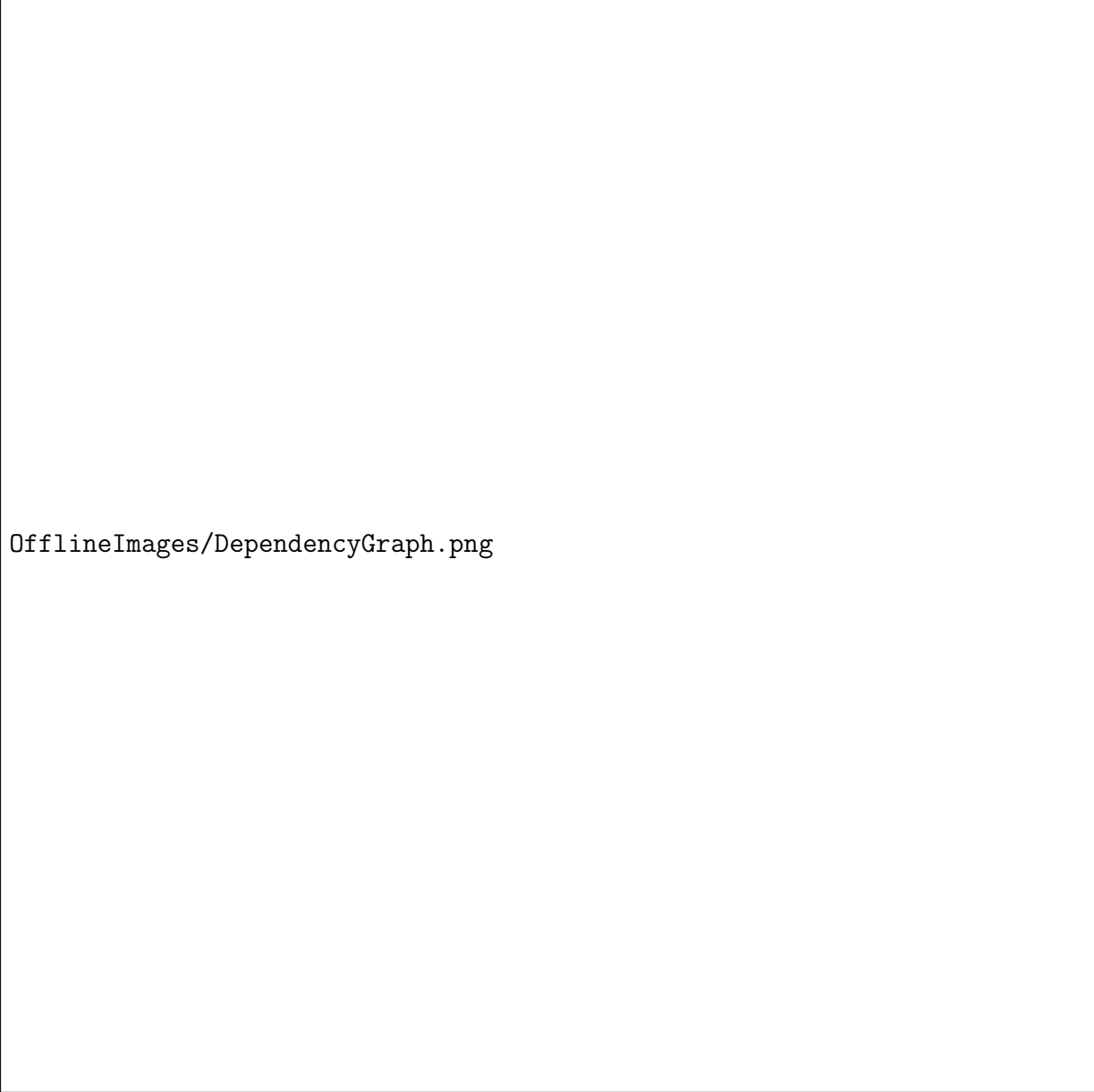
It is important to point out that, no matter how sophisticated, a tool to calculate maintainability will never match up to the understanding of a future maintainer and, as such, it is not possible to say definitively that our code is "highly maintainable" just because it falls in the first category for Visual Studio's Maintainability Index calculations.

Furthermore, many valid objections have been raised regarding the validity of the original index on which the Visual Studio implementation is based on [OH92], particularly regarding the strong correlation between the number of lines of code and a lower maintainability score.[RMT09][SAM12]

We ended our refactoring with the code metrics shown in Figure 6.6, with some small but consistent improvements, but decided not to refactor further as one might be tempted to edit the code in such a way that it would obtain the desired maintainability index, possibly ignoring that the code might actually become less readable.

"When a measure becomes a target, it ceases to be a good measure."

– Goodhart's law[Str97]



OfflineImages/DependencyGraph.png


Figure 6.1: Dependency graph of the 28 scripts in DubitaC. The structure is very complex but the subdivision to reduce the compilation time was necessary.



Figure 6.2: Left: the original dependency graph colored based on the 4 chosen groups. Right: New dependency graph after having condensed the scripts into overarching groups.

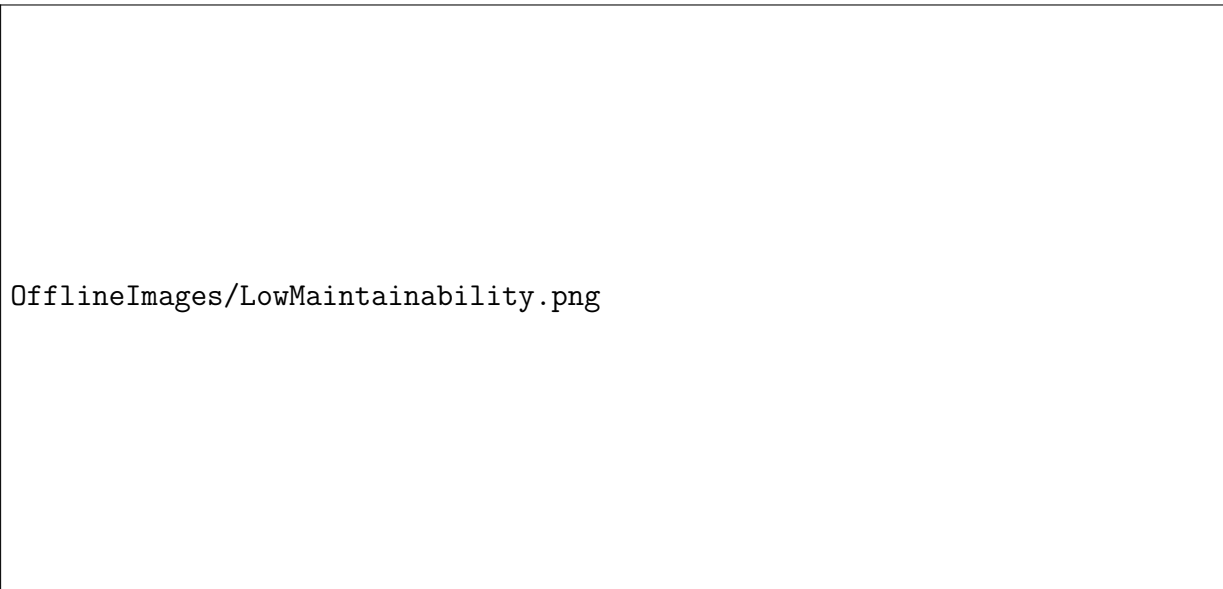


Figure 6.3: Left: the original dependency graph colored based on the 16 chosen groups. Right: New dependency graph after having condensed the scripts into overarching groups.



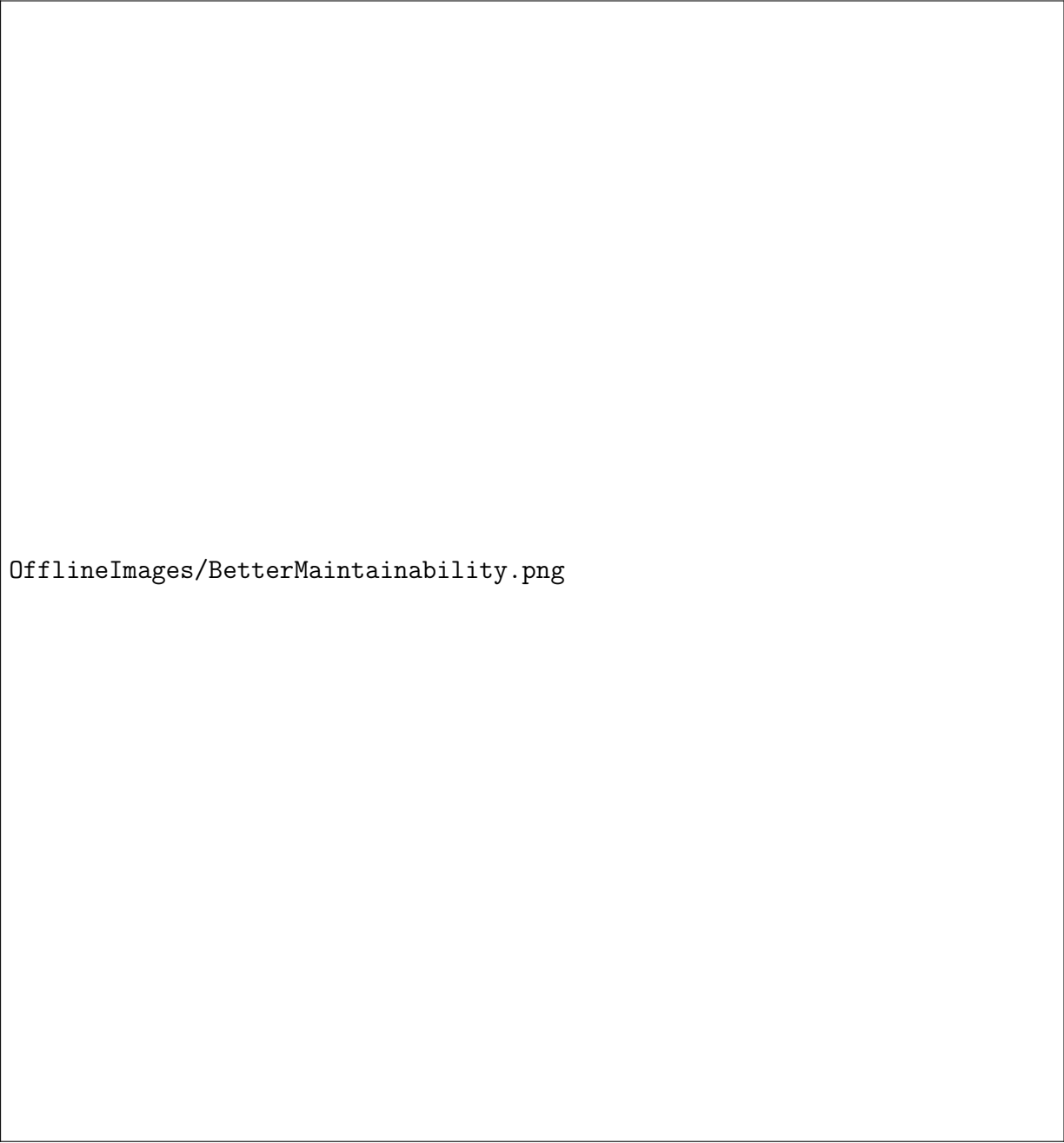
OfflineImages/GeneralCodeMetrics.png

Figure 6.4: Overall evaluation of all the scripts in DubitaC, the green square next to each of them refers to a "high" maintainability following Visual Studio's code metrics.



OfflineImages/LowMaintainability.png

Figure 6.5: A more specific evaluation of the "DoubtManager" class, showing the maintainability score of each declared method. A single problematic method is highlighted by the selection in blue, with a yellow triangle instead of a green square indicating "problematic" maintainability. For visualization purposes, the declarations of non-method elements of the class have been replaced by a series of dots.



OfflineImages/BetterMaintainability.png

Figure 6.6: Final maintainability score at release. No class score is below 55, and the highlighted, previously problematic, method sits above 30. For visualization purposes, the declarations of non-method elements of the class have been replaced by a series of dots.

Chapter 7

Limitations and future work

In this chapter we will highlight the shortcomings of the project and, if possible, pinpoint the causes. In the last section we will also expose some ideas of possible DubitaC extensions.

7.1 Limitations

Because of a lack of resources, the whole project was tested on not more than 2 machines. As the expected number of players is around 20, it was not possible to check if any unexpected behaviours arise when increasing the number of distributed clients connected to the server.

Because of the necessity of using third party components to be able to integrate non-local multiplayer, DubitaC can only be played on a local network.

Because of the Catch2 requirements, the installation has to include a folder containing the Catch2 header and the LLVM linker "lld".

Because of the need of starting a terminal execution, DubitaC can only be played on a Windows machine, where the terminal can be invoked with cmd.exe, and the g++ compiler must be present in the environment variables.

Because of the build behaviour of the Unity game engine, every modification to DubitaC, even the ones regarding the implementation of additional assets without changing the source code, require Unity to be applied and rebuilt.

Because of the test that we conduct to understand if an execution is reasonably stuck in an infinite loop, the C++ libraries:

- chrono.h

- `thread.h`
- `mutex.h`
- `condition_variable.h`

are automatically included and the players, not being aware of it, might forget to insert the relevant `#include` lines but receive no error during compilation.

Because of the way `LocalizableText` works, we could not use it for the player log, since terminal output and parsing messages are mixed into a single textfield the log cannot be translated and the default english language is always used.

Because of the lack of any C++ tutorial, by design, DubitaC can be appreciated only by players that already have a basic grasp on the language.

7.2 Possible extensions

For many extensions the systems are already in place, for example:

- Extending the languages available by creating and integrating more localization files.
- Extending the list of available Avatars by creating and integrating more sprites.
- Extending the list of available codeQuestions by creating and integrating more text files with the correct codeQuestion syntax.

To improve the "user" experience for an admin that would implement some of these extensions, it would be possible to create Unity specific or third party tools to streamline the integration of new content without the need of recompiling and reinstalling the game.

Others extensions would require some knowledge of other programming languages, for example:

- Researching and implementing a test framework different from `Catch2`.
- Researching and implementing the possibility of compiling and executing source code of other languages, like java, from terminal.

Lastly, some extensions would improve the general experience of playing DubitaC like:

- Adding more polish, like music or feedback for the user.

- Making it portable on other platforms by introducing code that would recognize the current platform and execute code specific for it, for example having the bash started instead of cmd on non-Windows machines.
- Introduce some multithreading, even if the game has little room for parallelization currently.

Chapter 8

Conclusions

TODO: [insert github repository link here]

TODO: [write conclusions]

Bibliography

- [Bil21] Jasmine Bilham. Case study: How duolingo utilises gamification to increase user interest. Online article, Jul 2021. URL: <https://raw.studio/blog/how-duolingo-utilises-gamification/>.
- [Boh15] Kim Bohyun. *Technology Reports*, volume 51, chapter 1-5, pages 5–36. ALA TechSource, Mar 2015. URL: <https://journals.ala.org/index.php/ltr/article/view/5629>.
- [con21] Multiple contributors. Code metrics values. Microsoft documentation, Jun 2021. URL: <https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022>.
- [Cor18] Tomorrow Corporation. 7 billion humans. Videogame, 2018. URL: <https://tomorrowcorporation.com/7billionhumans>.
- [DBC⁺14] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *NDSS*, volume 14, pages 23–26, 2014. URL: <https://www.cs.umd.edu/class/spring2017/cmsc8180/papers/tangled-web.pdf>.
- [DD17] Christo Dichev and Darina Dicheva. Gamifying education: what is known, what is believed and what remains uncertain: a critical review. *International Journal of Educational Technology in Higher Education*, 14(1):9, Feb 2017. URL: <https://doi.org/10.1186/s41239-017-0042-5>.
- [DH12] Nathan Doctor and Jake Hoffner. Code wars. Online platform, 2012. URL: <https://www.codewars.com>.
- [dt19] Unity development team. Assembly definitions. Unity documentation, 2019. URL: <https://docs.unity3d.com/2019.1/Documentation/Manual/ScriptCompilationAssemblyDefinitionFiles.html>.
- [dt22] Codewars development team. Gamification. Code wars Documentation, 2022. URL: <https://docs.codewars.com/gamification>.

- [HFA04] David Helgason, Nicholas Francis, and Joachim Ante. Unity. Game Engine, 2004. URL: <https://unity.com/>.
- [Hoř15] Martin Hořeňovský. Catch2. Test framework, 2015. URL: <https://github.com/catchorg/Catch2>.
- [HYHS13] Wendy Hsin-Yuan Huang and Dilip Soman. A practitioner’s guide to gamification of education. *academia.edu*, Dec 2013. URL: https://www.academia.edu/33219783/A_Practitioners_Guide_To_Gamification_Of_Education.
- [KAY14] Gabriela Kiryakova, Nadezhda Angelova, and Lina Yordanova. Gamification in education. In *9th International Balkan Education and Science Conference*, Oct 2014. URL: https://www.researchgate.net/publication/320234774_GAMIFICATION_IN_EDUCATION.
- [LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, Mar 2004. URL: <https://llvm.org/>.
- [LH11] von Ahn Luis and Severin Hacker. Duolingo. Online platform, 2011. URL: <https://www.duolingo.com/>.
- [OH92] P. Oman and J. Hagemester. Metrics for assessing a software system’s maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, Nov 1992. URL: <https://ieeexplore.ieee.org/document/242525>.
- [RMT09] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. pages 367–377, Oct 2009. URL: https://www.researchgate.net/publication/221494987_A_systematic_review_of_software_maintainability_prediction_and_metrics#read.
- [SAM12] Dag I. K. Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: A comparative case study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 107–110, Sep 2012. URL: <https://ieeexplore.ieee.org/document/6475403>.
- [SRM⁺20] Rodrigo Smiderle, Sandro José Rigo, Leonardo B. Marques, Jorge Arthur Peçanha de Miranda Coelho, and Patricia A. Jaques. The impact of gamification on students’ learning, engagement and behavior based on their personality traits. *Smart Learning Environments*, 7(1):3, Jan 2020. URL: <https://slejournal.springeropen.com/articles/10.1186/s40561-019-0098-x>.

- [Str97] Marilyn Strathern. ‘improving ratings’: audit in the british university system. *European review*, 5(3):305–321, 1997. URL: <https://archive.org/details/ImprovingRatingsAuditInTheBritishUniversitySystem>.
- [T⁺36] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, Nov 1936. URL: <https://www.wolframscience.com/prizes/tm23/images/Turing.pdf>.
- [Tec20] Unity Technologies. Netcode for gameobjects. Unity package, 2020. URL: <https://github.com/Unity-Technologies/com.unity.netcode.gameobjects>.
- [Uni21] Università di Genova. Introduction to computer programming. University course description, 2021. URL: <https://unige.it/en/off.f/2021/ins/46801>.
- [Unk] Unknown. I doubt it. Card game. URL: <https://bicyclecards.com/how-to-play/i-doubt-it/>.
- [Zac15] Zachtronics. TIS-100. Videogame, 2015. URL: <https://www.zachtronics.com/tis-100/>.
- [Zac16] Zachtronics. SHENZHEN I/O. Videogame, 2016. URL: <https://www.zachtronics.com/shenzhen-io/>.
- [Zac17] Zachtronics. Opus magnum. Videogame, 2017. URL: <https://www.zachtronics.com/opus-magnum/>.