Università di Genova
DIBRIS DIPARTIMENTO
DI INFORMATICA, BIOINGEGNERIA,
ROBOTICA E INGEGNERIA DEI SISTEMI

DubitaC: a serious game to support undergraduate programming courses

Lorenzo Tibaldi

Master Thesis

Università di Genova, DIBRIS Via Opera Pia, 13 16145 Genova, Italy
https://www.dibris.unige.it/

Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

DubitaC: a serious game to support undergraduate programming courses

Lorenzo Tibaldi

Advisors: Maura Cerioli & Gianna Reggio

Examiner: Giovanna Guerrini

February, 2022

Table of Contents

Chapter 1

Introduction

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

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

To achieve this we had to research, select and implement various gamification elements for a serious game called DubitaC.

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 supporting professors during beginner programming courses using C++ .

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

Chapter 2

Serious gaming and gamification

This chapter will contain the starting background for the rest of the paper.

Section 2.1 will act as a glossary, where we will introduce some terms that will be used in later chapters.

In Section 2.2, we will present the results obtained by the research conducted on gamification.

In Section 2.3 we will highlight some successful examples of gamification being employed in different domains.

Lastly, in Section 2.4, we will present some 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.

2.1 Introduction of the terminology

Each entry is presented in alphabetical order.

2.1.1 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.

2.1.2 Game engine

A **game engine** is a programming software, created with the goal of helping developers in making videogames.

All **game engine**s lower the barrier of entry in the game development world by taking care of the fundamental parts of *starting* a game.

In particular:

- An automatic continuous execution loop.
- An automatic continuous rendering loop.
- An high level interface for allocation and deallocation of data.
- A tick based physics system.
- Basic UI support.
- Simple creation of game builds.
- Compilation and execution environment for testing before building.

Most of the popular **game engine**s available today offer more features that set them apart from one another.

2.1.3 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:

Analyse 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 actions, just the singular one of pressing said button. Similarly, no **game loop** is present in a game that, for example, awards the user some points every 5 minutes and the only objective is reaching a 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 arbitrarely simple.

Lastly, game sessions will often contain multiple **game loops** of different sizes, returning to the previous chess example, let us consider a chess tournament and identify at least 3 different **game loops**:

- During a match, deciding and making a move like 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 continously 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 partecipating in a tournament, that contains the previous game loops and its sequence of actions is playing many different matches.

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.

2.1.4 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.

For example, an app that tracks the amount of steps taken during day can increase user retention by setting daily goals and rewarding the user with badges every time they reach said goal.

2.1.5 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 loop**s 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 deafeating 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.

2.1.6 Serious games

Taken from [KAY14]:

"**Serious games** are games designed for a specific purpose related to training, not just for fun. They possess all game elements, they look like games, but their objective is to achieve something that is predetermined."

This does not mean that **serious games** are opposed to providing entertainment, it only suggests that entertainment was not a priority during the product's development.

2.2 Literature and state of the art

Even though there is no scientific consensus regarding 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], reaserchers were able to correlate, on their study group, an overall improvement in all categories for subjects that would be defined as introverted, shy or non confrontational, personality traits that 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.

2.3 Gamified products

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.

Regarding online learning platforms, Code wars[DH12] 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.

Another example is the very popular online learning platform Duolingo.[LH11]

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

We also want to remark how some of the previous examples were employing gamification, mainly for user retention, 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 voluntarely 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 by design, since Code wars does not aim to be an entertaining videogame.[dt22]

2.4 Programming games and their limitations

The game studio Zachtronics created interesting and engaging programming games:

- 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. [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, simulating a multi threaded environment. [Cor18]

Figures 2.1, 2.2, 2.3 and 2.4 show an example game screen for each of the mentioned games taken from their Steam pages.[1]

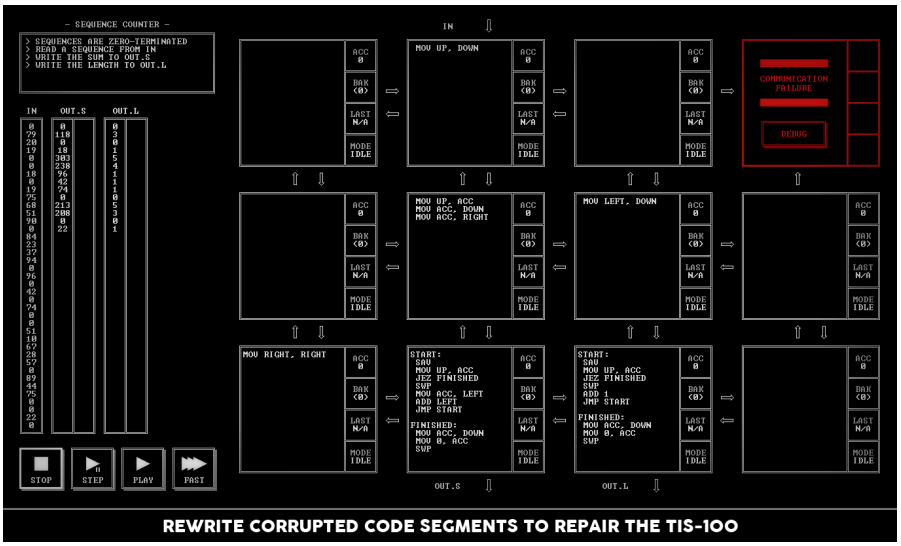


Figure 2.1: Example game screen of TIS-100. On the left from top to bottom, the player has access to: The current goal, inputs and outputs and a execution console, while in the rest of the interface, 12 cores are shown, each with their own space to assign assembly instructions.

Out of all the games mentioned in this section, 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 laguage in a game system.

While it is possible to argue that designing algorithms is language agnostic, programming is a problem solving skill that is heavily language dependant, meaning that such games are not directly improving programming skills in their controlled environment.

Conversely, we felt that giving students a concrete starting point using C++ would be more effective and make it easier to integrate DubitaC in an educational setting.

[1] `https://store.steampowered.com/`

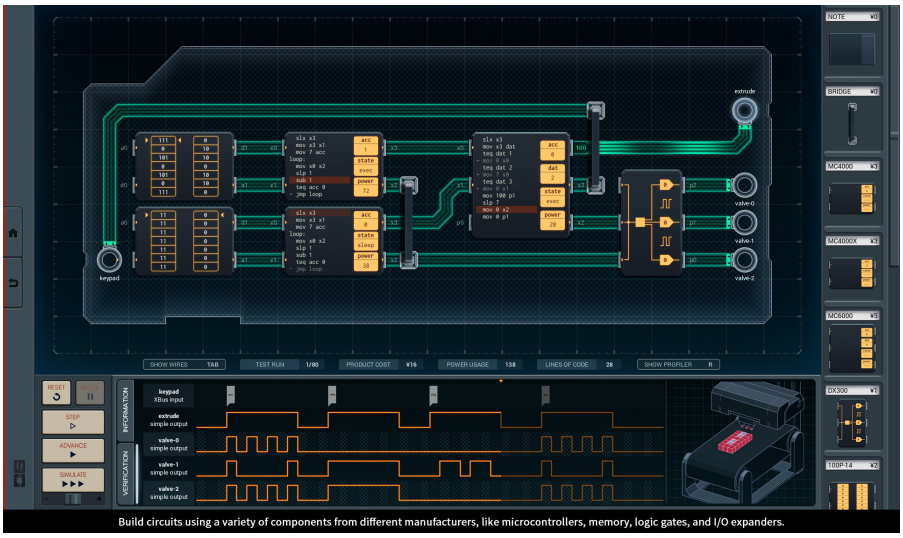


Figure 2.2: Example game screen of SHENZHEN I/O. On the top part, the main circuit is shown, where each component can be customized with pseudo-assembly code, on the right, the components that the player can buy and on the bottom, an execution console is shown alongside the circuitry inputs. The current goal can be found on a different separate panel.

Figure 2.3: Example game screen of Opus Magnum. On the left from top to bottom, the player has access to: The current goal "product", the input "reagents" at its disposal and the components that the player can buy, while in the center of the interface, the playing field is shown, where the player can place the bought components, lastly, on the bottom of the interface, a strip that can be filled with blocks of instructions for each component in the field is shown alongside the execution console. Opus Magnum is as much about programming the pistons movements, as it is about creating an efficient factory layout.

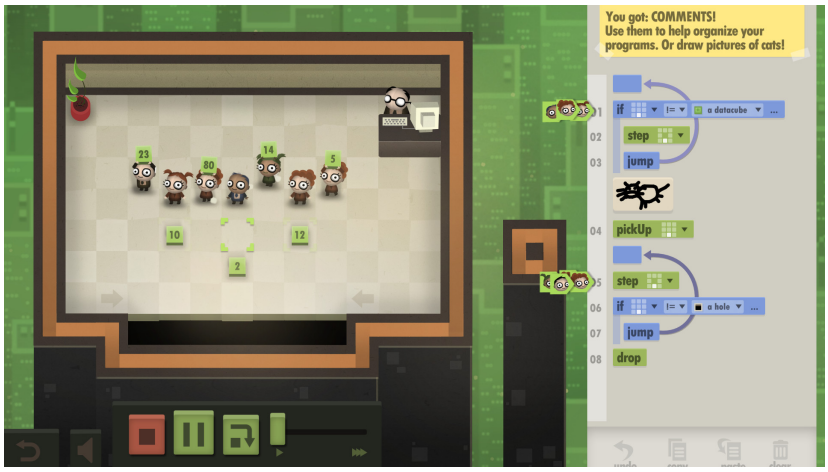


Figure 2.4: Example game screen of 7 Billion Humans. On the right, the player can insert basic coding blocks governing movement that will be applied to all "humans" in the level, while the rest of the interface shows the playing field, which the user cannot interact with directly, and an execution console. Almost all levels in 7 Billion Humans simulate multithreaded solutions, the one in the figure, for example, requires that all green datablocks are picked up and dropped in the hole at the bottom, with 7 "humans" following commands, the complexity of the solution can be reduced.

Chapter 3

Design of the game

In this chapter we will detail the design of DubitaC, from the initial requirements to the final game loop.

3.1 Initial requirements

The initial requirements for the game were:

1. Containing game loop that involved writing and understanding code.
2. Utilizing C++ as the starting language for the gameplay.
3. Avoiding a "trivia" focus for the game, for example: "What variable should go in this spot?".
4. Offering the possibility for struggling students to "buy" hints to help them create a solution.
5. Containing some gamification elements.
6. Having a graphical interface.

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

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

Additionally, to help future maintainers, DubitaC hould be:

- Well documented.
- Easy to extend.

3.2 Main game loop

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 form 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.

3.2.1 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.

3.3 Gamification elements

DubitaC implements the following gamification elements:

- A responsive graphical interface.
- A points system.
- A series of Avatars to choose from, that are publicly shown to the other players and are closely tied with the point system.
- An hint system, that is not too punishing and kept private.
- A multiplayer competitive environment.

To compile Table 3.1 we compared DubitaC to its equivalent educational setting, a university laboratory session where students are asked to complete a codeQuestion.

User enjoyment is targeted by:

- A more interesting graphical interface than a simple notepad.
- The "fun" competitive aspect of the multiplayer component.

User retention is targeted by:

- The points system, that rewards the player with a wider choice of Avatars for future game sessions giving them a sense of progression.
- The Avatars being public, displayed to other players as an "ego boost" marking the player progression.
- The "fun" competitive aspect of the multiplayer component.

User performance is targeted by:

- The hints system, which is forgivin and kept private, to encourage struggling player to use them, rather than giving up creating a solution.
- The multiplayer component requiring to analyse, understand and spot mistakes in other player's solutions, which does not happen in the traditional laboratory setting.

	User		
	Enjoyment	Retention	Performance
Graphical Interface	√	×	×
Points system	×	√	×
Public Avatars	×	√	×
Private hints	×	×	√
Multiplayer competition	√	√	√

Table 3.1: Expected improvements caused by gamification elements. User enjoyment can 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.

3.4 Example execution

DubitaC can be split into 5 different scenes, see Figure 3.1:

1. Main menu.
2. First round.
3. Second round.
4. Slideshow.
5. Final menu.

3.4.1 Main menu

The main menu manages the connectivity of the participants and setup of a game session, see Figure 3.2.

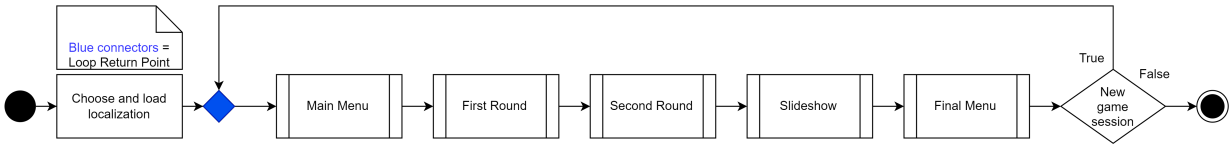


Figure 3.1: General flow of the game with 5 scenes where most of the gameplay is concentrated in the first and second rounds and there is a possibility to start as many game sessions as desired without having to restart the execution.

3.4.1.1 Shared view

A shared panel between server and clients lets users select their preferred language then, by continuing, the main menu shows different interfaces depending on the build.

3.4.1.2 Server view

The server loads and parses the codeQuestions available in the game following the expected format, see Figure 3.3.

The server's user (admin) selects the codeQuestion and the amount of time alloweded to complete it for the next game session. The admin **must** communicate the Ipv4 of the server to its players, to simplify this step, the Ipv4 address of the server is shown at the top of the interface, see Figure 3.4.

When a client tries to connect, the server authenticates the login credentials, See Figure 3.5 and, if they are correct, assigns the client to the first available lobby.

In case a client sends a message containing the new Avatar chosen, the server updates their choice.

When the admin is satisfied with the number of connected clients, it can start the game session.

At this point, the clients are rebalanced between lobbies, see Figure 3.6, and are given the information about the game session and which clients belong to the same lobby.

Lastly, the server synchronously loads the next scene for all clients and itself.

3.4.1.3 Client view

The players **need** to insert the Ipv4 address that they want to connect to in the relevant field. Additionally, before being able to request a connection, the user needs to insert some

valid login credentials, either previously known or, if correctly indicated by a checkbox, new credentials with a unique name and respecting length requirements for both username and password, see Figure 3.7.

If it wishes, the user can also access an option panel, where it can modify volume, maximum timeout and the path where to save user solutions, or a "How to play" panel, explaining the different scenes and gameplay of DubitaC, see Figure 3.8.

Lastly, until the server signals that the first round is about to start, a cosmetic panel will open where the user can select their Avatar from a list of available Avatars depending on the amount of points that the user had previously earned, see Figure 3.9.

3.4.2 First Round

In the first round the players will create their solutions for the given codeQuestion, see Figure 3.10.

3.4.2.1 Server view

The server waits for a timer counting down from the available time chosen in the previous scene.

In case a client sends a message containing their ready status, the server updates them in a list containing the order with which the clients have been signaling their ready status.

When the timer runs out, the server receives all user solutions and shares them with all clients in the relevant lobbies.

Lastly, the server synchronously loads the next scene for all clients and itself.

3.4.2.2 Client view

Each client is informed of which codeQuestion to solve, then they can interact with the notepad in front of them to write their solution.

A timer is started from the available time that the server had chosen before starting the game session.

A user can test its solution and signal that they are ready after a successful execution. If the solution failed to compile or did not pass some tests, a log will be filled with the relevant information, but the user will not be able to ready themselves, see Figure 3.11.

The user can open a hint panel where it can find the codeQuestion description and it will have the possibility to buy hints at the cost of some final progression points.

Additionally, the user can save its solution to the folder that they had chosen in the options in the previous scene.

When the timer runs out, the client sends its solution to the server.

3.4.3 Second round

In the second round the players read each other's solutions and create doubts against their possible mistakes, see Figure 3.12.

3.4.3.1 Server view

The server spawns the client Avatars following the ready list filled in the previous scene.

The server waits for a timer counting down from the available time chosen in the first scene multiplied by a balancing factor, depending on how many clients are in each lobby.

When the timer runs out, the server receives all user doubts.

The server combines all the solutions with the corresponding doubts, then sends them back to the original owner of the solution.

Afterwards, the server creates the "perfect" solutions and attaches the corresponding doubts.

The "perfect" solutions are then executed on the server and their result is parsed and saved.

Then, the results of the clients' executions are received, parsed and saved.

The clients' points are calculated and a new leaderboard is saved.

Lastly, the server signals that the sldeshow is about to start and synchronously loads the next scene for all clients and itself.

3.4.3.2 Client view

Receive all necessary information about the other clients in the lobby from the server.

A timer is started from the available time that the server had chosen in the first scene, multiplied by a balancing factor, depending on how many clients are in the current lobby.

The clients can select, from a list on the left side of the screen, a player from the same lobby to view its solution.

If the player has identified a possible mistake, they can create a doubt, see Figure 3.13.

When the timer runs out, the doubts are sent to the server.

Receive a string representing the solution with doubts is sent from the server.

The client compile and tests the solution and then sends back the result to the server.

3.4.4 Slideshow

During the slideshow, the player can see the evaluation of all doubts in the lobby and then a partial leaderboard, see Figure 3.14.

3.4.4.1 Server view

The server spawns all clients in the same order as last scene.

The server shares all doubts and the new leaderboard.

Depending on the number of doubts, the server waits for the slideshow to finish.

The server creates one last version for each user solution where the final test battery is included.

Then the final test batteries are executed and the results are parsed and saved.

The points are calculated and the final leaderboard is saved.

Before loading the next scene, the server saves the progress of each client in the database.

Lastly, the server signals that the final menu is about to be loaded and synchronously loads the next scene for all clients and itself.

3.4.4.2 Client view

The clients wait for the slideshow to finish, then the final leaderboard is shown and the clients wait once more for the next scene.

3.4.5 Final menu

The final menu displays the final leaderboard, manages the points distribution and final disconnections, see Figure 3.15

3.4.5.1 Server view

The server spawns all clients following the final leaderboard.

If it wishes, the admin can save all the client's solutions on the local machine.

If the admin disconnects, all connected clients and the server itself reloads the first scene: the main menu.

3.4.5.2 Client view

The final interface shows the final leaderboard and gives the possibility to save on the local machine the best solution of the lobby and the client's own solution.

If the player disconnects manually, the cosmetic panel opens showing the amount of points that the user now has and which Avatars will be unlocked for the next game session.

After a manual disconnection the client can go back and reload the main menu.

If the server is disconnected before the client can disconnect manually, the first scene is loaded: the main menu.

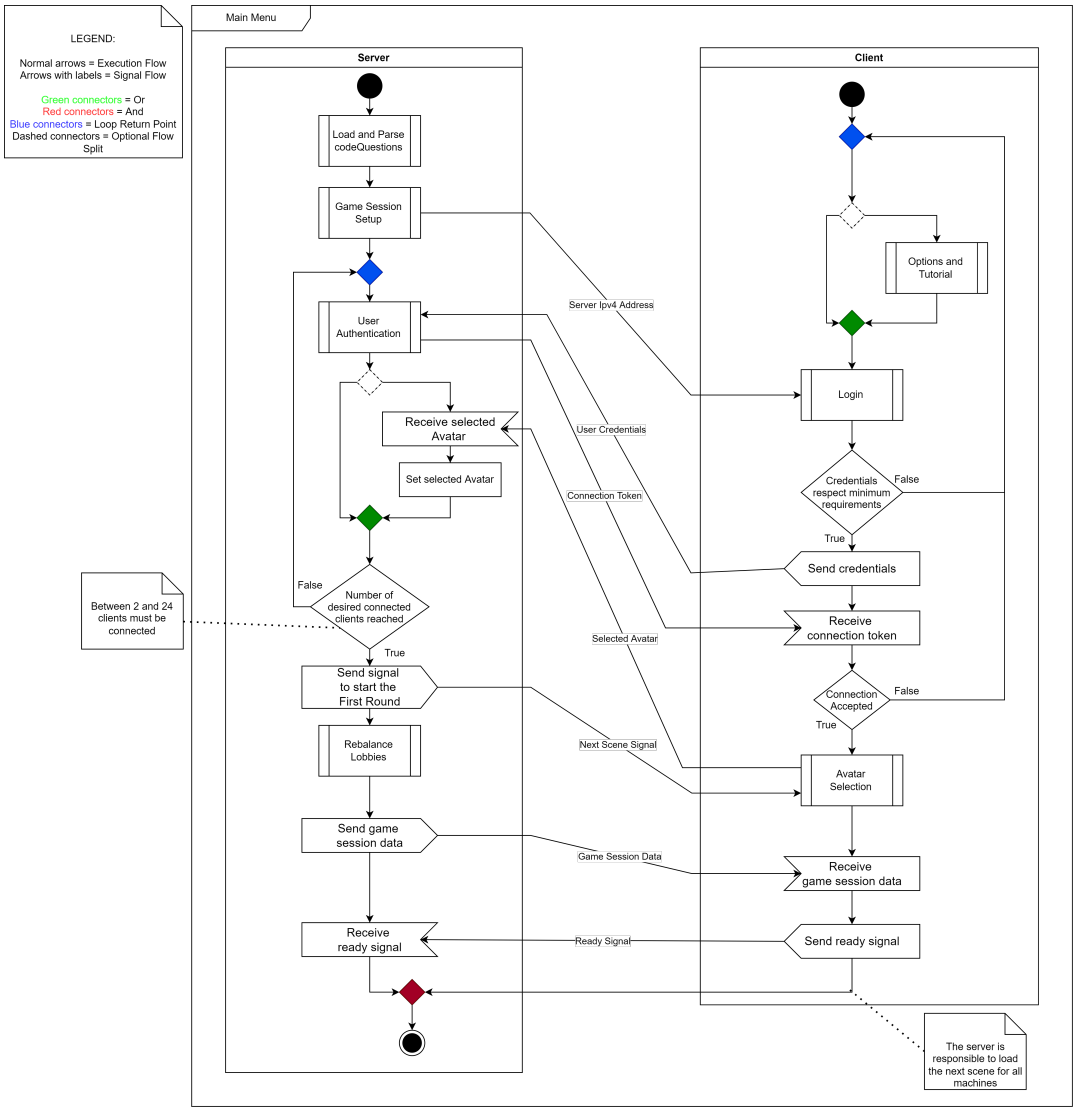


Figure 3.2: High level activity diagram of the main menu scene.

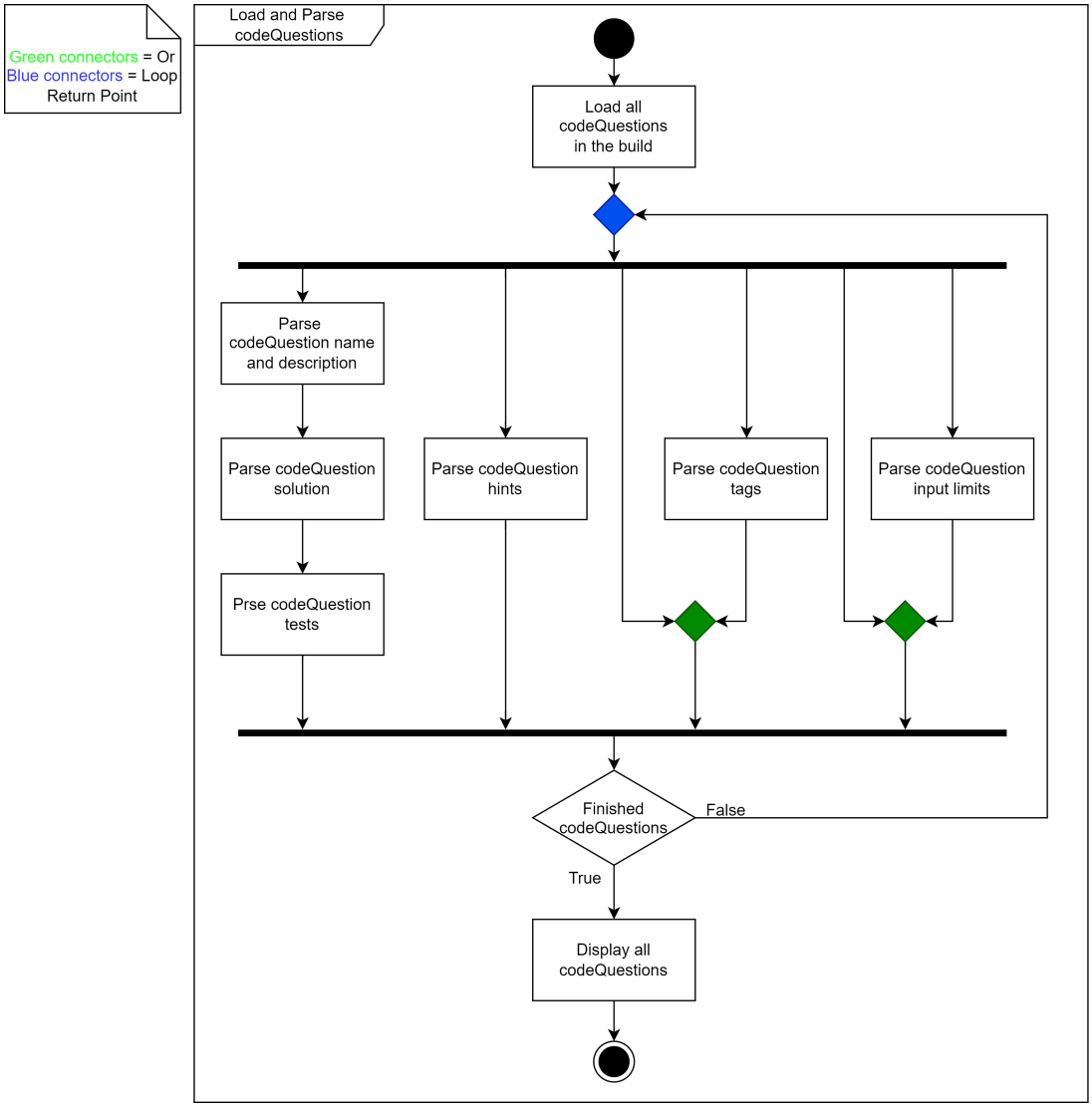


Figure 3.3: Activity diagram of the main menu subprocess "Load and Parse codeQuestions".

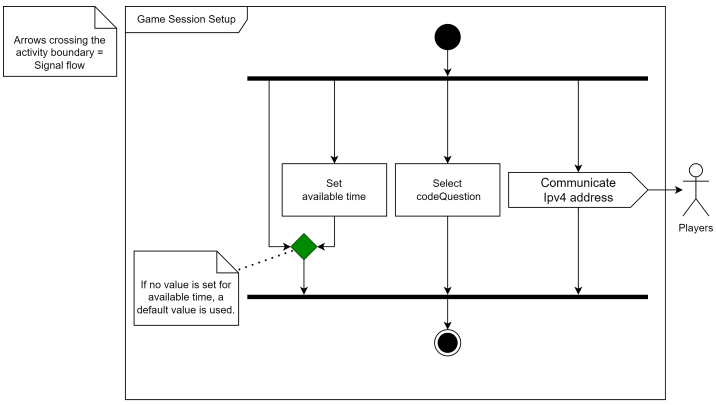


Figure 3.4: Activity diagram of the main menu subprocess "Game Session Setup".

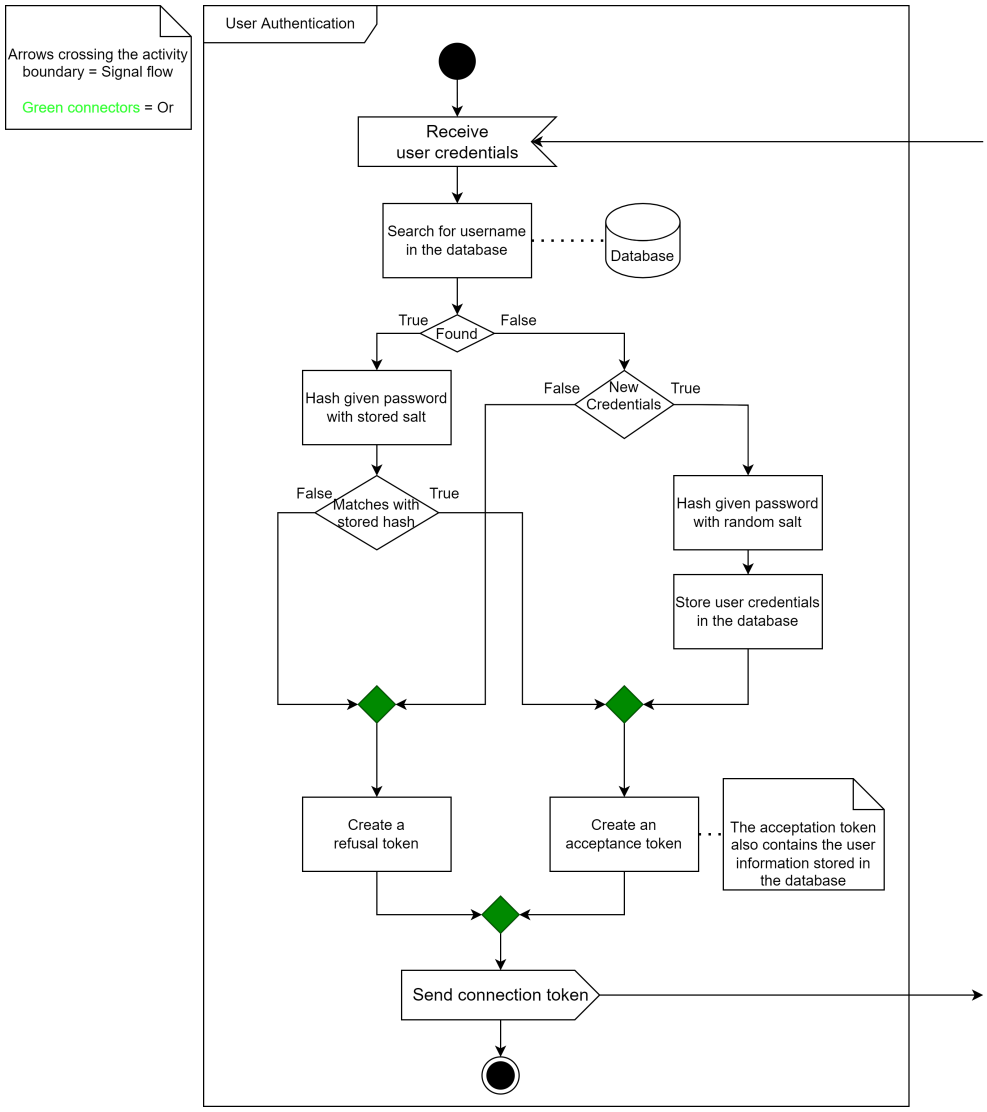


Figure 3.5: Activity diagram of the main menu subprocess "User Authentication".

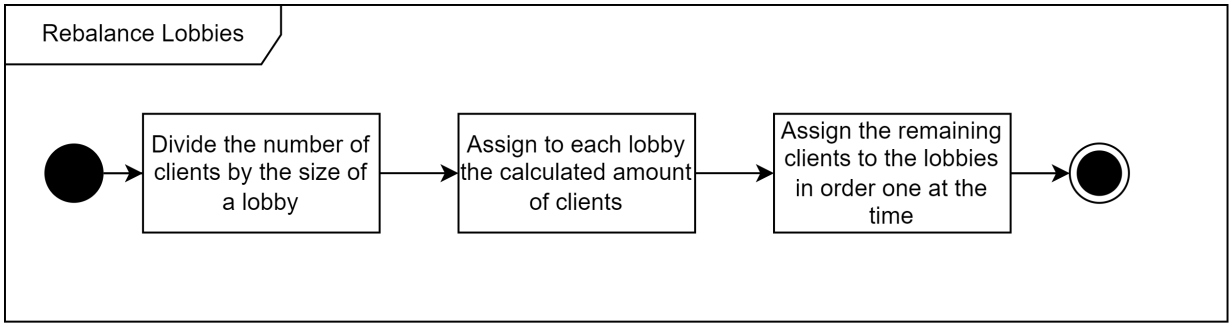


Figure 3.6: Activity diagram of the main Menu subprocess "Rebalance Lobbies".

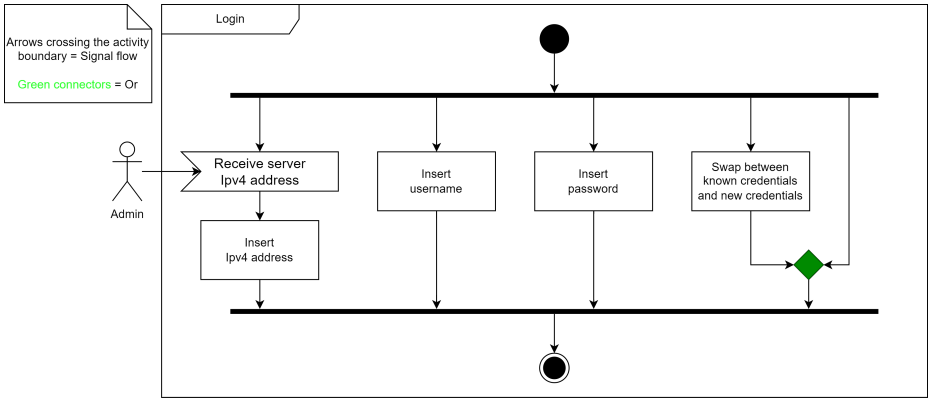


Figure 3.7: Activity diagram of the main menu subprocess "Login".

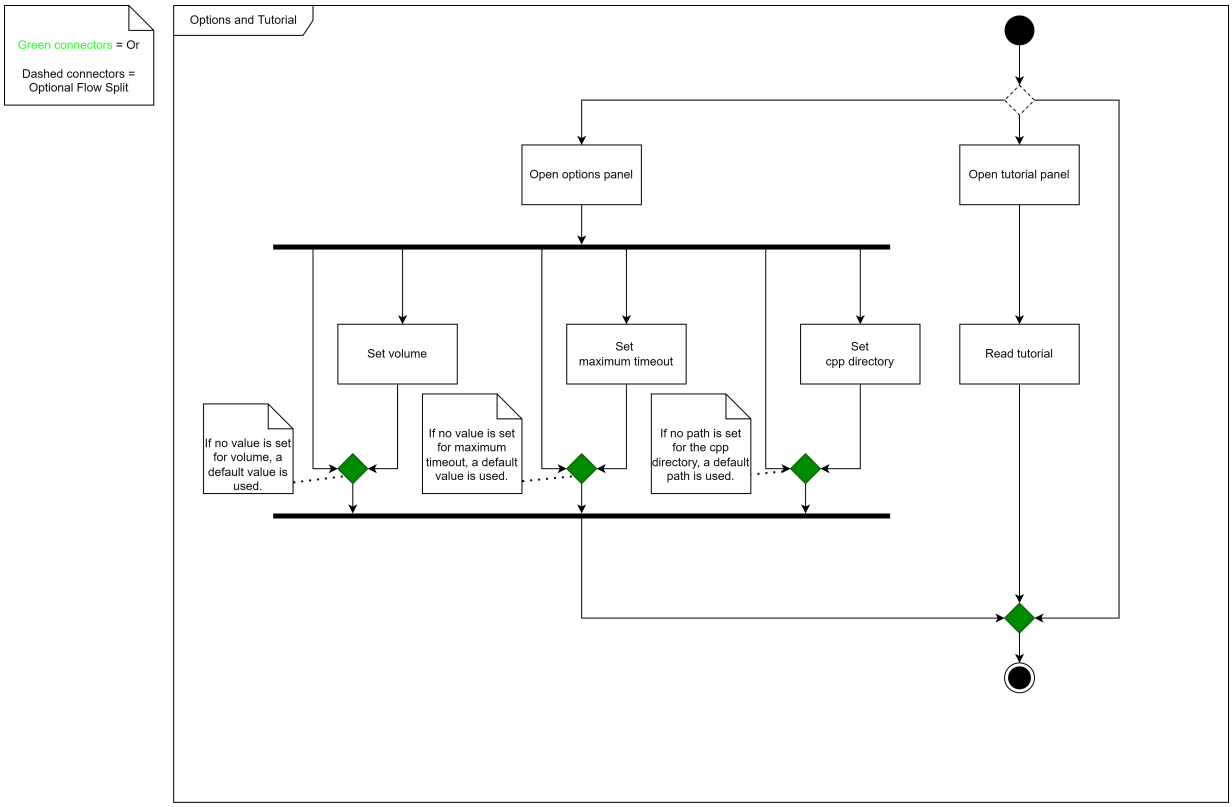


Figure 3.8: Activity diagram of the main menu subprocess "Options and Tutorial".

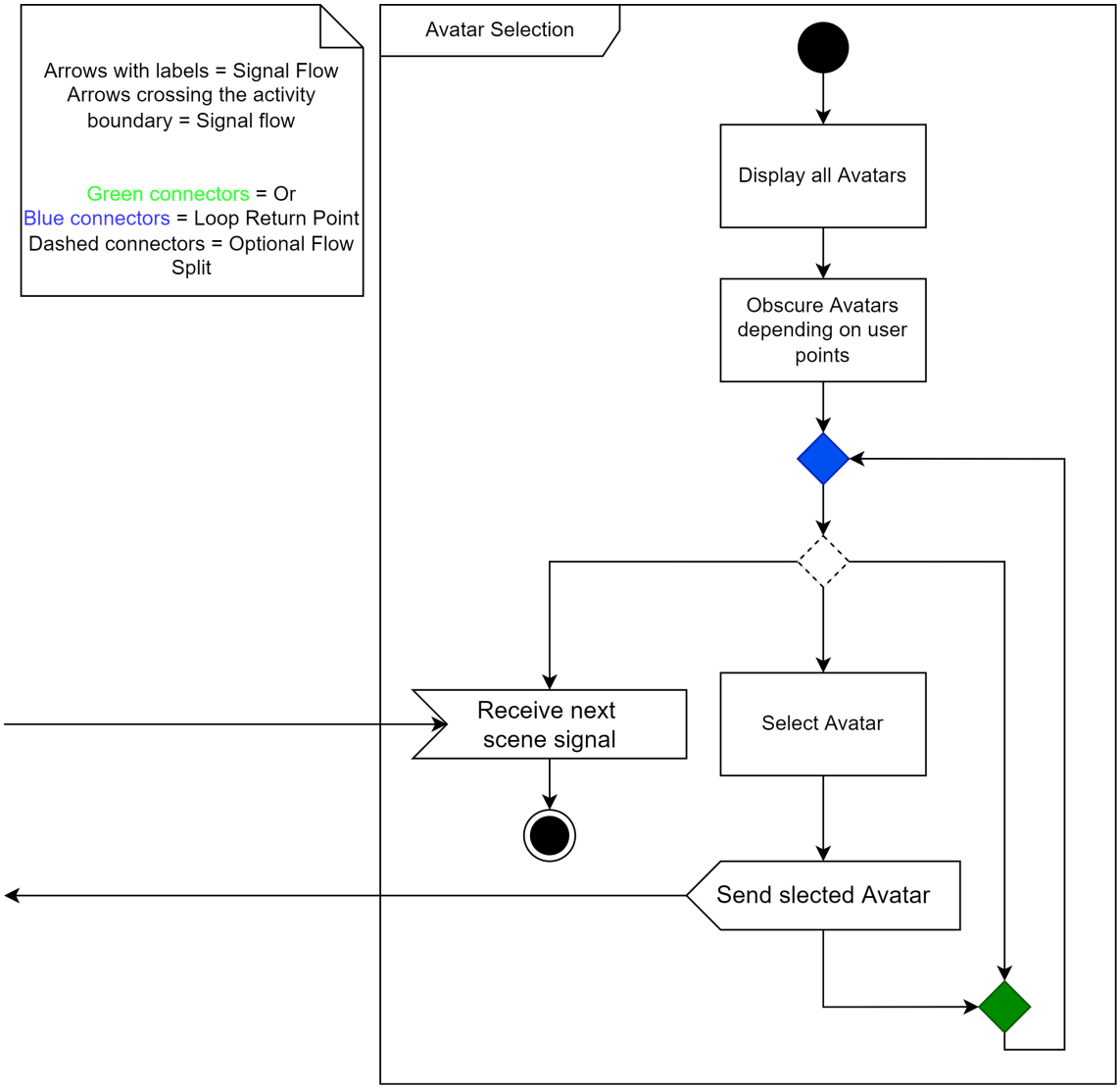


Figure 3.9: Activity diagram of the main menu subprocess "Avatar Selection".

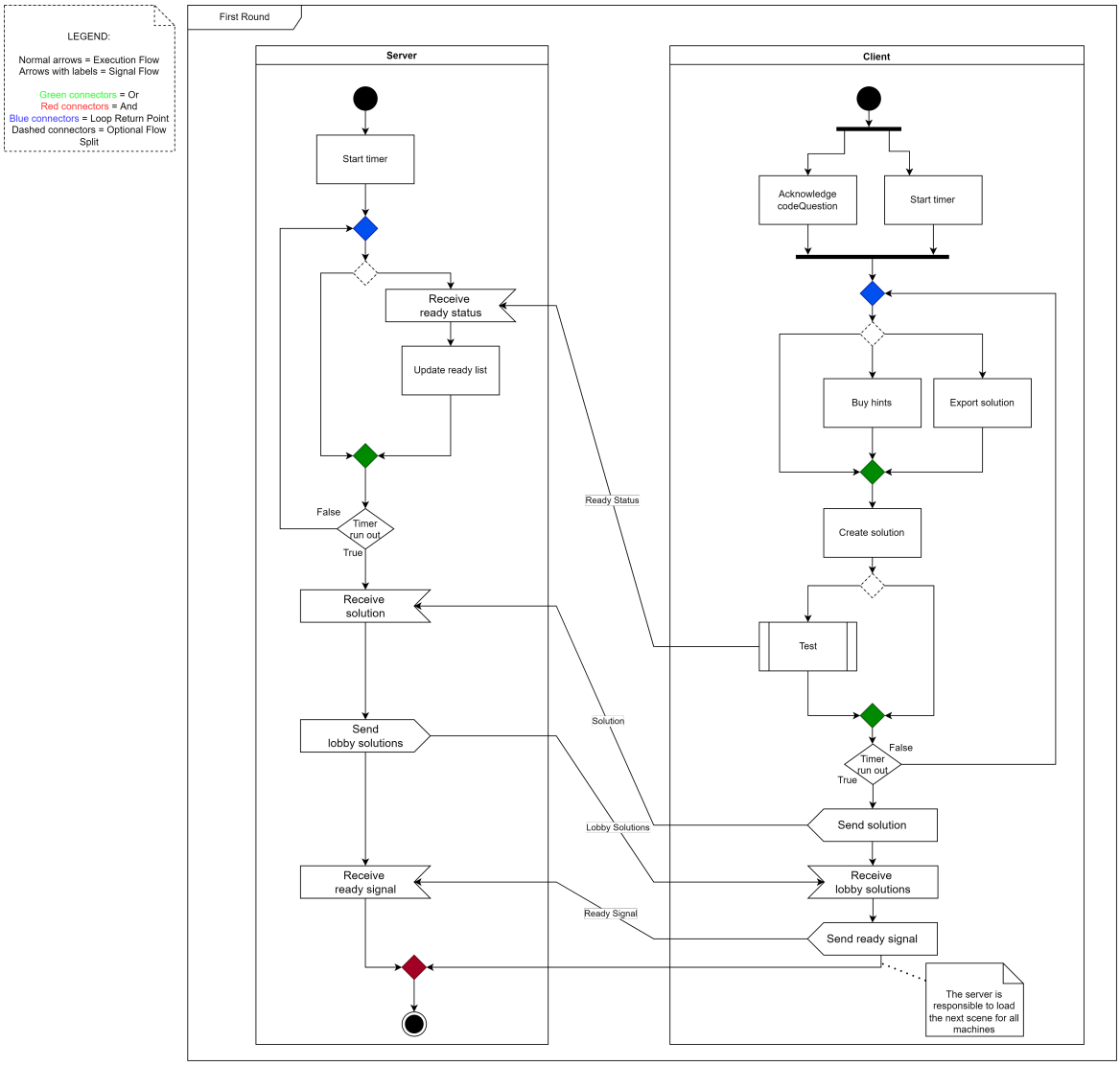


Figure 3.10: High level activity diagram of the first round scene.

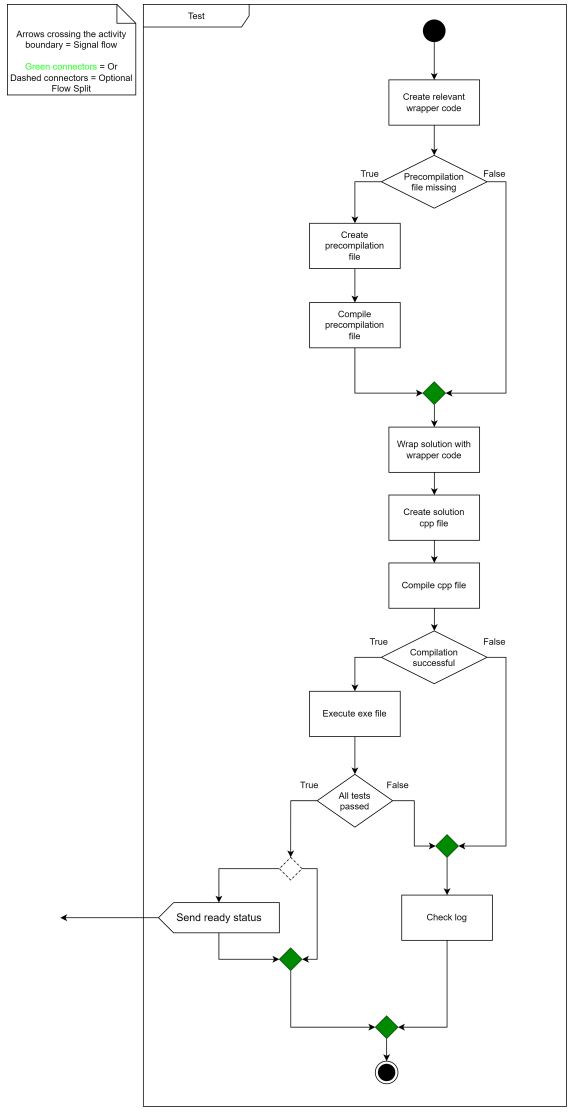


Figure 3.11: Activity diagram of the first round subprocess "Test".

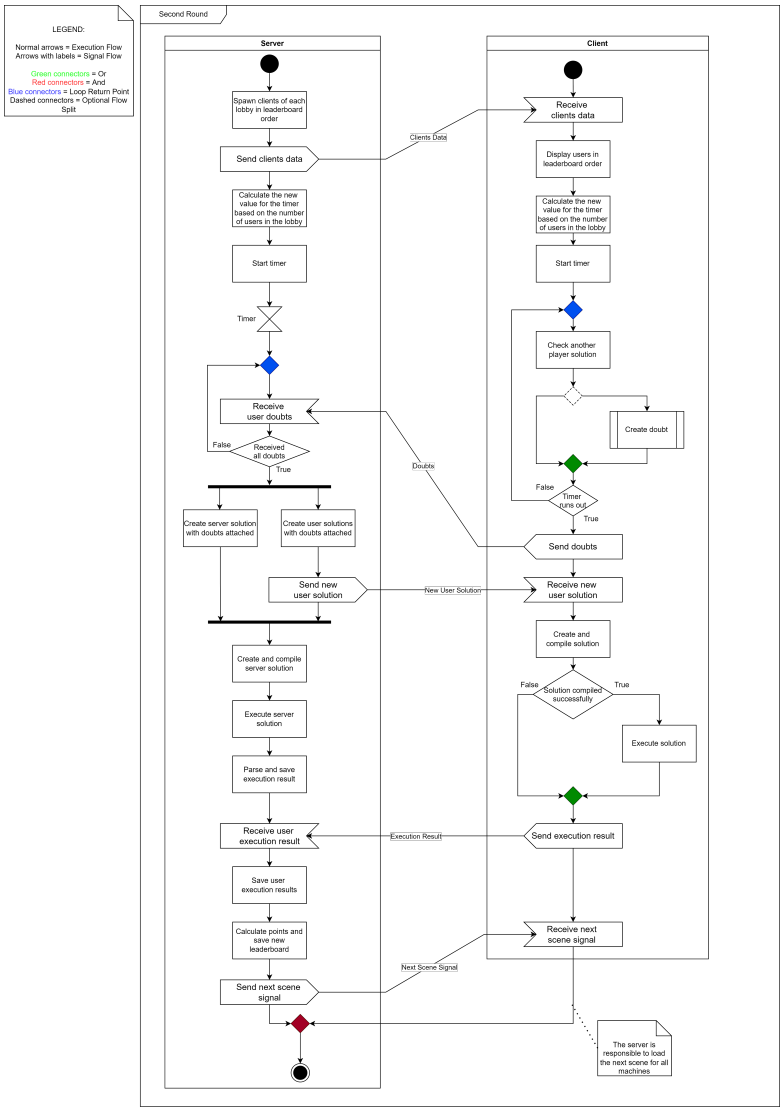


Figure 3.12: High level activity diagram of the second round scene.

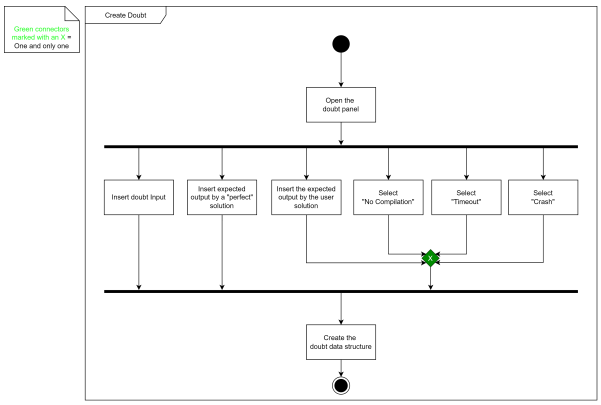


Figure 3.13: Activity diagram of the second round subprocess "Create Doubt".

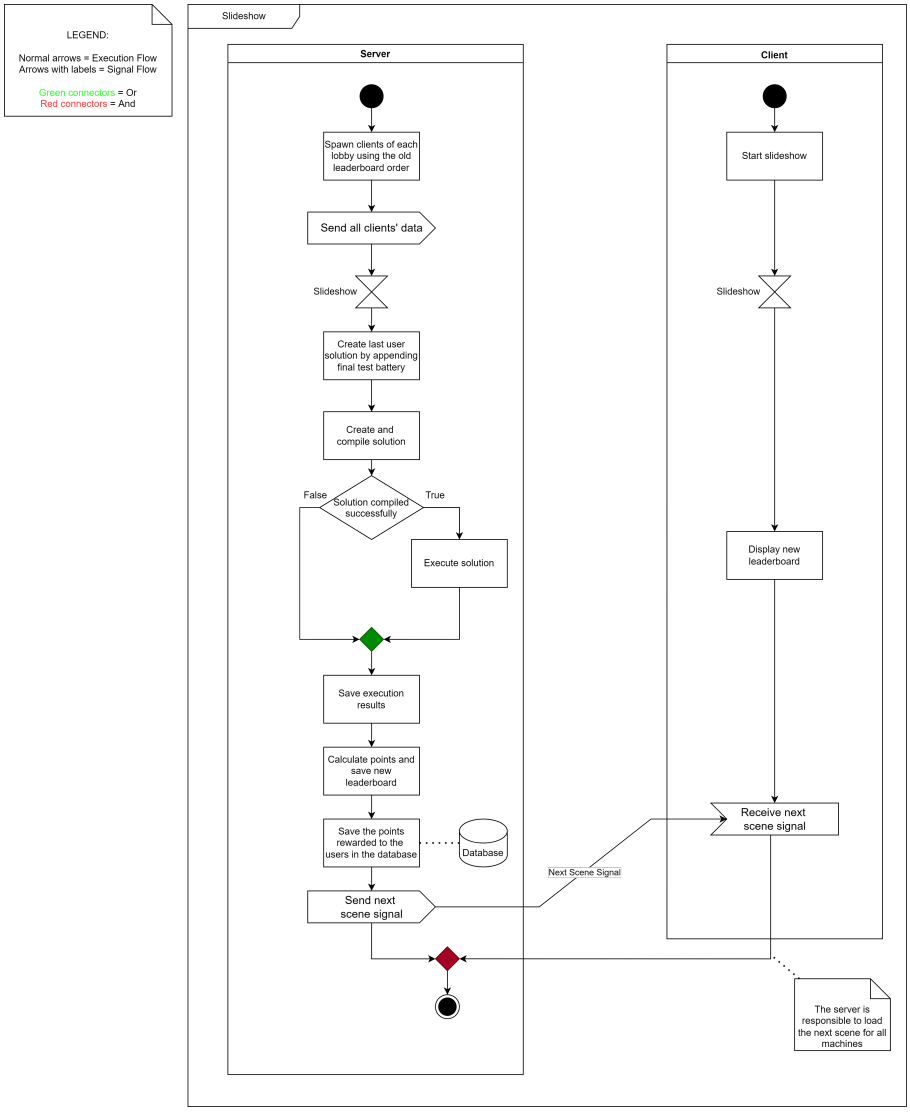


Figure 3.14: High level activity diagram of the slideshow scene.

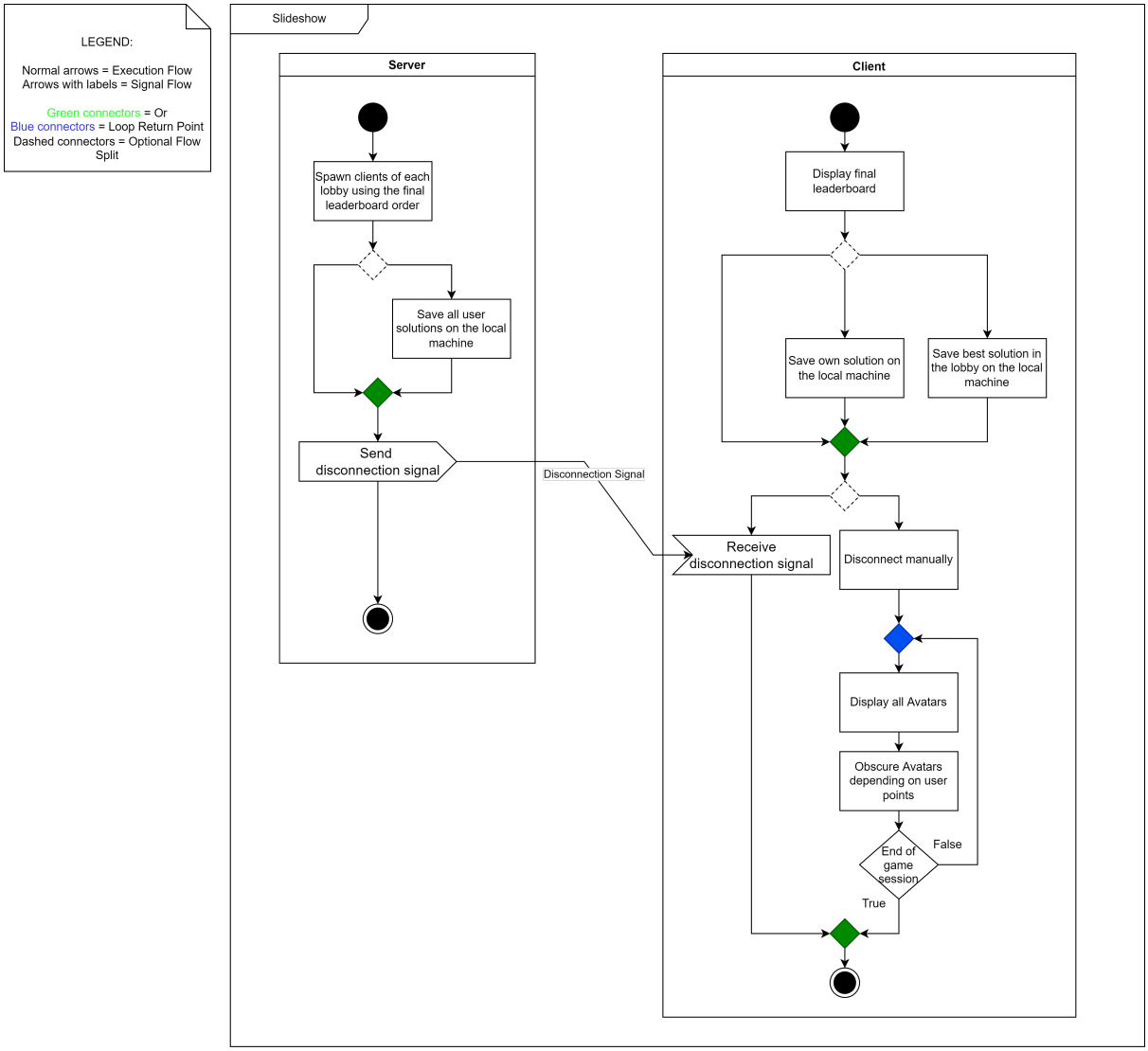


Figure 3.15: High level activity diagram of the Final menu scene.

Chapter 4

Game implementation

In this chapter we will detail the implementation of DubitaC, from the choice of software used to the details to the challenges that we had to overcome.

4.1 Possible implementation choices

Since we needed to create a graphical interface, we could choose between 2 different approaches:

- Chose a programming language and learn how to create a graphical interface.
- Utilize an existing game engine.

A choice between the 2 would also influence the general development of the game.

Additionally, since the game is online, we had to choose between:

- Handling the communications by ourselves, from establishing a connection to the packet transport.
- Utilize an existing solution that would take care of the low level implementation.

Regarding the main game loop, the solutions created by the players would need to be tested either by:

- Starting a new process for each different test.

- Utilize an existing testing framework to execute a single execution for all tests regarding a solution.

Lastly, because we wanted to create an installer, we had to find a compiler that could package all the required files.

4.2 Choices and motivations

In this section we motivate the choices made regarding Section 4.1.

4.2.1 Using a game engine

We decided to use a game engine so that we could focus our development on the main moving parts without having to worry about implementing our own low level solutions for rendering and allocation/deallocation, for example.

We chose to use Unity[HFA04] as our game engine of choice, since we were already familiar with the tool.

Unity offers a framework that follows the objects and components approach, the developer can create objects that can hold different components, while the components contain the logic that interacts with the main execution loop.

Additionally, the user can create its own scripts, using C# and a wide variety of .NET libraries, that can be attached to GameObjects and by interfacing with the main Unity procedures (like Start or Update) they will be treated like default Unity components.

The Unity execution loop can be seen in Figure 4.1.

4.2.2 Using a networking package

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 experimental.

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 4.4.3 for our implementation of the login system.

4.2.3 Using a test framework

We decided to use a test framework for C++ programs: Catch2.[Hoř15]

We chose this framework because of 3 main reasons:

- Lightweight execution.
- Single header integration.
- Free to use and constantly maintaned.

When using Catch2, the developer only needs to include the given header and write some tests regarding the functions that it wants to test, then at compile time Catch2 will take care to create an alternative main, so that at runtime, the executable would return to the terminal the results of the tests.

4.2.4 Using an installer creator

A Windows installer has been created to aid in the installation of DubitaC by using Inno Setup.[Rus97]

Inno Setup is a free software that aids users in creating their own installers for their application, it offers different customization options and a visual interface to create and compile the setup scripts.

Inno Setup is considered the best software to create Windows installers, even being used by professional companies, and was chosen because of this and ease of use.

4.3 Development process

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 for the first release started from scratch.

4.3.1 The first 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.

4.3.2 Architectural components

The final components architecture of DubitaC is composed of the main loop and some additional systems that we implemented to create a more complete final product, see Figure 4.2.

4.4 Detailed design: limits and solutions

In this section we present some obstacles that we had to overcome during the implementation of the game's systems.

4.4.1 Creating a valid cpp

Since the main loop consists of creating valid C++ programs that will be compiled and then run against different test batteries, we had to ensure that the user solutions would follow the correct format.

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.

4.4.2 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 4.4.7 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 Figure 3.13, 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.

4.4.2.1 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 successfull compilation does not return anything on stadard 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.

4.4.2.2 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 4.3 shows the whole wrapper that we wrote for the task, additionally, we will explain briefly all of its parts:

```
//_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.

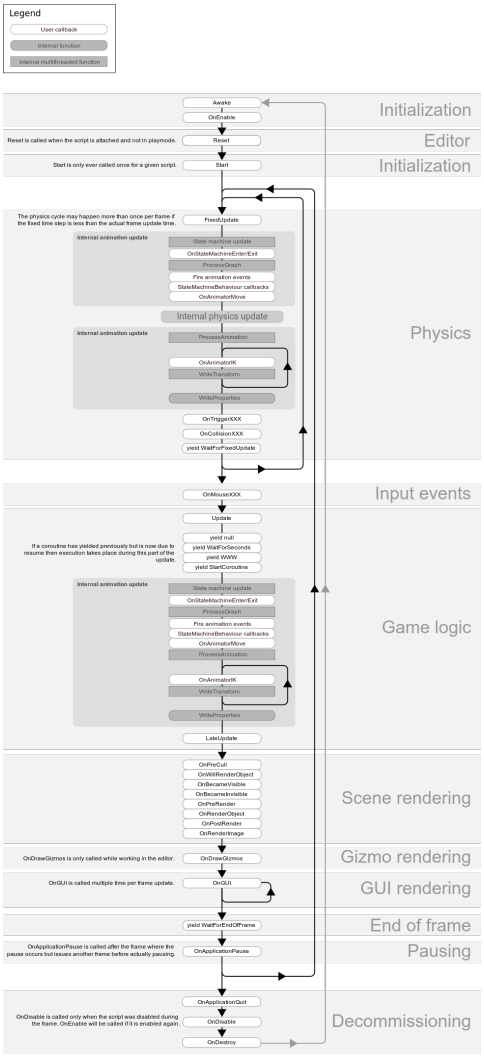


Figure 4.1: The complete execution flow employed by Unity. The complexity of the system is alleviated by the possibility of developers to interface only with the "white" actions in the graph.

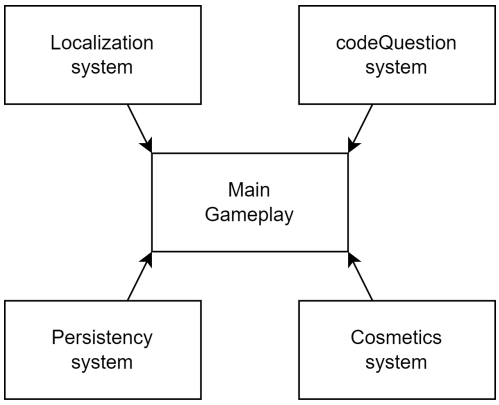


Figure 4.2: Component view of the game. The additional systems contribute in creating a more complete producte, but were not necessary for the development of the core gameplay.

```
//_type_// //_name_//(int __timeout__, //_full_arguments_//){
    std::mutex __mutex__;
    std::condition_variable __conVal__;
    std::exception_ptr __exPtr__ = nullptr;
    bool __wakeUp__ = false;
    //_type_// __retVal__;

    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();

    {
        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__;
}
```

Figure 4.3: 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.

```
{
    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 excepton 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.

4.4.2.3 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 4.4.2.2, 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
```

4.4.2.4 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);
};
```

4.4.2.5 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 procede 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 4.4.2.1, and parsed to calculate the points that needs to be assigned to the clients based on the performance of their solution and their doubts.

4.4.3 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.

4.4.3.1 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 pasword 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 cresentials is shown in Figure 3.5.

4.4.3.2 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 4.1 summarizes the considerations of this section.

<table>
<tr><th></th><th>Eavesdropper</th><th>Peeker</th></tr>
<tr><td>Plaintext communication and storage</td><td rowspan="2">No action needed</td><td>No action needed</td></tr>
<tr><td>Serverside hashing only</td><td rowspan="3">Short password attacks,
Lookup tables,
Rainbow tables
+ deducing identical passwords</td></tr>
<tr><td>Clientside hashing only</td><td rowspan="2">Short password attacks,
Lookup tables
and Rainbow tables</td></tr>
<tr><td>Hashing on both sides</td><td>Lookup tables,
Rainbow tables
+ deducing identical passwords</td></tr>
<tr><td>Salting and hashing on both sides</td><td colspan="2">Dictionary attacks or brute force</td></tr>
</table>

Table 4.1: Table detailing the possible techniques that the 2 types of malicious actor, that we considered in Section 4.4.3.2, could perform depending on the different decisions taken when handling the users credentials, specifically their passwords.

4.4.3.3 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 incompatability 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, anyCSV 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 enconding that we could find did not have the first 2 properties, indeed, because of backwards compatability 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 charachter 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 viualized, 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 corespondence 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.

4.4.3.4 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 attemps are not problematic.

4.4.4 The localization

In game development, and software development in general, the easiest apporach 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

4.4.4.1 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.

4.4.4.2 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.

4.4.4.3 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 4.1.

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.

4.4.5 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 4.4.2.2.

The list of possible labels are:

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

4.4.5.1 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.

4.4.5.2 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 distrupt 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 maount of seconds.

4.4.5.3 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 on 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:

```
i0 :: (:, 7)                    i0 < 7
i1 :: (2, 100)                  2 < i1 < 100
i2 :: [0, :)                    i2 ≥ 0
i3 :: {1914, 1939}              i3 ∈ {1914, 1939}
f0, f1 :: (0,1]                 0 < f0, f1 ≤ 1
f2 :: [0, 2022)                 0 ≤ f2 < 2022
f3 :: {3.14, 2.718}             f3 ∈ {3.14, 2.718}
c0, c1 :: {'a', 'b', 'c'}       c0, c1 ∈ {'a', 'b', 'c'}
```

```
s0, s1 :: [1, 5]                1 ≤ len(s0), len(s1) ≤ 5
s2 :: {"Hello", "world!"}       s2 ∈ {"Hello", "world!"}
```

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

4.4.6 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 enought 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.

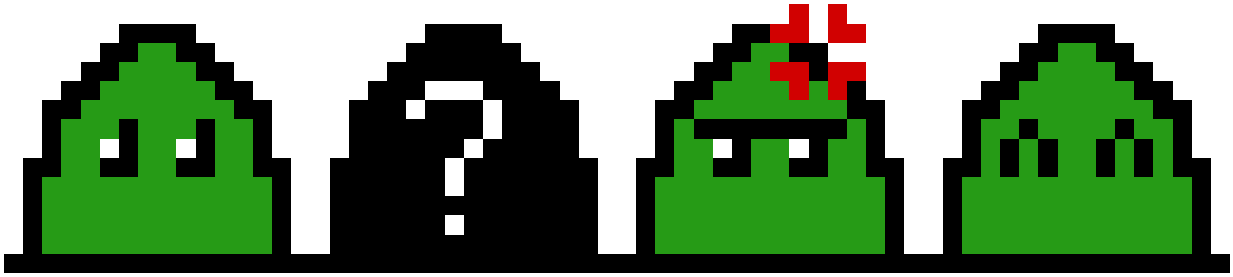


Figure 4.4: The "Slime" Avatar's sprite that can be selected during the game. The Avatar is repeated 4 times, once for each of the possible states: normal, locked, angry and happy.

4.4.7 The Windows installer

Because of the need to separate server and client build, we decided to create an installer that would provide all the requirements to be able to start playing as soon as possible.

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.

4.4.8 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.

4.4.8.1 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 FasBufferReader 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 bandwith of the clients' connections.

4.4.8.2 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 correcly 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.

4.4.8.3 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.

4.4.9 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.

4.4.9.1 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 negligeable, 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.

4.4.9.2 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.

4.4.9.3 Avoiding strings when possible

Because structs cannot contain reference types, See Section 4.4.8.1, 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 4.4.9.1, 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.

4.4.9.4 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 4.4.8.1, 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.

4.4.10 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.

4.4.10.1 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.

4.4.11 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 4.4.2, 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.

4.4.11.1 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.

Unfotunately, 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.

4.4.12 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.

4.4.12.1 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 analize 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 4.1 shows the formula to calculate the number and size of the doubts created with N number of players:

$$\begin{aligned} |D| &= N * (N-1) \\ \text{size}(D) * |D| &= 370 \text{ bytes } * (N^2 - N) \end{aligned} \tag{4.1}$$

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

4.4.12.2 Reducing the number of doubts

Taking the Equation 4.1 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 manageble and reducing the repetition of the doubting game loop, achieving both of our goals.

Equation 4.2 shows the improved formula for calculating the amount of doubts:

$$|D| = \sum_{i=0}^{3} n_i * (n_i - 1) \quad \text{for } 2 \leq n_i \leq 6 \tag{4.2}$$

Where n_i represents the number of players that have been assigned to the ith 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 analyse and doubt from 23 to just 5.

4.4.12.3 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 4.4.10, 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 4.3:

$$\begin{aligned} T_2 &= T_1 + \max(n) * 0.15 \\ T_4 &= (\max(n)^2 - \max(n)) * 4 \\ t &= T_1 + T_2 + T_3 + T_4 \end{aligned} \tag{4.3}$$

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 3.6, 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.

4.4.12.4 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 consdiderations, Equations 4.2 and 4.3 can be parametrized, as shown in Equation 4.4.

$$\begin{aligned} |D| &= \sum_{i=0}^{L} n_i * (n_i - 1) \quad \text{for } 2 \leq n_i \leq P \\ T_2 &= T_1 + \max(n) * \alpha \\ T_4 &= (\max(n)^2 - \max(n)) * W \\ t &= T_1 + T_2 + T_3 + T_4 \end{aligned} \tag{4.4}$$

In conclusion, thanks to the efforts detailed in Sections 4.4.10 and 4.4.12, 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.

4.4.13 Increasing maintainability

To ensure that DubitaC can be considered a maintainable product, we provided a documentation containing the description of every class and wrote the component scripts so that they can be easily modified.

As an additional approach in the interest of maintainability, we decided to take a look into code metrics for maintainability.

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.

The code metrics calculated for version DubitaC 1.0 can be seen in Figure 4.5.

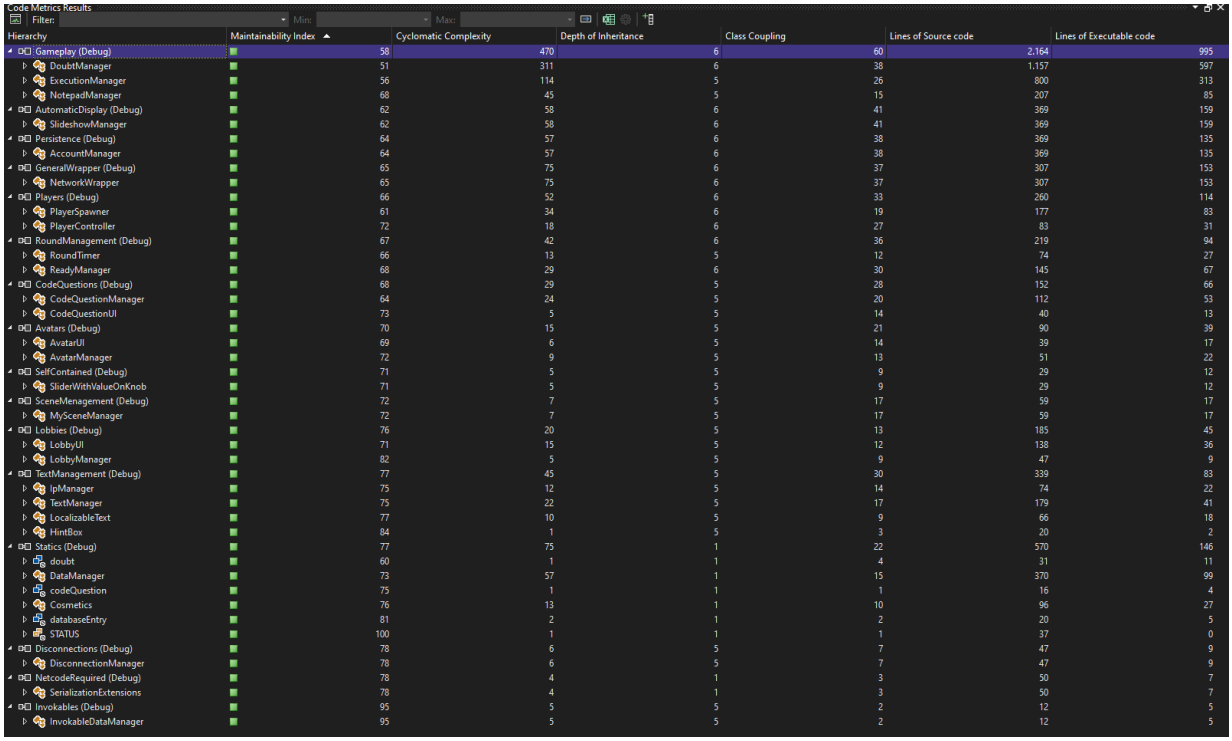
For each folder containing scripts, and each script itself, the results were promising, with the lowest maintainability index being 51 in the DoubtManager class.

It is also possible to go 1 step more in depth and analyse the code metrics of each function inside the classes, this time, however, the DoubtManager class contained a troubling method, highlighted in Figure 4.6.

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

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 betweeen the number of lines of code and a lower maintainability score.[RMT09][SAM12]



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
Gameplay (Debug)	58	470	6	60	2,164	995
DoubtManager	51	311	6	38	1,157	597
ExecutionManager	56	114	5	26	800	313
NotepadManager	68	45	5	15	207	85
AutomaticDisplay (Debug)	62	58	6	41	369	159
SlideshowManager	62	58	6	41	369	159
Persistence (Debug)	64	57	6	38	369	135
AccountManager	64	57	6	38	369	135
GeneralWrapper (Debug)	65	75	6	37	307	153
NetworkWrapper	65	75	6	37	307	153
Players (Debug)	66	52	6	33	260	114
PlayerSpawner	61	34	6	19	177	83
PlayerController	72	18	6	27	83	31
RoundManagement (Debug)	67	42	6	36	219	94
RoundTimer	66	13	5	12	74	27
ReadyManager	68	29	6	30	145	67
CodeQuestions (Debug)	68	29	5	28	152	66
CodeQuestionManager	64	24	5	20	112	53
CodeQuestionUI	73	5	5	14	40	13
Avatars (Debug)	70	15	5	21	90	39
AvatarUI	69	6	5	14	39	17
AvatarManager	72	9	5	13	51	22
SelfContained (Debug)	71	5	5	9	29	12
SliderWithValueOnKnob	71	5	5	9	29	12
SceneMenagement (Debug)	72	7	5	17	59	17
MySceneManager	72	7	5	17	59	17
Lobbies (Debug)	76	20	5	13	185	45
LobbyUI	71	15	5	12	138	36
LobbyManager	82	5	5	9	47	9
TextManagement (Debug)	77	45	5	30	339	83
IpManager	75	12	5	14	74	22
TextManager	75	22	5	17	179	41
LocalizableText	77	10	5	9	66	18
HintBox	84	1	5	3	20	2
Statics (Debug)	77	75	1	22	570	146
doubt	60	1	1	4	31	11
DataManager	73	57	1	15	370	99
codeQuestion	75	1	1	1	16	4
Cosmetics	76	13	1	10	96	27
databaseEntry	81	2	1	2	20	5
STATUS	100	1	1	1	37	0
Disconnections (Debug)	78	6	5	7	47	9
DisconnectionManager	78	6	5	7	47	9
NetcodeRequired (Debug)	78	4	1	3	50	7
SerializationExtensions	78	4	1	3	50	7
Invokables (Debug)	95	5	5	2	12	5
InvokableDataManager	95	5	5	2	12	5

Figure 4.5: 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.

We ended our refactoring with the code metrics shown in Figure 4.7, 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]

4.4.14 Current status of development

DubitaC has been developed as an Open Source project which can be found at: `TODO`

Containing source code, documentation, the 2 latest server and client builds, the Windows installer and this paper.

All the assets used have been created specifically for DubitaC and all the software used was free to use.

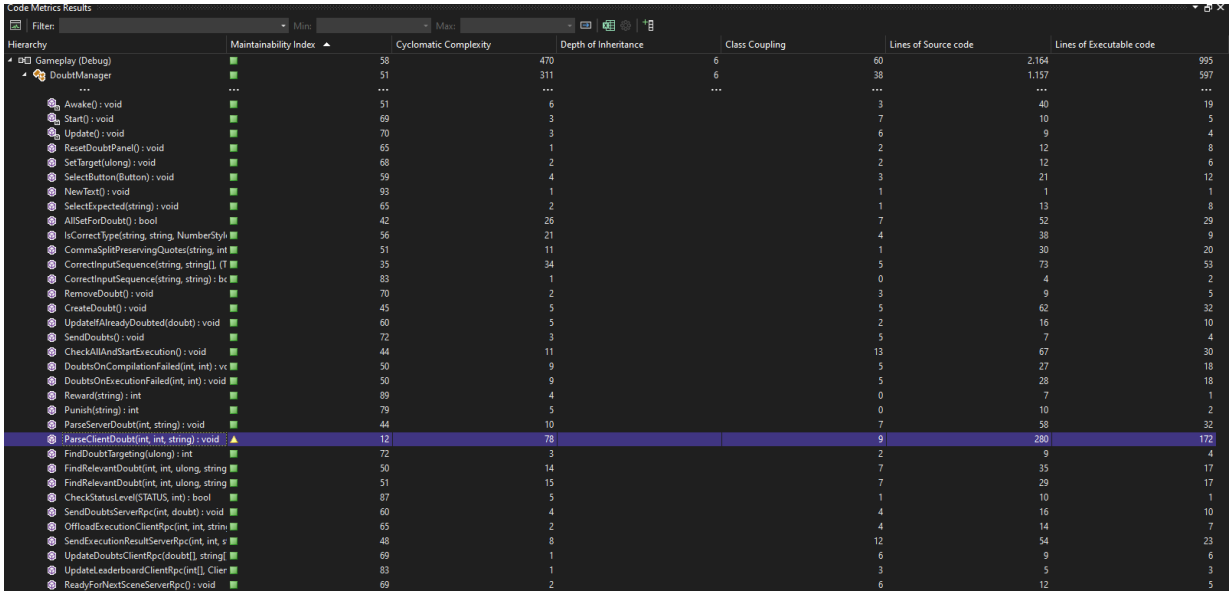
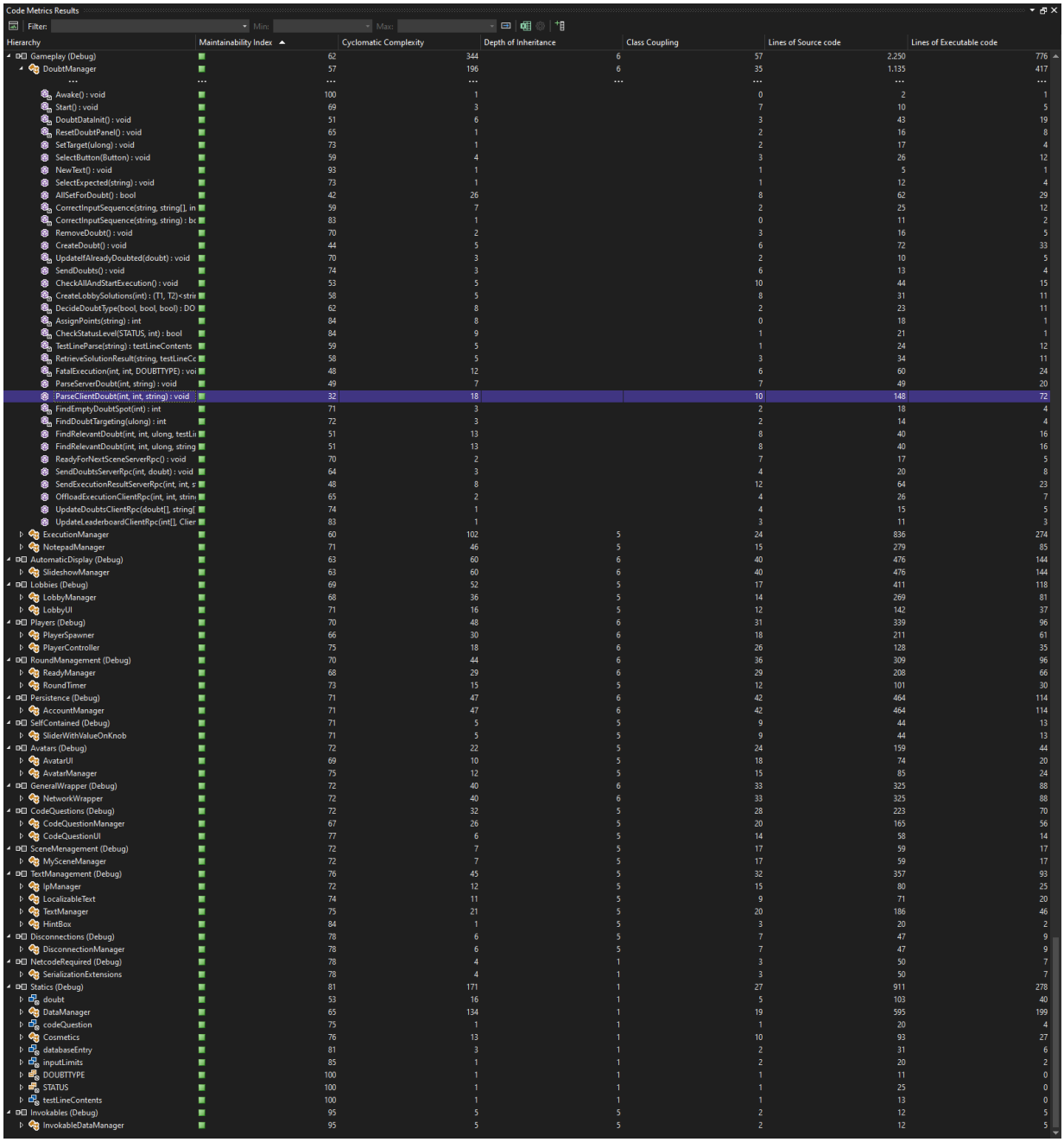


Figure 4.6: A more specific evaluation of the "DoubtManager" class, showing the maintainability score of each declared method. A single problematic method is highlighted by the selection in blue, with a yellow triangle instead of a green square indicating "problematic" maintainability. For visualization purposes, the declarations of non-method elements of the class have been replaced by a series of dots.

4.4.14.1 Current version

DubitaC is currently at version 1.2 containing:

- The complete implementation from connection to disconnection of the main gameplay.
- Various gamification elements so that the users would find DubitaC more engaging than its laboratory counterpart for studying basic C++ concepts.
- 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.
- A complete and easily extendable persistency system.
- 8 playable codeQuestions that can be easily extended by following the detailed format.
- 4 Avatars to choose from that can be easily extended by creating new sprites.



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code	Lines of Executable code
Gameplay (Debug)	62	344	6	57	2,250	776
DoubtManager	57	196	6	35	1,135	417
...	...	...	...	...	...	...
Awake() : void	100	1		0	2	1
Start() : void	69	3		7	10	5
DoubtDataInit() : void	51	6		3	43	19
ResetDoubtPanel() : void	65	1		2	16	8
SetTarget(ulong) : void	73	1		2	17	4
SelectButton(Button) : void	59	4		3	26	12
NewText() : void	93	1		1	5	1
SelectExpected(string) : void	73	1		1	12	4
AllSetForDoubt() : bool	42	26		8	62	29
CorrectInputSequence(string, string[], in	59	7		2	25	12
CorrectInputSequence(string, string) : bc	83	1		0	11	2
RemoveDoubt() : void	70	2		3	16	5
CreateDoubt() : void	44	5		6	72	33
UpdateIfAlreadyDoubted(doubt) : void	70	3		2	10	5
SendDoubts() : void	74	3		6	13	4
CheckAllAndStartExecution() : void	53	5		10	44	15
CreateLobbySolutions(int) : (T1, T2)<strir	58	5		8	31	11
DecideDoubtType(bool, bool, bool) : DO	62	8		2	23	11
AssignPoints(string) : int	84	8		0	18	1
CheckStatusLevel(STATUS, int) : bool	84	9		1	21	1
TestLineParse(string) : testLineContents	59	5		1	24	12
RetrieveSolutionResult(string, testLineCc	58	5		3	34	11
FatalExecution(int, int, DOUBTTYPE) : voi	48	12		6	60	24
ParseServerDoubt(int, string) : void	49	7		7	49	20
ParseClientDoubt(int, int, string) : void	32	18		10	148	72
FindEmptyDoubtSpot(int) : int	71	3		2	18	4
FindDoubtTargeting(ulong) : int	72	3		2	14	4
FindRelevantDoubt(int, int, ulong, testLi	51	13		8	40	16
FindRelevantDoubt(int, int, ulong, string	51	13		8	40	16
ReadyForNextSceneServerRpc() : void	70	2		7	17	5
SendDoubtsServerRpc(int, doubt) : void	64	3		4	20	8
SendExecutionResultServerRpc(int, int, s	48	8		12	64	23
OffloadExecutionClientRpc(int, int, strin	65	2		4	26	7
UpdateDoubtsClientRpc(doubt[], string[	74	1		4	15	5
UpdateLeaderboardClientRpc(int[], Clier	83	1		3	11	3
ExecutionManager	60	102	5	24	836	274
NotepadManager	71	46	5	15	279	85
AutomaticDisplay (Debug)	63	60	6	40	476	144
SlideshowManager	63	60	6	40	476	144
Lobbies (Debug)	69	52	5	17	411	118
LobbyManager	68	36	5	14	269	81
LobbyUI	71	16	5	12	142	37
Players (Debug)	70	48	6	31	339	96
PlayerSpawner	66	30	6	18	211	61
PlayerController	75	18	6	26	128	35
RoundManagement (Debug)	70	44	6	36	309	96
ReadyManager	68	29	6	29	208	66
RoundTimer	73	15	5	12	101	30
Persistence (Debug)	71	47	6	42	464	114
AccountManager	71	47	6	42	464	114
SelfContained (Debug)	71	5	5	9	44	13
SliderWithValueOnKnob	71	5	5	9	44	13
Avatars (Debug)	72	22	5	24	159	44
AvatarUI	69	10	5	18	74	20
AvatarManager	75	12	5	15	85	24
GeneralWrapper (Debug)	72	40	6	33	325	88
NetworkWrapper	72	40	6	33	325	88
CodeQuestions (Debug)	72	32	5	28	223	70
CodeQuestionManager	67	26	5	20	165	56
CodeQuestionUI	77	6	5	14	58	14
SceneMenagement (Debug)	72	7	5	17	59	17
MySceneManager	72	7	5	17	59	17
TextManagement (Debug)	76	45	5	32	357	93
IpManager	72	12	5	15	80	25
LocalizableText	74	11	5	9	71	20
TextManager	75	21	5	20	186	46
HintBox	84	1	5	3	20	2
Disconnections (Debug)	78	6	5	7	47	9
DisconnectionManager	78	6	5	7	47	9
NetcodeRequired (Debug)	78	4	1	3	50	7
SerializationExtensions	78	4	1	3	50	7
Statics (Debug)	81	171	1	27	911	278
doubt	53	16	1	5	103	40
DataManager	65	134	1	19	595	199
codeQuestion	75	1	1	1	20	4
Cosmetics	76	13	1	10	93	27
databaseEntry	81	3	1	2	31	6
inputLimits	85	1	1	2	20	2
DOUBTTYPE	100	1	1	1	11	0
STATUS	100	1	1	1	25	0
testLineContents	100	1	1	1	13	0
Invokables (Debug)	95	5	5	2	12	5
InvokableDataManager	95	5	5	2	12	5

Figure 4.7: Final maintainability score for DubitaC's current version. 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 5

Conclusions

To conclude our work, we present the limitation and some possible future extension in Section 5.1 followed by a final closing statement.

5.1 Limitations and future work

Currently, DubitaC suffers from the following 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 incresing 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 invokex with cmd.exe, and the g++ compiler must be present in the environment variables.
- 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.

Lastly, we offer some possible future extensions:

- 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" eperience 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.
- 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.
- 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.

5.2 Closing statement

In this thesis we detailed the complete process, from the starting goal of creating a serious game to support introductory programming courses, to the final implementation and obstacles that we had to face to create a satisfactory final product.

DubitaC is now a complete serious game and we encourage readers to give it a try and, if they so wish, contribute on the official Github repository.

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/cmsc818O/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.

[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. Hagemeister. 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.

[Rus97] Jordan Russell. Inno setup. Software for Windows installers, 1997. URL: https://jrsoftware.org/isinfo.php.

[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] Universitá 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/.

Appendix A

Reference manual

The reference manual offers some guidance for how to play and how to maintain DubitaC, with an additional troubleshooting section at the end.

A.1 How to play

Important parts are written in **bold**.

A.1.1 Players & Clients

A.1.1.1 Main menu

- Setup the preferred localization and options.
- Insert the server Ipv4 address, **must be shared by the server admin**.
- Insert the user credentials, if you do not have an account yet, insert the user credentials as normal but check the checkbox "Create new account".
- Press the "Play" button and avait for the connection to be accepted.
- Select an Avatar betweeen the available ones.

In case the connection is refused, for any reason, it will be possible to retry as much as needed.

The first round will be loaded when the server admin decides to start it.

A.1.1.2 First round

- Read the description of the codeQuestion.
- Type the solution in the notepad in the center of the screen, **do not change the function signature that is already written**.
- Test your solution by pressing the button "Test".
- Check the test results by pressing the button "Log".
- If needed, improve your solution to pass more tests.
- If available, press the button "Ready" to stop writing the solution and save your position in the leaderboard.
- At any moment, by pressing the "Hints" button, a panel containing the description of the codeQuestion and the possibility of buying hints will be opened.
- At any moment, by pressing the "Export" button, a copy of the solution will be saved on the local machine.

The second round will be loaded after the available time expires.

A.1.1.3 Second round

- Select one of the user avatars on the left to be able to read their solutions.
- If you spot a mistake, press the "Doubt" button to open the doubt panel.
- Create a doubt by filling out the panel and confirming with the "Doubt" button, **only one doubt can be created towards each user**.
- To remove a previously created doubt, select the target user and, in the doubt panel, press the top right button to remove any doubt towards the selected user.

A loading screen will appear when the available time expires.

The slideshow will be loaded after all clients and server have completed their executions.

A.1.1.4 Slideshow

- Observe all doubts that will be shown during the slideshow.

The intermediate leaderboard will appear when the slideshow finishes.

The final menu will be loaded when the server has completed the final executions.

A.1.1.5 Final menu

- If needed, by pressing one of the 2 "Export" buttons, the own solution or the best solution of the lobby will be saved on the local machine.
- Disconnect by pressing the "Disconnect" button in the top left corner.
- Go back to menu by pressing the "Back to menu" button in the top left corner.

The final menu shows the final leaderboard of the game session.

After having disconnected, a panel showing the amount of current points will be shown.

The main menu will be loaded after pressing on the "Back to menu" button.

A.1.1.6 Tips

The cost of the hints is minimal, always make sure to use them when some help is needed.

The list of users on the left during the second round is ordered using the current leaderboard, it is more advantageous to start doubting from the top, since that will increase the chances of being brought down.

While examining a solution, if no mistake can be found, it is more efficient to try to analyse another solution instead of trying to create a doubt for the current one.

A.1.2 Admin & Server

A.1.2.1 Main menu

- Setup the preferred localization.
- Share the Ipv4 address that is shown at the top of the screen, **this step is required**.

- If needed, insert the amount of available time to solve the codeQuestion.
- Select a codeQuestion from the list on the right, the tag filter can be used to quickly narrow down the number of options, **always select a codeQuestion that can be solved by the players based on the knowledge that they should have**.
- Start the game session, **At least 2 clients must be connected**.

The first round will be loaded when the "Start" button in the bottom right corner is pressed.

A.1.2.2 First round

- While waiting for the available time to run out, a panel showing the current progress can be read.

The second round will be loaded after the available time expires.

A.1.2.3 Second round

- While waiting for the available time to run out, a panel showing the current progress can be read.

The slideshow will be loaded after all clients and server have completed their executions.

A.1.2.4 Slideshow

- While waiting for the available time to run out, a panel showing the current progress can be read.

The final menu will be loaded when the server has completed the final executions.

A.1.2.5 Final menu

- If needed, by pressing the "Export" button, all user solutions will be saved on the local machine.
- Shutdown the server by pressing the "Disconnect" button in the top left corner.

The main menu will be loaded after pressing on the "Disconnect" button.

A.1.2.6 Tips

The admin interaction is minimal after the main menu, however, after the final menu has been loaded, it is important to save the user solution on the local machine, so that they might be analysed later. For this purpose the user solutions will contain a first comment line with the username of their author.

It might be wise to write somewhere the usernames that are involved in a game session, alongside their real names, for easier feedback distribution.

A.2 How to maintain

With the exception of troubleshooting, see Section A.3, the mainteinance of DubitaC should involve small tweaks to the source code to best fit the current needs, or extension of already existing systems.

A.2.1 Small tweaks

Every script that contains a class contains at the beginning of its property declaration some constants or readonly values that can be modified to apply small changes in the game executions.

All classes that contain such constants or readonly variables will explain their usage in the class summary at the top of the file, see Figure A.1.

```
/// <summary>
/// Class responsible to manage the interaction with the database and the validation of the connections with the server.
/// The maximum amount of points a user can have is stored in constant <see cref="maxPointsPossible"/>.
/// The string that is used during the clientside salting is stored in constant <see cref="sharedSalt"/>.
/// The Encoding that has been chosen is stored in the readonly field <see cref="currentEncoding"/>.
/// The Hash function that has been chosen is stored in the readonly field <see cref="hashAlgorithm"/>.
/// The separator in the database file is stored in constant <see cref="separator"/>.
/// The relative path to the database folder is stored in constant <see cref="relativeDatabasePath"/>.
/// The path to the database file is stored in constant <see cref="databaseFile"/>.
/// </summary>
[RequireComponent(typeof(NetworkObject))]
public class AccountManager : NetworkBehaviour {
    private const ushort maxPointsPossible = 9999;
    private const string sharedSalt = "11/09/2021-11/12/2021-28/03/2022";
    //Not allowed to save 2 classes as a const, so we make them readonly
    private readonly Encoding currentEncoding = new UTF8Encoding(true);
    private readonly HashAlgorithm hashAlgorithm = new SHA256Managed();
    private const string separator = ",";
    private const string relativeDatabasePath = "/Database";
    private const string databaseFile = "/accounts.csv";
```

Figure A.1: Example of a class that explains the available constants in the summary at the top.

A.2.2 Extending existing systems

A.2.2.1 Extending the localization system

The localization system is completely managed by the "TextManager" class.

To introduce a **new** localization:

- Create a file with:
  - A name following the format: 3 letters language identifier, underscore, localization.csv
  - The contents following the localization format: label between double quotes, comma, localized text between double quotes.
- Insert the file in the folder at path: Assets/Resources/Localization
- Create a sprite representing the new language with a country flag with a name following the format: 3 letters language identifier, underscore, icon.png
- Insert the sprite in the folder at path: Assets/Resources/OtherSprites
- Modify the value of the image in the Unity inspector to convert it from a texture to a sprite, see Figure A.2.
- Only in the main menu scene: extend the "TextManager" class' public property "Language Sprites" from the Unity inspector to include the new sprite, see Figure A.3.

To extend **existing** localizations, simply insert the new labels in all localization files following the localization format.

A.2.2.2 Extending the persistency system

The persistency system is completely managed by the "AccountManager" class, any modification to this class, that does not extend to other classes, must respect 2 restrictions:

- A function, that is public, void and at maximum with 1 parameter, needs to be exposed so that the submittion of user credentials can be executed by a button being pressed by the player, currently this function is called "Submit".
- A function with the same signature as the "ValidateLogin", see Figure A.4, must exist, so that the connection with the server can be validated using it.

If the first is not respected, the players will not be able to connect easily by pressing a button on the interface.

If the second is not respected, the server will accept any client that tries to connect, removing the user authentication step.

A.2.2.3 Extending the cosmetics system

The cosmetics system is managed by the static "Cosmetics" class for sprite initialization, while other classes are free to query it using its getters, for example classes "AvatarUI", "AvatarManager" and "NetworkWrapper".

To introduce a **new** Avatar:

- Create a new sprite with:
  - A name following the format: name of the sprite, underscore, number of points required to unlock.
  - Size of 64x16 pixels, divided in 4 squares of size 16x16, each containing a different Avatar expression: normal, obscured, angry, happy.
- Insert the sprite in the folder at path: Assets/Resources/AvatarSprites
- Modify the value of the image in the Unity inspector to convert it from a texture to a sprite, see Figure A.2.
- Press the "Sprite Editor" button on the inspector to slice the complete sprite into the 4 smaller sprites.
- Open the txt file at path: Assets/Resources/AvatarNames.txt
- Add the name of the new Avatar **in a new line**, making sure that the complete name does not contain the extension.

To change the name or point threshold of **existing** Avatars:

- Open the txt file at path: Assets/Resources/AvatarNames.txt
- Change the name of the target Avatars, making sure to respect the same format.
- Rename the Avatar sprite in path: Assets/Resources/AvatarSprites

Make sure that the Avatar sprite name and the name inserted in the "AvatarNames" text file always match.

A.2.2.4 Extending the codeQuestion system

The codeQuestion system is managed by the "CodeQuestionManager" class for the codeQuestions initialization. The "CodeQuestionUI" class is responsible for holding and displaying a single codeQuestion during the codeQuestion selection. Later, the "ExecutionManager" class , during startup parses the selected codeQuestion following the format detailed in Section 4.4.5.

To insert a **new** codeQuestion:

- Create and test the cpp file that corresponds to the "perfect" solution to the codeQuestion.
- Change the file extentsion from .cpp to .txt
- Insert the file in path: Assets/Resources/CodeQuestions

If the codeQuestion contents are malformed, an error is returned and continuing the execution of the game session might not be possible.

Make sure that the codeQuestion returns the expected values for all the tests.

To change the codeQuestion parsing, for example to add a possible label, the "ExecutionManager" class would need to be modified.

A.2.2.5 Extending the main execution

The main execution is completely managed by the "ExecutionManager" class, thanks to the procedure of spawning a parallel hidden process to compile and test user solutions.

This is achieved thanks to the .NET namespaces **System.Diagnostics** that let us create new processes and **System.Threading.Tasks** that let us wait asynchronously for a result, see Figure A.5.

From here, modifications regarding which processes to spawn can yield any desired result, for example extending the commands to work on other platforms, see Figure A.6.

Make sure that modifications to the process spawning process are secure and contained, since this part steps out of the Unity controlled environment, to avoid system problems.

A.3 Troubleshooting

Actions that can be taken to tackle each problem are written in **bold**, while additional information is displayed in normal text.

A.3.1 Players & Clients

A.3.1.1 There is no place to insert my user credentials

Only a client build will display the login menu.

Make sure that the build is a client build and not a server build.

When DubitaC is run, the first screen will contain a button with either "Continue as Server" or "Continue as Player" written on it, to be able to play the latter message should be displayed.

A.3.1.2 I cannot create a new account

It is possible to create a new account only if the credentials requirements are met:

- The username should be between 1 and 20 characters long.
- The username cannot be already in the database.
- The username cannot contain commas or whitespace.
- The password must be between 8 and 20 characters long.

Make sure that all requirements are met when inserting new user credentials.

A.3.1.3 I forgot my user credentials

DubitaC does not store passwords in a retrievable way.

In case of forgot password:

- **Ask the server admin to manually delete the relevant entry in the database and save the number of points that were earned.**

- **Recreate the account by making sure to check the "Create new account" checkbox and inserting the new username and password.**
- **Ask the server admin to manually modify the new entry in the database by inserting the old "points" value in the correct column.**

In case of forgot username: nothing can be done to retrieve the points that were earned, **create a new account**.

A.3.1.4 After pressing "Play" the connection takes a long time and fails

To be able to press "Play" a player has to insert a valid Ipv4 address in the relevant field, however, there is no guarantee that a server will be ready to listen on the other size.

Make sure that the inserted Ipv4 address is the same as the one that the server admin has shared.

Additionally there might be connection problems on the local machine.

Make sure that firewall and antivirus do not interfere with the outgoing connections.

A.3.1.5 I cannot select some of the Avatars

The Avatar selection is tied to the progression with the game, some Avatars will be obscured and non-interactive until a cetrain threshold of required points is reached.

Play more to unlock more Avatars.

A.3.1.6 I could not choose my Avatar in time

A random Avatar will be assigned.

If this happens frequently, **ask the server admin to wait a bit longer before starting a game session**.

A.3.1.7 I want to play with a friend but we are never in the same lobby

The lobby rebalancing system might remove some clients from a lobby to place them in another.

Clients that connect later are moved more often, **try to connect as soon as possible**, .

A.3.1.8 I cannot find my cpp file after I pressed on "Export"

The creation of permanent .cpp files will depend on the path that was inserted in the options menu.

If none were inserted, a default path has been used.

Check the directory at path: %AppData%/LocalLow/LorenzoTibaldi/DubitaC

A.3.1.9 The log is not in my chosen language

Because of some implementation limitations, the log cannot be localized in the current version.

What is written in the log are either easy sentences or what the terminal returned on the local machine.

A.3.1.10 The game hanged

To check if the game is hanging, look for moving parts in the interface, for example the loading screen or interactive intermediate leaderboard.

If nothing moves as expected, then the game hanged, **after waiting an appropriate amount of time, close the application since the game will not be able to be resumed**.

If the moving parts work, then the game is simply compiling and executing, **wait until the scene changes**.

A.3.1.11 I think my solution should not have timed out

The game will abort any execution that takes more than TIMEOUT amount of seconds, where TIMEOUT is the value that was selected in the options menu.

If none were selected, a default value of 3 seconds has been used.

In case a valid colution requires more than 3 seconds for an execution, **select a higher TIMEOUT value before starting the next game session**.

The maximum selectable TIMEUOT value corresponds to 9 seconds but if the solution takes longer we suggest **finding the parts of very inefficient code and refactor them**.

A.3.1.12 During the slideshow, a doubt reported that a solution returned "Unknown"

Because of the current dependency on Catch2, there is no way to detect what a function returned when a test expected a crash or timeout.

In the future, it could be possible that a new Catch2 version would offer tests that, even when checking for exceptions, could capture the returned value of a function, at that point it would be possible to extend the result parsing so that "Unknown" will not have to be used anymore.

A.3.1.13 I do not understand the doubt evaluations

The doubts can fall into 4 categories:

- Correct doubt, meaning that the doubter correctly guessed the output of the "perfect" solution and the output of the "wrong" target solution when given a certain input.
- Wrong doubt, meaning that the doubter incorrectly guessed the output of the "perfect" solution and the output of the "wrong" target solution when given a certain input, while the target solution returned the correct result.
- Both wrong, meaning that the doubter incorrectly guessed the output of the "perfect" solution and the output of the "wrong" target solution when given a certain input, while the target solution returned a wrong result.
- Half doubt, meaning that the doubter corectly guessed only one of the outputs between the "perfect" solution and the "wrong" target solution when given a certain input, while the target solution returned the correct result.

The reasoning behind the evaluation is that a doubter to be awarded points needs to be able to guess the correct execution of the codeQuestion and the wrong execution of another player, while a solution that is being doubted will only need to return the correct result and it will be awarded some points.

A.3.1.14 My final position in the leaderboard is wrong

The leaderboard cannot be wrong, remember that there are 3 different orderings of the leaderboard:

- The order of players based on their order of pressing the ready button in the first round.
- The order of players based on the points that they received after testing all the "[user]" doubts.
- The order of players based on the points that they received after testing all the "[final]" tests.

Between the end of the slideshow and the final leaderboard, the order can still change, depending on the final tests that are executed.

A.3.1.15 How many points did I win?

The number of points that players obtain after each game session depend on 2 factors:

- The number of hints bought during the first round.
- The final position on the leaderboard.

There is always a minimum number of points that will be awarded for each game session, but the higher your position in the final leaderboard, the more points will be awarded.

Additionally, in the final menu after pressing disconnect, it is possible to see how many points the user currently has without having to start a new game session.

A.3.1.16 When is it safe to disconnect?

The game saves user progress on the database after having created the final leaderboard, this means that **it is safe to disconnect once the final menu has been loaded**.

A disconection during the slideshow will *not* result in your progress being saved.

A.3.1.17 I was kicked back to the main menu

Clients are sent back to the main menu when they lose connection with the server.

All clientside and serverside disconnections are handled gracefully, but in the case of a clientside disconnection *the game might not continue for the clients that are still connected*, see Section A.3.3.3.

A.3.2 Admin & Server

A.3.2.1 There is no place to select the codeQuestions

Only a server build will display the codeQuestion selection menu.

Make sure that the build is a server build and not a client build.

When DubitaC is run, the first screen will contain a button with either "Continue as Server" or "Continue as Player" written on it, to be able to select the codeQuestion the former message should be displayed.

A.3.2.2 Can I move the clients between lobbies manually?

No, manual lobby rebalancing is not supported in the current version.

A.3.2.3 I forgot to set the available time

A default amount of 600 seconds will be set as the available time.

In case 10 minutes are too long or too short, **close and reopen the application, so that a new game session can be created**.

A.3.2.4 I cannot find user solutions after I pressed on "Export"

The user solutions are saved in the default path.

Check the directory at path: $InstallationDirectory$/DubitaC_Data/Cpps

Where $InstallationDirectory$ is the folder where the game is installed, the one containing the file DubitaC.exe.

A.3.2.5 I cannot find the database

The database should alway be at the default path.

Check the directory at path: $InstallationDirectory$/DubitaC_Data/Database

Where $InstallationDirectory$ is the folder where the game is installed, the one containing the file DubitaC.exe.

In case the database is not in this path, a new empty one will be automatically created the next time DubitaC is run.

A.3.2.6 I accidentally deleted the database

Unfortunately, there is no redundancy measure for the database in the current version, this means that when the database is deleted, all of its information is lost.

Try to restore the database file in other ways.

A.3.3 Shared

A.3.3.1 I want to play DubitaC over the internet

The current version does not support over the internet connections.

Setup a LAN to start playing.

A.3.3.2 Red lines appeared at the bottom of the screen

The red lines represent errror or exceptions in the execution.

If it is possible to continue playing, **Press on the "Close" button that appeard alongside the red lines and continue playing**.

If it is impossible to continue playing:

- **Go to path: %AppData%/LocalLow/LorenzoTibaldi/DubitaC**
- **Copy the file called Player.log**
- **Contact a game maintainer and provide them the log file.**
- **Close the application.**

A.3.3.3 The executions are taking an infinite amount of time

If it became obvious that the game is not proceeding to the next scene, it is most likely not due to the executions continuing infinitely, but because one of the clients has disconnected during some critical networking moments.

The current version cannot recover from such a failure, we suggest to **close and reopen the application to start a new game session**.

DubitaC requires a stable local network connection to ensure a smooth game experience.

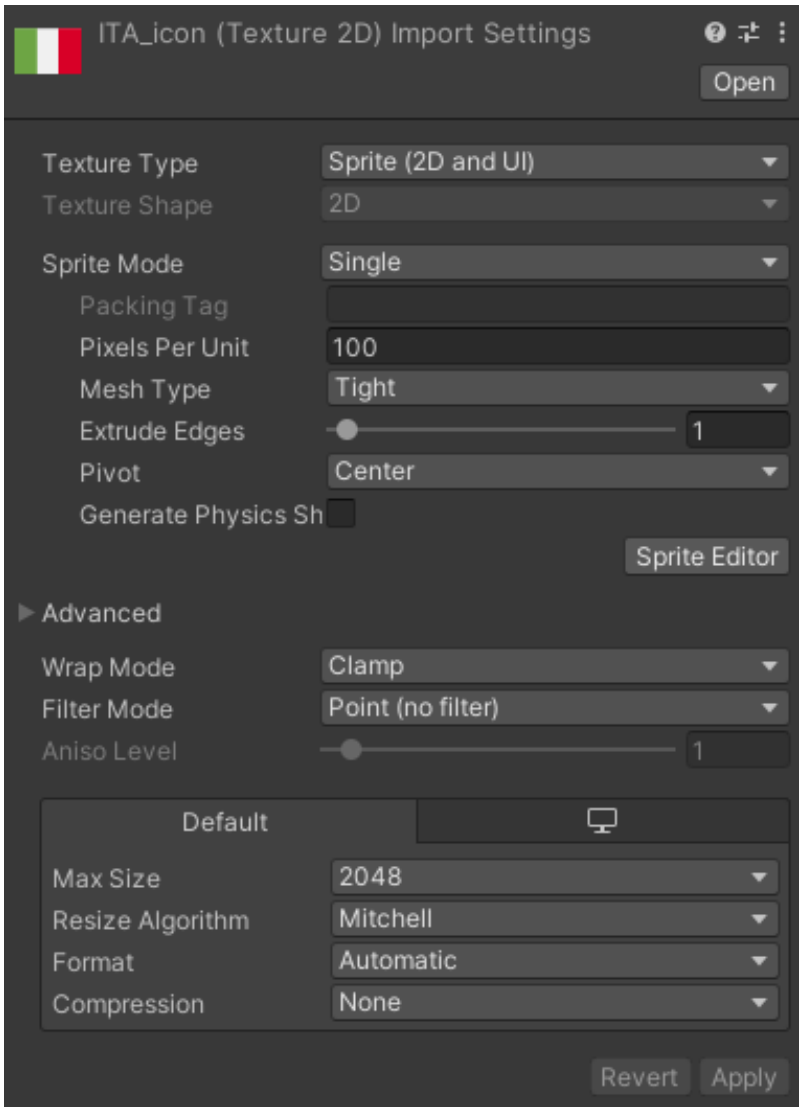


Figure A.2: Suggested parameters to convert a simple png texture into an usable sprite. After choosing these parameters it is necessary to press the "Apply" button at the bottom to save the changes.

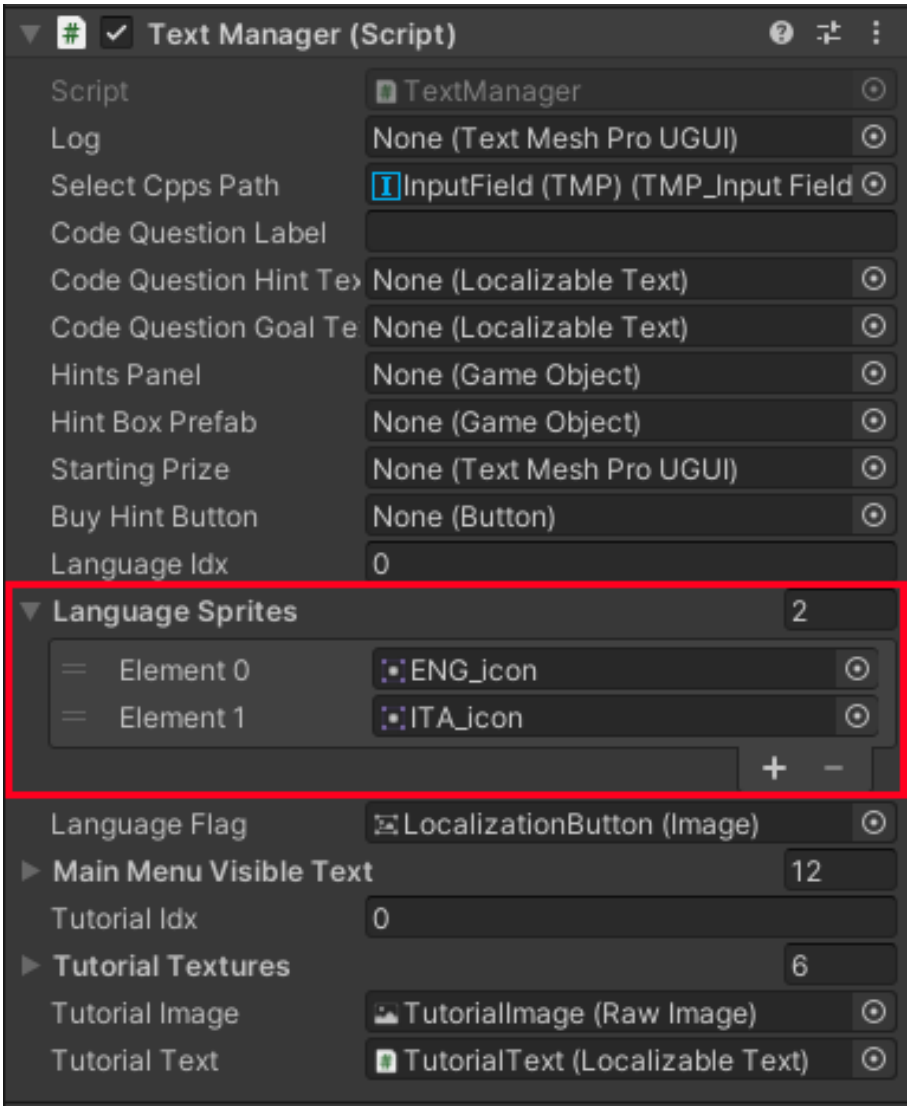


Figure A.3: View from the inpector of the "TextManager" component on the "TextManager" GameObject during the main menu scene. The highlighte red area can be expanded by clicking on the "+" button, this will create a new slot where it will be possible to drag the new sprite.

```
/// <summary>
/// Function required to validate the connection to the server.
/// The connection will be accepted if:
/// The sent username and password are valid OR
/// The sent username is new and the newAccount boolean is true.
/// And refused otherwise, if someone with the same username is already connected to the server then the newest client is rejected.
/// </summary>
/// <param name="connectionData">Array of bytes representing the data sent from client to server to validate the connection.</param>
/// <param name="clientId">Id of the client that requests a connection with the server.</param>
/// <param name="callback">Mandatory callback parameter, it signals to the server the outcome of the validation.</param>
private void ValidateLogin(byte[] connectionData, ulong clientId, NetworkManager.ConnectionApprovedDelegate callback) {
```

Figure A.4: Signature of the function used to validate a client login, shown alongside its summary.

```
/// <summary>
/// Utility function to setup a <see cref="Process"/>, start it, wait for it to finish and dispose of it.
/// The function is 'async' because the execution of the command could take arbitrarely long.
/// The function returns a <see cref="Task"/> because we need the result of the execution.
/// </summary>
/// <param name="executionProgram">Name of the program to start.</param>
/// <param name="command">Command that will be given to the <paramref name="executionProgram"/> as argument.</param>
/// <returns>The result of the execution as it would have been printed to standard output.</returns>
private async Task<string> ExecuteProcess(string executionProgram, string command) {
    Process exeProcess = new Process();
    exeProcess.StartInfo.WorkingDirectory = Application.persistentDataPath + "/";
    exeProcess.StartInfo.CreateNoWindow = true;
    exeProcess.StartInfo.UseShellExecute = false;
    exeProcess.StartInfo.RedirectStandardOutput = true;

    exeProcess.StartInfo.FileName = executionProgram;
    exeProcess.StartInfo.Arguments = command;

    exeProcess.Start();

    string commandResult = await exeProcess.StandardOutput.ReadToEndAsync();

    exeProcess.WaitForExit();
    exeProcess.Dispose();

    return commandResult;
}
```

Figure A.5: Simple function requiring only the name of the process to start and the arguments to give to it. Because of the async implementation, the rest of the execution is not stalled while the process works in the background. The standard output to the terminal is redirected into a string that is returned by the function, so that the terminal output can be easily parsed.

```
    /// <summary>
    /// Function to compile the user solution.
    /// The function is 'async' because the compilation takes some time (usually between 10 and 30 seconds).
    /// The function returns a <see cref="Task"/> because we need the result of the compilation.
    /// </summary>
    /// <param name="fileName">The name of the file to compile, with extension excluded.</param>
    /// <returns>The result of the compilation, if it is not empty something went wrong.</returns>
    public async Task<string> Compile(string fileName) {
        string executionProgram;
        string command;

#if UNITY_STANDALONE_LINUX
        executionProgram = "bash";
        command = "g++ -O3 -g0 -Werror -Wall -fuse-ld=lld catch_main.o " + fileName + ".cpp -o " + fileName + " 2>&1; exit";

#else
        executionProgram = "cmd.exe";
        command = "/C g++ -O3 -g0 -Werror -Wall -fuse-ld=lld catch_main.o " + fileName + ".cpp -o " + fileName + " 2>&1";
#endif

        return await ExecuteProcess(executionProgram, command);
    }
```

Figure A.6: The, currently in use, compilation function, which could use directives like "#if UNITY_STANDALONE_LINUX" to execute different commands based on the detected platform. Unfortunately we were not able to test if using this command in bash would let us play DubitaC correctly on Linux platforms.