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/

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

March, 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.

Particularly, between the many possible activities that can be conducted, serious games can prove useful tools to maintain the students engaged and to help them understand the basic concepts of programming.

As serious games have been implemented and put under scrutiny in different fields, it is not unreasonable to think that they could be successfully employed to support the teaching of introductory programming courses.

Students are often demotivated when taking their first steps in the programming world since programming requires them to learn both a different way of thinking and the specific syntax of their first programming language. A serious game that would, by using all the techniques that are employed by games, keep students engaged long enough to acquire the basics of both would reduce the number of students that abandon their studies and, hopefully, put them on the right path to become competent programmers.

Specifically, we wanted to create a game where students receive a programming exercise to solve by writing code, then, by introducing a multiplayer component, students are tasked to analyse and understand the solutions created by the other students that took part in the game. From there, the students are tasked to find mistakes in the solutions they have read and are rewarded when correctly doing so.

Such an approach would be beneficial to teach students both to avoid creating low-quality code and to understand code written by someone else, which we believe to be extremely important foundations for aspiring programmers.

Furthermore, since we believe that such a game should be practical and should not require

constant supervision by an instructor, we believe a mostly automated computer game to be a perfect candidate for a concrete implementation.

In this thesis, we describe our work to create **DubitaC**, a serious game aimed at helping students to learn introductory programming concepts 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]

The current development using the Unity game engine has reached version 1.3.0 and is available for Windows PC on the project's GitHub repository:

`https://github.com/WeLikeIke/DubitaC/tree/v1.3.0`

1.1 Structure of the thesis

Following this introduction, the thesis is divided into 4 chapters:

- In Chapter 2 we explain the state of the art and literature on gamification and serious games, followed by examples of games that can be considered programming games and their shortcomings.
- In Chapter 3 we describe our main goals and present an overview of the game that we invented.
- In Chapter 4 we give a complete overview of choices, implementations, and challenges that we had to overcome to develop the DubitaC computer game.
- Lastly, Chapter 5 contains some possible future extensions and a closing statement about the project's journey.

Additionally, a reference manual of the game is provided as appendix A.

Chapter 2

Serious gaming and gamification

In this chapter we first present the results obtained by the research conducted on gamification and serious games, then we highlight some successful examples of gamification being employed in different domains.

In Section 2.3, we present some examples of products that can be considered games about programming, although that does not necessarily mean that they teach about a programming language that could be used in the real world.

2.1 Literature and state of the art

2.1.1 Gamification

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

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

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 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 two different types of gamification elements:

- Self elements.

- Social elements.

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

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

Additionally, in [SRM+20], researchers were able to correlate, on their study group, an overall improvement in all categories for subjects that would be defined as 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.

2.1.2 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 is not a priority during the product's development.

A first recognition of the importance of using games for training can be found in [Abt87], where the author remarks how teenagers would be less averse to learning if it was presented within a framework similar to competitive or cooperative games and that adults would benefit from it in the same way, for example for the military training of soldiers.

Additional surveys [SJB07][LEES14] compiled a taxonomy of serious games and highlight their wide range of application domains while comparing some of the already existing products and exposing the state of the market at their time of writing.

2.2 Gamified products

Chapters 2 and 4 of [Boh15] present a wide variety of products that 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 programming challenges to be solved using a language of choice among many possible supported ones.

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.[vH11]

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

We also want to remark how some of the previous examples employ 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 encourage the user to interact with other parts of the community.
- An optional clan system that lets users easily follow other users' progress, introducing a social aspect to the platform.
- The only interactions between users happen voluntarily with votes and comments but there is no obvious competitiveness in the system.

It should be pointed out that Code wars does not employ any gamification elements aimed at increasing user enjoyment directly. This is by design since Code wars does not aim to be an entertaining videogame.[Cod22]

2.3 Programming games and their limitations

In this section, Figures 2.1-2.4, referring to the games mentioned below, are taken from their respective Steam pages.[1]

The game studio Zachtronics created interesting and engaging programming games. Here we shortly introduce some of them.

TIS-100, released in 2015, asks the user to learn how to creatively solve programming problems in assembly. [Zac15]

An example game screen is shown in Figure 2.1.

On the left side of the screen, from top to bottom, the player has access to the current goal, inputs and outputs and an execution console, while in the rest of the interface, 12 cores are shown, each with their own space to assign assembly instructions.

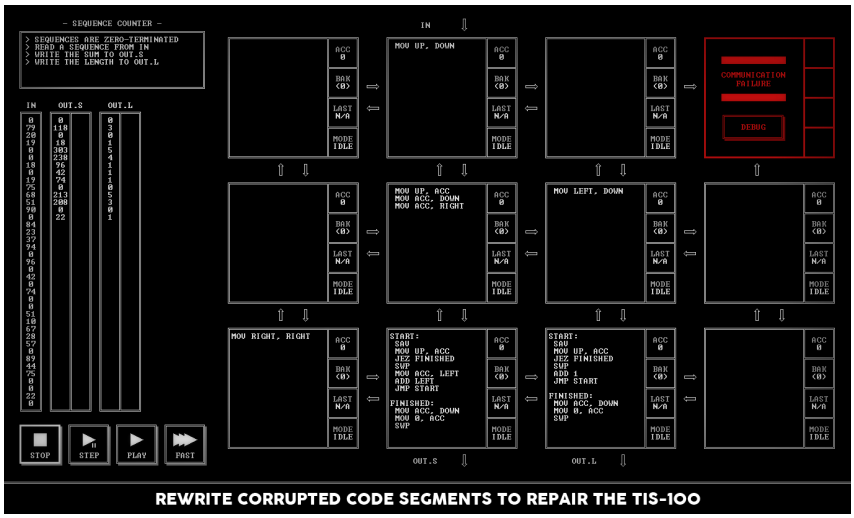


Figure 2.1: Example game screen of TIS-100.

In 2016, Zachtronics moved away from a real-world programming language. Their game SHENZHEN I/O asks the user to learn some circuitry basics and an ad hoc programming language closely resembling assembly. [Zac16]

An example game screen is shown in Figure 2.2.

On the top part of the screen, the main circuit is shown, where each component can be customized with pseudo-assembly code. On the right, the player can buy components and integrate them into its solution and on the bottom, an execution console is shown alongside

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

the circuitry inputs. The current goal can be found on a separate panel and cannot be currently seen.

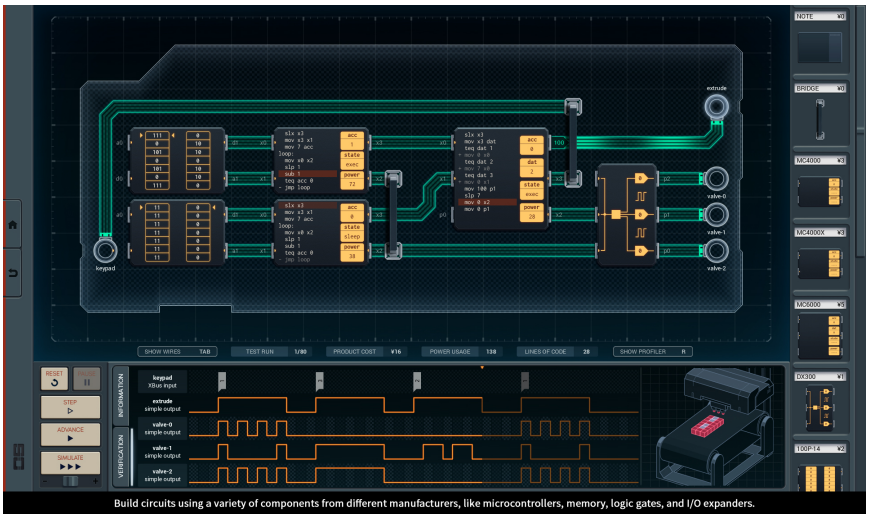


Figure 2.2: Example game screen of SHENZHEN I/O.

Lastly, in 2017, Zachtronics 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]

An example game screen is shown in Figure 2.3.

On the left side of the screen, 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 centre of the interface, the playing field is shown, where the player can place the bought components. Lastly, at 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 movements of the pistons, as it is about creating an efficient factory layout.

Another example, that was released in 2018 by the Tomorrow Corporation game studio, 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 multithreaded environment. [Cor18]

An example game screen is shown in Figure 2.4.

On the right side of the screen, 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 fig-

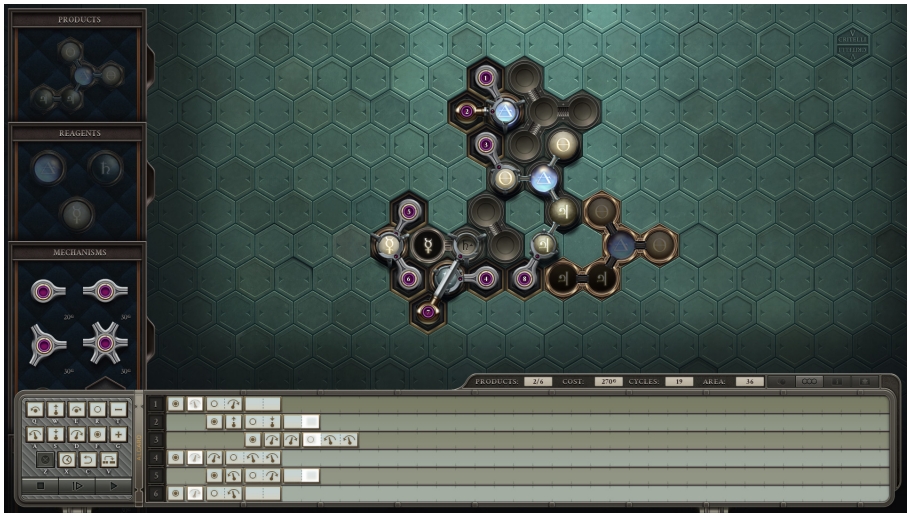


Figure 2.3: Example game screen of Opus Magnum.

ure, for example, requires that all green datablocks are picked up and dropped in the hole at the bottom. The level contains seven "humans" following the user commands which reduces the needed complexity of the solution.

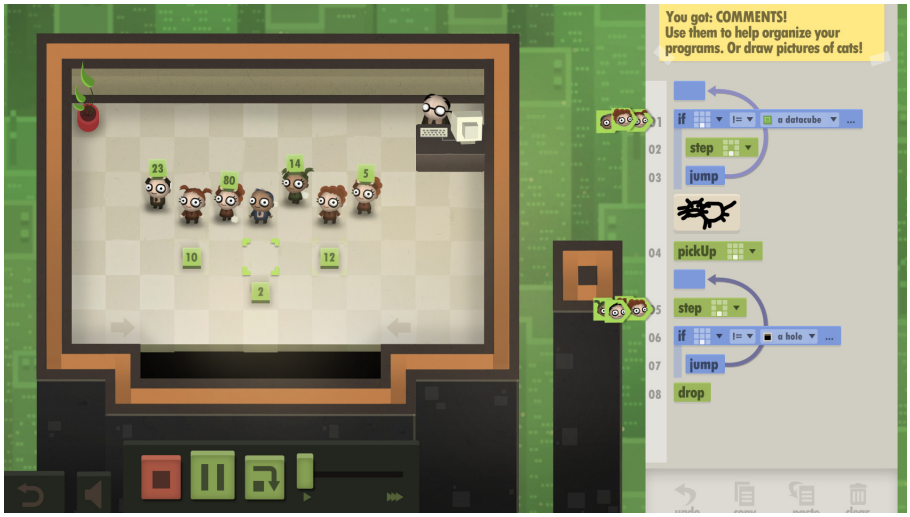


Figure 2.4: Example game screen of 7 Billion Humans..

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 toy language to be used in the controlled environment of a game, instead of wrapping an existing language in a game system.

While it is possible to argue that 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 our game in an educational setting.

Chapter 3

Creation of the game

In this chapter, we detail the initial goals and the abstract description of DubitaC.

3.1 Main goals

We decided to develop a serious game to support students that are attending introductory programming courses.

Our main use scenario corresponds to a university laboratory session, where students are given programming exercises and are left on their own while solving them with the occasional help from instructors.

Our main goals were:

- Helping students to learn introductory programming concepts using a real programming language.
- Keeping students engaged, especially the ones that might struggle and quickly lose interest in the subject.

The students should learn that understanding the requirements of a programming exercise and taking their time to create good and correct code are fundamental principles of programming that should be internalized.

At the same time, we want students to learn how to read and understand code written by others, which is a skill that will be very useful to have during their programming careers.

With these goals in mind, we created DubitaC.

3.2 Game overview

3.2.1 High level game description

The game is played in multiplayer self-contained game sessions between a minimum of two players to a maximum of 24 players which are divided into four lobbies of at most six players each. Additionally, an admin is required during the game session.

The main flow of the game, shown in Figure 3.1, is subdivided in four main phase:

- Setup
- Solution creation
- Doubting
- Final evaluation

Let us briefly describe the activities taking place in each of them.

During the setup:

- The admin selects an appropriate programming challenge for the players, considering their current programming knowledge, and the time at their disposal.
- At the same time, players can choose an avatar that will represent them during the game session between the available ones.
- Lastly, the admin shares all details regarding the selected programming challenge to the players.

During the solution creation round:

- The players create a solution to the given challenge by writing code under a time limit.
- At any point, the players can test their solution against a basic test battery.
- Struggling students can ask the admin for a limited amount of predetermined hints to help them create their solution.
- Players that are satisfied with their solution can notify the admin that they have finished by submitting their solution.

- At the end of the round, the admin collects all player solutions and shares them with the other players in the same lobby.

During the doubting round:

- An intermediate leaderboard is shown to the player, containing the order in which the players have completed their solution in the previous part.
- The players analyse the solutions of the other players in the same lobby.
- If a mistake is spotted, the player can create a doubt against the target solution. Only one doubt is allowed from each player towards each solution. Each doubt on another player solution execution consists of three parts: the input, the expected erroneous output by the given solution and the correct output a reference solution should provide on that input.
- At the end of the round, the admin collects all player doubts.

During the final evaluation:

- For each doubt, the admin creates two tests: the sanity check test, asserting that on the given input the reference implementation yields the correct output, and the doubting test, asserting that on the given input the doubted implementation yields the expected incorrect output.
- The admin creates a test battery containing all sanity check tests and evaluates it against the reference solution.
- The admin runs the doubting tests on the doubted solutions.
- The admin collects all test results.
- The admin evaluates all player solutions against a final test battery and calculates the final points for each player.
- The final leaderboard is shown to the players, containing the order of the players based on the scorekeeping of the previous step.
- The game session concludes with the awarding of points to the players depending on their position on the leaderboard and the number of hints that each player received.

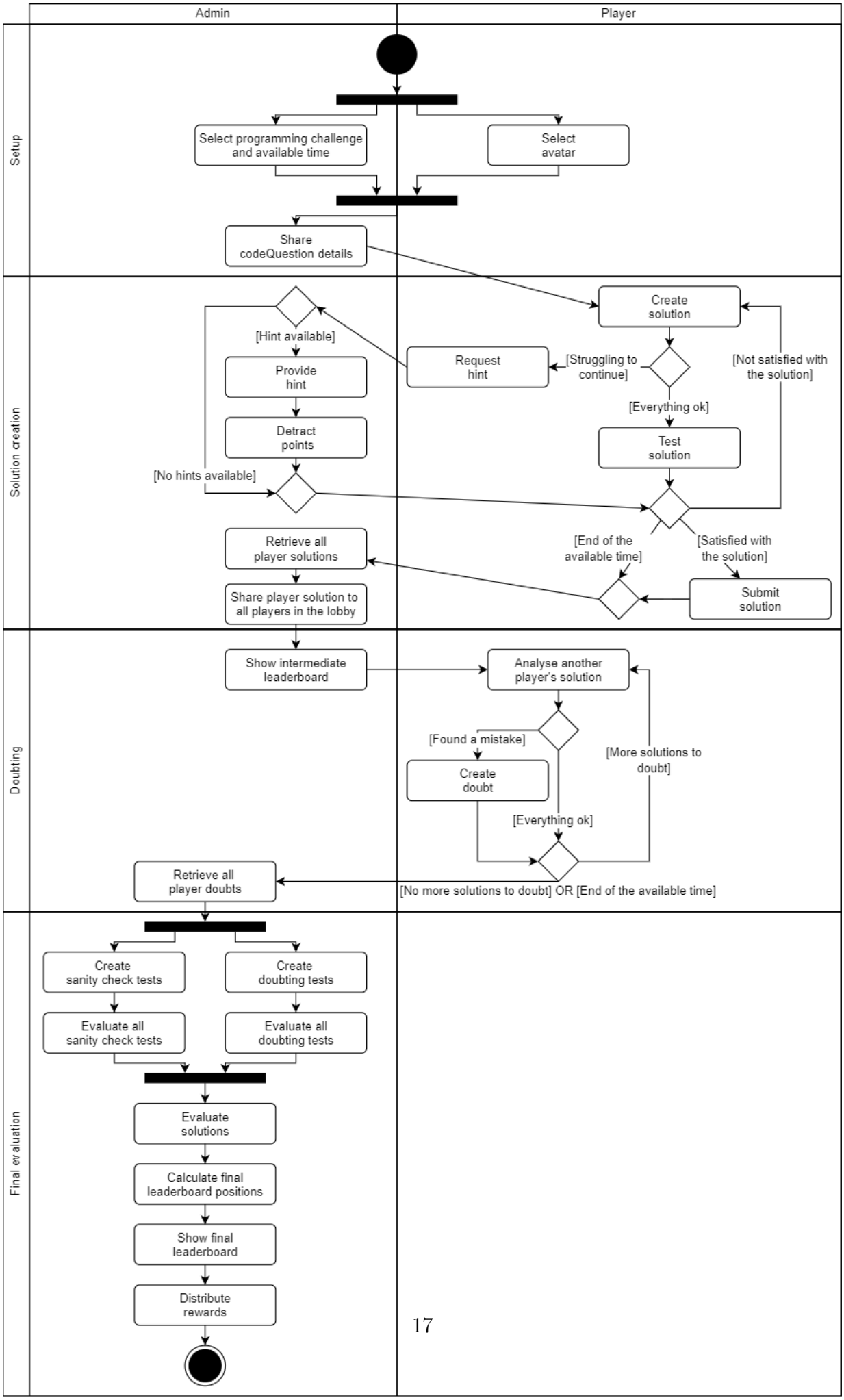


Figure 3.1: Activity diagram of the high level game flow.

3.2.2 Finer details

3.2.2.1 Introducing the programming challenges

The programming challenges to be used in the setup are the core of the game. They are predefined so that the admin can simply select one of them to start the game.

Each programming challenge requires the implementation of a C++ function[1] and consists of:

- The description of the challenge, which is the request that the players need to fulfil by writing code, for example, "Print Hello world!".
- The function signature, consisting of the return type, the function name and the function parameters.
- The reference solution to the challenge, which is assumed to be correct.
- The basic tests that are provided to the players to help them refine their solutions.
- The final tests used to evaluate the submitted solutions.
- The list of available, challenge specific, hints.
- The list of restrictions on the acceptable inputs, if any, representing preconditions on the actual parameters for the function to be implemented.

Each of the above can be categorized, depending on if the players have access to it or not, like in Table 3.1.

	Shared with the players
Challenge description	√
Function signature	√
Reference solution	×
Baic test battery	√
Final test battery	×
Hints list	Single hints are shared if requested
Input restrictions	√

Table 3.1: Categorization of the programming challenges details.

[1]We chose C++ for our specific use case, but the game can be easily extended to utilize a different programming language.

The input restrictions are part of the challenge definition, being constraints on the function to be implemented and hence must be shared with the players. The input restrictions have an impact on which doubts can be created and doubts not respecting them are not allowed.

3.2.2.2 Introducing doubts

Doubts capture the users' understanding of another player's solution. Players create them to pinpoint mistakes in the code, aiming to gain a great number of points.

Two different types of doubts are available to the players:

- Compilation doubts.
- Execution doubts, which are those discussed earlier.

Compilation doubts are not described in the previous section and do not entail the definition of the tests. In principle, since all players have access to compilers, we could expect all submitted solutions to be syntactically correct. Since teaching experience has shown that this is not always the case, we have also included this kind of doubts, which are simply evaluated by running the compiler on the doubted solution, to drill students into using *all* available tools.

If players correctly identify that a solution does not compile, they can create a "compilation doubt".

To create a correct "execution doubt", instead, players will need to correctly identify which inputs are problematic for the target solution. The user is given a choice between:

- The solution returns a value X that is wrong.
- The solution goes into an infinite loop.
- The solution crashes or terminates unexpectedly.

At the same time, the players will need to provide the correct output, proving that they understand the requirements of the given challenge.

3.2.2.3 Evaluating performance

The evaluation of the players' performance encompasses their results in all phases of the game.

The challenge solving part works like a Code wars kata[2]:

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

During the solution creation round, the players can compile and test their own solution against a basic test battery that contains very simple tests, which do not consider any edge case, without incurring in penalties.

The evaluation of the correctness of the solutions during the final part is conducted on a final test battery that includes the edge cases that the players have to recognize to solve the challenge correctly.

Lastly, the evaluation of the player doubts is conducted on a separate test battery which will be different for each player solution.

3.2.2.4 Calculating points

During the final evaluation step, the players are placed on the leaderboard depending on the results of the evaluation of their solution against the final test battery and the player doubts targeted at them.

Starting from the intermediate order after both evaluations have been completed, the players are:

- Awarded points:
  - For each final test that their solution has passed.
  - For each doubt that they created that was correct.
- Detracted points:
  - For each hint that they have requested.
  - For each final test that their solution has failed.
  - For each doubt that they created that was incorrect.
  - For each doubt targeting their own solution that was correct.

A "compilation doubt" is considered correct when the target solution does not compile and incorrect otherwise.

[2]Taken from the Japanese word used to refer to individual exercises in martial arts, Kata is the name that Code wars uses to refer to their programming challenges.

An “execution doubt” is considered correct when both the sanity check test and the doubting test on the target solution have passed and incorrect otherwise.

For the final leaderboard, the players are ordered in descending order of points, with the first place being the player that earned the most points in the lobby.

In general, the players are incentivized to create as many correct doubts as possible, since each correct doubt improves their position in the leaderboard while, simultaneously, lowering the positions of the doubted players.

Because of this doubting component, players are taught that quickly writing low-quality code is very detrimental since other slower players will overtake them on the leaderboard after having doubted their solution.

At the same time, we want to teach students to understand their peers' code instead of just creating as many doubts as possible. For this reason, each wrong doubt will lower the position on the leaderboard of the doubter instead of the doubted player.

3.2.2.5 Time considerations

The two main rounds of the game are time-limited because we wanted to maintain the length of a game session reasonable.

Specifically, the decision of splitting the players into smaller lobbies was motivated by the necessity of limiting the waiting times to avoid boredom and to reduce the number of solutions that each player can analyse.

Additionally, by introducing time limits on both rounds, we obtain two side effects:

- Since the first round is time-limited, the programming challenges should be small and contained, inducing students to focus on key aspects of elementary programming constructs.
- Since the second round is time-limited, such time limit might be too strict for some players and some strategic components regarding which players to doubt can arise, for example, “Is it better to doubt players lower in the leaderboard hoping that their solutions contain glaring mistakes, or doubting players at the top of the leaderboard hoping that they can be overtaken once the doubts have dragged them down?”.

If during experimental tests of the game, such strategies are deemed problematic and too frequent among players, it is possible to adopt some countermeasures. For example, giving more weight to the number of points gained by a correct doubt against players near the top of the leaderboard.

3.2.2.6 Engagement considerations

As a serious game, DubitaC contains elements typically found in games, for example:

- A points system.
- A series of avatars to choose from, that are publicly shown to the other players.
- A hint system, that is not too punishing and kept private.
- A multiplayer competitive environment.

To link more strongly the first two elements, we decided to introduce a minimum threshold of points required to be able to select some of the avatars. The motivational advantage of doing so are multiple:

- As the points system becomes important outside of the single game session, players will be more willing to try and reach the top of the leaderboard to obtain more points.
- Since points are now directly tied to the user progression, the avatar availability becomes tied to the user progression as well, incentivizing players to try to unlock more "expensive" avatars.
- Since the unavailable avatars are obscured, the players' curiosity is leveraged to drive them to play more and discover what the locked avatars look like.

To compile Table 3.2 we compared DubitaC to its equivalent educational setting, a university laboratory session where students are asked to complete a programming challenge.

Our considerations depend heavily on the personality of every single student, meaning that they cannot be broadly applied to every user expecting the same positive results.

User enjoyment, which refers to elements that would make the game "more fun", is targeted by:

- The "fun" competitive aspect of the multiplayer component.

User retention, which refers to elements that increase the players' motivation to play more, 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, which refers to elements that improve the learning experience making it "easier" or "deeper", is targeted by:

- The hints system, which is forgiving and kept private to encourage struggling players 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
Points system	×	√	×
Public avatars	×	√	×
Private hints	×	×	√
Multiplayer competition	√	√	√

Table 3.2: Expected improvements caused by the different game elements in DubitaC.

Chapter 4

The DubitaC computer game

In this chapter, we will detail how we have designed and implemented a computer game version of DubitaC.

Our main goal regarding DubitaC as a computer game was to reduce the workload of the instructors, in the role of admins, during a game session.

Specifically, by automating the majority of the game, instructors will be free to supervise what their students are doing and be able to gather some indirect feedback that can be implemented to improve future lessons.

At the same time, the automation of the game sessions reduces waiting times, making the game more enjoyable for the students.

Since our target environment was a laboratory setting, we decided to implement DubitaC as a distributed computer game where the players and the admin will play each on their own machines which are connected between them over a network.

For the development process, we decided to use the Unity [HFA04] game engine, guided by our familiarity with the tool and to avoid low-level implementation details like rendering and allocation/deallocation, for example.

Two main parts of the game, the management of a distributed environment and the rapid testing of player solutions, required some initial evaluation to decide what would be the best way to tackle them.

Additionally, we developed a graphical interface for DubitaC so that the visual feedback to the players would engage them during their challenge-solving parts and to offer a simple way to manage the client connections and the game session setup.

In the following sections we will describe:

- The options available, the motivations and final decisions regarding the main parts of the game that we mentioned previously: the distributed environment and the testing of solutions.
- The main implementation parts of the creation of the concrete computer game:
  - The subdivision of the main gameplay in Unity scenes.
  - Considerations on the role of the admin and the automation of the game sessions.
  - Considerations on the freedom given to the players during the solution creation.
  - The implementation of the generation of testable executables starting from the player solutions.
  - The implementation of the management of the players' progress and avatars.
  - The implementation of the programming challenges.
- The implementation of a Windows installer.
- The status of the development at the time of writing.

4.1 Implementation choices

4.1.1 The distributed environment

Because we wanted to develop DubitaC as a distributed game, 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.

Since we had no experience in establishing over-the-network communication in Unity, we decided to take advantage of a new experimental package inside Unity: Netcode for GameObjects.[Tec20]

The package offers the basics of a Server-Client architecture with only two ways to interact between 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 synchronized over the network.

The package offers the option to ensure that the communications are 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.

DubitaC was developed using version 1.0.0-pre5 of the Netcode for GameObjects package but, because of its ongoing development, its API is being updated frequently. Future maintainers might have to modify some of the game's functions that might no longer correctly interface with the package internals.

As a side remark, we had to implement a lower-level extension of a couple of functions that the package utilizes for the serialization of data to be sent over the network. 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`

But, at the time of writing, the issue has neither been tackled nor resolved yet.

4.1.2 The testing

The testing procedure is at the core of the entire gameplay but its execution is not trivial. 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.

Spawning new processes and managing their pooling and termination would give us a more efficient general execution flow.

However, the requirement to make sure that every execution would be carried out completely and their result correctly sent back to the main process would make the code more complex than needed.

Instead, we decided to streamline the testing procedure by taking advantage of a test framework for C++ programs: Catch2.[Hoř15]

We chose this framework because of three 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, containing all the code required to make use of all Catch2 functionalities, at the top of the ".cpp" file to test.

After writing some tests, Catch2 will take care to create an alternative main at compile time which will make sure that each test is run in a completely "clean" environment and each environment is completely disposed before running the next test.

Lastly, at runtime, the executable would run the alternative main and return to the terminal the results of all the tests.

Because of the framework-specific test syntax, each test case must be created ad hoc from the users' doubts.

DubitaC was developed using version 2.13.8[1] of the Catch2 test framework, which is the latest stable release at the time of writing. A new version 3.0.0 is being actively worked on and future maintainers might want to migrate to this major release once it will be considered stable.

4.2 From abstract to concrete

In this section, we will detail the necessary steps that we took to create DubitaC in its computer game form.

We will refer to the programming challenges as "codeQuestions" from this point onward.

[1] Available here: `https://github.com/catchorg/Catch2/releases/tag/v2.13.8`

4.2.1 Subdivision in scenes

In the context of Unity development, scenes are used to separate both the graphical elements and the game logic of different game steps. When a scene is loaded, first all the components of the previous scene are unloaded and then, after the unloading process has finished, the graphical elements of the new scene are instantiated and the scripts containing the game logic are started.

Following the description of Section 3.2.1, it would be natural to implement 4 scenes, each representing one game phase.

Upon further inspection, however, we decided to split the final phase into two separate scenes: one consisting of a slideshow showing to the players all the doubts that have been created by the lobby and another consisting of the final evaluation and rewards distribution.

Figure 4.1 shows the split that we performed on the "Final evaluation" phase of the original activity diagram.

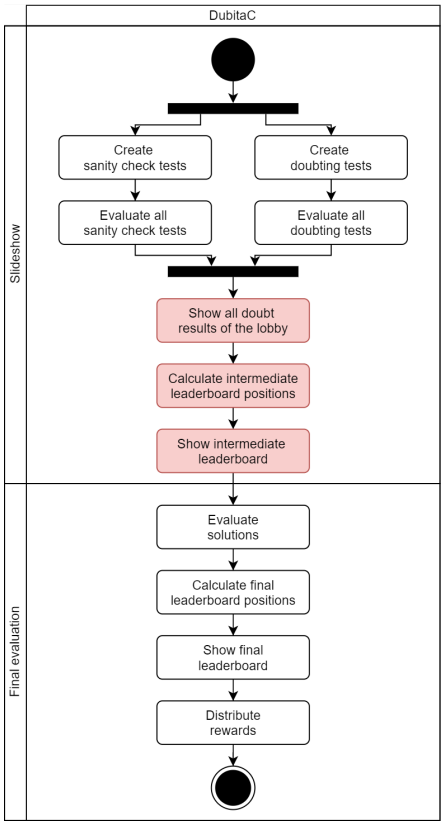


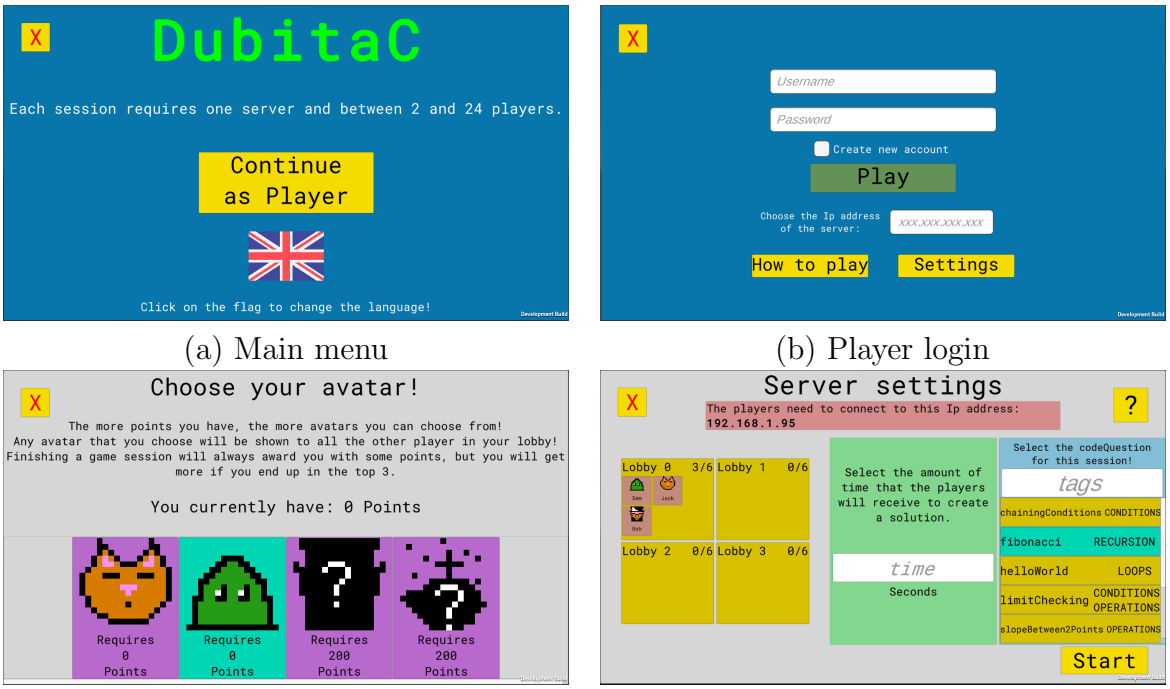
Figure 4.1: Activity diagram of the last phase that has been split in 2 phases: "Slideshow" and "Final evaluation". The red actions were not present in the original activity diagram.

The reason behind this choice lies in the "private" nature of the doubt creation and execution. To make sure that the players are exposed to as many doubts as possible and, at the same time, making sure that they are notified of the result of each doubt, we implemented a slideshow clearly indicating:

- Which player created the doubt.
- Which player is being doubted.
- What the doubt expected.
- In case of a "compilation doubt":
  - If the target solution compiled.
- In case of a "execution doubt":
  - What the target solution returned when given which inputs.
  - What the reference solution returned when given the same inputs.
- A final evaluation of the doubt, indicating if the doubt was correct or wrong.

Additionally, we decided to show an intermediate leaderboard between the evaluation of the doubts and the evaluation of the solutions against the final test battery to highlight which players performed best during the doubting round and to remind the players that the final leaderboard will also judge their solutions meaning that the leaderboard might change once again.

Each scene contains various interfaces that the players and the admin will interact with. Figures 4.2-4.6 show the most important interfaces currently present in DubitaC.



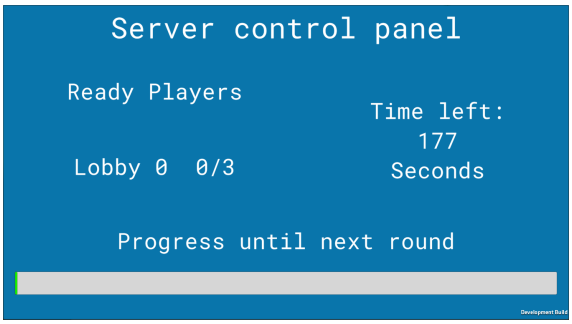
(a) Main menu

(b) Player login

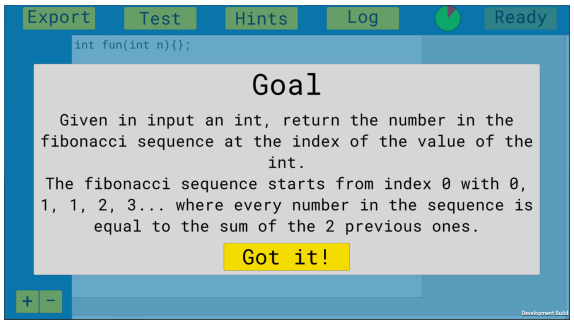
(c) Avatar selection

(d) Game session setup

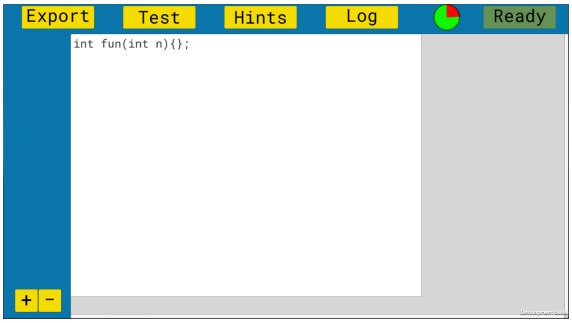
Figure 4.2: Interfaces found in the "Setup" scene. Interfaces (a), (b) and (c) are interfaces shown to the players while interfaces (a), with a different label in the central button, and (d) are shown to the admin.



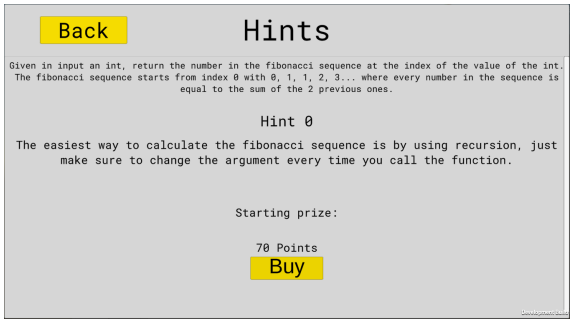
(a) Game session monitoring



(b) Current goal panel

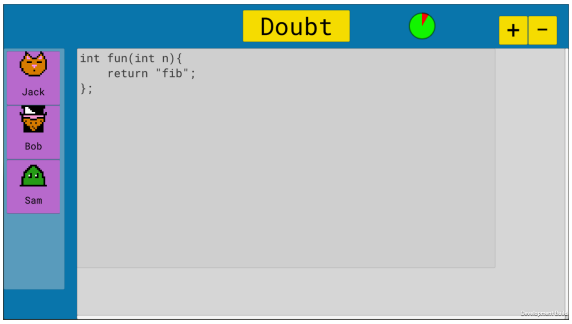


(c) Solution creation

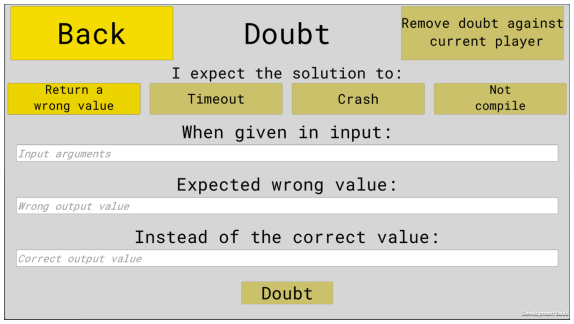


(d) Hints panel

Figure 4.3: Interfaces found in the "Solution creation" scene. Interface (a) is shown to the admin while interfaces (b), (c) and (d) are shown to the players. Various similar interfaces to (a) are shown to the admin during all the subsequent scenes.



(a) Doubting



(b) Doubt panel

Figure 4.4: Interfaces found in the "Doubting" scene. Both interfaces are only shown to the players.

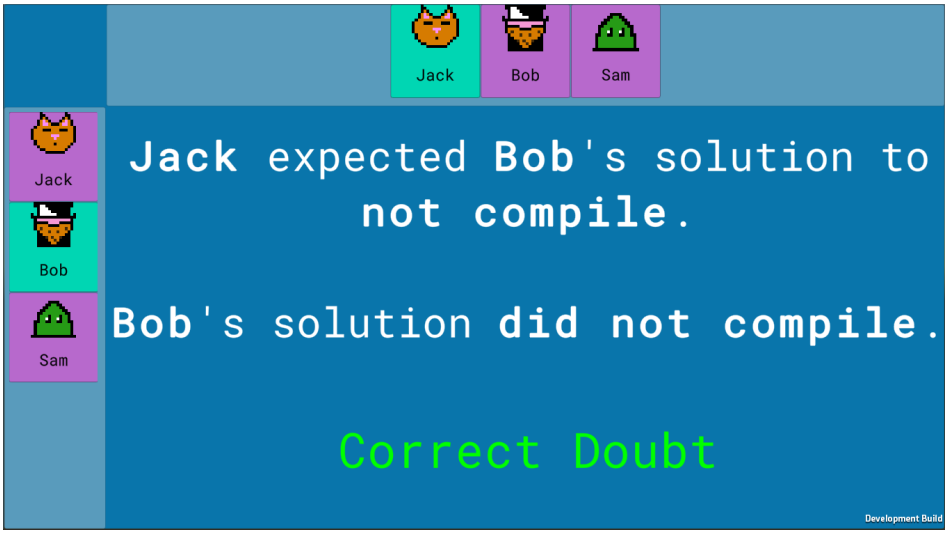


Figure 4.5: Slideshow interface found in the "Slideshow" scene. The interface is only shown to the players.

Figure 4.6: Final leaderboard found in the "Final evaluation" scene. The interface is only shown to the players.

4.2.2 The role of the admin

In our abstract description of DubitaC, the admin was a central role that ensured the communication between all players of a lobby.

In our concrete implementation, following our goal mentioned at the beginning of the chapter, the role of the admin has been reduced considerably and is now only tasked with setting up the initial codeQuestion.

Note, for example, how Figure 4.1 describes the activities executed by DubitaC, instead of the admin, since all activities in those scenes have been completely automated.

Since the codeQuestion setup does not require an expert to be completed, the admin role could be assigned to a junior instructor while more senior instructors can attend to more important tasks, like identifying the most common mistakes of the players' solutions to be able to conduct a wrap-up session at the end of the game.

Figure 4.7 shows the original game flow and highlights which actions have been automated in the current implementation.

Furthermore, it could be possible to extend the game so that no admin interaction would be required by:

- Introducing the amount of time available to solve a codeQuestion as a codeQuestion detail.
- Randomizing the choice of a codeQuestion between the available ones.

By offering both methods, it would still be possible to play with an admin setting up the appropriate codeQuestions and, at the same time, make players that want to play in complete autonomy able to do so. The only requirement for the autonomous version would be a running server to accept the connection of the clients.

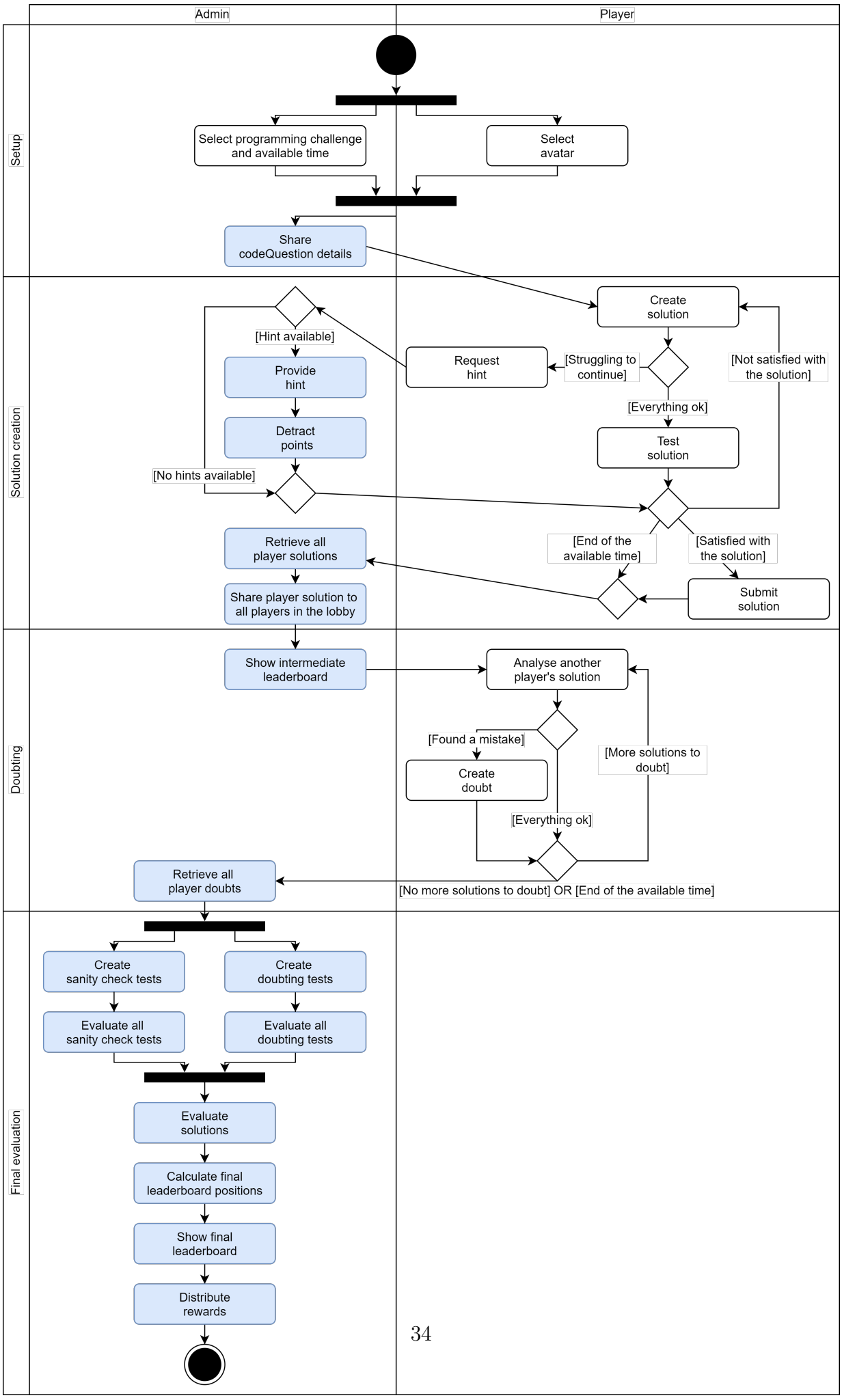


Figure 4.7: Activity diagram of the abstract version of DubitaC. The actions highlighted in blue have been completely automated in our implementation.

4.2.3 Considerations on the sandbox aspect of solution creation

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

Since the correct format depends on the test framework, we had to ensure that we could test any compilable player solution without exposing the specifics to the players. We decided to automatically generate the testable solutions from the player solutions by appending some relevant code that would let us proceed with testing, no matter the specific code that the players created.

This sandbox approach lets us save, compile and execute any player solution, provided that the players are able to write a compilable C++ solution, while at the same time, letting us take advantage of the chosen test framework.

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 undesirable behaviour is occurring.

We were able to implement a timeout that would work with any player solution, provided that they do not change the starting function's signature, meaning that we could give almost complete freedom to the players during the creation of their own solutions.

4.2.4 Creating a valid Catch2 program

Catch2 prides itself on 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:

- Adding the Catch2 header to the possible libraries that the g++ compiler can link.
- Including the Catch2 header file in the source code of all the files that contain tests (not necessarily the same files containing the functions to be tested).
- Either creating a new file or extending the existing target file with a Catch2 directive that would signal to the compiler where the tests are located.

Regarding the first point, during installation, we provide a folder with all the necessary files to correctly compile the user solutions, refer to Section 4.3 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 our test cases 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 contains the tests and it will create the alternative main accordingly during compilation.

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.

To concretize the abstract creation of the player doubts delineated in Section 3.2.1 during the doubting round, we needed to create a test for all the possible doubt types described in Section 3.2.2.2.

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.

Except for the first type, there is no Catch2 function that could be used to easily test these doubts and additional measures had to be adopted.

4.2.4.1 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 be able to test them in the first place, as such, it is possible to intercept the result that would be displayed to the standard output after the execution of the compilation command and parse it for our needs.

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

However, a doubt that expects a solution to not compile when the target solution compiles correctly, instead, will be lost unless a test is created for it.

For this, we create a dummy test that will always succeed so that the corresponding doubt is not lost when parsing the test results.

4.2.4.2 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 an arbitrary program will terminate when given an arbitrary input with no additional restrictions.

Since the problem of non-termination cannot be solved, we shifted our focus to detect if an execution terminates, or not, given a time limit.

This timeout detection can be considered equivalent to the non-termination detection since the codeQuestions that the students have to solve are simple enough that an execution should not even take a second.

The general approach consists of creating a wrapper function that implements the timeout using two different threads and then, using macro expansion, the wrapper can be adapted to work on any arbitrary solution.

The preliminary implementation of our timeout follows these steps:

- Creating a C++ wrapper function that takes advantage of a separate thread to execute the function to be tested. The new thread will be aborted by the main thread after a predetermined amount of seconds if the execution has not terminated until that point.
- Stripping the wrapper of any specifics of the function to be tested and the timeout amount.
- Substituting the stripped keywords with placeholder macros, making sure to use different labels for the various components.

We refer to the resulting function as **generic wrapper** since every function-specific component has been substituted by a generic label.

As an example, the first line of the generic wrapper is:

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

Where every label, represented by the format "//_label_name_//", refers to a specific component of an arbitrary function signature.

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 represents a user-definable amount of seconds that the function will be allowed to run before being aborted and, since it does not follow the format, is not a label.

Additionally, the format of the labels has been chosen purposefully to include the double forward slashes that represent the start of a comment in C++ files. In this way, a compilation will always fail if the wrapper has not been configured correctly since the comments would interfere with the file's syntax.

Once the generic wrapper is ready, it is possible to consider its integration in the testing framework:

- Given a user created solution to a codeQuestion.
- Modify the generic wrapper by substituting all labels with their function-specific counterparts.
- Append the obtained wrapper, which we refer to as **specific wrapper**, to the user solution so that it will be included in the compilation.

Lastly, if the tests for the user solution refer to the original function, the specific wrapper will not be executed and the timeout detection will not work. For this reason, every test **needs** to refer to the specific wrapper so that possible infinite loops can be detected and their execution aborted.

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

- Because the function that needs to be tested is the wrapper function and not the original one, when creating new programming challenges to add to the game, the tests will need to be written by a maintainer that must keep in mind such limitations.
- 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.

Figure 4.8 shows the whole generic wrapper, additionally, we will explain briefly all of its parts:

```
//_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 4.8: The whole wrapper used to make sure that any compilable C++ code can be run as a Catch2 executable with a timeout.

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

Inside the wrapper 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 separate 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 form "& name" and separated by commas.

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

After the setup, the thread is immediately started.

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

return __retVal__;
```

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

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

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

If neither happens, the execution was successful and the wrapper function will return the value returned by the target function in the other thread.

The wrapper lets 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.

4.2.4.3 Detecting unexpected 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 4.2.4.2, 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
```

4.2.4.4 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 "execution doubts".
- 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 correct solution of 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 "compilation doubt" expecting the solution to not compile is tracked with:

```
TEST_CASE("clientId", "[user]"){
    SUCCEED(clientId);
};
```

The test will always pass and the doubter with id equal to clientId will be punished, since the solution had to be compiled to pass the test, the doubt was incorrect.

The other doubts, that require a correctly completed compilation, are called "execution doubts".

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

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

In case the test passes, it means that the doubter correctly predicted the output of the target solution.

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

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

In case the test passes, it means that the function threw the exception called "Timeout" which does not exist in the standard libraries, meaning that it was thrown by our timeout procedure.

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

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

In case the test passes, it means that the function threw an exception that is not called "Timeout".

Regarding fatal crashes, as explained in Section 4.2.4.3, Catch2 will handle them automatically, no matter which tests were present.

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

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

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

This is similar to the doubt expecting a wrong return value and, when it passes, indicates that the doubter was able to correctly identify the return value of the reference solution when given the chosen input.

4.2.4.5 Executions and results

For the implementation of the abstract evaluation detailed in Section 3.2.2.3, we rely on a three steps approach:

- Given a player solution, append to it the specific wrapper created in Section 4.2.4.2 and the test cases created in Section 4.2.4.4.
- Compile the resulting ".cpp" file.
- If the compilation was successful, run the executable that was created using the appropriate tags.

Catch2 offers the possibility of executing only a part of the tests in a file by specifying any combination of tags before execution, for example after a file containing one test tagged as "[base]" and one test tagged as "[final]" is compiled, it is possible to execute only one of the two tests by specifying their respective tag. If both tags are specified both tests will be executed and, if no tags are specified, all tests in the file will be executed, regardless of their tag.

In DubitaC we take advantage of the tag system to reduce the waiting time that might be generated by longer executions.

After the relevant tests have been created, the executions depend on the current scene:

- If a user is testing its own solution, the base tests are appended and then the solution is compiled and executed using tag "[base]".
- 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 server executes, for each user solution, only the tests with tag "[final]".

Regarding the second point, the server needs to send back the solutions to the clients because the clients do not have the data required to create and append the test from the correct doubts to their solution. It is more convenient to send a single string to each client than to share the data required to complete the setup process locally.

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. In case the test fatally crashed, the error raised from the system is returned instead.

Lastly, all the results that would be displayed on the standard output are intercepted, like in Section 4.2.4.1, and parsed to calculate the points that need to be assigned to the clients based on the performance of their solution and their doubts.

4.2.4.6 Reducing the compilation time

At the beginning of DubitaC development, the compilation of a solution with tests would take up to 3 minutes.

This waiting time was unacceptable, as the game would need to provide the results in a reasonable timeframe so that the players would not get bored during a long wait.

We were able to determine that the long compilation was 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.

Regarding the first point, one major drawback of using Catch2 is the need to compile a "main" Catch2 file that would initialize the test framework.

In Section 4.2.4, we mentioned adding the main directive at the top of the user solution.

While that statement is not incorrect, it would mean that the environment is reinitialized at every compilation, increasing significantly the compilation time.

The proper way consist of creating a simple auxiliary cpp file 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"
```

We then compile this new ".cpp" file only once at the beginning of a game session, using the following terminal command:

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

Where the "-c" flag is used to create a linkable file instead of an executable file.

Now we can compile our solutions, as many times as we want, alongside this linkable file to obtain a considerable speedup. Bringing down the compilation time from around 3 minutes to slightly more than 2 minutes.

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

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

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

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

The Catch2 community suggests using 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 file alongside our linkable file, the total compilation time decreased from around 2 minutes to a, much more reasonable, 20 seconds.

4.2.5 Managing the players' progress

4.2.5.1 The avatars

As we previously stated in Section 3.2.2.6, the points awarded to the players at the end of a game session will influence the availability of the avatars that the players can select during a future game session.

We implemented the avatar concept of Section 3.2.1 by using small sprites that are shown to the players from the doubting round until the end of the game 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 sprites 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 unavailable ones are obscured making the reward of knowing what the avatar represents more interesting.

An example sprite is shown in Figure 4.9, each avatar sprite is a 64x16 png image that can be split into 4 different 16x16 "states":

- A standard avatar that is shown throughout the game.
- An obscured version of the avatar that is only shown if the player has not enough points to select it.
- An "angry" state that is only shown when the player does not reach the top 3 at the end of a game session.
- A "happy" state that is only shown when the player reaches the top 3 at the end of a game session.

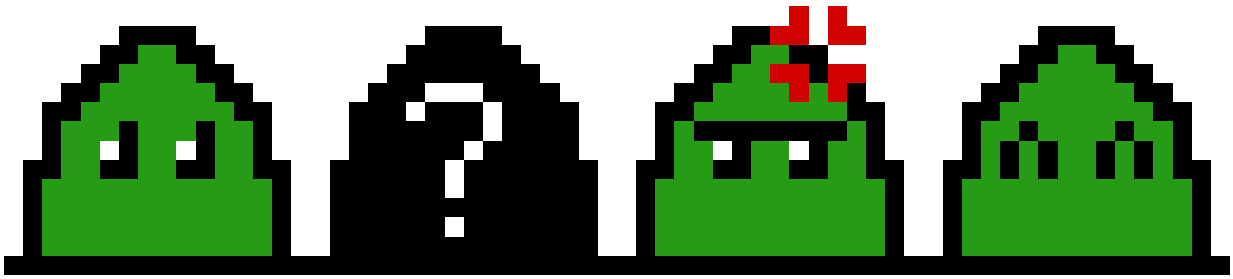


Figure 4.9: The "Slime" avatar's sprite that can be selected during the game. The avatar is repeated 4 times, once for each of the possible states: normal, locked, angry and happy.

4.2.5.2 Saving the progress

Regarding the storage of the players' progress, we implemented a simple authentication system where all unique login credentials are stored alongside the player's progress on a local database file.

Since the database file is stored locally on the admin's machine, maintainers and admins can access and edit the database, if needed, but cannot retrieve any of the users' passwords, since they have been stored securely following the best practices detailed in [Hor21].

4.2.6 The codeQuestions implementation

Aside from the players and the admin, it is possible to identify the additional role of the maintainer of the game which does not participate in the game sessions but is responsible for the extension of its parts. Their main task consists in creating new codeQuestion while making sure to correctly include all of their details described in Section 3.2.2.1, and to add them to the already available list.

To streamline the creation process, we implemented a modular system to convert files that contain reference C++ solutions into codeQuestions that can be added, loaded and solved during a game session.

We introduce the concept of **detail label**, which is a string following a specific format that indicates that the following text, until the next label or the end of the file is met, fulfils the role represented by the label.

In the current version, the number of existing detail labels is seven, however, they differ slightly from the previously mentioned codeQuestion details.

Additionally, the labels can be categorized as:

- "Mandatory", that must be present in the file otherwise the codeQuestion will not be loaded.
- "Recommended", that are needed to correctly experience a DubitaC game session, but the codeQuestion will be loaded anyway if they are missing.
- "Optional", that might be needed for some codeQuestions, but their absence will never interfere with the loading procedure.

In Table 4.1, we give a brief summary of each label.

Additionally, a localization label corresponding to the description of the codeQuestion must be always present as the first line of the file.

	Category	Summary
//_main_//	Mandatory	Marks the function signature of the tested function.
//_question_//	Optional	In case the codeQuestion requires some kind of setup, marks the reference function.
//_tags_//	Recommended	Marks a comma separated list of tags used during the filtering of the codeQuestions.
//_hints_//	Recommended	Marks a comma separated list of localization labels that will be used for the hints asked by the players.
//_base_//	Recommended	Marks the test case that players can use against their solutions during the first round.
//_final_//	Recommended	Marks the test case that will be used during the final evaluation against each player solution.
//_limits_//	Optional	Marks the beginning of a block detailing the limits for the main function arguments.

Table 4.1: Detail labels that can be used in a codeQuestion file.

4.2.6.1 Example process to create a codeQuestion

Let us assume that the desired codeQuestion represents the programming challenge:

> Given a single digit natural number, return the smallest prime that is bigger or equal than it.

The request is easy and, because of the restriction on the input, there is no need to implement a prime detector. A possible C++ solution might be:

```
int fun(int n){
  int primes[5] = {2, 3, 5, 7, 11};

  for(int i = 0; i < 5; i++){
    if(n <= primes[i]){return primes[i];}
  }
}
```

To make sure that the solution satisfies all of our requirements, we should create 2 test cases following Catch2's syntax, one containing some basic cases and one containing both basic cases and edge cases.

```
TEST_CASE("Basic", "[base]"){
  CHECK(fun(3) == 3);
  CHECK(fun(6) == 7);
}

TEST_CASE("Edge", "[final]"){
  CHECK(fun(0) == 2);
  CHECK(fun(5) == 5);
  CHECK(fun(9) == 11);
}
```

Now, assuming that the alternative linkable file described in Section 4.2.4.6 already exists, we add to the top of our solution:

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

Then we compile and run the solution, if all tests have passed, we can continue with the conversion following these steps:

- Modify the file extension from ".cpp" to ".txt".
- Add the localization label at the top of the file, for example //_small_primes//.
- Add the //_main_// label above the function signature.
- Add the //_base_// label above the "[base]" test case.
- Add the //_final_// label above the "[final]" test case.
- Remove the "catch.hpp" include.

Additionally, the test cases should be modified slightly to "enable" the use of our wrapper created in Section 4.2.4.2:

```
TEST_CASE("Basic", "[base]"){
  CHECK(fun(TIMEOUT, 3) == 3);
  CHECK(fun(TIMEOUT, 6) == 7);
}

TEST_CASE("Edge", "[final]"){
  CHECK(fun(TIMEOUT, 0) == 2);
  CHECK(fun(TIMEOUT, 5) == 5);
  CHECK(fun(TIMEOUT, 9) == 11);
}
```

At this point, the codeQuestion conversion is completed and it could already be included in the correct folder to be loaded during the next game session.

As good practice, some additional steps should be taken so that the codeQuestion details are more complete:

- Add to each localization file the "_small_primes" label and its corresponding text, for example, the one at the beginning of this section could be used for the English localization.
- Add the //_tags_// label followed by at least one tag that describes the codeQuestion category, for example, LOOPS.
- Add the //_hint_// label followed by at least one localization label referring to a hint that could be given to struggling players, for example, "_small_primes_hint_1".
- Add to each localization file the hints labels and their corresponding text, for example, "Remember that 1 is not prime".

Lastly, to respect the limitations on the input, it is possible to add the //_limits_// label followed by some limits written in the format explained in Section 4.2.6.2.

For this codeQuestion one of the possible ways to write the limit is:

```
n :: [0, 10)
```

After following the whole procedure, the original solution file has been completely converted into a correct codeQuestion containing:

```
//_small_primes//

//_tags_//
LOOPS

//_hints_//
_small_primes_hint_1

//_limits_//
n :: (0, 10)

//_main_//
int fun(int n){
  int primes[5] = {2, 3, 5, 7, 11} ;

  for(int i = 0; i < 5; i++){
    if(n <= primes[i]){return primes[i];}
  }
}

//_base_//
TEST_CASE("Basic", "[base]"){
  CHECK(fun(TIMEOUT, 3) == 3);
  CHECK(fun(TIMEOUT, 6) == 7);
}

//_final_//
TEST_CASE("Edge", "[final]"){
  CHECK(fun(TIMEOUT, 0) == 2);
  CHECK(fun(TIMEOUT, 5) == 5);
  CHECK(fun(TIMEOUT, 9) == 11);
}
```

4.2.6.2 The limits format

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

It is possible to limit the inputs that can be given to a function using the following syntax:

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

Where the list of variableNames contains at least one variable and all of them must use the same identifier of one of the parameters in the function signature.

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 two 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.

4.3 The Windows installer

To facilitate the usage of the application we decided to create a Windows installer so that users will only need to download and run a single executable file to be immediately ready to play.

The installer has been created using Inno Setup [Rus97], which is a free software that aids users in creating their own installers for their applications, it offers different customization options and a visual interface to create and compile the setup scripts. It was chosen because of its popularity and ease of use.

The installer will ask between:

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

Each choice will install the corresponding build and, only in the case of a Server installation, setup the database file.

The installation includes:

- A DubitaC build, with all the necessary resources to start the game and an executable file to start playing.
- 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 provided "UpToDateMinGw" directory.

4.4 Current status of development

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

`https://github.com/WeLikeIke/DubitaC/tree/v1.3.0`

The repository contains:

- All the source code and assets used.
- The documentation containing the description of every class and method of each script.
- The two latest server and client builds.
- The Windows installer as the latest release.
- This thesis paper.

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

DubitaC is currently at version 1.3 containing:

- The complete implementation of the main gameplay.
- Various elements to increase students' engagement while understanding basic C++ concepts.
- 2 fully localized languages, English and Italian.
- An offline authentication system using a local database.
- 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.

Chapter 5

Conclusions

To conclude our work, we present the limitations and some possible future extensions followed by a final closing statement.

5.1 Limitations

Currently, DubitaC suffers from the following limitations:

- Because of how the tests are evaluated, it is currently not possible to distinguish an unexpected exception from a correctly thrown one, meaning that the codeQuestions should not ask the players to throw exceptions when correct to do so.
- Because of a lack of resources, the whole project was tested on not more than 2 machines. As the expected number of players is around 20, it was not possible to check if any unexpected behaviours arise when increasing the number of distributed clients connected to the server.
- Because of the Catch2 requirements, the installation has to include a folder containing the Catch2 header and the LLVM linker "lld".
- 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. This may introduce errors in the automatic evaluation. Indeed, if the solution needs any of those libraries, and the players forget to include them, during compilation they will receive no error warning them of their mistake, nor the static error will be spotted by the automatic evaluation, and, finally, compilation doubts against it will fail, though they should succeed.

- Because of the way the localization system works, it was not possible to use it for the player log. Since terminal output and parsed messages are mixed into a single textfield, the log could not be translated and the default English language is always used.

5.2 Future work

DubitaC can be extended in various parts, here we offer some possible suggestions for future work:

- Extending the doubting options, and corresponding testing, for the players by letting them doubt on specific exceptions.
- Extending the languages available by creating and integrating more localization files.
- Extending the list of available avatars by creating and integrating more sprites.
- Extending the list of available codeQuestions by creating and integrating more text files with the correct codeQuestion syntax.
- To improve the user experience for a maintainer implementing some of these extensions, an interface or a Unity specific tool can be created 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 from the terminal, like java for example.
- Adding more polish, like music or feedback for the user.
- Making the game 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.

5.3 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 to the official Github repository.

Bibliography

[Abt87] Clark C Abt. *Serious games*. University press of America, 1987. URL: https://link.springer.com/book/10.1007/978-3-319-40612-1.

[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.

[Cod22] Codewars development team. Gamification. Code wars Documentation, 2022. URL: https://docs.codewars.com/gamification.

[Cor18] Tomorrow Corporation. 7 billion humans. Videogame, 2018. URL: https://tomorrowcorporation.com/7billionhumans.

[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.

[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.

[Hor21] Taylor Hornby. Salted password hashing - doing it right. Online website for security research and development, Sep 2021. URL: https://crackstation.net/hashing-security.htm.

[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/.

[LEES14] Fedwa Laamarti, Mohamad Eid, and Abdulmotaleb El Saddik. An overview of serious games. *International Journal of Computer Games Technology*, 2014. URL: https://dl.acm.org/doi/pdf/10.1155/2014/358152.

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

[SJB07] Tarja Susi, Mikael Johannesson, and Per Backlund. Serious games: An overview. 2007. URL: https://www.diva-portal.org/smash/get/diva2:2416/fulltext01.pdf.

[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.

[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.

[vH11] Luis von Ahn and Severin Hacker. Duolingo. Online platform, 2011. URL: https://www.duolingo.com/.

[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 Setup phase

- 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 between the available ones.

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

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

A.1.1.2 Solution creation round

- Read the description of the codeQuestion.
- Type the solution in the notepad in the centre 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 doubting 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 evaluation phase

- 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 evaluation phase shows the final leaderboard of the game session.

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

The setup phase 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 doubting round is ordered using the current leaderboard, it is more advantageous to start doubting from the top since that will increase the chances of bringing them 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 Setup pahes

- 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 solution creation round will be loaded when the "Start" button in the bottom right corner is pressed.

A.1.2.2 Solution creation round

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

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

A.1.2.3 Doubting 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 evaluation phase will be loaded when the server has completed the final executions.

A.1.2.5 Final Evaluation

- 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 setup phase will be loaded after pressing on the "Disconnect" button.

A.1.2.6 Tips

The admin interaction is minimal after the setup phase, however, after the final evaluation phase has been loaded, it is important to save the user solutions 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

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

DubitaC was developed using Unity version 2020.3.30f11[1], which is considered the 2020 LTS (Long Term Support) release, meaning that newer features and fixes will be available until March 2023. Afterwards, it is recommended to migrate the game to a newer Unity version.

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.

A.2.2 Extending existing systems

A.2.2.1 Extending the localization

The localization system is completely managed by the Unity localization package.[2]

[1]Unity editor available here: `https://unity3d.com/unity/qa/lts-releases`

[2]Package manual available here: `https://docs.unity3d.com/Packages/com.unity.localization@1.0/manual/index.html`

```
/// <summary>
/// Class that manages the notepads, implements autocentering and Undo-Redo.
/// The maximum amount of zoom possible is stored in constant <see cref="maxAllowedZoom"/>.
/// The minimum amount of zoom possible is stored in constant <see cref="minAllowedZoom"/>.
/// The zoom granularity is stored in constant <see cref="zoomStepSize"/>.
/// The offset required to avoid that the cursor goes off screen while typing is stored in constant <see cref="typingOffset"/>.
/// </summary>
public class NotepadManager : MonoBehaviour {
    private const float maxAllowedZoom = 2f;
    private const float minAllowedZoom = 0.2f;
    private const float zoomStepSize   = 0.2f;
    private const float typingOffset   = 150f;
```

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

To introduce a **new** localization you would need to provide:

- A sprite of the country flag representing the language being added.
- A localization CSV file containing "label - comma - localized text" in each row.

From then on, it is only necessary to add the support for your chosen locale and to update the relative tables:

- The "CountryFlags" table by adding the new sprite.
- The "Strings" table by importing the CSV.

To extend **existing** localizations, simply insert the new labels in all localization files following the localization format and then reimport all CSV files in the "Strings" table.

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 submission 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.2, 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.

```
/// <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.2: Signature of the function used to validate a client login, shown alongside its summary.

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 into 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.3.
- 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 4.2.6.

To insert a **new** codeQuestion:

- Create and test the cpp file that corresponds to the reference solution to the codeQuestion.
- Follow the conversion steps detailed in Section 4.2.6.1.
- 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.4.

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.5.

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 certain 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 solution requires more than 3 seconds for an execution, **select a higher TIMEOUT value before starting the next game session**.

The maximum selectable TIMEOUT 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 value"

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 value" 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 reference 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 reference 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 reference 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 correctly guessed only one of the outputs between the reference 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 depends 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 disconnection 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 always 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 errors or exceptions in the execution.

If it is possible to continue playing, **Press on the "Close" button that appeared 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.

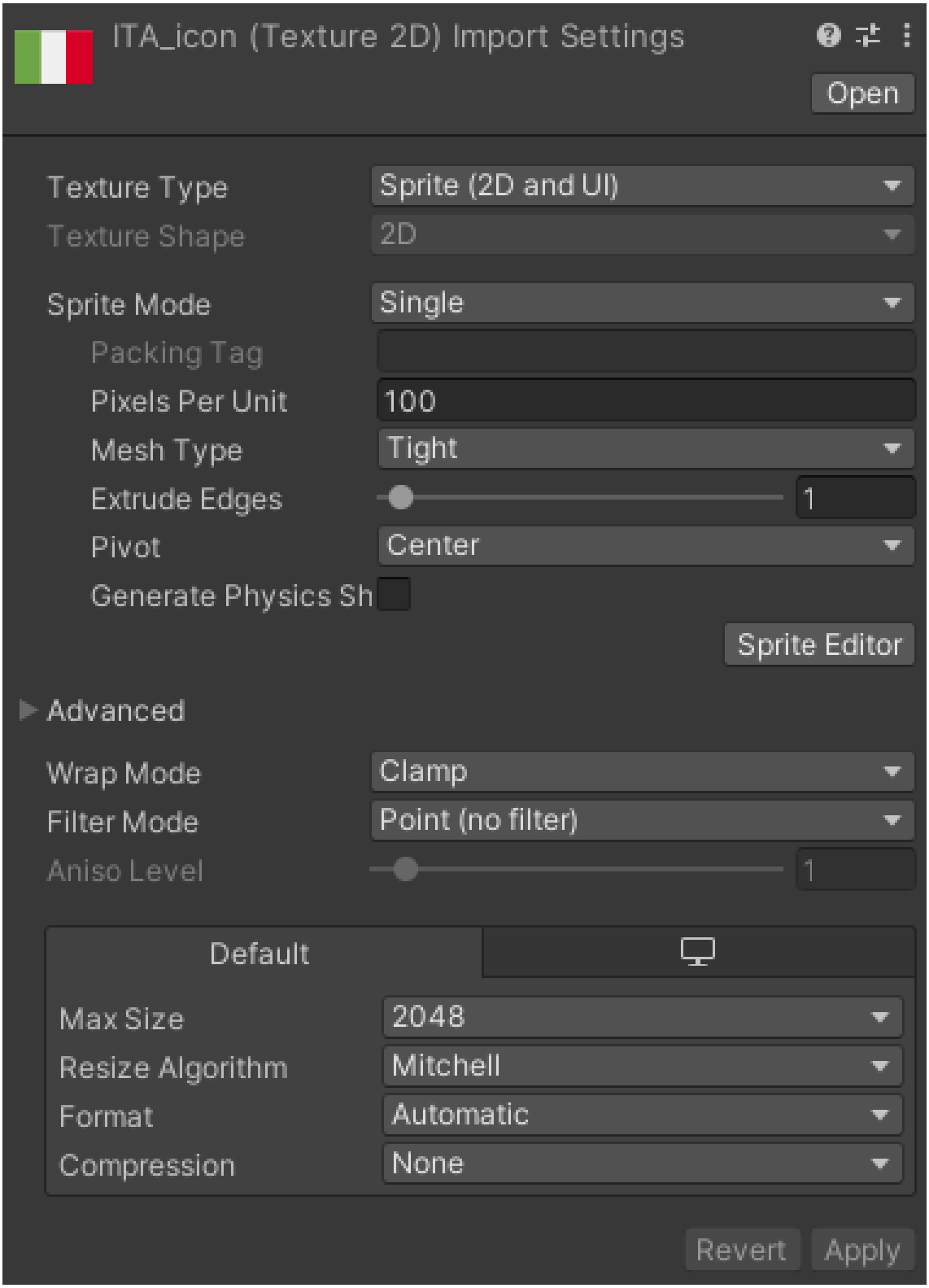


Figure A.3: Settings to be used to convert an image into a sprite using the Unity inspector.

```
/// <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.4: Asynchronous function used to spawn a terminal and execute arbitrary commands on it.

```
    /// <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.5: Example function that uses directives like "#if UNITY_STANDALONE_LINUX" to execute different commands based on the detected platform.